

The Swift Programming Language

Swift 5.1 Edition



About Swift

Swift is a fantastic way to write software, whether it's for phones, desktops, servers, or anything else that runs code. It's a safe, fast, and interactive programming language that combines the best in modern language thinking with wisdom from the wider Apple engineering culture and the diverse contributions from its open-source community. The compiler is optimized for performance and the language is optimized for development, without compromising on either.

Swift is friendly to new programmers. It's an industrial-quality programming language that's as expressive and enjoyable as a scripting language. Writing Swift code in a playground lets you experiment with code and see the results immediately, without the overhead of building and running an app.

Swift defines away large classes of common programming errors by adopting modern programming patterns:

- Variables are always initialized before use.
- Array indices are checked for out-of-bounds errors.
- Integers are checked for overflow.
- Optionals ensure that nil values are handled explicitly.
- Memory is managed automatically.
- Error handling allows controlled recovery from unexpected failures.

Swift code is compiled and optimized to get the most out of modern hardware. The syntax and standard library have been designed based on the guiding principle that the obvious way to write your code should also perform the best. Its combination of safety and speed make Swift an excellent choice for everything from "Hello, world!" to an entire operating system.

Swift combines powerful type inference and pattern matching with a modern, lightweight syntax, allowing complex ideas to be expressed in a clear and

concise manner. As a result, code is not just easier to write, but easier to read and maintain as well.

Swift has been years in the making, and it continues to evolve with new features and capabilities. Our goals for Swift are ambitious. We can't wait to see what you create with it.

Version Compatibility

This book describes Swift 5.1, the default version of Swift that's included in Xcode 11. You can use Xcode 11 to build targets that are written in either Swift 5.1, Swift 4.2, or Swift 4.

When you use Xcode 11 to build Swift 4 and Swift 4.2 code, most Swift 5.1 functionality is available. That said, the following changes are available only to code that uses Swift 5.1 or later:

- Functions that return an opaque type require the Swift 5.1 runtime.
- The try? expression doesn't introduce an extra level of optionality to expressions that already return optionals.
- Large integer literal initialization expressions are inferred to be of the correct integer type. For example, UInt64(0xffff_ffff_ffff) evaluates to the correct value rather than overflowing.

A target written in Swift 5.1 can depend on a target that's written in Swift 4.2 or Swift 4, and vice versa. This means, if you have a large project that's divided into multiple frameworks, you can migrate your code from Swift 4 to Swift 5.1 one framework at a time.

A Swift Tour

Tradition suggests that the first program in a new language should print the words "Hello, world!" on the screen. In Swift, this can be done in a single line:

```
print("Hello, world!")
// Prints "Hello, world!"
```

If you have written code in C or Objective-C, this syntax looks familiar to you—in Swift, this line of code is a complete program. You don't need to import a separate library for functionality like input/output or string handling. Code written at global scope is used as the entry point for the program, so you don't need a main() function. You also don't need to write semicolons at the end of every statement.

This tour gives you enough information to start writing code in Swift by showing you how to accomplish a variety of programming tasks. Don't worry if you don't understand something—everything introduced in this tour is explained in detail in the rest of this book.

NOTE

On a Mac with Xcode installed, or on an iPad with Swift Playgrounds, you can open this chapter as a playground. Playgrounds allow you to edit the code listings and see the result immediately.

Download Playground

Simple Values

Use <u>let</u> to make a <u>constant</u> and <u>var</u> to make a <u>variable</u>. The value of a constant doesn't need to be known at compile time, but you must <u>assign it</u> a value <u>exactly</u> <u>once</u>. This means you can use constants to name a value that you determine once but use in many places.

```
var myVariable = 42
myVariable = 50
let myConstant = 42
```

let implicitInteger = 70

A constant or variable must have the same type as the value you want to assign to it. However, you don't always have to write the type explicitly. Providing a value when you create a constant or variable lets the compiler infer its type. In the example above, the compiler infers that myVariable is an integer because its initial value is an integer.

If the initial value doesn't provide enough information (or if there is no initial value), specify the type by writing it after the variable, separated by a colon.

```
2 let implicitDouble = 70.0
3 let explicitDouble: Double = 70

EXPERIMENT
Create a constant with an explicit type of Float and a value of 4.
```

Values are never implicitly converted to another type. If you need to convert a value to a different type, explicitly make an instance of the desired type.

```
1 let label = "The width is "
2 let width = 94
3 let widthLabel = label + String(width)
```

```
EXPERIMENT

Try removing the conversion to String from the last line. What error do you get?
```

There's an even simpler way to include values in strings: Write the value in parentheses, and write a backslash $(\)$ before the parentheses. For example:

```
1 let apples = 3
2 let oranges = 5
3 let appleSummary = "I have \((apples)\) apples."
4 let fruitSummary = "I have \((apples + oranges)\) pieces of fruit."
```

```
EXPERIMENT

Use \( ) to include a floating-point calculation in a string and to include someone's name in a greeting.
```

Use three double quotation marks (""") for strings that take up multiple lines. Indentation at the start of each quoted line is removed, as long as it matches the indentation of the closing quotation marks. For example:

```
1 let quotation = """
2 I said "I have \(apples) apples."
3 And then I said "I have \(apples + oranges) pieces of fruit."
4 """
```

Create arrays and dictionaries using brackets ([]), and access their elements by writing the index or key in brackets. A comma is allowed after the last element.

Arrays automatically grow as you add elements.

```
shoppingList.append("blue paint")
print(shoppingList)
```

To create an empty array or dictionary, use the initializer syntax.

```
1 let emptyArray = [String]()
2 let emptyDictionary = [String: Float]()
```

If type information can be inferred, you can write an empty array as [] and an empty dictionary as [:]—for example, when you set a new value for a variable or pass an argument to a function.

```
shoppingList = (]

coccupations = (:)
```

Control Flow

Use if and switch to make conditionals, and use for-in, while, and repeat-while to make loops. Parentheses around the condition or loop variable are optional. Braces around the body are required.

```
let individualScores = [75, 43, 103, 87, 12]
1
    var teamScore = 0
2
    for score in individualScores {
3
        if score > 50 {
4
             teamScore += 3
5
        } else {
6
             teamScore += 1
8
        }
9
    }
10
    print(teamScore)
```

```
11 // Prints "11"
```

In an if statement, the conditional must be a Boolean expression—this means that code such as if score { . . . } is an error, not an implicit comparison to zero.

You can use if and let together to work with values that might be missing. These values are represented as optionals. An optional value either contains a value or contains nil to indicate that a value is missing. Write a question mark (?) after the type of a value to mark the value as optional.

```
var optionalString: String? = "Hello"
1
   print(optionalString == nil)
2
3
   // Prints "false"
4
   var optionalName: String? = "John Appleseed"
5
   var greeting = "Hello!"
6
7
   if let name = optionalName {
        greeting = "Hello, \(name)"
8
9
   }
```

EXPERIMENT

Change optionalName to nil. What greeting do you get? Add an else clause that sets a different greeting if optionalName is nil.

If the optional value is nil, the conditional is false and the code in braces is skipped. Otherwise, the optional value is unwrapped and assigned to the constant after let, which makes the unwrapped value available inside the block of code.

Another way to handle optional values is to provide a default value using the ?? operator. If the optional value is missing, the default value is used instead.

```
1 let nickName: String? = nil
2 let fullName: String = "John Appleseed"
```

Switches support any kind of data and a wide variety of comparison operations—they aren't limited to integers and tests for equality.

```
1
    let vegetable = "red pepper"
2
    switch vegetable {
    case "celery":
3
        print("Add some raisins and make ants on a log.")
4
    case ("cucumber", ("watercress"):
        print("That would make a good tea sandwich.")
6
7
    case let x where x.hasSuffix("pepper"):
        print("Is it a spicy (x)?")
8
    default:
9
         print("Everything tastes good in soup.")
10
    }
11
    // Prints "Is it a spicy red pepper?"
12
```

```
EXPERIMENT

Try removing the default case. What error do you get?
```

Notice how let can be used in a pattern to assign the value that matched the pattern to a constant.

After executing the code inside the switch case that matched, the program exits from the switch statement. Execution doesn't continue to the next case, so there is no need to explicitly break out of the switch at the end of each case's code.

You use for-in to iterate over items in a dictionary by providing a pair of names to use for each key-value pair. Dictionaries are an unordered collection, so their keys and values are iterated over in an arbitrary order.

```
1 let interestingNumbers = [
```

```
2
         "Prime": [2, 3, 5, 7, 11, 13],
         "Fibonacci": [1, 1, 2, 3, 5, 8],
3
         "Square": [1, 4, 9, 16, 25],
4
5
    ]
    var largest = 0
6
7
    for (kind, numbers) in interestingNumbers {
         for number in numbers {
8
             if number > largest {
9
                 largest = number
10
             }
11
         }
12
13
     }
14
    print(largest)
15
     // Prints "25"
```

EXPERIMENT

Add another variable to keep track of which kind of number was the largest, as well as what that largest number was.

Use while to repeat a block of code until a condition changes. The condition of a loop can be at the end instead, ensuring that the loop is run at least once.

```
1
    var n = 2
2
    while n < 100  {
        n *= 2
3
4
    }
    print(n)
5
    // Prints "128"
6
7
8
    var m = 2
    repeat {
9
```

You can keep an index in a loop by using ... to make a range of indexes.

```
1  var total = 0
2  for i in 0..<4 {
3    total += i
4  }
5  print(total)
6  // Prints "6"</pre>
```

Use ... < to make a range that omits its upper value, and use to make a range that includes both values.

Functions and Closures

Use func to declare a function. Call a function by following its name with a list of arguments in parentheses. Use -> to separate the parameter names and types from the function's return type.

```
func greet(person: String, day: String) -> String {
   return "Hello \(person), today is \(day)."
}
greet(person: "Bob", day: "Tuesday")
```

EXPERIMENT

Remove the day parameter. Add a parameter to include today's lunch special in the greeting.

By default, functions use their parameter names as labels for their arguments. Write a custom argument label before the parameter name, or write to use no argument label.

```
func greet(_ person: String, on day: String) -> String {
   return "Hello \(person), today is \(day)."
}
greet("John", on: "Wednesday")
```

Use a tuple to make a compound value—for example, to return multiple values from a function. The elements of a tuple can be referred to either by name or by number.

```
func calculateStatistics(scores: [Int]) -> (min: Int, max: Int,
1
          sum: Int) {
         var min = scores[0]
2
         var max = scores[0]
3
         var sum = 0
4
5
         for score in scores {
6
7
             if score > max {
                 max = score
8
9
             } else if score < min {</pre>
                 min = score
10
11
             }
             sum += score
12
13
         }
14
15
         return (min, max, sum)
16
     }
17
     let statistics = calculateStatistics(scores: [5, 3, 100, 3,
```

```
9])

18 print(statistics.sum)

19 // Prints "120"

20 print(statistics.2)

21 // Prints "120"
```

Functions can be nested. Nested functions have access to variables that were declared in the outer function. You can use nested functions to organize the code in a function that is long or complex.

```
func returnFifteen() -> Int {
    var y = 10

func add() {
    y += 5
}

add()

return y
}

returnFifteen()
```

Functions are a first-class type. This means that a function can return another function as its value.

```
func makeIncrementer() -> ((Int) -> Int) {
   func addOne(number: Int) -> Int {
     return 1 + number
}

return addOne

var increment = makeIncrementer()

increment(7)
```

A function can take another function as one of its arguments.

```
1
     func hasAnyMatches(list: [Int], condition: (Int) -> Bool) ->
            Bool {
         for item in list {
 2
              if condition(item) {
 3
                   return true
 5
              }
 6
         }
         return false
 7
8
     func lessThanTen(number: Int) -> Bool {
9
          return number < 10
10
11
     var numbers = [20, 19, 7, 12]
12
     hasAnyMatches(list: numbers, condition: <a href="lessThanTen">lessThanTen</a>)
13
```

Functions are actually a special case of closures: blocks of code that can be called later. The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it is executed—you saw an example of this already with nested functions. You can write a closure without a name by surrounding code with braces ({}). Use in to separate the arguments and return type from the body.

```
numbers.map({ (number: Int) -> Int in
let result = 3 * number
return result
})
```

EXPERIMENT

Rewrite the closure to return zero for all odd numbers.

You have several options for writing closures more concisely. When a closure's type is already known, such as the callback for a delegate, you can omit the type of its parameters, its return type, or both. Single statement closures implicitly return the value of their only statement.

```
1 let mappedNumbers = numbers.map({ number in 3 * number })
2 print(mappedNumbers)
3 // Prints "[60, 57, 21, 36]"
```

You can refer to parameters by number instead of by name—this approach is especially useful in very short closures. A closure passed as the last argument to a function can appear immediately after the parentheses. When a closure is the only argument to a function, you can omit the parentheses entirely.

```
let sortedNumbers = numbers.sorted { $0 > $1 }
print(sortedNumbers)
// Prints "[20, 19, 12, 7]"
```

Objects and Classes

Use class followed by the class's name to create a class. A property declaration in a class is written the same way as a constant or variable declaration, except that it is in the context of a class. Likewise, method and function declarations are written the same way.

```
class Shape {
    var numberOfSides = 0

func simpleDescription() -> String {
    return "A shape with \(numberOfSides) sides."
}
```

Add a constant property with let, and add another method that takes an argument.

Create an instance of a class by putting parentheses after the class name. Use dot syntax to access the properties and methods of the instance.

```
var shape = Shape()
shape.numberOfSides = 7
var shapeDescription = shape.simpleDescription()
```

This version of the Shape class is missing something important: an initializer to set up the class when an instance is created. Use init to create one.

```
class NamedShape {
1
2
         var numberOfSides: Int = 0
3
         var name: String
4
5
         init(name: String) {
             self.name = name
6
7
         }
8
         func simpleDescription() -> String {
9
             return "A shape with \(numberOfSides) sides."
10
         }
11
12
    }
```

Notice how self is used to distinguish the name property from the name argument to the initializer. The arguments to the initializer are passed like a function call when you create an instance of the class. Every property needs a value assigned —either in its declaration (as with numberOfSides) or in the initializer (as with name).

Use deinit to create a deinitializer if you need to perform some cleanup before

the object is deallocated.

Subclasses include their superclass name after their class name, separated by a colon. There is no requirement for classes to subclass any standard root class, so you can include or omit a superclass as needed.

Methods on a subclass that override the superclass's implementation are marked with override—overriding a method by accident, without override, is detected by the compiler as an error. The compiler also detects methods with override that don't actually override any method in the superclass.

```
1
    class Square: NamedShape {
         var sideLength: Double
2
3
         init(sideLength: Double, name: String) {
4
             self.sideLength = sideLength
5
             super init(name: name)
6
             numberOfSides = 4
7
8
9
         func area() -> Double {
10
             return sideLength * sideLength
11
         }
12
13
         override func simpleDescription() -> String {
14
             return "A square with sides of length \((sideLength)."
15
         }
16
    }
17
    let test = Square(sideLength: 5.2, name: "my test square")
18
19
    test.area()
20
     test.simpleDescription()
```

EXPERIMENT

Make another subclass of NamedShape called Circle that takes a radius and a name as arguments to its initializer. Implement an area() and a simpleDescription() method on the Circle class.

In addition to simple properties that are stored, properties can have a getter and a setter.

```
1
    class EquilateralTriangle: NamedShape {
2
         var sideLength: Double = 0.0
3
         init(sideLength: Double, name: String) {
4
             self.sideLength = sideLength
5
6
             super.init(name: name)
             numberOfSides = 3
7
         }
8
9
         var perimeter: Double {
10
11
             get {
                 return 3.0 * sideLength
12
13
             }
14
             set {
                 sideLength = newValue / 3.0
15
16
             }
17
18
         override func simpleDescription() -> String {
19
             return "An equilateral triangle with sides of length \
20
           (sideLength)."
21
         }
22
     }
```

In the setter for perimeter, the new value has the implicit name newValue. You can provide an explicit name in parentheses after set.

Notice that the initializer for the EquilateralTriangle class has three different steps:

- 1. Setting the value of properties that the subclass declares.
- 2. Calling the superclass's initializer.
- 3. Changing the value of properties defined by the superclass. Any additional setup work that uses methods, getters, or setters can also be done at this point.

If you don't need to compute the property but still need to provide code that is run before and after setting a new value, use willset and didset. The code you provide is run any time the value changes outside of an initializer. For example, the class below ensures that the side length of its triangle is always the same as the side length of its square.

```
class TriangleAndSquare {
   var triangle: EquilateralTriangle {
      willSet {
        square.sideLength = newValue.sideLength
      }
}
```

```
7
        var square: Square {
            willSet {
8
                 triangle.sideLength = newValue.sideLength
9
             }
10
11
         }
         init(size: Double, name: String) {
12
             square = Square(sideLength: size, name: name)
13
             triangle = EquilateralTriangle(sideLength: size, name:
14
          name)
        }
15
16
    }
17
    var triangleAndSquare = TriangleAndSquare(size: 10, name:
          "another test shape")
     print(triangleAndSquare.square.sideLength)
18
    // Prints "10.0"
19
    print(triangleAndSquare.triangle.sideLength)
20
    // Prints "10.0"
21
22
    triangleAndSquare.square = Square(sideLength: 50, name:
          "larger square")
    print(triangleAndSquare.triangle.sideLength)
23
    // Prints "50.0"
24
```

When working with optional values, you can write? before operations like methods, properties, and subscripting. If the value before the? is nil, everything after the? is ignored and the value of the whole expression is nil. Otherwise, the optional value is unwrapped, and everything after the? acts on the unwrapped value. In both cases, the value of the whole expression is an optional value.

Enumerations and Structures

Use enum to create an enumeration. Like classes and all other named types, enumerations can have methods associated with them.

```
1
    enum Rank: Int {
 2
         case ace = 1
 3
         case two, three, four, five, six, seven, eight, nine, ten
 4
         case jack, queen, king
 5
         func simpleDescription() -> String {
 6
 7
             switch self {
 8
             case .ace:
                 return "ace"
9
10
             case jack:
                 return "jack"
11
12
             case queen:
                 return "queen"
13
             case king:
14
                 return "king"
15
16
             default:
                 return String(self.rawValue)
17
18
             }
         }
19
20
    }
    let ace = Rank.ace
21
    let aceRawValue = ace.rawValue
22
```

EXPERIMENT

Write a function that compares two Rank values by comparing their raw values.

By default, Swift assigns the raw values starting at zero and incrementing by one each time, but you can change this behavior by explicitly specifying values. In the example above, Ace is explicitly given a raw value of 1, and the rest of the raw values are assigned in order. You can also use strings or floating-point numbers as the raw type of an enumeration. Use the rawValue property to access the raw value of an enumeration case.

Use the init?(rawValue:) initializer to make an instance of an enumeration from a raw value. It returns either the enumeration case matching the raw value or nil if there is no matching Rank.

```
if let convertedRank = Rank(rawValue: 3) {
    let threeDescription = convertedRank.simpleDescription()
}
```

The case values of an enumeration are actual values, not just another way of writing their raw values. In fact, in cases where there isn't a meaningful raw value, you don't have to provide one.

```
1
     enum Suit {
         case spades, hearts, diamonds, clubs
 2
 3
         func simpleDescription() -> String {
 4
 5
             switch self {
             case .spades:
 6
 7
                  return "spades"
 8
             case hearts:
                  return "hearts"
 9
             case diamonds:
10
                  return "diamonds"
11
             case .clubs:
12
                  return "clubs"
13
14
             }
```

```
15  }
16  }
17  let hearts = Suit.hearts
18  let heartsDescription = hearts.simpleDescription()
```

EXPERIMENT

Add a color() method to Suit that returns "black" for spades and clubs, and returns "red" for hearts and diamonds.

Notice the two ways that the hearts case of the enumeration is referred to above: When assigning a value to the hearts constant, the enumeration case Suit.hearts is referred to by its full name because the constant doesn't have an explicit type specified. Inside the switch, the enumeration case is referred to by the abbreviated form .hearts because the value of self is already known to be a suit. You can use the abbreviated form anytime the value's type is already known.

If an enumeration has raw values, those values are determined as part of the declaration, which means every instance of a particular enumeration case always has the same raw value. Another choice for enumeration cases is to have values associated with the case—these values are determined when you make the instance, and they can be different for each instance of an enumeration case. You can think of the associated values as behaving like stored properties of the enumeration case instance. For example, consider the case of requesting the sunrise and sunset times from a server. The server either responds with the requested information, or it responds with a description of what went wrong.

```
1  enum ServerResponse {
2    case result(String, String)
3    case failure(String)
4  }
5 
6  let success = ServerResponse.result("6:00 am", "8:09 pm")
7  let failure = ServerResponse.failure("Out of cheese.")
```

```
8
    switch success {
9
    case let .result(sunrise, sunset):
10
        print("Sunrise is at \(sunrise) and sunset is at \
11
          (sunset).")
    case let .failure(message):
12
        print("Failure... \(message)")
13
14
    }
    // Prints "Sunrise is at 6:00 am and sunset is at 8:09 pm."
15
```

```
EXPERIMENT
```

Add a third case to ServerResponse and to the switch.

Notice how the sunrise and sunset times are extracted from the ServerResponse value as part of matching the value against the switch cases.

Use struct to create a structure. Structures support many of the same behaviors as classes, including methods and initializers. One of the most important differences between structures and classes is that structures are always copied when they are passed around in your code, but classes are passed by reference.

```
1
    struct Card {
        var rank: Rank
2
        var suit: Suit
3
        func simpleDescription() -> String {
4
            return "The \(rank.simpleDescription()) of \
5
          (suit.simpleDescription())"
        }
6
7
   }
   let threeOfSpades = Card(rank: .three, suit: .spades)
8
    let threeOfSpadesDescription =
9
```

```
EXPERIMENT
```

Write a function that returns an array containing a full deck of cards, with one card of each combination of rank and suit.

Protocols and Extensions

Use protocol to declare a protocol.

```
protocol ExampleProtocol {
    var simpleDescription: String { get }
    mutating func adjust()
}
```

Classes, enumerations, and structs can all adopt protocols.

```
1
    class SimpleClass: ExampleProtocol {
        var simpleDescription: String = "A very simple class."
2
        var anotherProperty: Int = 69105
3
        func adjust() {
4
             simpleDescription += " Now 100% adjusted."
5
        }
6
7
    }
    var a = SimpleClass()
8
    a.adjust()
9
    let aDescription = a.simpleDescription
10
11
12
    struct SimpleStructure: ExampleProtocol {
13
        var simpleDescription: String = "A simple structure"
```

EXPERIMENT

Add another requirement to ExampleProtocol. What changes do you need to make to SimpleClass and SimpleStructure so that they still conform to the protocol?

Notice the use of the mutating keyword in the declaration of SimpleStructure to mark a method that modifies the structure. The declaration of SimpleClass doesn't need any of its methods marked as mutating because methods on a class can always modify the class.

Use extension to add functionality to an existing type, such as new methods and computed properties. You can use an extension to add protocol conformance to a type that is declared elsewhere, or even to a type that you imported from a library or framework.

```
1
    extension Int: ExampleProtocol {
         var simpleDescription: String {
2
             return "The number \(self)"
3
4
         mutating func adjust() {
5
             self += 42
         }
 7
8
    }
9
    print(7.simpleDescription)
     // Prints "The number 7"
10
```

Write an extension for the Double type that adds an absoluteValue property.

You can use a protocol name just like any other named type—for example, to create a collection of objects that have different types but that all conform to a single protocol. When you work with values whose type is a protocol type, methods outside the protocol definition are not available.

Even though the variable protocolvalue has a runtime type of SimpleClass, the compiler treats it as the given type of ExampleProtocol. This means that you can't accidentally access methods or properties that the class implements in addition to its protocol conformance.

Error Handling

You represent errors using any type that adopts the Error protocol.

```
1 enum PrinterError: Error {
2    case outOfPaper
3    case noToner
4    case onFire
5 }
```

Use throw to throw an error and throws to mark a function that can throw an error. If you throw an error in a function, the function returns immediately and the code that called the function handles the error.

```
func send(job: Int, toPrinter printerName: String) throws ->
    String {
    if printerName == "Never Has Toner" {
        throw PrinterError.noToner
    }
    return "Job sent"
}
```

There are several ways to handle errors. One way is to use do-catch. Inside the do block, you mark code that can throw an error by writing try in front of it. Inside the catch block, the error is automatically given the name error unless you give it a different name.

```
do {
   let printerResponse = try send(job: 1040, toPrinter: "Bi
        Sheng")
   print(printerResponse)
4   } catch {
        print(error)
6   }
7   // Prints "Job sent"
```

```
EXPERIMENT

Change the printer name to "Never Has Toner", so that the send(job:toPrinter:) function throws an error.
```

You can provide multiple catch blocks that handle specific errors. You write a pattern after catch just as you do after case in a switch.

```
do {
   let printerResponse = try send(job: 1440, toPrinter:
        "Gutenberg")
```

```
3
         print(printerResponse)
    } catch PrinterError.onFire {
4
         print("I'll just put this over here, with the rest of the
5
           fire.")
6
    } catch let printerError as PrinterError {
7
         print("Printer error: \(printerError\).")
    } catch {
8
         print(error)
9
10
    }
    // Prints "Job sent"
11
```

EXPERIMENT

Add code to throw an error inside the do block. What kind of error do you need to throw so that the error is handled by the first catch block? What about the second and third blocks?

Another way to handle errors is to use try? to convert the result to an optional. If the function throws an error, the specific error is discarded and the result is nil. Otherwise, the result is an optional containing the value that the function returned.

Use defer to write a block of code that is executed after all other code in the function, just before the function returns. The code is executed regardless of whether the function throws an error. You can use defer to write setup and cleanup code next to each other, even though they need to be executed at different times.

```
var fridgeIsOpen = false
let fridgeContent = ["milk", "eggs", "leftovers"]
```

```
3
    func fridgeContains(_ food: String) -> Bool {
4
         fridgeIsOpen = true
 5
         defer {
 6
             fridgeIsOpen = false
 7
         }
 8
 9
         let result = fridgeContent.contains(food)
10
         return result
11
12
    }
    fridgeContains("banana")
13
14
    print(fridgeIsOpen)
    // Prints "false"
15
```

Generics

Write a name inside angle brackets to make a generic function or type.

```
1
    func makeArray<Item>(repeating item: Item, numberOfTimes: Int)
          -> [Item] {
        var result = [Item]()
2
        for _ in 0..<numberOfTimes {</pre>
3
            result.append(item)
4
5
        }
        return result
6
7
   }
   makeArray(repeating: "knock", numberOfTimes: 4)
```

You can make generic forms of functions and methods, as well as classes,

enumerations, and structures.

```
// Reimplement the Swift standard library's optional type
enum OptionalValue<Wrapped> {
   case none
   case some(Wrapped)
}
var possibleInteger: OptionalValue<Int> = .none
possibleInteger = .some(100)
```

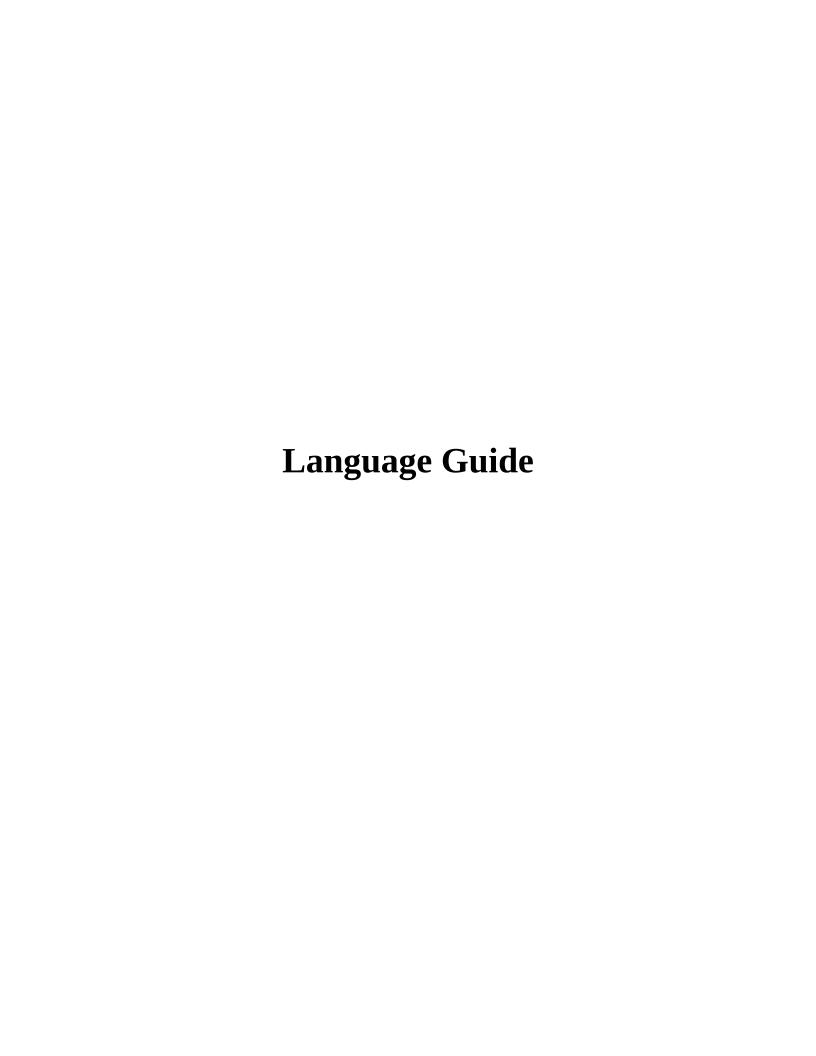
Use where right before the body to specify a list of requirements—for example, to require the type to implement a protocol, to require two types to be the same, or to require a class to have a particular superclass.

```
1
    func anyCommonElements<T: Sequence, U: Sequence>(_ lhs: T, _
           rhs: U) -> Bool
         where T.Element: Equatable, T.Element == U.Element
2
3
    {
         for lhsItem in lhs {
4
5
             for rhsItem in rhs {
6
                 if lhsItem == rhsItem {
7
                     return true
                 }
8
             }
9
10
         }
         return false
11
12
    }
     anyCommonElements([1, 2, 3], [3])
13
```

EXPERIMENT

Modify the any Common Elements ($_{:}$:) function to make a function that returns an array of the elements that any two sequences have in common.

Writing <T: Equatable> is the same as writing <T> ... where T: Equatable.



The Basics

Swift is a new programming language for iOS, macOS, watchOS, and tvOS app development. Nonetheless, many parts of Swift will be familiar from your experience of developing in C and Objective-C.

Swift provides its own versions of all fundamental C and Objective-C types, including Int for integers, Double and Float for floating-point values, Bool for Boolean values, and String for textual data. Swift also provides powerful versions of the three primary collection types, Array, Set, and Dictionary, as described in Collection Types.

Like C, Swift uses variables to store and refer to values by an identifying name. Swift also makes extensive use of variables whose values can't be changed. These are known as constants, and are much more powerful than constants in C. Constants are used throughout Swift to make code safer and clearer in intent when you work with values that don't need to change.

In addition to familiar types, Swift introduces advanced types not found in Objective-C, such as tuples. Tuples enable you to create and pass around groupings of values. You can use a tuple to return multiple values from a function as a single compound value.

Swift also introduces optional types, which handle the absence of a value. Optionals say either "there is a value, and it equals x" or "there isn't a value at all". Using optionals is similar to using nil with pointers in Objective-C, but they work for any type, not just classes. Not only are optionals safer and more expressive than nil pointers in Objective-C, they're at the heart of many of Swift's most powerful features.

Swift is a *type-safe* language, which means the language helps you to be clear about the types of values your code can work with. If part of your code requires a String, type safety prevents you from passing it an Int by mistake. Likewise, type safety prevents you from accidentally passing an optional String to a piece of code that requires a non-optional String. Type safety helps you catch and fix errors as early as possible in the development process.

Constants and Variables

Constants and variables associate a name (such as

maximumNumberOfLoginAttempts or welcomeMessage) with a value of a particular type (such as the number 10 or the string "Hello"). The value of a *constant* can't be changed once it's set, whereas a *variable* can be set to a different value in the future.

Declaring Constants and Variables

Constants and variables must be declared before they're used. You declare constants with the let keyword and variables with the var keyword. Here's an example of how constants and variables can be used to track the number of login attempts a user has made:

```
1 let maximumNumberOfLoginAttempts = 10
2 var currentLoginAttempt = 0
```

This code can be read as:

"Declare a new constant called maximumNumberOfLoginAttempts, and give it a value of 10. Then, declare a new variable called currentLoginAttempt, and give it an initial value of 0."

In this example, the maximum number of allowed login attempts is declared as a constant, because the maximum value never changes. The current login attempt counter is declared as a variable, because this value must be incremented after each failed login attempt.

You can declare multiple constants or multiple variables on a single line, separated by commas:

```
var \times = 0.0, y = 0.0, z = 0.0
```

NOTE

If a stored value in your code won't change, always declare it as a constant with the let keyword.

Type Annotations

You can provide a *type annotation* when you declare a constant or variable, to be clear about the kind of values the constant or variable can store. Write a type annotation by placing a colon after the constant or variable name, followed by a space, followed by the name of the type to use.

This example provides a type annotation for a variable called welcomeMessage, to indicate that the variable can store String values:

```
var welcomeMessage: String
```

The colon in the declaration means "...of type...," so the code above can be read as:

"Declare a variable called welcomeMessage that is of type String."

The phrase "of type String" means "can store any String value." Think of it as meaning "the type of thing" (or "the kind of thing") that can be stored.

The welcomeMessage variable can now be set to any string value without error:

```
welcomeMessage = "Hello"
```

You can define multiple related variables of the same type on a single line, separated by commas, with a single type annotation after the final variable name:

```
var red, green, blue: Double
```

NOTE

It's rare that you need to write type annotations in practice. If you provide an initial value for a constant or variable at the point that it's defined, Swift can almost always infer the type to be used for that constant or variable, as described in Type Safety and Type Inference. In the welcomeMessage example above, no initial value is provided, and so the type of the welcomeMessage variable is

Naming Constants and Variables

Constant and variable names can contain almost any character, including Unicode characters:

```
1 let π = 3.14159
2 let = ""
3 let 𝔞 = "dogcow"
```

Constant and variable names can't contain whitespace characters, mathematical symbols, arrows, private-use Unicode scalar values, or line- and box-drawing characters. Nor can they begin with a number, although numbers may be included elsewhere within the name.

Once you've declared a constant or variable of a certain type, you can't declare it again with the same name, or change it to store values of a different type. Nor can you change a constant into a variable or a variable into a constant.

```
NOTE
```

If you need to give a constant or variable the same name as a reserved Swift keyword, surround the keyword with backticks (`) when using it as a name. However, avoid using keywords as names unless you have absolutely no choice.

You can change the value of an existing variable to another value of a compatible type. In this example, the value of friendlyWelcome is changed from "Hello!" to "Bonjour!":

```
var friendlyWelcome = "Hello!"

friendlyWelcome = "Bonjour!"

// friendlyWelcome is now "Bonjour!"
```

Unlike a variable, the value of a constant can't be changed after it's set.

Attempting to do so is reported as an error when your code is compiled:

```
1 let languageName = "Swift"
2 languageName = "Swift++"
3 // This is a compile-time error: languageName cannot be changed.
```

Printing Constants and Variables

You can print the current value of a constant or variable with the print(_:separator:terminator:) function:

```
print(friendlyWelcome)
// Prints "Bonjour!"
```

The print(_:separator:terminator:) function is a global function that prints one or more values to an appropriate output. In Xcode, for example, the print(_:separator:terminator:) function prints its output in Xcode's "console" pane. The separator and terminator parameter have default values, so you can omit them when you call this function. By default, the function terminates the line it prints by adding a line break. To print a value without a line break after it, pass an empty string as the terminator—for example, print(someValue, terminator: ""). For information about parameters with default values, see Default Parameter Values.

Swift uses *string interpolation* to include the name of a constant or variable as a placeholder in a longer string, and to prompt Swift to replace it with the current value of that constant or variable. Wrap the name in parentheses and escape it with a backslash before the opening parenthesis:

All options you can use with string interpolation are described in <u>String Interpolation</u>.

Comments

Use comments to include nonexecutable text in your code, as a note or reminder to yourself. Comments are ignored by the Swift compiler when your code is compiled.

Comments in Swift are very similar to comments in C. Single-line comments begin with two forward-slashes (//):

```
// This is a comment.
```

Multiline comments start with a forward-slash followed by an asterisk (/*) and end with an asterisk followed by a forward-slash (*/):

```
/* This is also a comment
but is written over multiple lines. */
```

Unlike multiline comments in C, multiline comments in Swift can be nested inside other multiline comments. You write nested comments by starting a multiline comment block and then starting a second multiline comment within the first block. The second block is then closed, followed by the first block:

```
/* This is the start of the first multiline comment.
/* This is the second, nested multiline comment. */
This is the end of the first multiline comment. */
```

Nested multiline comments enable you to comment out large blocks of code quickly and easily, even if the code already contains multiline comments.

Semicolons

Unlike many other languages, Swift doesn't require you to write a semicolon (;) after each statement in your code, although you can do so if you wish. However, semicolons *are* required if you want to write multiple separate statements on a single line:

```
let cat = "♥"; print(cat)
// Prints "♥"
```

Integers

Integers are whole numbers with no fractional component, such as 42 and -23. Integers are either *signed* (positive, zero, or negative) or *unsigned* (positive or zero).

Swift provides signed and unsigned integers in 8, 16, 32, and 64 bit forms. These integers follow a naming convention similar to C, in that an 8-bit unsigned integer is of type UInt8, and a 32-bit signed integer is of type Int32. Like all types in Swift, these integer types have capitalized names.

Integer Bounds

You can access the minimum and maximum values of each integer type with its min and max properties:

```
let minValue = UInt8.min // minValue is equal to 0, and is of
          type UInt8
let maxValue = UInt8.max // maxValue is equal to 255, and is
          of type UInt8
```

The values of these properties are of the appropriate-sized number type (such as UInt8 in the example above) and can therefore be used in expressions alongside

other values of the same type.

Int

In most cases, you don't need to pick a specific size of integer to use in your code. Swift provides an additional integer type, Int, which has the same size as the current platform's native word size:

- On a 32-bit platform, Int is the same size as Int32.
- On a 64-bit platform, Int is the same size as Int64.

Unless you need to work with a specific size of integer, always use Int for integer values in your code. This aids code consistency and interoperability. Even on 32-bit platforms, Int can store any value between -2,147,483,648 and 2,147,483,647, and is large enough for many integer ranges.

UInt

Swift also provides an unsigned integer type, UInt, which has the same size as the current platform's native word size:

- On a 32-bit platform, UInt is the same size as UInt32.
- On a 64-bit platform, UInt is the same size as UInt64.

NOTE

Use UInt only when you specifically need an unsigned integer type with the same size as the platform's native word size. If this isn't the case, Int is preferred, even when the values to be stored are known to be nonnegative. A consistent use of Int for integer values aids code interoperability, avoids the need to convert between different number types, and matches integer type inference, as described in Type Safety and Type Inference.

Floating-Point Numbers

Floating-point numbers are numbers with a fractional component, such as 3.14159, 0.1, and -273.15.

Floating-point types can represent a much wider range of values than integer types, and can store numbers that are much larger or smaller than can be stored in an Int. Swift provides two signed floating-point number types:

- Double represents a 64-bit floating-point number.
- Float represents a 32-bit floating-point number.

NOTE

Double has a precision of at least 15 decimal digits, whereas the precision of Float can be as little as 6 decimal digits. The appropriate floating-point type to use depends on the nature and range of values you need to work with in your code. In situations where either type would be appropriate, Double is preferred.

Type Safety and Type Inference

Swift is a *type-safe* language. A type safe language encourages you to be clear about the types of values your code can work with. If part of your code requires a String, you can't pass it an Int by mistake.

Because Swift is type safe, it performs *type checks* when compiling your code and flags any mismatched types as errors. This enables you to catch and fix errors as early as possible in the development process.

Type-checking helps you avoid errors when you're working with different types of values. However, this doesn't mean that you have to specify the type of every constant and variable that you declare. If you don't specify the type of value you need, Swift uses *type inference* to work out the appropriate type. Type inference enables a compiler to deduce the type of a particular expression automatically when it compiles your code, simply by examining the values you provide.

Because of type inference, Swift requires far fewer type declarations than languages such as C or Objective-C. Constants and variables are still explicitly

typed, but much of the work of specifying their type is done for you.

Type inference is particularly useful when you declare a constant or variable with an initial value. This is often done by assigning a *literal value* (or *literal*) to the constant or variable at the point that you declare it. (A literal value is a value that appears directly in your source code, such as 42 and 3.14159 in the examples below.)

For example, if you assign a literal value of 42 to a new constant without saying what type it is, Swift infers that you want the constant to be an Int, because you have initialized it with a number that looks like an integer:

```
1 let meaningOfLife = 42
2 // meaningOfLife is inferred to be of type Int
```

Likewise, if you don't specify a type for a floating-point literal, Swift infers that you want to create a Double:

```
1 let pi = 3.14159
2 // pi is inferred to be of type Double
```

Swift always chooses Double (rather than Float) when inferring the type of floating-point numbers.

If you combine integer and floating-point literals in an expression, a type of pouble will be inferred from the context:

```
1 let anotherPi = 3 + 0.14159
2 // anotherPi is also inferred to be of type Double
```

The literal value of 3 has no explicit type in and of itself, and so an appropriate output type of Double is inferred from the presence of a floating-point literal as part of the addition.

Numeric Literals

Integer literals can be written as:

- A *decimal* number, with no prefix
- A *binary* number, with a 0b prefix
- An *octal* number, with a 00 prefix
- A *hexadecimal* number, with a 0x prefix

All of these integer literals have a decimal value of 17:

Floating-point literals can be decimal (with no prefix), or hexadecimal (with a 0× prefix). They must always have a number (or hexadecimal number) on both sides of the decimal point. Decimal floats can also have an optional *exponent*, indicated by an uppercase or lowercase e; hexadecimal floats must have an exponent, indicated by an uppercase or lowercase p.

For decimal numbers with an exponent of exp, the base number is multiplied by 10^{exp} :

- 1.25e2 means 1.25×10^2 , or 125.0.
- 1.25e-2 means 1.25 x 10⁻², or 0.0125.

For hexadecimal numbers with an exponent of exp, the base number is multiplied by 2^{exp} :

• $0 \times Fp2$ means 15×2^2 , or 60.0.

• $0 \times Fp-2$ means 15×2^{-2} , or 3.75.

All of these floating-point literals have a decimal value of 12.1875:

```
1 let decimalDouble = 12.1875
2 let exponentDouble = 1.21875e1
3 let hexadecimalDouble = 0xC.3p0
```

Numeric literals can contain extra formatting to make them easier to read. Both integers and floats can be padded with extra zeros and can contain underscores to help with readability. Neither type of formatting affects the underlying value of the literal:

```
1 let paddedDouble = 000123.456
2 let oneMillion = 1_000_000
3 let justOverOneMillion = 1_000_000.000_000_1
```

Numeric Type Conversion

Use the Int type for all general-purpose integer constants and variables in your code, even if they're known to be nonnegative. Using the default integer type in everyday situations means that integer constants and variables are immediately interoperable in your code and will match the inferred type for integer literal values.

Use other integer types only when they're specifically needed for the task at hand, because of explicitly sized data from an external source, or for performance, memory usage, or other necessary optimization. Using explicitly sized types in these situations helps to catch any accidental value overflows and implicitly documents the nature of the data being used.

Integer Conversion

The range of numbers that can be stored in an integer constant or variable is different for each numeric type. An Int8 constant or variable can store numbers between -128 and 127, whereas a UInt8 constant or variable can store numbers between 0 and 255. A number that won't fit into a constant or variable of a sized integer type is reported as an error when your code is compiled:

```
1 let cannotBeNegative: UInt8 = -1
2 // UInt8 cannot store negative numbers, and so this will report
          an error
3 let tooBig: Int8 = Int8.max + 1
4 // Int8 cannot store a number larger than its maximum value,
5 // and so this will also report an error
```

Because each numeric type can store a different range of values, you must opt in to numeric type conversion on a case-by-case basis. This opt-in approach prevents hidden conversion errors and helps make type conversion intentions explicit in your code.

To convert one specific number type to another, you initialize a new number of the desired type with the existing value. In the example below, the constant twoThousand is of type UInt16, whereas the constant one is of type UInt8. They can't be added together directly, because they're not of the same type. Instead, this example calls UInt16(one) to create a new UInt16 initialized with the value of one, and uses this value in place of the original:

```
1 let twoThousand: UInt16 = 2_000
2 let one: UInt8 = 1
3 let twoThousandAndOne = twoThousand + UInt16(one)
```

Because both sides of the addition are now of type UInt16, the addition is allowed. The output constant (twoThousandAndOne) is inferred to be of type UInt16, because it's the sum of two UInt16 values.

SomeType(ofInitialValue) is the default way to call the initializer of a Swift type and pass in an initial value. Behind the scenes, UInt16 has an initializer that

accepts a UInt8 value, and so this initializer is used to make a new UInt16 from an existing UInt8. You can't pass in *any* type here, however—it has to be a type for which UInt16 provides an initializer. Extending existing types to provide initializers that accept new types (including your own type definitions) is covered in Extensions.

Integer and Floating-Point Conversion

Conversions between integer and floating-point numeric types must be made explicit:

```
1 let three = 3
2 let pointOneFourOneFiveNine = 0.14159
3 let pi = Double(three) + pointOneFourOneFiveNine
4 // pi equals 3.14159, and is inferred to be of type Double
```

Here, the value of the constant three is used to create a new value of type <code>Double</code>, so that both sides of the addition are of the same type. Without this conversion in place, the addition would not be allowed.

Floating-point to integer conversion must also be made explicit. An integer type can be initialized with a Double or Float value:

```
1 let integerPi = Int(pi)
2 // integerPi equals 3, and is inferred to be of type Int
```

Floating-point values are always truncated when used to initialize a new integer value in this way. This means that 4.75 becomes 4, and -3.9 becomes -3.

NOTE

The rules for combining numeric constants and variables are different from the rules for numeric literals. The literal value 3 can be added directly to the literal value 0.14159, because number literals don't have an explicit type in and of themselves. Their type is inferred only at the point that they're evaluated by the compiler.

Type Aliases

Type aliases define an alternative name for an existing type. You define type aliases with the typealias keyword.

Type aliases are useful when you want to refer to an existing type by a name that is contextually more appropriate, such as when working with data of a specific size from an external source:

```
typealias AudioSample = UInt16
```

Once you define a type alias, you can use the alias anywhere you might use the original name:

```
var maxAmplitudeFound = AudioSample.min
// maxAmplitudeFound is now 0
```

Here, AudioSample is defined as an alias for UInt16. Because it's an alias, the call to AudioSample.min actually calls UInt16.min, which provides an initial value of 0 for the maxAmplitudeFound variable.

Booleans

Swift has a basic *Boolean* type, called Bool. Boolean values are referred to as *logical*, because they can only ever be true or false. Swift provides two Boolean constant values, true and false:

```
1 let orangesAreOrange = true
2 let turnipsAreDelicious = false
```

The types of orangesAreOrange and turnipsAreDelicious have been inferred as Bool from the fact that they were initialized with Boolean literal values. As with Int and Double above, you don't need to declare constants or variables as Bool if you set them to true or false as soon as you create them. Type inference helps

make Swift code more concise and readable when it initializes constants or variables with other values whose type is already known.

Boolean values are particularly useful when you work with conditional statements such as the if statement:

```
if turnipsAreDelicious {
   print("Mmm, tasty turnips!")
} else {
   print("Eww, turnips are horrible.")
}
// Prints "Eww, turnips are horrible."
```

Conditional statements such as the if statement are covered in more detail in Control Flow.

Swift's type safety prevents non-Boolean values from being substituted for Bool. The following example reports a compile-time error:

```
1 let i = 1
2 if i {
3    // this example will not compile, and will report an error
4 }
```

However, the alternative example below is valid:

```
1 let i = 1
2 if i == 1 {
3    // this example will compile successfully
4 }
```

The result of the i == 1 comparison is of type Bool, and so this second example passes the type-check. Comparisons like i == 1 are discussed in Basic Operators.

As with other examples of type safety in Swift, this approach avoids accidental errors and ensures that the intention of a particular section of code is always clear.

Tuples

Tuples group multiple values into a single compound value. The values within a tuple can be of any type and don't have to be of the same type as each other.

In this example, (404, "Not Found") is a tuple that describes an *HTTP status code*. An HTTP status code is a special value returned by a web server whenever you request a web page. A status code of 404 Not Found is returned if you request a webpage that doesn't exist.

```
let http404Error = (404, "Not Found")
// http404Error is of type (Int, String), and equals (404, "Not Found")
```

The (404, "Not Found") tuple groups together an Int and a String to give the HTTP status code two separate values: a number and a human-readable description. It can be described as "a tuple of type (Int, String)".

You can create tuples from any permutation of types, and they can contain as many different types as you like. There's nothing stopping you from having a tuple of type (Int, Int, Int), or (String, Bool), or indeed any other permutation you require.

You can *decompose* a tuple's contents into separate constants or variables, which you then access as usual:

```
1 let (statusCode, statusMessage) = http404Error
2 print("The status code is \((statusCode)"))
3 // Prints "The status code is 404"
4 print("The status message is \((statusMessage)"))
```

```
5 // Prints "The status message is Not Found"
```

If you only need some of the tuple's values, ignore parts of the tuple with an underscore (_) when you decompose the tuple:

```
1 let (justTheStatusCode, _) = http404Error
2 print("The status code is \( (justTheStatusCode)")
3 // Prints "The status code is 404"
```

Alternatively, access the individual element values in a tuple using index numbers starting at zero:

```
print("The status code is \((http404Error.0)")

// Prints "The status code is 404"

print("The status message is \((http404Error.1)"))

// Prints "The status message is Not Found"
```

You can name the individual elements in a tuple when the tuple is defined:

```
let http200Status = (statusCode: 200, description: "OK")
```

If you name the elements in a tuple, you can use the element names to access the values of those elements:

```
print("The status code is \((http200Status.statusCode)")

// Prints "The status code is 200"

print("The status message is \((http200Status.description)"))

// Prints "The status message is OK"
```

Tuples are particularly useful as the return values of functions. A function that tries to retrieve a web page might return the (Int, String) tuple type to describe the success or failure of the page retrieval. By returning a tuple with two distinct values, each of a different type, the function provides more useful information about its outcome than if it could only return a single value of a single type. For

more information, see <u>Functions with Multiple Return Values</u>.

NOTE

Tuples are useful for simple groups of related values. They're not suited to the creation of complex data structures. If your data structure is likely to be more complex, model it as a class or structure, rather than as a tuple. For more information, see Structures and Classes.

Optionals

You use *optionals* in situations where a value may be absent. An optional represents two possibilities: Either there *is* a value, and you can unwrap the optional to access that value, or there *isn't* a value at all.

NOTE

The concept of optionals doesn't exist in C or Objective-C. The nearest thing in Objective-C is the ability to return nil from a method that would otherwise return an object, with nil meaning "the absence of a valid object." However, this only works for objects—it doesn't work for structures, basic C types, or enumeration values. For these types, Objective-C methods typically return a special value (such as NSNotFound) to indicate the absence of a value. This approach assumes that the method's caller knows there's a special value to test against and remembers to check for it. Swift's optionals let you indicate the absence of a value for *any type at all*, without the need for special constants.

Here's an example of how optionals can be used to cope with the absence of a value. Swift's Int type has an initializer which tries to convert a String value into an Int value. However, not every string can be converted into an integer. The string "123" can be converted into the numeric value 123, but the string "hello, world" doesn't have an obvious numeric value to convert to.

The example below uses the initializer to try to convert a String into an Int:

```
1 let possibleNumber = "123"
2 let convertedNumber = Int(possibleNumber)
3 // convertedNumber is inferred to be of type "Int?", or "optional Int"
```

Because the initializer might fail, it returns an *optional* Int, rather than an Int. An optional Int is written as Int?, not Int. The question mark indicates that the value it contains is optional, meaning that it might contain *some* Int value, or it might contain *no value at all*. (It can't contain anything else, such as a Bool value or a String value. It's either an Int, or it's nothing at all.)

nil

You set an optional variable to a valueless state by assigning it the special value nil:

```
var serverResponseCode: Int? = 404
// serverResponseCode contains an actual Int value of 404
serverResponseCode = nil
// serverResponseCode now contains no value
```

NOTE

You can't use nil with non-optional constants and variables. If a constant or variable in your code needs to work with the absence of a value under certain conditions, always declare it as an optional value of the appropriate type.

If you define an optional variable without providing a default value, the variable is automatically set to nil for you:

```
var surveyAnswer: String?
// surveyAnswer is automatically set to nil
```

NOTE

Swift's nil isn't the same as nil in Objective-C. In Objective-C, nil is a pointer to a nonexistent object. In Swift, nil isn't a pointer—it's the absence of a value of a certain type. Optionals of *any* type can be set to nil, not just object types.

If Statements and Forced Unwrapping

You can use an if statement to find out whether an optional contains a value by comparing the optional against nil. You perform this comparison with the "equal to" operator (==) or the "not equal to" operator (!=).

If an optional has a value, it's considered to be "not equal to" nil:

```
if convertedNumber != nil {
    print("convertedNumber contains some integer value.")
}

// Prints "convertedNumber contains some integer value."
```

Once you're sure that the optional *does* contain a value, you can access its underlying value by adding an exclamation mark (!) to the end of the optional's name. The exclamation mark effectively says, "I know that this optional definitely has a value; please use it." This is known as *forced unwrapping* of the optional's value:

For more about the if statement, see **Control Flow**.

```
NOTE
```

Trying to use! to access a nonexistent optional value triggers a runtime error. Always make sure that an optional contains a non-nil value before using! to force-unwrap its value.

Optional Binding

You use optional binding to find out whether an optional contains a value, and if

so, to make that value available as a temporary constant or variable. Optional binding can be used with if and while statements to check for a value inside an optional, and to extract that value into a constant or variable, as part of a single action. if and while statements are described in more detail in Control Flow.

Write an optional binding for an if statement as follows:

```
if let constantName = someOptional {
    statements
}
```

You can rewrite the possibleNumber example from the Optionals section to use optional binding rather than forced unwrapping:

```
if let actualNumber = Int(possibleNumber) {
   print("The string \"\(possibleNumber)\" has an integer
       value of \(actualNumber)")
} else {
   print("The string \"\(possibleNumber)\" could not be
       converted to an integer")
}
// Prints "The string "123" has an integer value of 123"
```

This code can be read as:

"If the optional Int returned by Int(possibleNumber) contains a value, set a new constant called actualNumber to the value contained in the optional."

If the conversion is successful, the actualNumber constant becomes available for use within the first branch of the if statement. It has already been initialized with the value contained *within* the optional, and so there's no need to use the ! suffix to access its value. In this example, actualNumber is simply used to print the result of the conversion.

You can use both constants and variables with optional binding. If you wanted to

manipulate the value of actualNumber within the first branch of the if statement, you could write if var actualNumber instead, and the value contained within the optional would be made available as a variable rather than a constant.

You can include as many optional bindings and Boolean conditions in a single if statement as you need to, separated by commas. If any of the values in the optional bindings are nil or any Boolean condition evaluates to false, the whole if statement's condition is considered to be false. The following if statements are equivalent:

```
if let firstNumber = Int("4"), let secondNumber = Int("42"),
1
           firstNumber < secondNumber && secondNumber < 100 {</pre>
         print("\(firstNumber) < \(secondNumber) < 100")</pre>
 2
 3
    }
     // Prints "4 < 42 < 100"
 4
 5
    if let firstNumber = Int("4") {
 6
         if let secondNumber = Int("42") {
 7
             if firstNumber < secondNumber && secondNumber < 100 {</pre>
 8
                  print("\(firstNumber) < \(secondNumber) < 100")</pre>
 9
              }
10
11
         }
12
     }
13
     // Prints "4 < 42 < 100"
```

NOTE

Constants and variables created with optional binding in an if statement are available only within the body of the if statement. In contrast, the constants and variables created with a guard statement are available in the lines of code that follow the guard statement, as described in Early Exit.

Implicitly Unwrapped Optionals

As described above, optionals indicate that a constant or variable is allowed to have "no value". Optionals can be checked with an if statement to see if a value exists, and can be conditionally unwrapped with optional binding to access the optional's value if it does exist.

Sometimes it's clear from a program's structure that an optional will *always* have a value, after that value is first set. In these cases, it's useful to remove the need to check and unwrap the optional's value every time it's accessed, because it can be safely assumed to have a value all of the time.

These kinds of optionals are defined as *implicitly unwrapped optionals*. You write an implicitly unwrapped optional by placing an exclamation mark (String!) rather than a question mark (String?) after the type that you want to make optional.

Implicitly unwrapped optionals are useful when an optional's value is confirmed to exist immediately after the optional is first defined and can definitely be assumed to exist at every point thereafter. The primary use of implicitly unwrapped optionals in Swift is during class initialization, as described in Unowned References and Implicitly Unwrapped Optional Properties.

An implicitly unwrapped optional is a normal optional behind the scenes, but can also be used like a non-optional value, without the need to unwrap the optional value each time it's accessed. The following example shows the difference in behavior between an optional string and an implicitly unwrapped optional string when accessing their wrapped value as an explicit String:

You can think of an implicitly unwrapped optional as giving permission for the optional to be unwrapped automatically whenever it's used. Rather than placing an exclamation mark after the optional's name each time you use it, you place an exclamation mark after the optional's type when you declare it.

NOTE

If an implicitly unwrapped optional is nil and you try to access its wrapped value, you'll trigger a runtime error. The result is exactly the same as if you place an exclamation mark after a normal optional that doesn't contain a value.

You can still treat an implicitly unwrapped optional like a normal optional, to check if it contains a value:

```
if assumedString != nil {
    print(assumedString!)
}

// Prints "An implicitly unwrapped optional string."
```

You can also use an implicitly unwrapped optional with optional binding, to check and unwrap its value in a single statement:

```
if let definiteString = assumedString {
   print(definiteString)
}

// Prints "An implicitly unwrapped optional string."
```

NOTE

Don't use an implicitly unwrapped optional when there's a possibility of a variable becoming nil at a later point. Always use a normal optional type if you need to check for a nil value during the lifetime of a variable.

Error Handling

You use *error handling* to respond to error conditions your program may encounter during execution.

In contrast to optionals, which can use the presence or absence of a value to communicate success or failure of a function, error handling allows you to determine the underlying cause of failure, and, if necessary, propagate the error to another part of your program.

When a function encounters an error condition, it *throws* an error. That function's caller can then *catch* the error and respond appropriately.

```
func canThrowAnError() throws {
    // this function may or may not throw an error
}
```

A function indicates that it can throw an error by including the throws keyword in its declaration. When you call a function that can throw an error, you prepend the try keyword to the expression.

Swift automatically propagates errors out of their current scope until they're handled by a catch clause.

```
do {
   try canThrowAnError()
   // no error was thrown
} catch {
   // an error was thrown
}
```

A do statement creates a new containing scope, which allows errors to be propagated to one or more catch clauses.

Here's an example of how error handling can be used to respond to different error conditions:

```
func makeASandwich() throws {
1
         // ...
2
3
    }
4
    do {
5
         try makeASandwich()
6
7
         eatASandwich()
    } catch SandwichError.outOfCleanDishes {
8
9
         washDishes()
     } catch SandwichError missingIngredients(let ingredients) {
10
         buyGroceries(ingredients)
11
12
     }
```

In this example, the <code>makeASandwich()</code> function will throw an error if no clean dishes are available or if any ingredients are missing. Because <code>makeASandwich()</code> can throw an error, the function call is wrapped in a try expression. By wrapping the function call in a do statement, any errors that are thrown will be propagated to the provided <code>catch</code> clauses.

If no error is thrown, the eatASandwich() function is called. If an error is thrown and it matches the SandwichError.outOfCleanDishes case, then the washDishes() function will be called. If an error is thrown and it matches the SandwichError.missingIngredients case, then the buyGroceries(_:) function is called with the associated [String] value captured by the catch pattern.

Throwing, catching, and propagating errors is covered in greater detail in <u>Error Handling</u>.

Assertions and Preconditions

Assertions and preconditions are checks that happen at runtime. You use them to make sure an essential condition is satisfied before executing any further code. If

the Boolean condition in the assertion or precondition evaluates to true, code execution continues as usual. If the condition evaluates to false, the current state of the program is invalid; code execution ends, and your app is terminated.

You use assertions and preconditions to express the assumptions you make and the expectations you have while coding, so you can include them as part of your code. Assertions help you find mistakes and incorrect assumptions during development, and preconditions help you detect issues in production.

In addition to verifying your expectations at runtime, assertions and preconditions also become a useful form of documentation within the code. Unlike the error conditions discussed in Error Handling above, assertions and preconditions aren't used for recoverable or expected errors. Because a failed assertion or precondition indicates an invalid program state, there's no way to catch a failed assertion.

Using assertions and preconditions isn't a substitute for designing your code in such a way that invalid conditions are unlikely to arise. However, using them to enforce valid data and state causes your app to terminate more predictably if an invalid state occurs, and helps make the problem easier to debug. Stopping execution as soon as an invalid state is detected also helps limit the damage caused by that invalid state.

The difference between assertions and preconditions is in when they're checked: Assertions are checked only in debug builds, but preconditions are checked in both debug and production builds. In production builds, the condition inside an assertion isn't evaluated. This means you can use as many assertions as you want during your development process, without impacting performance in production.

Debugging with Assertions

You write an assertion by calling the assert("::file:line:">assert("::file:line:) function from the Swift standard library. You pass this function an expression that evaluates to true or false and a message to display if the result of the condition is false. For example:

```
1 let age = -3
```

```
2 assert(age >= 0, "A person's age can't be less than zero.")
3 // This assertion fails because -3 is not >= 0.
```

In this example, code execution continues if age >= 0 evaluates to true, that is, if the value of age is nonnegative. If the value of age is negative, as in the code above, then age >= 0 evaluates to false, and the assertion fails, terminating the application.

You can omit the assertion message—for example, when it would just repeat the condition as prose.

```
assert(age >= 0)
```

If the code already checks the condition, you use the assertionFailure(:file:line:) function to indicate that an assertion has failed. For example:

```
if age > 10 {
    print("You can ride the roller-coaster or the ferris
        wheel.")
} else if age >= 0 {
    print("You can ride the ferris wheel.")
} else {
    assertionFailure("A person's age can't be less than zero.")
}
```

Enforcing Preconditions

Use a precondition whenever a condition has the potential to be false, but must *definitely* be true for your code to continue execution. For example, use a precondition to check that a subscript is not out of bounds, or to check that a function has been passed a valid value.

You write a precondition by calling the precondition(: :file:line:) function.

You pass this function an expression that evaluates to true or false and a message to display if the result of the condition is false. For example:

```
// In the implementation of a subscript...
precondition(index > 0, "Index must be greater than zero.")
```

You can also call the preconditionFailure(:file:line:) function to indicate
that a failure has occurred—for example, if the default case of a switch was
taken, but all valid input data should have been handled by one of the switch's
other cases.

NOTE

If you compile in unchecked mode (-Ounchecked), preconditions aren't checked. The compiler assumes that preconditions are always true, and it optimizes your code accordingly. However, the fatalError(_:file:line:) function always halts execution, regardless of optimization settings.

You can use the fatalError(_:file:line:) function during prototyping and early development to create stubs for functionality that hasn't been implemented yet, by writing fatalError("Unimplemented") as the stub implementation. Because fatal errors are never optimized out, unlike assertions or preconditions, you can be sure that execution always halts if it encounters a stub implementation.

Basic Operators

An *operator* is a special symbol or phrase that you use to check, change, or combine values. For example, the addition operator (+) adds two numbers, as in let i = 1 + 2, and the logical AND operator (&&) combines two Boolean values, as in if enteredDoorCode && passedRetinaScan.

Swift supports most standard C operators and improves several capabilities to eliminate common coding errors. The assignment operator (=) doesn't return a value, to prevent it from being mistakenly used when the equal to operator (==) is intended. Arithmetic operators (+, -, *, /, % and so forth) detect and disallow value overflow, to avoid unexpected results when working with numbers that become larger or smaller than the allowed value range of the type that stores them. You can opt in to value overflow behavior by using Swift's overflow operators, as described in Overflow Operators.

Swift also provides range operators that aren't found in C, such as a...
b and a...b, as a shortcut for expressing a range of values.

This chapter describes the common operators in Swift. <u>Advanced Operators</u> covers Swift's advanced operators, and describes how to define your own custom operators and implement the standard operators for your own custom types.

Terminology

Operators are unary, binary, or ternary:

- *Unary* operators operate on a single target (such as -a). Unary *prefix* operators appear immediately before their target (such as !b), and unary *postfix* operators appear immediately after their target (such as c!).
- *Binary* operators operate on two targets (such as 2 + 3) and are *infix* because they appear in between their two targets.

• *Ternary* operators operate on three targets. Like C, Swift has only one ternary operator, the ternary conditional operator (a ? b : c).

The values that operators affect are *operands*. In the expression 1 + 2, the + symbol is a binary operator and its two operands are the values 1 and 2.

Assignment Operator

The *assignment operator* (a = b) initializes or updates the value of a with the value of b:

```
1 let b = 10
2 var a = 5
3 a = b
4 // a is now equal to 10
```

If the right side of the assignment is a tuple with multiple values, its elements can be decomposed into multiple constants or variables at once:

```
1 let (x, y) = (1, 2)
2 // x is equal to 1, and y is equal to 2
```

Unlike the assignment operator in C and Objective-C, the assignment operator in Swift does not itself return a value. The following statement is not valid:

```
if x = y {
    // This is not valid, because x = y does not return a
    value.
}
```

This feature prevents the assignment operator (=) from being used by accident when the equal to operator (==) is actually intended. By making $if \times = y$ invalid, Swift helps you to avoid these kinds of errors in your code.

Arithmetic Operators

Swift supports the four standard *arithmetic operators* for all number types:

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)

Unlike the arithmetic operators in C and Objective-C, the Swift arithmetic operators don't allow values to overflow by default. You can opt in to value overflow behavior by using Swift's overflow operators (such as a &+ b). See Overflow Operators.

The addition operator is also supported for ${\tt String}$ concatenation:

```
"hello, " + "world" // equals "hello, world"
```

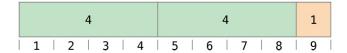
Remainder Operator

The *remainder operator* (a % b) works out how many multiples of b will fit inside a and returns the value that is left over (known as the *remainder*).

```
NOTE
```

The remainder operator (%) is also known as a *modulo operator* in other languages. However, its behavior in Swift for negative numbers means that, strictly speaking, it's a remainder rather than a modulo operation.

Here's how the remainder operator works. To calculate 9 % 4, you first work out how many 4s will fit inside 9:



You can fit two 4s inside 9, and the remainder is 1 (shown in orange).

In Swift, this would be written as:

To determine the answer for a % b, the % operator calculates the following equation and returns remainder as its output:

$$a = (b \times some \ multiplier) + remainder$$

where some multiplier is the largest number of multiples of b that will fit inside a.

Inserting 9 and 4 into this equation yields:

$$9 = (4 \times 2) + 1$$

The same method is applied when calculating the remainder for a negative value of a:

Inserting -9 and 4 into the equation yields:

$$-9 = (4 \times -2) + -1$$

giving a remainder value of -1.

The sign of ${\tt b}$ is ignored for negative values of ${\tt b}$. This means that a % ${\tt b}$ and a % - ${\tt b}$ always give the same answer.

Unary Minus Operator

The sign of a numeric value can be toggled using a prefixed -, known as the *unary minus operator*:

The unary minus operator (-) is prepended directly before the value it operates on, without any white space.

Unary Plus Operator

The *unary plus operator* (+) simply returns the value it operates on, without any change:

```
1 let minusSix = -6
2 let alsoMinusSix = +minusSix // alsoMinusSix equals -6
```

Although the unary plus operator doesn't actually do anything, you can use it to provide symmetry in your code for positive numbers when also using the unary minus operator for negative numbers.

Compound Assignment Operators

Like C, Swift provides *compound assignment operators* that combine assignment (=) with another operation. One example is the *addition assignment operator* (+=):

```
1 var a = 1
2 a += 2
```

```
3 // a is now equal to 3
```

The expression a += 2 is shorthand for a = a + 2. Effectively, the addition and the assignment are combined into one operator that performs both tasks at the same time.

```
NOTE

The compound assignment operators don't return a value. For example, you can't write let b = a += 2.
```

For information about the operators provided by the Swift standard library, see Operator Declarations.

Comparison Operators

Swift supports all standard C *comparison operators*:

- Equal to (a == b)
- Not equal to (a != b)
- Greater than (a > b)
- Less than (a < b)
- Greater than or equal to (a >= b)
- Less than or equal to (a <= b)

```
NOTE
```

Swift also provides two *identity operators* (=== and !==), which you use to test whether two object references both refer to the same object instance. For more information, see <u>Identity Operators</u>.

Each of the comparison operators returns a Bool value to indicate whether or not the statement is true:

```
1  1 == 1  // true because 1 is equal to 1
2  != 1  // true because 2 is not equal to 1
3  2 > 1   // true because 2 is greater than 1
4  1 < 2   // true because 1 is less than 2
5  1 >= 1  // true because 1 is greater than or equal to 1
6  2 <= 1  // false because 2 is not less than or equal to 1</pre>
```

Comparison operators are often used in conditional statements, such as the if statement:

```
1 let name = "world"
2 if name == "world" {
3    print("hello, world")
4 } else {
5    print("I'm sorry \(name), but I don't recognize you")
6 }
7 // Prints "hello, world", because name is indeed equal to "world".
```

For more about the if statement, see <u>Control Flow</u>.

You can compare two tuples if they have the same type and the same number of values. Tuples are compared from left to right, one value at a time, until the comparison finds two values that aren't equal. Those two values are compared, and the result of that comparison determines the overall result of the tuple comparison. If all the elements are equal, then the tuples themselves are equal. For example:

In the example above, you can see the left-to-right comparison behavior on the first line. Because 1 is less than 2, (1, "zebra") is considered less than (2, "apple"), regardless of any other values in the tuples. It doesn't matter that "zebra" isn't less than "apple", because the comparison is already determined by the tuples' first elements. However, when the tuples' first elements are the same, their second elements *are* compared—this is what happens on the second and third line.

Tuples can be compared with a given operator only if the operator can be applied to each value in the respective tuples. For example, as demonstrated in the code below, you can compare two tuples of type (String, Int) because both String and Int values can be compared using the < operator. In contrast, two tuples of type (String, Bool) can't be compared with the < operator because the < operator can't be applied to Bool values.

```
("blue", -1) < ("purple", 1)  // OK, evaluates to true
("blue", false) < ("purple", true) // Error because < can't
compare Boolean values</pre>
```

NOTE

The Swift standard library includes tuple comparison operators for tuples with fewer than seven elements. To compare tuples with seven or more elements, you must implement the comparison operators yourself.

Ternary Conditional Operator

The *ternary conditional operator* is a special operator with three parts, which takes the form question ? answer1 : answer2. It's a shortcut for evaluating one of two expressions based on whether question is true or false. If question is true, it evaluates answer1 and returns its value; otherwise, it evaluates answer2

and returns its value.

The ternary conditional operator is shorthand for the code below:

```
if question {
    answer1
} else {
    answer2
}
```

Here's an example, which calculates the height for a table row. The row height should be 50 points taller than the content height if the row has a header, and 20 points taller if the row doesn't have a header:

```
1 let contentHeight = 40
2 let hasHeader = true
3 let rowHeight = contentHeight + (hasHeader ? 50 : 20)
4 // rowHeight is equal to 90
```

The example above is shorthand for the code below:

```
1 let contentHeight = 40
2 let hasHeader = true
3 let rowHeight: Int
4 if hasHeader {
5 rowHeight = contentHeight + 50
6 } else {
7 rowHeight = contentHeight + 20
8 }
9 // rowHeight is equal to 90
```

The first example's use of the ternary conditional operator means that rowHeight can be set to the correct value on a single line of code, which is more concise

than the code used in the second example.

The ternary conditional operator provides an efficient shorthand for deciding which of two expressions to consider. Use the ternary conditional operator with care, however. Its conciseness can lead to hard-to-read code if overused. Avoid combining multiple instances of the ternary conditional operator into one compound statement.

Nil-Coalescing Operator

The *nil-coalescing operator* (a ?? b) unwraps an optional a if it contains a value, or returns a default value b if a is nil. The expression a is always of an optional type. The expression b must match the type that is stored inside a.

The nil-coalescing operator is shorthand for the code below:

```
a != nil ? a! : b
```

The code above uses the ternary conditional operator and forced unwrapping (a!) to access the value wrapped inside a when a is not nil, and to return b otherwise. The nil-coalescing operator provides a more elegant way to encapsulate this conditional checking and unwrapping in a concise and readable form.

```
NOTE

If the value of a is non-nil, the value of b is not evaluated. This is known as short-circuit evaluation.
```

The example below uses the nil-coalescing operator to choose between a default color name and an optional user-defined color name:

```
1 let defaultColorName = "red"
2 var userDefinedColorName: String? // defaults to nil
3
```

```
var colorNameToUse = userDefinedColorName ?? defaultColorName

// userDefinedColorName is nil, so colorNameToUse is set to the
    default of "red"
```

The userDefinedColorName variable is defined as an optional String, with a default value of nil. Because userDefinedColorName is of an optional type, you can use the nil-coalescing operator to consider its value. In the example above, the operator is used to determine an initial value for a String variable called colorNameToUse. Because userDefinedColorName is nil, the expression userDefinedColorName ?? defaultColorName returns the value of defaultColorName, or "red".

If you assign a non-nil value to userDefinedColorName and perform the nil-coalescing operator check again, the value wrapped inside userDefinedColorName is used instead of the default:

Range Operators

Swift includes several *range operators*, which are shortcuts for expressing a range of values.

Closed Range Operator

The *closed range operator* (a...b) defines a range that runs from a to b, and includes the values a and b. The value of a must not be greater than b.

The closed range operator is useful when iterating over a range in which you

want all of the values to be used, such as with a for-in loop:

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}

// 1 times 5 is 5

// 2 times 5 is 10

// 3 times 5 is 15

// 4 times 5 is 20

// 5 times 5 is 25
```

For more about for-in loops, see **Control Flow**.

Half-Open Range Operator

The *half-open range operator* (a..<b) defines a range that runs from a to b, but doesn't include b. It's said to be *half-open* because it contains its first value, but not its final value. As with the closed range operator, the value of a must not be greater than b. If the value of a is equal to b, then the resulting range will be empty.

Half-open ranges are particularly useful when you work with zero-based lists such as arrays, where it's useful to count up to (but not including) the length of the list:

```
1 let names = ["Anna", "Alex", "Brian", "Jack"]
2 let count = names.count
3 for i in 0..<count {
    print("Person \(i + 1) is called \(names[i])")
5 }
6 // Person 1 is called Anna
7 // Person 2 is called Alex</pre>
```

```
8 // Person 3 is called Brian9 // Person 4 is called Jack
```

Note that the array contains four items, but 0..<count only counts as far as 3 (the index of the last item in the array), because it's a half-open range. For more about arrays, see <u>Arrays</u>.

One-Sided Ranges

The closed range operator has an alternative form for ranges that continue as far as possible in one direction—for example, a range that includes all the elements of an array from index 2 to the end of the array. In these cases, you can omit the value from one side of the range operator. This kind of range is called a *one-sided range* because the operator has a value on only one side. For example:

```
for name in names[2...] {
1
 2
         print(name)
 3
    }
    // Brian
 4
    // Jack
 5
 6
7
    for name in names[...2] {
8
         print(name)
    }
10
     // Anna
     // Alex
11
12
     // Brian
```

The half-open range operator also has a one-sided form that's written with only its final value. Just like when you include a value on both sides, the final value isn't part of the range. For example:

```
for name in names[..<2] {
   print(name)
}

// Anna
// Alex</pre>
```

One-sided ranges can be used in other contexts, not just in subscripts. You can't iterate over a one-sided range that omits a first value, because it isn't clear where iteration should begin. You *can* iterate over a one-sided range that omits its final value; however, because the range continues indefinitely, make sure you add an explicit end condition for the loop. You can also check whether a one-sided range contains a particular value, as shown in the code below.

```
1 let range = ...5
2 range.contains(7) // false
3 range.contains(4) // true
4 range.contains(-1) // true
```

Logical Operators

Logical operators modify or combine the Boolean logic values true and false. Swift supports the three standard logical operators found in C-based languages:

- Logical NOT (!a)
- Logical AND (a && b)
- Logical OR (a || b)

Logical NOT Operator

The logical NOT operator (!a) inverts a Boolean value so that true becomes

false, and false becomes true.

The logical NOT operator is a prefix operator, and appears immediately before the value it operates on, without any white space. It can be read as "not a", as seen in the following example:

```
1 let allowedEntry = false
2 if !allowedEntry {
3    print("ACCESS DENIED")
4 }
5 // Prints "ACCESS DENIED"
```

The phrase if !allowedEntry can be read as "if not allowed entry." The subsequent line is only executed if "not allowed entry" is true; that is, if allowedEntry is false.

As in this example, careful choice of Boolean constant and variable names can help to keep code readable and concise, while avoiding double negatives or confusing logic statements.

Logical AND Operator

The *logical AND operator* (a && b) creates logical expressions where both values must be true for the overall expression to also be true.

If either value is false, the overall expression will also be false. In fact, if the *first* value is false, the second value won't even be evaluated, because it can't possibly make the overall expression equate to true. This is known as *short-circuit evaluation*.

This example considers two Bool values and only allows access if both values are true:

```
1 let enteredDoorCode = true
2 let passedRetinaScan = false
```

```
if enteredDoorCode && passedRetinaScan {
   print("Welcome!")
} else {
   print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

Logical OR Operator

The *logical OR operator* (a | | b) is an infix operator made from two adjacent pipe characters. You use it to create logical expressions in which only *one* of the two values has to be true for the overall expression to be true.

Like the Logical AND operator above, the Logical OR operator uses short-circuit evaluation to consider its expressions. If the left side of a Logical OR expression is true, the right side is not evaluated, because it can't change the outcome of the overall expression.

In the example below, the first Bool value (hasDoorKey) is false, but the second value (knowsOverridePassword) is true. Because one value is true, the overall expression also evaluates to true, and access is allowed:

```
1
   let hasDoorKey = false
   let knows0verridePassword = true
2
   if hasDoorKey || knowsOverridePassword {
3
        print("Welcome!")
4
    } else {
5
        print("ACCESS DENIED")
6
7
   }
    // Prints "Welcome!"
8
```

Combining Logical Operators

You can combine multiple logical operators to create longer compound expressions:

This example uses multiple && and || operators to create a longer compound expression. However, the && and || operators still operate on only two values, so this is actually three smaller expressions chained together. The example can be read as:

If we've entered the correct door code and passed the retina scan, or if we have a valid door key, or if we know the emergency override password, then allow access.

Based on the values of enteredDoorCode, passedRetinaScan, and hasDoorKey, the first two subexpressions are false. However, the emergency override password is known, so the overall compound expression still evaluates to true.

```
NOTE
```

The Swift logical operators && and | | are left-associative, meaning that compound expressions with multiple logical operators evaluate the leftmost subexpression first.

Explicit Parentheses

It's sometimes useful to include parentheses when they're not strictly needed, to make the intention of a complex expression easier to read. In the door access example above, it's useful to add parentheses around the first part of the

compound expression to make its intent explicit:

The parentheses make it clear that the first two values are considered as part of a separate possible state in the overall logic. The output of the compound expression doesn't change, but the overall intention is clearer to the reader. Readability is always preferred over brevity; use parentheses where they help to make your intentions clear.

Strings and Characters

A string is a series of characters, such as "hello, world" or "albatross". Swift strings are represented by the String type. The contents of a String can be accessed in various ways, including as a collection of Character values.

Swift's string and character types provide a fast, Unicode-compliant way to work with text in your code. The syntax for string creation and manipulation is lightweight and readable, with a string literal syntax that is similar to C. String concatenation is as simple as combining two strings with the + operator, and string mutability is managed by choosing between a constant or a variable, just like any other value in Swift. You can also use strings to insert constants, variables, literals, and expressions into longer strings, in a process known as string interpolation. This makes it easy to create custom string values for display, storage, and printing.

Despite this simplicity of syntax, Swift's string type is a fast, modern string implementation. Every string is composed of encoding-independent Unicode characters, and provides support for accessing those characters in various Unicode representations.

NOTE

Swift's String type is bridged with Foundation's NSString class. Foundation also extends String to expose methods defined by NSString. This means, if you import Foundation, you can access those NSString methods on String without casting.

For more information about using String with Foundation and Cocoa, see <u>Bridging Between</u> <u>String and NSString</u>.

String Literals

You can include predefined string values within your code as *string literals*. A string literal is a sequence of characters surrounded by double quotation marks (").

Use a string literal as an initial value for a constant or variable:

```
let someString = "Some string literal value"
```

Note that Swift infers a type of String for the someString constant because it's initialized with a string literal value.

Multiline String Literals

If you need a string that spans several lines, use a multiline string literal—a sequence of characters surrounded by three double quotation marks:

```
1 let quotation = """
2 The White Rabbit put on his spectacles. "Where shall I begin,
3 please your Majesty?" he asked.
4
5 "Begin at the beginning," the King said gravely, "and go on
6 till you come to the end; then stop."
7 """
```

A multiline string literal includes all of the lines between its opening and closing quotation marks. The string begins on the first line after the opening quotation marks (""") and ends on the line before the closing quotation marks, which means that neither of the strings below start or end with a line break:

```
1 let singleLineString = "These are the same."
2 let multilineString = """
3 These are the same.
4 """
```

When your source code includes a line break inside of a multiline string literal, that line break also appears in the string's value. If you want to use line breaks to make your source code easier to read, but you don't want the line breaks to be

part of the string's value, write a backslash (\) at the end of those lines:

To make a multiline string literal that begins or ends with a line feed, write a blank line as the first or last line. For example:

```
1 let lineBreaks = """
2
3 This string starts with a line break.
4 It also ends with a line break.
5
6 """
```

A multiline string can be indented to match the surrounding code. The whitespace before the closing quotation marks (""") tells Swift what whitespace to ignore before all of the other lines. However, if you write whitespace at the beginning of a line in addition to what's before the closing quotation marks, that whitespace *is* included.

```
Space ignored

Appears in string

Let linesWithIndentation = """

NNN This line doesn't begin with whitespace.

NNN This line begins with four spaces.

NNN This line doesn't begin with whitespace.
```

In the example above, even though the entire multiline string literal is indented,

the first and last lines in the string don't begin with any whitespace. The middle line has more indentation than the closing quotation marks, so it starts with that extra four-space indentation.

Special Characters in String Literals

String literals can include the following special characters:

- The escaped special characters \0 (null character), \\ (backslash), \t (horizontal tab), \n (line feed), \r (carriage return), \" (double quotation mark) and \' (single quotation mark)
- An arbitrary Unicode scalar value, written as $\u\{n\}$, where n is a 1–8 digit hexadecimal number (Unicode is discussed in <u>Unicode</u> below)

The code below shows four examples of these special characters. The wisewords constant contains two escaped double quotation marks. The dollarSign, blackHeart, and sparklingHeart constants demonstrate the Unicode scalar format:

Because multiline string literals use three double quotation marks instead of just one, you can include a double quotation mark (") inside of a multiline string literal without escaping it. To include the text """ in a multiline string, escape at least one of the quotation marks. For example:

```
1 let threeDoubleQuotationMarks = """
2 Escaping the first quotation mark \"""
```

```
3 Escaping all three quotation marks \"\"\"
4 """
```

Extended String Delimiters

You can place a string literal within *extended delimiters* to include special characters in a string without invoking their effect. You place your string within quotation marks (") and surround that with number signs (#). For example, printing the string literal #"Line 1\nLine 2"# prints the line feed escape sequence (\n) rather than printing the string across two lines.

If you need the special effects of a character in a string literal, match the number of number signs within the string following the escape character (\). For example, if your string is #"Line 1\nLine 2"# and you want to break the line, you can use #"Line 1\#nLine 2"# instead. Similarly, ###"Line1\###nLine2"### also breaks the line.

String literals created using extended delimiters can also be multiline string literals. You can use extended delimiters to include the text """ in a multiline string, overriding the default behavior that ends the literal. For example:

```
1 let threeMoreDoubleQuotationMarks = #"""
2 Here are three more double quotes: """
3 """#
```

Initializing an Empty String

To create an empty String value as the starting point for building a longer string, either assign an empty string literal to a variable, or initialize a new String instance with initializer syntax:

```
var emptyString = ""
// empty string literal
```

Find out whether a String value is empty by checking its Boolean isEmpty property:

```
if emptyString.isEmpty {
    print("Nothing to see here")
}

// Prints "Nothing to see here"
```

String Mutability

You indicate whether a particular String can be modified (or *mutated*) by assigning it to a variable (in which case it can be modified), or to a constant (in which case it can't be modified):

```
variableString = "Horse"
variableString += " and carriage"

// variableString is now "Horse and carriage"

let constantString = "Highlander"

constantString += " and another Highlander"

// this reports a compile-time error - a constant string cannot be modified
```

NOTE

This approach is different from string mutation in Objective-C and Cocoa, where you choose between two classes (NSString and NSMutableString) to indicate whether a string can be mutated.

Strings Are Value Types

Swift's string type is a *value type*. If you create a new string value, that string value is *copied* when it's passed to a function or method, or when it's assigned to a constant or variable. In each case, a new copy of the existing string value is created, and the new copy is passed or assigned, not the original version. Value types are described in <u>Structures and Enumerations Are Value Types</u>.

Swift's copy-by-default string behavior ensures that when a function or method passes you a string value, it's clear that you own that exact string value, regardless of where it came from. You can be confident that the string you are passed won't be modified unless you modify it yourself.

Behind the scenes, Swift's compiler optimizes string usage so that actual copying takes place only when absolutely necessary. This means you always get great performance when working with strings as value types.

Working with Characters

You can access the individual Character values for a String by iterating over the string with a for-in loop:

```
for character in "Dog!" {
    print(character)
}

// D

// 0

// g

// !

//
```

The for-in loop is described in For-In Loops.

Alternatively, you can create a stand-alone Character constant or variable from a single-character string literal by providing a Character type annotation:

```
let exclamationMark: Character = "!"
```

String values can be constructed by passing an array of Character values as an argument to its initializer:

```
1 let catCharacters: [Character] = ["C", "a", "t", "!", "☺"]
2 let catString = String(catCharacters)
3 print(catString)
4 // Prints "Cat!☺"
```

Concatenating Strings and Characters

String values can be added together (or *concatenated*) with the addition operator (+) to create a new String value:

```
1 let string1 = "hello"
2 let string2 = " there"
3 var welcome = string1 + string2
4 // welcome now equals "hello there"
```

You can also append a String value to an existing String variable with the addition assignment operator (+=):

```
var instruction = "look over"
instruction += string2
// instruction now equals "look over there"
```

You can append a Character value to a String variable with the String type's append() method:

```
1 let exclamationMark: Character = "!"
2 welcome.append(exclamationMark)
3 // welcome now equals "hello there!"
```

NOTE

You can't append a String or Character to an existing Character variable, because a Character value must contain a single character only.

If you're using multiline string literals to build up the lines of a longer string, you want every line in the string to end with a line break, including the last line. For example:

```
let badStart = """
 1
 2
     one
 3
     two
     \Pi \Pi \Pi
 4
     let end = """
 5
 6
     three
     0.00
 7
     print(badStart + end)
8
     // Prints two lines:
     // one
10
     // twothree
11
12
13
     let goodStart = """
14
     one
15
     two
16
     0.000
17
18
     print(goodStart + end)
     // Prints three lines:
19
```

```
20 // one
21 // two
22 // three
```

In the code above, concatenating badStart with end produces a two-line string, which isn't the desired result. Because the last line of badStart doesn't end with a line break, that line gets combined with the first line of end. In contrast, both lines of goodStart end with a line break, so when it's combined with end the result has three lines, as expected.

String Interpolation

String interpolation is a way to construct a new String value from a mix of constants, variables, literals, and expressions by including their values inside a string literal. You can use string interpolation in both single-line and multiline string literals. Each item that you insert into the string literal is wrapped in a pair of parentheses, prefixed by a backslash (\):

In the example above, the value of multiplier is inserted into a string literal as \ (multiplier). This placeholder is replaced with the actual value of multiplier when the string interpolation is evaluated to create an actual string.

The value of multiplier is also part of a larger expression later in the string. This expression calculates the value of Double(multiplier) * 2.5 and inserts the result (7.5) into the string. In this case, the expression is written as \ (Double(multiplier) * 2.5) when it's included inside the string literal.

You can use extended string delimiters to create strings containing characters

that would otherwise be treated as a string interpolation. For example:

To use string interpolation inside a string that uses extended delimiters, match the number of number signs before the backslash to the number of number signs at the beginning and end of the string. For example:

```
print(#"6 times 7 is \#(6 * 7)."#)
// Prints "6 times 7 is 42."
```

NOTE

The expressions you write inside parentheses within an interpolated string can't contain an unescaped backslash (\), a carriage return, or a line feed. However, they can contain other string literals.

Unicode

Unicode is an international standard for encoding, representing, and processing text in different writing systems. It enables you to represent almost any character from any language in a standardized form, and to read and write those characters to and from an external source such as a text file or web page. Swift's String and Character types are fully Unicode-compliant, as described in this section.

Unicode Scalar Values

Behind the scenes, Swift's native String type is built from *Unicode scalar values*. A Unicode scalar value is a unique 21-bit number for a character or modifier, such as U+0061 for LATIN SMALL LETTER A ("a"), or U+1F425 for FRONT-FACING BABY CHICK ("").

Note that not all 21-bit Unicode scalar values are assigned to a character—some scalars are reserved for future assignment or for use in UTF-16 encoding. Scalar values that have been assigned to a character typically also have a name, such as LATIN SMALL LETTER A and FRONT-FACING BABY CHICK in the examples above.

Extended Grapheme Clusters

Every instance of Swift's Character type represents a single *extended grapheme cluster*. An extended grapheme cluster is a sequence of one or more Unicode scalars that (when combined) produce a single human-readable character.

Here's an example. The letter é can be represented as the single Unicode scalar é (LATIN SMALL LETTER E WITH ACUTE, or U+00E9). However, the same letter can also be represented as a *pair* of scalars—a standard letter e (LATIN SMALL LETTER E, or U+0065), followed by the COMBINING ACUTE ACCENT scalar (U+0301). The COMBINING ACUTE ACCENT scalar is graphically applied to the scalar that precedes it, turning an e into an é when it's rendered by a Unicode-aware text-rendering system.

In both cases, the letter é is represented as a single Swift Character value that represents an extended grapheme cluster. In the first case, the cluster contains a single scalar; in the second case, it's a cluster of two scalars:

Extended grapheme clusters are a flexible way to represent many complex script characters as a single Character value. For example, Hangul syllables from the Korean alphabet can be represented as either a precomposed or decomposed sequence. Both of these representations qualify as a single Character value in Swift:

```
1 let precomposed: Character = "\u{D55C}" //
```

Extended grapheme clusters enable scalars for enclosing marks (such as COMBINING ENCLOSING CIRCLE, or U+20DD) to enclose other Unicode scalars as part of a single Character value:

```
1 let enclosedEAcute: Character = "\u{E9}\u{20DD}"
2 // enclosedEAcute is én
```

Unicode scalars for regional indicator symbols can be combined in pairs to make a single Character value, such as this combination of REGIONAL INDICATOR SYMBOL LETTER U (U+1F1FA) and REGIONAL INDICATOR SYMBOL LETTER S (U+1F1F8):

```
1 let regionalIndicatorForUS: Character = "\u{1F1FA}\u{1F1F8}"
2 // regionalIndicatorForUS is ==
```

Counting Characters

To retrieve a count of the Character values in a string, use the count property of the string:

Note that Swift's use of extended grapheme clusters for Character values means that string concatenation and modification may not always affect a string's character count.

For example, if you initialize a new string with the four-character word cafe, and then append a COMBINING ACUTE ACCENT (U+0301) to the end of the string, the resulting string will still have a character count of 4, with a fourth character of é, not e:

```
var word = "cafe"
print("the number of characters in \(word) is \(word.count)")

// Prints "the number of characters in cafe is 4"

word += "\u{301}" // COMBINING ACUTE ACCENT, U+0301

print("the number of characters in \(word) is \(word.count)")

// Prints "the number of characters in café is 4"
```

NOTE

Extended grapheme clusters can be composed of multiple Unicode scalars. This means that different characters—and different representations of the same character—can require different amounts of memory to store. Because of this, characters in Swift don't each take up the same amount of memory within a string's representation. As a result, the number of characters in a string can't be calculated without iterating through the string to determine its extended grapheme cluster boundaries. If you are working with particularly long string values, be aware that the count property must iterate over the Unicode scalars in the entire string in order to determine the characters for that string.

The count of the characters returned by the count property isn't always the same as the length property of an NSString that contains the same characters. The length of an NSString is based on the number of 16-bit code units within the string's UTF-16 representation and not the number of Unicode extended grapheme clusters within the string.

Accessing and Modifying a String

You access and modify a string through its methods and properties, or by using subscript syntax.

String Indices

Each String value has an associated *index type*, String.Index, which corresponds to the position of each Character in the string.

As mentioned above, different characters can require different amounts of memory to store, so in order to determine which Character is at a particular position, you must iterate over each Unicode scalar from the start or end of that string. For this reason, Swift strings can't be indexed by integer values.

Use the startIndex property to access the position of the first Character of a String. The endIndex property is the position after the last character in a String. As a result, the endIndex property isn't a valid argument to a string's subscript. If a String is empty, startIndex and endIndex are equal.

You access the indices before and after a given index using the index(before:) and index(after:) methods of String. To access an index farther away from the given index, you can use the index(_:offsetBy:) method instead of calling one of these methods multiple times.

You can use subscript syntax to access the Character at a particular String index.

```
let greeting = "Guten Tag!"
1
2
    greeting[greeting.startIndex]
3
    // G
    greeting[greeting.index(before: greeting.endIndex)]
    //!
5
    greeting[greeting.index(after: greeting.startIndex)]
6
7
    // u
    let index = greeting.index(greeting.startIndex, offsetBy: 7)
8
    greeting[index]
10
    // a
```

Attempting to access an index outside of a string's range or a Character at an index outside of a string's range will trigger a runtime error.

```
greeting[greeting.endIndex] // Error
greeting.index(after: greeting.endIndex) // Error
```

Use the indices property to access all of the indices of individual characters in a string.

```
for index in greeting.indices {
    print("\(greeting[index]) ", terminator: "")
}
// Prints "G u t e n T a g ! "
```

NOTE

You can use the startIndex and endIndex properties and the index(before:), index(after:), and index(_:offsetBy:) methods on any type that conforms to the Collection protocol. This includes String, as shown here, as well as collection types such as Array, Dictionary, and Set.

Inserting and Removing

To insert a single character into a string at a specified index, use the <code>insert(_:at:)</code> method, and to insert the contents of another string at a specified index, use the <code>insert(contentsOf:at:)</code> method.

```
var welcome = "hello"
welcome.insert("!", at: welcome.endIndex)
// welcome now equals "hello!"

welcome.insert(contentsOf: " there", at: welcome.index(before: welcome.endIndex))
// welcome now equals "hello there!"
```

To remove a single character from a string at a specified index, use the remove(at:) method, and to remove a substring at a specified range, use the

removeSubrange(_:) method:

```
NOTE
```

You can use the insert(_:at:), insert(contentsOf:at:), remove(at:), and removeSubrange(_:) methods on any type that conforms to the RangeReplaceableCollection protocol. This includes String, as shown here, as well as collection types such as Array, Dictionary, and Set.

Substrings

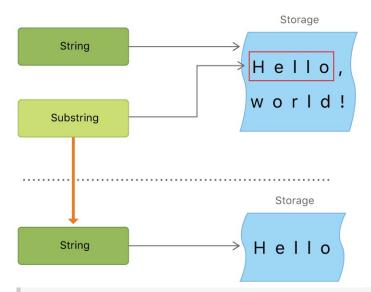
When you get a substring from a string—for example, using a subscript or a method like prefix(_:)—the result is an instance of <u>Substring</u>, not another string. Substrings in Swift have most of the same methods as strings, which means you can work with substrings the same way you work with strings. However, unlike strings, you use substrings for only a short amount of time while performing actions on a string. When you're ready to store the result for a longer time, you convert the substring to an instance of <u>String</u>. For example:

```
1 let greeting = "Hello, world!"
2 let index = greeting.firstIndex(of: ",") ?? greeting.endIndex
3 let beginning = greeting[..<index]
4 // beginning is "Hello"
5</pre>
```

6 // Convert the result to a String for long-term storage.
7 let newString = String(beginning)

Like strings, each substring has a region of memory where the characters that make up the substring are stored. The difference between strings and substrings is that, as a performance optimization, a substring can reuse part of the memory that's used to store the original string, or part of the memory that's used to store another substring. (Strings have a similar optimization, but if two strings share memory, they are equal.) This performance optimization means you don't have to pay the performance cost of copying memory until you modify either the string or substring. As mentioned above, substrings aren't suitable for long-term storage—because they reuse the storage of the original string, the entire original string must be kept in memory as long as any of its substrings are being used.

In the example above, greeting is a string, which means it has a region of memory where the characters that make up the string are stored. Because beginning is a substring of greeting, it reuses the memory that greeting uses. In contrast, newString is a string—when it's created from the substring, it has its own storage. The figure below shows these relationships:



NOTE

Both String and Substring conform to the <u>StringProtocol</u> protocol, which means it's often convenient for string-manipulation functions to accept a <u>StringProtocol</u> value. You can call such functions with either a <u>String</u> or <u>Substring</u> value.

Comparing Strings

Swift provides three ways to compare textual values: string and character equality, prefix equality, and suffix equality.

String and Character Equality

String and character equality is checked with the "equal to" operator (==) and the "not equal to" operator (!=), as described in Comparison Operators:

```
1 let quotation = "We're a lot alike, you and I."
2 let sameQuotation = "We're a lot alike, you and I."
3 if quotation == sameQuotation {
4    print("These two strings are considered equal")
5 }
6 // Prints "These two strings are considered equal"
```

Two String values (or two Character values) are considered equal if their extended grapheme clusters are *canonically equivalent*. Extended grapheme clusters are canonically equivalent if they have the same linguistic meaning and appearance, even if they're composed from different Unicode scalars behind the scenes.

For example, LATIN SMALL LETTER E WITH ACUTE (U+00E9) is canonically equivalent to LATIN SMALL LETTER E (U+0065) followed by COMBINING ACUTE ACCENT (U+0301). Both of these extended grapheme clusters are valid ways to represent the character é, and so they're considered to be canonically equivalent:

```
// "Voulez-vous un café?" using LATIN SMALL LETTER E WITH ACUTE
let eAcuteQuestion = "Voulez-vous un caf\u{E9}?"

// "Voulez-vous un café?" using LATIN SMALL LETTER E and
COMBINING ACUTE ACCENT
```

Conversely, LATIN CAPITAL LETTER A (U+0041, or "A"), as used in English, is *not* equivalent to CYRILLIC CAPITAL LETTER A (U+0410, or "A"), as used in Russian. The characters are visually similar, but don't have the same linguistic meaning:

```
1 let latinCapitalLetterA: Character = "\u{41}"
2 
3 let cyrillicCapitalLetterA: Character = "\u{0410}"
4 
5 if latinCapitalLetterA != cyrillicCapitalLetterA {
6 print("These two characters are not equivalent.")
7 }
8 // Prints "These two characters are not equivalent."
```

NOTE

String and character comparisons in Swift are not locale-sensitive.

Prefix and Suffix Equality

To check whether a string has a particular string prefix or suffix, call the string's hasPrefix(_:) and hasSuffix(_:) methods, both of which take a single argument of type String and return a Boolean value.

The examples below consider an array of strings representing the scene locations from the first two acts of Shakespeare's *Romeo and Juliet*:

```
1
    let romeoAndJuliet = [
2
        "Act 1 Scene 1: Verona, A public place",
        "Act 1 Scene 2: Capulet's mansion",
3
        "Act 1 Scene 3: A room in Capulet's mansion",
         "Act 1 Scene 4: A street outside Capulet's mansion",
5
         "Act 1 Scene 5: The Great Hall in Capulet's mansion",
6
        "Act 2 Scene 1: Outside Capulet's mansion",
7
        "Act 2 Scene 2: Capulet's orchard",
8
        "Act 2 Scene 3: Outside Friar Lawrence's cell",
9
         "Act 2 Scene 4: A street in Verona",
10
         "Act 2 Scene 5: Capulet's mansion",
11
         "Act 2 Scene 6: Friar Lawrence's cell"
12
13
    ]
```

You can use the hasPrefix(_:) method with the romeoAndJuliet array to count the number of scenes in Act 1 of the play:

```
var act1SceneCount = 0

for scene in romeoAndJuliet {
   if scene.hasPrefix("Act 1 ") {
      act1SceneCount += 1
   }

print("There are \((act1SceneCount) ) scenes in Act 1")

// Prints "There are 5 scenes in Act 1"
```

Similarly, use the hasSuffix(_:) method to count the number of scenes that take place in or around Capulet's mansion and Friar Lawrence's cell:

```
var mansionCount = 0
var cellCount = 0
```

```
3
    for scene in romeoAndJuliet {
        if scene.hasSuffix("Capulet's mansion") {
4
5
             mansionCount += 1
        } else if scene.hasSuffix("Friar Lawrence's cell") {
6
7
             cellCount += 1
        }
8
9
    }
    print("\(mansionCount) mansion scenes; \(cellCount) cell
10
          scenes")
    // Prints "6 mansion scenes; 2 cell scenes"
11
```

NOTE

The hasPrefix(_:) and hasSuffix(_:) methods perform a character-by-character canonical equivalence comparison between the extended grapheme clusters in each string, as described in String and Character Equality.

Unicode Representations of Strings

When a Unicode string is written to a text file or some other storage, the Unicode scalars in that string are encoded in one of several Unicode-defined *encoding forms*. Each form encodes the string in small chunks known as *code units*. These include the UTF-8 encoding form (which encodes a string as 8-bit code units), the UTF-16 encoding form (which encodes a string as 16-bit code units), and the UTF-32 encoding form (which encodes a string as 32-bit code units).

Swift provides several different ways to access Unicode representations of strings. You can iterate over the string with a for-in statement, to access its individual Character values as Unicode extended grapheme clusters. This process is described in Working with Characters.

Alternatively, access a String value in one of three other Unicode-compliant

representations:

- A collection of UTF-8 code units (accessed with the string's utf8 property)
- A collection of UTF-16 code units (accessed with the string's utf16 property)
- A collection of 21-bit Unicode scalar values, equivalent to the string's UTF-32 encoding form (accessed with the string's unicodeScalars property)

Each example below shows a different representation of the following string, which is made up of the characters D, o, g, !! (DOUBLE EXCLAMATION MARK, or Unicode scalar U+203C), and the character (DOG FACE, or Unicode scalar U+1F436):

```
let dogString = "Dog!!"
```

UTF-8 Representation

You can access a UTF-8 representation of a String by iterating over its utf8 property. This property is of type String.UTF8View, which is a collection of unsigned 8-bit (UInt8) values, one for each byte in the string's UTF-8 representation:

Character	D U+0044	0 U+006F	g U+0067	!! U+203C			₩ U+1F436			
UTF-8 Code Unit	68	111	103	226	128	188	240	159	144	182
Position	0	1	2	3	4	5	6	7	8	9

```
for codeUnit in dogString.utf8 {
   print("\(codeUnit\)", terminator: "")
```

```
3  }
4  print("")
5  // Prints "68 111 103 226 128 188 240 159 144 182 "
```

In the example above, the first three decimal codeUnit values (68, 111, 103) represent the characters D, o, and g, whose UTF-8 representation is the same as their ASCII representation. The next three decimal codeUnit values (226, 128, 188) are a three-byte UTF-8 representation of the DOUBLE EXCLAMATION MARK character. The last four codeUnit values (240, 159, 144, 182) are a four-byte UTF-8 representation of the DOG FACE character.

UTF-16 Representation

You can access a UTF-16 representation of a String by iterating over its utf16 property. This property is of type String.UTF16View, which is a collection of unsigned 16-bit (UInt16) values, one for each 16-bit code unit in the string's UTF-16 representation:

Character	D U+0044	O g U+006F U+0067		!! U+203C	€ U+1F436		
UTF-16 Code Unit	68	111	103	8252	55357	56374	
Position	0	1	2	3	4	5	

```
for codeUnit in dogString.utf16 {
    print("\(codeUnit) ", terminator: "")
}

print("")
// Prints "68 111 103 8252 55357 56374 "
```

Again, the first three codeUnit values (68, 111, 103) represent the characters D, o, and g, whose UTF-16 code units have the same values as in the string's UTF-8 representation (because these Unicode scalars represent ASCII characters).

The fourth codeUnit value (8252) is a decimal equivalent of the hexadecimal value 203C, which represents the Unicode scalar U+203C for the DOUBLE EXCLAMATION MARK character. This character can be represented as a single code unit in UTF-16.

The fifth and sixth codeUnit values (55357 and 56374) are a UTF-16 surrogate pair representation of the DOG FACE character. These values are a high-surrogate value of U+D83D (decimal value 55357) and a low-surrogate value of U+DC36 (decimal value 56374).

Unicode Scalar Representation

You can access a Unicode scalar representation of a String value by iterating over its unicodeScalars property. This property is of type UnicodeScalarView, which is a collection of values of type UnicodeScalar.

Each UnicodeScalar has a value property that returns the scalar's 21-bit value, represented within a UInt32 value:

Character	D U+0044	0 U+006F	g U+0067	!! U+203C	€ U+1F436
Unicode Scalar Code Unit	68	111	103	8252	128054
Position	0	1	2	3	4

```
for scalar in dogString.unicodeScalars {
    print("\(scalar.value) ", terminator: "")
```

```
3  }
4  print("")
5  // Prints "68 111 103 8252 128054 "
```

The value properties for the first three UnicodeScalar values (68, 111, 103) once again represent the characters D, o, and g.

The fourth codeUnit value (8252) is again a decimal equivalent of the hexadecimal value 203C, which represents the Unicode scalar U+203C for the DOUBLE EXCLAMATION MARK character.

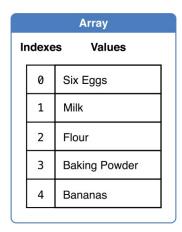
The value property of the fifth and final UnicodeScalar, 128054, is a decimal equivalent of the hexadecimal value 1F436, which represents the Unicode scalar U+1F436 for the DOG FACE character.

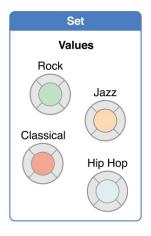
As an alternative to querying their value properties, each UnicodeScalar value can also be used to construct a new String value, such as with string interpolation:

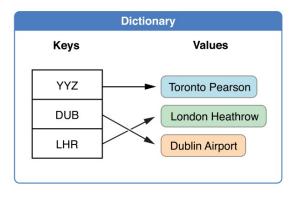
```
for scalar in dogString.unicodeScalars {
    print("\(scalar) ")
}
// D
// D
// 0
// g
// !!
// !!
```

Collection Types

Swift provides three primary *collection types*, known as arrays, sets, and dictionaries, for storing collections of values. Arrays are ordered collections of values. Sets are unordered collections of unique values. Dictionaries are unordered collections of key-value associations.







Arrays, sets, and dictionaries in Swift are always clear about the types of values and keys that they can store. This means that you cannot insert a value of the wrong type into a collection by mistake. It also means you can be confident about the type of values you will retrieve from a collection.

NOTE

Swift's array, set, and dictionary types are implemented as *generic collections*. For more about generic types and collections, see <u>Generics</u>.

Mutability of Collections

If you create an array, a set, or a dictionary, and assign it to a variable, the collection that is created will be *mutable*. This means that you can change (or *mutate*) the collection after it's created by adding, removing, or changing items in the collection. If you assign an array, a set, or a dictionary to a constant, that

collection is *immutable*, and its size and contents cannot be changed.

NOTE

It is good practice to create immutable collections in all cases where the collection does not need to change. Doing so makes it easier for you to reason about your code and enables the Swift compiler to optimize the performance of the collections you create.

Arrays

An *array* stores values of the same type in an ordered list. The same value can appear in an array multiple times at different positions.

NOTE

Swift's Array type is bridged to Foundation's NSArray class.

For more information about using Array with Foundation and Cocoa, see <u>Bridging Between Array</u> and <u>NSArray</u>.

Array Type Shorthand Syntax

The type of a Swift array is written in full as Array<Element>, where Element is the type of values the array is allowed to store. You can also write the type of an array in shorthand form as [Element]. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of an array.

Creating an Empty Array

You can create an empty array of a certain type using initializer syntax:

```
var someInts = [Int]()
print("someInts is of type [Int] with \((someInts.count)))
```

```
items.")
3 // Prints "someInts is of type [Int] with 0 items."
```

Note that the type of the someInts variable is inferred to be [Int] from the type of the initializer.

Alternatively, if the context already provides type information, such as a function argument or an already typed variable or constant, you can create an empty array with an empty array literal, which is written as [] (an empty pair of square brackets):

```
someInts.append(3)
// someInts now contains 1 value of type Int
someInts = []
// someInts is now an empty array, but is still of type [Int]
```

Creating an Array with a Default Value

Swift's Array type also provides an initializer for creating an array of a certain size with all of its values set to the same default value. You pass this initializer a default value of the appropriate type (called repeating): and the number of times that value is repeated in the new array (called count):

```
var threeDoubles = Array(repeating: 0.0, count: 3)
// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]
```

Creating an Array by Adding Two Arrays Together

You can create a new array by adding together two existing arrays with compatible types with the addition operator (+). The new array's type is inferred from the type of the two arrays you add together:

```
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
```

Creating an Array with an Array Literal

You can also initialize an array with an *array literal*, which is a shorthand way to write one or more values as an array collection. An array literal is written as a list of values, separated by commas, surrounded by a pair of square brackets:

```
[value 1, value 2, value 3]
```

The example below creates an array called shoppingList to store String values:

```
var shoppingList: [String] = ["Eggs", "Milk"]
// shoppingList has been initialized with two initial items
```

The shoppingList variable is declared as "an array of string values", written as [String]. Because this particular array has specified a value type of String, it is allowed to store String values only. Here, the shoppingList array is initialized with two String values ("Eggs" and "Milk"), written within an array literal.

NOTE

The shoppingList array is declared as a variable (with the var introducer) and not a constant (with the let introducer) because more items are added to the shopping list in the examples below.

In this case, the array literal contains two String values and nothing else. This matches the type of the shoppingList variable's declaration (an array that can only contain String values), and so the assignment of the array literal is permitted as a way to initialize shoppingList with two initial items.

Thanks to Swift's type inference, you don't have to write the type of the array if you're initializing it with an array literal containing values of the same type. The initialization of shoppingList could have been written in a shorter form instead:

```
var shoppingList = ["Eggs", "Milk"]
```

Because all values in the array literal are of the same type, Swift can infer that [String] is the correct type to use for the shoppingList variable.

Accessing and Modifying an Array

You access and modify an array through its methods and properties, or by using subscript syntax.

To find out the number of items in an array, check its read-only count property:

```
print("The shopping list contains \((shoppingList.count)\)
    items.")
// Prints "The shopping list contains 2 items."
```

Use the Boolean is Empty property as a shortcut for checking whether the count property is equal to 0:

```
if shoppingList.isEmpty {
    print("The shopping list is empty.")
} else {
    print("The shopping list is not empty.")
}
// Prints "The shopping list is not empty."
```

You can add a new item to the end of an array by calling the array's append(_:) method:

```
shoppingList.append("Flour")
```

```
2 // shoppingList now contains 3 items, and someone is making
pancakes
```

Alternatively, append an array of one or more compatible items with the addition assignment operator (+=):

```
shoppingList += ["Baking Powder"]
// shoppingList now contains 4 items
shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
// shoppingList now contains 7 items
```

Retrieve a value from the array by using *subscript syntax*, passing the index of the value you want to retrieve within square brackets immediately after the name of the array:

```
var firstItem = shoppingList[0]
// firstItem is equal to "Eggs"
```

```
NOTE
```

The first item in the array has an index of 0, not 1. Arrays in Swift are always zero-indexed.

You can use subscript syntax to change an existing value at a given index:

```
shoppingList[0] = "Six eggs"

// the first item in the list is now equal to "Six eggs" rather
than "Eggs"
```

When you use subscript syntax, the index you specify needs to be valid. For example, writing <code>shoppingList[shoppingList.count] = "Salt"</code> to try to append an item to the end of the array results in a runtime error.

You can also use subscript syntax to change a range of values at once, even if the replacement set of values has a different length than the range you are replacing. The following example replaces "Chocolate Spread", "Cheese", and "Butter"

with "Bananas" and "Apples":

```
shoppingList[4...6] = ["Bananas", "Apples"]
// shoppingList now contains 6 items
```

To insert an item into the array at a specified index, call the array's insert(_:at:) method:

```
shoppingList.insert("Maple Syrup", at: 0)
// shoppingList now contains 7 items
// "Maple Syrup" is now the first item in the list
```

This call to the insert(_:at:) method inserts a new item with a value of "Maple Syrup" at the very beginning of the shopping list, indicated by an index of 0.

Similarly, you remove an item from the array with the remove(at:) method. This method removes the item at the specified index and returns the removed item (although you can ignore the returned value if you do not need it):

```
1 let mapleSyrup = shoppingList.remove(at: 0)
2 // the item that was at index 0 has just been removed
3 // shoppingList now contains 6 items, and no Maple Syrup
4 // the mapleSyrup constant is now equal to the removed "Maple Syrup" string
```

NOTE

If you try to access or modify a value for an index that is outside of an array's existing bounds, you will trigger a runtime error. You can check that an index is valid before using it by comparing it to the array's count property. The largest valid index in an array is count - 1 because arrays are indexed from zero—however, when count is θ (meaning the array is empty), there are no valid indexes.

Any gaps in an array are closed when an item is removed, and so the value at index 0 is once again equal to "Six eggs":

```
firstItem = shoppingList[0]
// firstItem is now equal to "Six eggs"
```

If you want to remove the final item from an array, use the removeLast() method rather than the remove(at:) method to avoid the need to query the array's count property. Like the remove(at:) method, removeLast() returns the removed item:

Iterating Over an Array

You can iterate over the entire set of values in an array with the for-in loop:

```
for item in shoppingList {
   print(item)
}

// Six eggs

// Milk

// Flour

// Baking Powder

// Bananas
```

If you need the integer index of each item as well as its value, use the <code>enumerated()</code> method to iterate over the array instead. For each item in the array, the <code>enumerated()</code> method returns a tuple composed of an integer and the item. The integers start at zero and count up by one for each item; if you enumerate over a whole array, these integers match the items' indices. You can decompose the tuple into temporary constants or variables as part of the iteration:

```
for (index, value) in shoppingList.enumerated() {
1
        print("Item \(index + 1): \(value)")
2
3
   }
   // Item 1: Six eggs
4
   // Item 2: Milk
5
   // Item 3: Flour
6
7
   // Item 4: Baking Powder
   // Item 5: Bananas
8
```

For more about the for-in loop, see <u>For-In Loops</u>.

Sets

A *set* stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items is not important, or when you need to ensure that an item only appears once.

```
NOTE
```

Swift's Set type is bridged to Foundation's NSSet class.

For more information about using Set with Foundation and Cocoa, see <u>Bridging Between Set and NSSet</u>.

Hash Values for Set Types

A type must be *hashable* in order to be stored in a set—that is, the type must provide a way to compute a *hash value* for itself. A hash value is an Int value that is the same for all objects that compare equally, such that if a == b, it follows that a hashvalue == b hashvalue.

All of Swift's basic types (such as String, Int, Double, and Bool) are hashable by default, and can be used as set value types or dictionary key types. Enumeration

case values without associated values (as described in **Enumerations**) are also hashable by default.

NOTE

You can use your own custom types as set value types or dictionary key types by making them conform to the Hashable protocol from Swift's standard library. Types that conform to the Hashable protocol must provide a gettable Int property called hashValue. The value returned by a type's hashValue property is not required to be the same across different executions of the same program, or in different programs.

Because the Hashable protocol conforms to Equatable, conforming types must also provide an implementation of the equals operator (==). The Equatable protocol requires any conforming implementation of == to be an equivalence relation. That is, an implementation of == must satisfy the following three conditions, for all values a, b, and c:

```
a == a (Reflexivity)
a == b implies b == a (Symmetry)
a == b && b == c implies a == c (Transitivity)
```

For more information about conforming to protocols, see **Protocols**.

Set Type Syntax

The type of a Swift set is written as Set<Element>, where Element is the type that the set is allowed to store. Unlike arrays, sets do not have an equivalent shorthand form.

Creating and Initializing an Empty Set

You can create an empty set of a certain type using initializer syntax:

```
var letters = Set<Character>()
print("letters is of type Set<Character> with \(letters.count)
    items.")

// Prints "letters is of type Set<Character> with 0 items."
```

The type of the letters variable is inferred to be Set<Character>, from the type of the initializer.

Alternatively, if the context already provides type information, such as a function argument or an already typed variable or constant, you can create an empty set with an empty array literal:

Creating a Set with an Array Literal

You can also initialize a set with an array literal, as a shorthand way to write one or more values as a set collection.

The example below creates a set called favoriteGenres to store String values:

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip
hop"]

// favoriteGenres has been initialized with three initial items
```

The favoriteGenres variable is declared as "a set of String values", written as Set<String>. Because this particular set has specified a value type of String, it is only allowed to store String values. Here, the favoriteGenres set is initialized with three String values ("Rock", "Classical", and "Hip hop"), written within an array literal.

NOTE

The favoriteGenres set is declared as a variable (with the var introducer) and not a constant (with the let introducer) because items are added and removed in the examples below.

A set type cannot be inferred from an array literal alone, so the type Set must be explicitly declared. However, because of Swift's type inference, you don't have to write the type of the set's elements if you're initializing it with an array literal that contains values of just one type. The initialization of favoriteGenres could have been written in a shorter form instead:

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]
```

Because all values in the array literal are of the same type, Swift can infer that Set<String> is the correct type to use for the favoriteGenres variable.

Accessing and Modifying a Set

You access and modify a set through its methods and properties.

To find out the number of items in a set, check its read-only count property:

```
print("I have \(favoriteGenres.count) favorite music genres.")
// Prints "I have 3 favorite music genres."
```

Use the Boolean is Empty property as a shortcut for checking whether the count property is equal to 0:

```
if favoriteGenres.isEmpty {
   print("As far as music goes, I'm not picky.")
} else {
   print("I have particular music preferences.")
}
// Prints "I have particular music preferences."
```

You can add a new item into a set by calling the set's insert(_:) method:

```
favoriteGenres.insert("Jazz")
// favoriteGenres now contains 4 items
```

You can remove an item from a set by calling the set's remove(_:) method, which removes the item if it's a member of the set, and returns the removed value, or returns nil if the set did not contain it. Alternatively, all items in a set can be removed with its removeAll() method.

```
if let removedGenre = favoriteGenres.remove("Rock") {
   print("\(removedGenre)? I'm over it.")
} else {
   print("I never much cared for that.")
}
// Prints "Rock? I'm over it."
```

To check whether a set contains a particular item, use the contains(_:) method.

```
if favoriteGenres.contains("Funk") {
    print("I get up on the good foot.")
} else {
    print("It's too funky in here.")
}
// Prints "It's too funky in here."
```

Iterating Over a Set

You can iterate over the values in a set with a for-in loop.

```
for genre in favoriteGenres {
   print("\(genre)")
}

// Classical
// Jazz
// Hip hop
```

For more about the for-in loop, see <u>For-In Loops</u>.

Swift's Set type does not have a defined ordering. To iterate over the values of a set in a specific order, use the sorted() method, which returns the set's elements as an array sorted using the < operator.

```
for genre in favoriteGenres.sorted() {
    print("\(genre)")
}

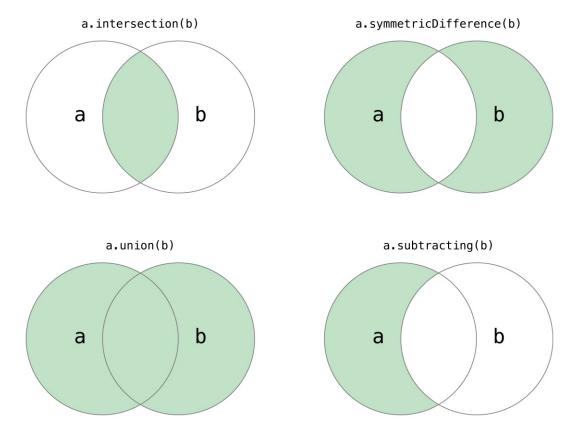
// Classical
// Hip hop
// Jazz
```

Performing Set Operations

You can efficiently perform fundamental set operations, such as combining two sets together, determining which values two sets have in common, or determining whether two sets contain all, some, or none of the same values.

Fundamental Set Operations

The illustration below depicts two sets—a and b—with the results of various set operations represented by the shaded regions.



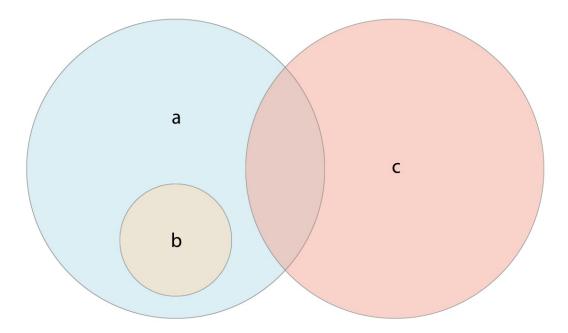
- Use the intersection(_:) method to create a new set with only the values common to both sets.
- Use the symmetricDifference(_:) method to create a new set with values in either set, but not both.
- Use the union(_:) method to create a new set with all of the values in both sets.
- Use the subtracting(_:) method to create a new set with values not in the specified set.

```
1 let oddDigits: Set = [1, 3, 5, 7, 9]
2 let evenDigits: Set = [0, 2, 4, 6, 8]
3 let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]
4
5 oddDigits.union(evenDigits).sorted()
```

```
6  // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
7  oddDigits.intersection(evenDigits).sorted()
8  // []
9  oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
10  // [1, 9]
11  oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
12  // [1, 2, 9]
```

Set Membership and Equality

The illustration below depicts three sets—a, b and c—with overlapping regions representing elements shared among sets. Set a is a *superset* of set b, because a contains all elements in b. Conversely, set b is a *subset* of set a, because all elements in b are also contained by a. Set b and set c are *disjoint* with one another, because they share no elements in common.



• Use the "is equal" operator (==) to determine whether two sets contain all of the same values.

- Use the isSubset(of:) method to determine whether all of the values of a set are contained in the specified set.
- Use the isSuperset(of:) method to determine whether a set contains all of the values in a specified set.
- Use the isStrictSubset(of:) or isStrictSuperset(of:) methods to determine whether a set is a subset or superset, but not equal to, a specified set.
- Use the isDisjoint(with:) method to determine whether two sets have no values in common.

```
1 let houseAnimals: Set = ["" , """]
2 let farmAnimals: Set = [""", "" , "" , """, """]
3 let cityAnimals: Set = ["" , ""]
4
5 houseAnimals.isSubset(of: farmAnimals)
6 // true
7 farmAnimals.isSuperset(of: houseAnimals)
8 // true
9 farmAnimals.isDisjoint(with: cityAnimals)
10 // true
```

Dictionaries

A *dictionary* stores associations between keys of the same type and values of the same type in a collection with no defined ordering. Each value is associated with a unique *key*, which acts as an identifier for that value within the dictionary. Unlike items in an array, items in a dictionary do not have a specified order. You use a dictionary when you need to look up values based on their identifier, in much the same way that a real-world dictionary is used to look up the definition for a particular word.

NOTE

Swift's Dictionary type is bridged to Foundation's NSDictionary class.

For more information about using Dictionary with Foundation and Cocoa, see <u>Bridging Between Dictionary and NSDictionary</u>.

Dictionary Type Shorthand Syntax

The type of a Swift dictionary is written in full as Dictionary<Key, Value>, where Key is the type of value that can be used as a dictionary key, and Value is the type of value that the dictionary stores for those keys.

NOTE

A dictionary Key type must conform to the Hashable protocol, like a set's value type.

You can also write the type of a dictionary in shorthand form as [Key: Value]. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of a dictionary.

Creating an Empty Dictionary

As with arrays, you can create an empty Dictionary of a certain type by using initializer syntax:

```
var namesOfIntegers = [Int: String]()
// namesOfIntegers is an empty [Int: String] dictionary
```

This example creates an empty dictionary of type [Int: String] to store human-readable names of integer values. Its keys are of type Int, and its values are of type String.

If the context already provides type information, you can create an empty dictionary with an empty dictionary literal, which is written as [:] (a colon inside a pair of square brackets):

```
namesOfIntegers[16] = "sixteen"
// namesOfIntegers now contains 1 key-value pair
namesOfIntegers = [:]
// namesOfIntegers is once again an empty dictionary of type
[Int: String]
```

Creating a Dictionary with a Dictionary Literal

You can also initialize a dictionary with a *dictionary literal*, which has a similar syntax to the array literal seen earlier. A dictionary literal is a shorthand way to write one or more key-value pairs as a <code>Dictionary</code> collection.

A *key-value pair* is a combination of a key and a value. In a dictionary literal, the key and value in each key-value pair are separated by a colon. The key-value pairs are written as a list, separated by commas, surrounded by a pair of square brackets:

```
[ key 1: value 1, key 2: value 2, key 3: value 3]
```

The example below creates a dictionary to store the names of international airports. In this dictionary, the keys are three-letter International Air Transport Association codes, and the values are airport names:

The airports dictionary is declared as having a type of [String: String], which means "a Dictionary whose keys are of type String, and whose values are also of type String".

NOTE

The airports dictionary is declared as a variable (with the var introducer), and not a constant (with the let introducer), because more airports are added to the dictionary in the examples below.

The airports dictionary is initialized with a dictionary literal containing two key-value pairs. The first pair has a key of "YYZ" and a value of "Toronto Pearson". The second pair has a key of "DUB" and a value of "Dublin".

This dictionary literal contains two String: String pairs. This key-value type matches the type of the airports variable declaration (a dictionary with only String keys, and only String values), and so the assignment of the dictionary literal is permitted as a way to initialize the airports dictionary with two initial items.

As with arrays, you don't have to write the type of the dictionary if you're initializing it with a dictionary literal whose keys and values have consistent types. The initialization of airports could have been written in a shorter form instead:

```
var airports = ["YYZ": "Toronto Pearson", "DUB": "Dublin"]
```

Because all keys in the literal are of the same type as each other, and likewise all values are of the same type as each other, Swift can infer that [String: String] is the correct type to use for the airports dictionary.

Accessing and Modifying a Dictionary

You access and modify a dictionary through its methods and properties, or by using subscript syntax.

As with an array, you find out the number of items in a Dictionary by checking its read-only count property:

```
print("The airports dictionary contains \(airports.count)
    items.")

// Prints "The airports dictionary contains 2 items."
```

Use the Boolean isEmpty property as a shortcut for checking whether the count property is equal to 0:

```
if airports.isEmpty {
    print("The airports dictionary is empty.")
} else {
    print("The airports dictionary is not empty.")
}
// Prints "The airports dictionary is not empty."
```

You can add a new item to a dictionary with subscript syntax. Use a new key of the appropriate type as the subscript index, and assign a new value of the appropriate type:

```
airports["LHR"] = "London"
// the airports dictionary now contains 3 items
```

You can also use subscript syntax to change the value associated with a particular key:

```
1 airports["LHR"] = "London Heathrow"
2 // the value for "LHR" has been changed to "London Heathrow"
```

As an alternative to subscripting, use a dictionary's updateValue(_:forKey:) method to set or update the value for a particular key. Like the subscript examples above, the updateValue(_:forKey:) method sets a value for a key if none exists, or updates the value if that key already exists. Unlike a subscript, however, the updateValue(_:forKey:) method returns the *old* value after performing an update. This enables you to check whether or not an update took place.

The updateValue(_:forKey:) method returns an optional value of the dictionary's value type. For a dictionary that stores String values, for example, the method returns a value of type String?, or "optional String". This optional value contains the old value for that key if one existed before the update, or nil if no value existed:

```
if let oldValue = airports.updateValue("Dublin Airport",
```

```
forKey: "DUB") {

print("The old value for DUB was \(oldValue).")

}

// Prints "The old value for DUB was Dublin."
```

You can also use subscript syntax to retrieve a value from the dictionary for a particular key. Because it is possible to request a key for which no value exists, a dictionary's subscript returns an optional value of the dictionary's value type. If the dictionary contains a value for the requested key, the subscript returns an optional value containing the existing value for that key. Otherwise, the subscript returns nil:

```
if let airportName = airports["DUB"] {
   print("The name of the airport is \((airportName)."))
} else {
   print("That airport is not in the airports dictionary.")}
}
// Prints "The name of the airport is Dublin Airport."
```

You can use subscript syntax to remove a key-value pair from a dictionary by assigning a value of nil for that key:

```
airports["APL"] = "Apple International"
// "Apple International" is not the real airport for APL, so
          delete it
airports["APL"] = nil
// APL has now been removed from the dictionary
```

Alternatively, remove a key-value pair from a dictionary with the removeValue(forKey:) method. This method removes the key-value pair if it exists and returns the removed value, or returns nil if no value existed:

```
1 if let removedValue = airports.removeValue(forKey: "DUB") {
```

```
print("The removed airport's name is \((removedValue).")
} else {
print("The airports dictionary does not contain a value for DUB.")
}
// Prints "The removed airport's name is Dublin Airport."
```

Iterating Over a Dictionary

You can iterate over the key-value pairs in a dictionary with a for-in loop. Each item in the dictionary is returned as a (key, value) tuple, and you can decompose the tuple's members into temporary constants or variables as part of the iteration:

```
for (airportCode, airportName) in airports {
   print("\(airportCode): \(airportName)")
}

// LHR: London Heathrow
// YYZ: Toronto Pearson
```

For more about the for-in loop, see <u>For-In Loops</u>.

You can also retrieve an iterable collection of a dictionary's keys or values by accessing its keys and values properties:

```
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}

// Airport code: LHR

// Airport code: YYZ

for airportName in airports.values {
```

```
8     print("Airport name: \(airportName)")
9  }
10    // Airport name: London Heathrow
11    // Airport name: Toronto Pearson
```

If you need to use a dictionary's keys or values with an API that takes an Array instance, initialize a new array with the keys or values property:

```
1 let airportCodes = [String](airports.keys)
2 // airportCodes is ["LHR", "YYZ"]
3
4 let airportNames = [String](airports.values)
5 // airportNames is ["London Heathrow", "Toronto Pearson"]
```

Swift's Dictionary type does not have a defined ordering. To iterate over the keys or values of a dictionary in a specific order, use the <code>sorted()</code> method on its keys or values property.

Control Flow

Swift provides a variety of control flow statements. These include while loops to perform a task multiple times; if, guard, and switch statements to execute different branches of code based on certain conditions; and statements such as break and continue to transfer the flow of execution to another point in your code.

Swift also provides a for-in loop that makes it easy to iterate over arrays, dictionaries, ranges, strings, and other sequences.

Swift's switch statement is considerably more powerful than its counterpart in many C-like languages. Cases can match many different patterns, including interval matches, tuples, and casts to a specific type. Matched values in a switch case can be bound to temporary constants or variables for use within the case's body, and complex matching conditions can be expressed with a where clause for each case.

For-In Loops

You use the for-in loop to iterate over a sequence, such as items in an array, ranges of numbers, or characters in a string.

This example uses a for-in loop to iterate over the items in an array:

```
1 let names = ["Anna", "Alex", "Brian", "Jack"]
2 for name in names {
    print("Hello, \(name)!")
4 }
5 // Hello, Anna!
6 // Hello, Brian!
```

```
8 // Hello, Jack!
```

You can also iterate over a dictionary to access its key-value pairs. Each item in the dictionary is returned as a (key, value) tuple when the dictionary is iterated, and you can decompose the (key, value) tuple's members as explicitly named constants for use within the body of the for-in loop. In the code example below, the dictionary's keys are decomposed into a constant called animalName, and the dictionary's values are decomposed into a constant called legCount.

```
1 let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
2 for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
4 }
5 // cats have 4 legs
6 // ants have 6 legs
7 // spiders have 8 legs
```

The contents of a Dictionary are inherently unordered, and iterating over them does not guarantee the order in which they will be retrieved. In particular, the order you insert items into a Dictionary doesn't define the order they are iterated. For more about arrays and dictionaries, see <u>Collection Types</u>.

You can also use for-in loops with numeric ranges. This example prints the first few entries in a five-times table:

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}

// 1 times 5 is 5

// 2 times 5 is 10

// 3 times 5 is 15

// 4 times 5 is 20

// 5 times 5 is 25
```

The sequence being iterated over is a range of numbers from 1 to 5, inclusive, as indicated by the use of the closed range operator (...). The value of index is set to the first number in the range (1), and the statements inside the loop are executed. In this case, the loop contains only one statement, which prints an entry from the five-times table for the current value of index. After the statement is executed, the value of index is updated to contain the second value in the range (2), and the print(_:separator:terminator:) function is called again. This process continues until the end of the range is reached.

In the example above, index is a constant whose value is automatically set at the start of each iteration of the loop. As such, index does not have to be declared before it is used. It is implicitly declared simply by its inclusion in the loop declaration, without the need for a let declaration keyword.

If you don't need each value from a sequence, you can ignore the values by using an underscore in place of a variable name.

```
1  let base = 3
2  let power = 10
3  var answer = 1
4  for _ in 1...power {
     answer *= base
6  }
7  print("\(base) to the power of \(power) is \(answer)")
8  // Prints "3 to the power of 10 is 59049"
```

The example above calculates the value of one number to the power of another (in this case, 3 to the power of 10). It multiplies a starting value of 1 (that is, 3 to the power of 0) by 3, ten times, using a closed range that starts with 1 and ends with 10. For this calculation, the individual counter values each time through the loop are unnecessary—the code simply executes the loop the correct number of times. The underscore character (_) used in place of a loop variable causes the individual values to be ignored and does not provide access to the current value during each iteration of the loop.

In some situations, you might not want to use closed ranges, which include both

endpoints. Consider drawing the tick marks for every minute on a watch face. You want to draw 60 tick marks, starting with the 0 minute. Use the half-open range operator (. . <) to include the lower bound but not the upper bound. For more about ranges, see Range Operators.

```
1 let minutes = 60
2 for tickMark in 0..<minutes {
3     // render the tick mark each minute (60 times)
4 }</pre>
```

Some users might want fewer tick marks in their UI. They could prefer one mark every 5 minutes instead. Use the stride(from:to:by:) function to skip the unwanted marks.

Closed ranges are also available, by using stride(from:through:by:) instead:

While Loops

A while loop performs a set of statements until a condition becomes false. These kinds of loops are best used when the number of iterations is not known before the first iteration begins. Swift provides two kinds of while loops:

- while evaluates its condition at the start of each pass through the loop.
- repeat-while evaluates its condition at the end of each pass through the loop.

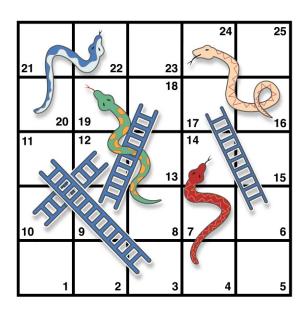
While

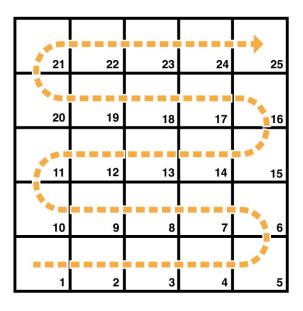
A while loop starts by evaluating a single condition. If the condition is true, a set of statements is repeated until the condition becomes false.

Here's the general form of a while loop:

```
while condition {
    statements
}
```

This example plays a simple game of *Snakes and Ladders* (also known as *Chutes and Ladders*):





The rules of the game are as follows:

- The board has 25 squares, and the aim is to land on or beyond square 25.
- The player's starting square is "square zero", which is just off the bottom-left corner of the board.
- Each turn, you roll a six-sided dice and move by that number of squares, following the horizontal path indicated by the dotted arrow above.
- If your turn ends at the bottom of a ladder, you move up that ladder.
- If your turn ends at the head of a snake, you move down that snake.

The game board is represented by an array of Int values. Its size is based on a constant called finalSquare, which is used to initialize the array and also to check for a win condition later in the example. Because the players start off the board, on "square zero", the board is initialized with 26 zero Int values, not 25.

```
1 let finalSquare = 25
2 var board = [Int](repeating: 0, count: finalSquare + 1)
```

Some squares are then set to have more specific values for the snakes and ladders. Squares with a ladder base have a positive number to move you up the board, whereas squares with a snake head have a negative number to move you back down the board.

Square 3 contains the bottom of a ladder that moves you up to square 11. To represent this, <code>board[03]</code> is equal to <code>+08</code>, which is equivalent to an integer value of 8 (the difference between 3 and <code>11</code>). To align the values and statements, the unary plus operator (<code>+i</code>) is explicitly used with the unary minus operator (<code>-i</code>) and numbers lower than <code>10</code> are padded with zeros. (Neither stylistic technique is

strictly necessary, but they lead to neater code.)

```
1
     var square = 0
    var diceRoll = 0
 2
    while square < finalSquare {</pre>
 3
         // roll the dice
 4
         diceRoll += 1
 5
 6
         if diceRoll == 7 { diceRoll = 1 }
 7
         // move by the rolled amount
         square += diceRoll
 8
         if square < board.count {</pre>
 9
             // if we're still on the board, move up or down for a
10
           snake or a ladder
             square += board[square]
11
         }
12
13
     }
     print("Game over!")
14
```

The example above uses a very simple approach to dice rolling. Instead of generating a random number, it starts with a diceRoll value of 0. Each time through the while loop, diceRoll is incremented by one and is then checked to see whether it has become too large. Whenever this return value equals 7, the dice roll has become too large and is reset to a value of 1. The result is a sequence of diceRoll values that is always 1, 2, 3, 4, 5, 6, 1, 2 and so on.

After rolling the dice, the player moves forward by diceRoll squares. It's possible that the dice roll may have moved the player beyond square 25, in which case the game is over. To cope with this scenario, the code checks that square is less than the board array's count property. If square is valid, the value stored in board[square] is added to the current square value to move the player up or down any ladders or snakes.

If this check is not performed, board[square] might try to access a value outside the bounds of the board array, which would trigger a runtime error.

The current while loop execution then ends, and the loop's condition is checked to see if the loop should be executed again. If the player has moved on or beyond square number 25, the loop's condition evaluates to false and the game ends.

A while loop is appropriate in this case, because the length of the game is not clear at the start of the while loop. Instead, the loop is executed until a particular condition is satisfied.

Repeat-While

The other variation of the while loop, known as the repeat-while loop, performs a single pass through the loop block first, *before* considering the loop's condition. It then continues to repeat the loop until the condition is false.

```
NOTE

The repeat-while loop in Swift is analogous to a do-while loop in other languages.
```

Here's the general form of a repeat-while loop:

```
repeat {
    statements
} while condition
```

Here's the *Snakes and Ladders* example again, written as a repeat-while loop rather than a while loop. The values of finalSquare, board, square, and diceRoll are initialized in exactly the same way as with a while loop.

```
1 let finalSquare = 25
2 var board = [Int](repeating: 0, count: finalSquare + 1)
3 board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
```

In this version of the game, the *first* action in the loop is to check for a ladder or a snake. No ladder on the board takes the player straight to square 25, and so it isn't possible to win the game by moving up a ladder. Therefore, it's safe to check for a snake or a ladder as the first action in the loop.

At the start of the game, the player is on "square zero". board[0] always equals 0 and has no effect.

```
1
     repeat {
 2
         // move up or down for a snake or ladder
         square += board[square]
 3
         // roll the dice
 4
         diceRoll += 1
 5
         if diceRoll == 7 { diceRoll = 1 }
 6
 7
         // move by the rolled amount
         square += diceRoll
 8
9
     } while square < finalSquare</pre>
     print("Game over!")
10
```

After the code checks for snakes and ladders, the dice is rolled and the player is moved forward by diceRoll squares. The current loop execution then ends.

The loop's condition (while square < finalsquare) is the same as before, but this time it's not evaluated until the *end* of the first run through the loop. The structure of the repeat-while loop is better suited to this game than the while loop in the previous example. In the repeat-while loop above, square += board[square] is always executed *immediately after* the loop's while condition confirms that square is still on the board. This behavior removes the need for the array bounds check seen in the while loop version of the game described earlier.

Conditional Statements

It is often useful to execute different pieces of code based on certain conditions. You might want to run an extra piece of code when an error occurs, or to display a message when a value becomes too high or too low. To do this, you make parts of your code *conditional*.

Swift provides two ways to add conditional branches to your code: the if statement and the switch statement. Typically, you use the if statement to evaluate simple conditions with only a few possible outcomes. The switch statement is better suited to more complex conditions with multiple possible permutations and is useful in situations where pattern matching can help select an appropriate code branch to execute.

If

In its simplest form, the if statement has a single if condition. It executes a set of statements only if that condition is true.

```
var temperatureInFahrenheit = 30

if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
}

// Prints "It's very cold. Consider wearing a scarf."</pre>
```

The example above checks whether the temperature is less than or equal to 32 degrees Fahrenheit (the freezing point of water). If it is, a message is printed. Otherwise, no message is printed, and code execution continues after the if statement's closing brace.

The if statement can provide an alternative set of statements, known as an *else clause*, for situations when the if condition is false. These statements are indicated by the else keyword.

```
temperatureInFahrenheit = 40
```

```
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else {
    print("It's not that cold. Wear a t-shirt.")
}
// Prints "It's not that cold. Wear a t-shirt."</pre>
```

One of these two branches is always executed. Because the temperature has increased to 40 degrees Fahrenheit, it is no longer cold enough to advise wearing a scarf and so the else branch is triggered instead.

You can chain multiple if statements together to consider additional clauses.

```
temperatureInFahrenheit = 90
1
2
   if temperatureInFahrenheit <= 32 {</pre>
        print("It's very cold. Consider wearing a scarf.")
3
4
   } else if temperatureInFahrenheit >= 86 {
        print("It's really warm. Don't forget to wear sunscreen.")
5
6
   } else {
7
        print("It's not that cold. Wear a t-shirt.")
8
   // Prints "It's really warm. Don't forget to wear sunscreen."
```

Here, an additional if statement was added to respond to particularly warm temperatures. The final else clause remains, and it prints a response for any temperatures that are neither too warm nor too cold.

The final else clause is optional, however, and can be excluded if the set of conditions does not need to be complete.

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
   print("It's very cold. Consider wearing a scarf.")</pre>
```

```
4  } else if temperatureInFahrenheit >= 86 {
5    print("It's really warm. Don't forget to wear sunscreen.")
6  }
```

Because the temperature is neither too cold nor too warm to trigger the if or else if conditions, no message is printed.

Switch

A switch statement considers a value and compares it against several possible matching patterns. It then executes an appropriate block of code, based on the first pattern that matches successfully. A switch statement provides an alternative to the if statement for responding to multiple potential states.

In its simplest form, a switch statement compares a value against one or more values of the same type.

```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,
    value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

Every switch statement consists of multiple possible *cases*, each of which begins with the case keyword. In addition to comparing against specific values, Swift provides several ways for each case to specify more complex matching patterns. These options are described later in this chapter.

Like the body of an if statement, each case is a separate branch of code execution. The switch statement determines which branch should be selected.

This procedure is known as *switching* on the value that is being considered.

Every switch statement must be *exhaustive*. That is, every possible value of the type being considered must be matched by one of the switch cases. If it's not appropriate to provide a case for every possible value, you can define a default case to cover any values that are not addressed explicitly. This default case is indicated by the default keyword, and must always appear last.

This example uses a switch statement to consider a single lowercase character called someCharacter:

```
let someCharacter: Character = "z"
1
2
    switch someCharacter {
    case "a":
3
        print("The first letter of the alphabet")
4
    case "z":
5
        print("The last letter of the alphabet")
6
7
    default:
        print("Some other character")
8
9
    }
    // Prints "The last letter of the alphabet"
10
```

The switch statement's first case matches the first letter of the English alphabet, a, and its second case matches the last letter, z. Because the switch must have a case for every possible character, not just every alphabetic character, this switch statement uses a default case to match all characters other than a and z. This provision ensures that the switch statement is exhaustive.

No Implicit Fallthrough

In contrast with switch statements in C and Objective-C, switch statements in Swift do not fall through the bottom of each case and into the next one by default. Instead, the entire switch statement finishes its execution as soon as the first matching switch case is completed, without requiring an explicit break statement. This makes the switch statement safer and easier to use than the one

in C and avoids executing more than one switch case by mistake.

```
NOTE
```

Although break is not required in Swift, you can use a break statement to match and ignore a particular case or to break out of a matched case before that case has completed its execution. For details, see <u>Break in a Switch Statement</u>.

The body of each case *must* contain at least one executable statement. It is not valid to write the following code, because the first case is empty:

```
1 let anotherCharacter: Character = "a"
2 switch anotherCharacter {
3 case "a": // Invalid, the case has an empty body
4 case "A":
5 print("The letter A")
6 default:
7 print("Not the letter A")
8 }
9 // This will report a compile-time error.
```

Unlike a switch statement in C, this switch statement does not match both "a" and "A". Rather, it reports a compile-time error that case "a": does not contain any executable statements. This approach avoids accidental fallthrough from one case to another and makes for safer code that is clearer in its intent.

To make a switch with a single case that matches both "a" and "A", combine the two values into a compound case, separating the values with commas.

```
1 let anotherCharacter: Character = "a"
2 switch anotherCharacter {
3 case "a", "A":
4 print("The letter A")
5 default:
```

```
print("Not the letter A")

// Prints "The letter A"
```

For readability, a compound case can also be written over multiple lines. For more information about compound cases, see Compound Cases.

```
NOTE
```

To explicitly fall through at the end of a particular Switch case, use the fallthrough keyword, as described in <u>Fallthrough</u>.

Interval Matching

Values in switch cases can be checked for their inclusion in an interval. This example uses number intervals to provide a natural-language count for numbers of any size:

```
let approximateCount = 62
1
    let countedThings = "moons orbiting Saturn"
2
    let naturalCount: String
3
    switch approximateCount {
5
    case 0:
         naturalCount = "no"
6
7
    case 1 <5:
         naturalCount = "a few"
8
    case 5..<12:
         naturalCount = "several"
10
11
     case 12. <100:
12
         naturalCount = "dozens of"
     case 100 .. <1000:
13
         naturalCount = "hundreds of"
14
     default:
15
```

```
naturalCount = "many"

print("There are \(naturalCount) \(countedThings).")

// Prints "There are dozens of moons orbiting Saturn."
```

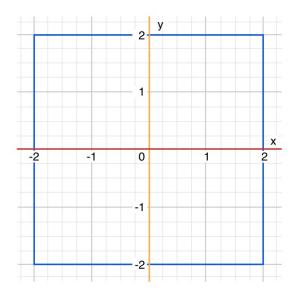
In the above example, approximateCount is evaluated in a switch statement. Each case compares that value to a number or interval. Because the value of approximateCount falls between 12 and 100, naturalCount is assigned the value "dozens of", and execution is transferred out of the switch statement.

Tuples

You can use tuples to test multiple values in the same switch statement. Each element of the tuple can be tested against a different value or interval of values. Alternatively, use the underscore character (_), also known as the wildcard pattern, to match any possible value.

The example below takes an (x, y) point, expressed as a simple tuple of type (Int, Int), and categorizes it on the graph that follows the example.

```
1
    let somePoint = (1, 1)
    switch somePoint {
2
    case (0, 0):
3
         print("\(somePoint) is at the origin")
4
    case (_, 0):
5
         print("\(somePoint) is on the x-axis")
6
7
    case (0, _):
         print("\(somePoint) is on the y-axis")
8
    case (-2...2, -2...2):
         print("\(somePoint) is inside the box")
10
     default:
11
         print("\(somePoint) is outside of the box")
12
13
    }
```



The switch statement determines whether the point is at the origin (0, 0), on the red x-axis, on the orange y-axis, inside the blue 4-by-4 box centered on the origin, or outside of the box.

Unlike C, Swift allows multiple switch cases to consider the same value or values. In fact, the point (0, 0) could match all *four* of the cases in this example. However, if multiple matches are possible, the first matching case is always used. The point (0, 0) would match case (0, 0) first, and so all other matching cases would be ignored.

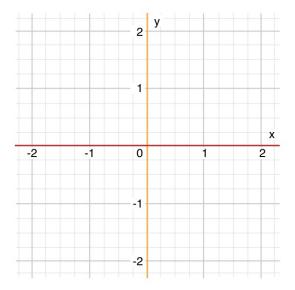
Value Bindings

A switch case can name the value or values it matches to temporary constants or variables, for use in the body of the case. This behavior is known as *value binding*, because the values are bound to temporary constants or variables within the case's body.

The example below takes an (x, y) point, expressed as a tuple of type (Int, Int), and categorizes it on the graph that follows:

```
1 let anotherPoint = (2, 0)
2 switch anotherPoint {
```

```
3
    case (let x, 0):
        print("on the x-axis with an x value of (x)")
4
    case (0, let y):
5
        print("on the y-axis with a y value of (y)")
6
7
    case let (x, y):
        print("somewhere else at (\(x), \(y))")
8
    }
9
    // Prints "on the x-axis with an x value of 2"
10
```



The switch statement determines whether the point is on the red x-axis, on the orange y-axis, or elsewhere (on neither axis).

The three switch cases declare placeholder constants \times and y, which temporarily take on one or both tuple values from anotherPoint. The first case, case (let \times , 0), matches any point with a y value of 0 and assigns the point's x value to the temporary constant x. Similarly, the second case, case (0, let y), matches any point with an x value of 0 and assigns the point's y value to the temporary constant y.

After the temporary constants are declared, they can be used within the case's code block. Here, they are used to print the categorization of the point.

This switch statement does not have a default case. The final case, case let

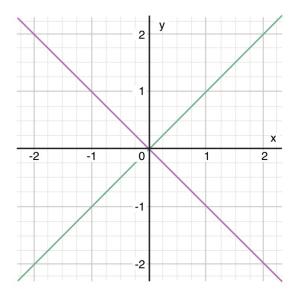
(x, y), declares a tuple of two placeholder constants that can match any value. Because anotherPoint is always a tuple of two values, this case matches all possible remaining values, and a default case is not needed to make the switch statement exhaustive.

Where

A switch case can use a where clause to check for additional conditions.

The example below categorizes an (x, y) point on the following graph:

```
1
   let yetAnotherPoint = (1, -1)
   switch yetAnotherPoint {
2
3
   case let (x, y) where x == y:
        print("(\(x), \(y))) is on the line x == y")
4
   case let (x, y) where x == -y:
5
        print("(\(x), \(y))) is on the line x == -y")
6
7
   case let (x, y):
        print("(\(x), \(y))) is just some arbitrary point")
8
9
   }
    // Prints "(1, -1) is on the line x == -y"
```



The switch statement determines whether the point is on the green diagonal line where x == y, on the purple diagonal line where x == -y, or neither.

The three switch cases declare placeholder constants \times and y, which temporarily take on the two tuple values from yetAnotherPoint. These constants are used as part of a where clause, to create a dynamic filter. The switch case matches the current value of point only if the where clause's condition evaluates to true for that value.

As in the previous example, the final case matches all possible remaining values, and so a default case is not needed to make the switch statement exhaustive.

Compound Cases

Multiple switch cases that share the same body can be combined by writing several patterns after case, with a comma between each of the patterns. If any of the patterns match, then the case is considered to match. The patterns can be written over multiple lines if the list is long. For example:

```
let someCharacter: Character = "e"
1
    switch someCharacter {
2
    case "a", "e", "i", "o", "u":
3
4
        print("\(someCharacter) is a vowel")
    case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
5
         "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
6
        print("\(someCharacter) is a consonant")
7
    default:
8
        print("\(someCharacter) is not a vowel or a consonant")
10
    }
    // Prints "e is a vowel"
11
```

The switch statement's first case matches all five lowercase vowels in the English language. Similarly, its second case matches all lowercase English consonants. Finally, the default case matches any other character.

Compound cases can also include value bindings. All of the patterns of a compound case have to include the same set of value bindings, and each binding has to get a value of the same type from all of the patterns in the compound case. This ensures that, no matter which part of the compound case matched, the code in the body of the case can always access a value for the bindings and that the value always has the same type.

```
1 let stillAnotherPoint = (9, 0)
2 switch stillAnotherPoint {
3 case (let distance, 0), (0, let distance):
4  print("On an axis, \(distance) from the origin")
5 default:
6  print("Not on an axis")
7 }
8 // Prints "On an axis, 9 from the origin"
```

The case above has two patterns: (let distance, 0) matches points on the x-axis and (0, let distance) matches points on the y-axis. Both patterns include a binding for distance and distance is an integer in both patterns—which means that the code in the body of the case can always access a value for distance.

Control Transfer Statements

Control transfer statements change the order in which your code is executed, by transferring control from one piece of code to another. Swift has five control transfer statements:

- continue
- break
- fallthrough

- return
- throw

The continue, break, and fallthrough statements are described below. The return statement is described in <u>Functions</u>, and the throw statement is described in <u>Propagating Errors Using Throwing Functions</u>.

Continue

The continue statement tells a loop to stop what it is doing and start again at the beginning of the next iteration through the loop. It says "I am done with the current loop iteration" without leaving the loop altogether.

The following example removes all vowels and spaces from a lowercase string to create a cryptic puzzle phrase:

```
1
    let puzzleInput = "great minds think alike"
    var puzzleOutput = ""
2
    let charactersToRemove: [Character] = ["a", "e", "i", "o", "u",
3
          " "]
    for character in puzzleInput {
4
        if charactersToRemove.contains(character) {
5
             continue
6
7
        }
8
        puzzleOutput.append(character)
    }
9
    print(puzzleOutput)
10
    // Prints "grtmndsthnklk"
11
```

The code above calls the continue keyword whenever it matches a vowel or a space, causing the current iteration of the loop to end immediately and to jump straight to the start of the next iteration.

Break

The break statement ends execution of an entire control flow statement immediately. The break statement can be used inside a switch or loop statement when you want to terminate the execution of the switch or loop statement earlier than would otherwise be the case.

Break in a Loop Statement

When used inside a loop statement, break ends the loop's execution immediately and transfers control to the code after the loop's closing brace (}). No further code from the current iteration of the loop is executed, and no further iterations of the loop are started.

Break in a Switch Statement

When used inside a switch statement, break causes the switch statement to end its execution immediately and to transfer control to the code after the switch statement's closing brace (}).

This behavior can be used to match and ignore one or more cases in a switch statement. Because Swift's switch statement is exhaustive and does not allow empty cases, it is sometimes necessary to deliberately match and ignore a case in order to make your intentions explicit. You do this by writing the break statement as the entire body of the case you want to ignore. When that case is matched by the switch statement, the break statement inside the case ends the switch statement's execution immediately.

NOTE

A switch case that contains only a comment is reported as a compile-time error. Comments are not statements and do not cause a Switch case to be ignored. Always use a break statement to ignore a Switch case.

The following example switches on a Character value and determines whether it represents a number symbol in one of four languages. For brevity, multiple values are covered in a single switch case.

```
let numberSymbol: Character = "" // Chinese symbol for the
1
           number 3
2
    var possibleIntegerValue: Int?
3
    switch numberSymbol {
    case "1", "\", "" , "<sub>\"</sub>:
4
5
         possibleIntegerValue = 1
    case "2", "Y", "" , "a":
6
7
         possibleIntegerValue = 2
    case "3", "٣", "" , "<sub>□</sub>":
8
9
         possibleIntegerValue = 3
    case "4", "&", "" , "o":
10
         possibleIntegerValue = 4
11
12
     default:
13
         break
14
    }
15
    if let integerValue = possibleIntegerValue {
16
         print("The integer value of \((numberSymbol)) is \
           (integerValue).")
    } else {
17
         print("An integer value could not be found for \
18
           (numberSymbol).")
19
    }
    // Prints "The integer value of is 3."
20
```

This example checks numberSymbol to determine whether it is a Latin, Arabic, Chinese, or Thai symbol for the numbers 1 to 4. If a match is found, one of the switch statement's cases sets an optional Int? variable called possibleIntegerValue to an appropriate integer value.

After the switch statement completes its execution, the example uses optional binding to determine whether a value was found. The possibleIntegerValue

variable has an implicit initial value of nil by virtue of being an optional type, and so the optional binding will succeed only if possibleIntegerValue was set to an actual value by one of the switch statement's first four cases.

Because it's not practical to list every possible Character value in the example above, a default case handles any characters that are not matched. This default case does not need to perform any action, and so it is written with a single break statement as its body. As soon as the default case is matched, the break statement ends the switch statement's execution, and code execution continues from the if let statement.

Fallthrough

In Swift, switch statements don't fall through the bottom of each case and into the next one. That is, the entire switch statement completes its execution as soon as the first matching case is completed. By contrast, C requires you to insert an explicit break statement at the end of every switch case to prevent fallthrough. Avoiding default fallthrough means that Swift switch statements are much more concise and predictable than their counterparts in C, and thus they avoid executing multiple switch cases by mistake.

If you need C-style fallthrough behavior, you can opt in to this behavior on a case-by-case basis with the fallthrough keyword. The example below uses fallthrough to create a textual description of a number.

```
let integerToDescribe = 5
1
   var description = "The number \(integerToDescribe) is"
2
   switch integerToDescribe {
3
4
   case 2, 3, 5, 7, 11, 13, 17, 19:
        description += " a prime number, and also"
5
6
        fallthrough
7
   default:
8
        description += " an integer."
   }
9
```

```
print(description)

// Prints "The number 5 is a prime number, and also an integer."
```

This example declares a new String variable called description and assigns it an initial value. The function then considers the value of integerToDescribe using a switch statement. If the value of integerToDescribe is one of the prime numbers in the list, the function appends text to the end of description, to note that the number is prime. It then uses the fallthrough keyword to "fall into" the default case as well. The default case adds some extra text to the end of the description, and the switch statement is complete.

Unless the value of integerToDescribe is in the list of known prime numbers, it is not matched by the first switch case at all. Because there are no other specific cases, integerToDescribe is matched by the default case.

After the switch statement has finished executing, the number's description is printed using the print(_:separator:terminator:) function. In this example, the number 5 is correctly identified as a prime number.

NOTE

The fallthrough keyword does not check the case conditions for the Switch case that it causes execution to fall into. The fallthrough keyword simply causes code execution to move directly to the statements inside the next case (or default case) block, as in C's standard Switch statement behavior.

Labeled Statements

In Swift, you can nest loops and conditional statements inside other loops and conditional statements to create complex control flow structures. However, loops and conditional statements can both use the break statement to end their execution prematurely. Therefore, it is sometimes useful to be explicit about which loop or conditional statement you want a break statement to terminate. Similarly, if you have multiple nested loops, it can be useful to be explicit about which loop the continue statement should affect.

To achieve these aims, you can mark a loop statement or conditional statement with a *statement label*. With a conditional statement, you can use a statement label with the break statement to end the execution of the labeled statement. With a loop statement, you can use a statement label with the break or continue statement to end or continue the execution of the labeled statement.

A labeled statement is indicated by placing a label on the same line as the statement's introducer keyword, followed by a colon. Here's an example of this syntax for a while loop, although the principle is the same for all loops and switch statements:

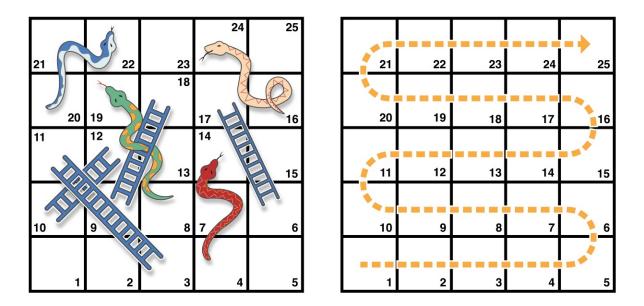
```
label name: while condition {
    statements
}
```

The following example uses the break and continue statements with a labeled while loop for an adapted version of the *Snakes and Ladders* game that you saw earlier in this chapter. This time around, the game has an extra rule:

• To win, you must land *exactly* on square 25.

If a particular dice roll would take you beyond square 25, you must roll again until you roll the exact number needed to land on square 25.

The game board is the same as before.



The values of finalSquare, board, square, and diceRoll are initialized in the same way as before:

```
1 let finalSquare = 25
2 var board = [Int](repeating: 0, count: finalSquare + 1)
3 board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
4 board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
5 var square = 0
6 var diceRoll = 0
```

This version of the game uses a while loop and a switch statement to implement the game's logic. The while loop has a statement label called gameLoop to indicate that it is the main game loop for the Snakes and Ladders game.

The while loop's condition is while square != finalSquare, to reflect that you must land exactly on square 25.

```
gameLoop: while square != finalSquare {
diceRoll += 1
```

```
3
         if diceRoll == 7 { diceRoll = 1 }
         switch square + diceRoll {
4
5
         case finalSquare:
             // diceRoll will move us to the final square, so the
6
           game is over
7
             break gameLoop
         case let newSquare where newSquare > finalSquare:
8
             // diceRoll will move us beyond the final square, so
9
           roll again
             continue gameLoop
10
         default:
11
             // this is a valid move, so find out its effect
12
13
             square += diceRoll
             square += board[square]
14
15
         }
16
     }
17
     print("Game over!")
```

The dice is rolled at the start of each loop. Rather than moving the player immediately, the loop uses a switch statement to consider the result of the move and to determine whether the move is allowed:

- If the dice roll will move the player onto the final square, the game is over. The break gameLoop statement transfers control to the first line of code outside of the while loop, which ends the game.
- If the dice roll will move the player *beyond* the final square, the move is invalid and the player needs to roll again. The continue gameLoop statement ends the current while loop iteration and begins the next iteration of the loop.
- In all other cases, the dice roll is a valid move. The player moves forward by diceRoll squares, and the game logic checks for any snakes and ladders.

The loop then ends, and control returns to the while condition to decide whether another turn is required.

NOTE

If the break statement above did not use the gameLoop label, it would break out of the switch statement, not the while statement. Using the gameLoop label makes it clear which control statement should be terminated.

It is not strictly necessary to use the <code>gameLoop</code> label when calling <code>continue</code> <code>gameLoop</code> to jump to the next iteration of the loop. There is only one loop in the game, and therefore no ambiguity as to which loop the <code>continue</code> statement will affect. However, there is no harm in using the <code>gameLoop</code> label with the <code>continue</code> statement. Doing so is consistent with the label's use alongside the <code>break</code> statement and helps make the <code>game</code>'s logic clearer to read and understand.

Early Exit

A guard statement, like an if statement, executes statements depending on the Boolean value of an expression. You use a guard statement to require that a condition must be true in order for the code after the guard statement to be executed. Unlike an if statement, a guard statement always has an else clause—the code inside the else clause is executed if the condition is not true.

```
1
    func greet(person: [String: String]) {
         guard let name = person["name"] else {
2
3
             return
         }
4
5
         print("Hello \(name)!")
6
7
         guard let location = person["location"] else {
8
             print("I hope the weather is nice near you.")
9
             return
10
11
         }
```

```
12
        print("I hope the weather is nice in \(location).")
13
14
    }
15
     greet(person: ["name": "John"])
16
17
    // Prints "Hello John!"
    // Prints "I hope the weather is nice near you."
18
    greet(person: ["name": "Jane", "location": "Cupertino"])
19
    // Prints "Hello Jane!"
20
    // Prints "I hope the weather is nice in Cupertino."
21
```

If the guard statement's condition is met, code execution continues after the guard statement's closing brace. Any variables or constants that were assigned values using an optional binding as part of the condition are available for the rest of the code block that the guard statement appears in.

If that condition is not met, the code inside the else branch is executed. That branch must transfer control to exit the code block in which the guard statement appears. It can do this with a control transfer statement such as return, break, continue, or throw, or it can call a function or method that doesn't return, such as fatalError(_:file:line:).

Using a guard statement for requirements improves the readability of your code, compared to doing the same check with an if statement. It lets you write the code that's typically executed without wrapping it in an else block, and it lets you keep the code that handles a violated requirement next to the requirement.

Checking API Availability

Swift has built-in support for checking API availability, which ensures that you don't accidentally use APIs that are unavailable on a given deployment target.

The compiler uses availability information in the SDK to verify that all of the

APIs used in your code are available on the deployment target specified by your project. Swift reports an error at compile time if you try to use an API that isn't available.

You use an *availability condition* in an if or guard statement to conditionally execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

```
if #available(iOS 10, macOS 10.12, *) {
    // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on
    macOS
} else {
    // Fall back to earlier iOS and macOS APIs
}
```

The availability condition above specifies that in iOS, the body of the if statement executes only in iOS 10 and later; in macOS, only in macOS 10.12 and later. The last argument, *, is required and specifies that on any other platform, the body of the if executes on the minimum deployment target specified by your target.

In its general form, the availability condition takes a list of platform names and versions. You use platform names such as ios, macos, watchos, and tvos—for the full list, see <u>Declaration Attributes</u>. In addition to specifying major version numbers like iOS 8 or macOS 10.10, you can specify minor versions numbers like iOS 11.2.6 and macOS 10.13.3.

```
if #available( platform name version , ..., *) {
    statements to execute if the APIs are available
} else {
    fallback statements to execute if the APIs are unavailable
}
```

Functions

Functions are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to "call" the function to perform its task when needed.

Swift's unified function syntax is flexible enough to express anything from a simple C-style function with no parameter names to a complex Objective-C-style method with names and argument labels for each parameter. Parameters can provide default values to simplify function calls and can be passed as in-out parameters, which modify a passed variable once the function has completed its execution.

Every function in Swift has a type, consisting of the function's parameter types and return type. You can use this type like any other type in Swift, which makes it easy to pass functions as parameters to other functions, and to return functions from functions. Functions can also be written within other functions to encapsulate useful functionality within a nested function scope.

Defining and Calling Functions

When you define a function, you can optionally define one or more named, typed values that the function takes as input, known as *parameters*. You can also optionally define a type of value that the function will pass back as output when it is done, known as its *return type*.

Every function has a *function name*, which describes the task that the function performs. To use a function, you "call" that function with its name and pass it input values (known as *arguments*) that match the types of the function's parameters. A function's arguments must always be provided in the same order as the function's parameter list.

The function in the example below is called <code>greet(person:)</code>, because that's what it does—it takes a person's name as input and returns a greeting for that person.

To accomplish this, you define one input parameter—a String value called person—and a return type of String, which will contain a greeting for that person:

```
func greet(person: String) -> String {
   let greeting = "Hello, " + person + "!"
   return greeting
}
```

All of this information is rolled up into the function's *definition*, which is prefixed with the func keyword. You indicate the function's return type with the *return arrow* -> (a hyphen followed by a right angle bracket), which is followed by the name of the type to return.

The definition describes what the function does, what it expects to receive, and what it returns when it is done. The definition makes it easy for the function to be called unambiguously from elsewhere in your code:

```
print(greet(person: "Anna"))
// Prints "Hello, Anna!"
print(greet(person: "Brian"))
// Prints "Hello, Brian!"
```

You call the <code>greet(person:)</code> function by passing it a <code>String</code> value after the <code>person</code> argument label, such as <code>greet(person: "Anna")</code>. Because the function returns a <code>String</code> value, <code>greet(person:)</code> can be wrapped in a call to the <code>print(_:separator:terminator:)</code> function to print that string and see its return value, as shown above.

```
{\tt NOTE}
```

The print(_:separator:terminator:) function doesn't have a label for its first argument, and its other arguments are optional because they have a default value. These variations on function syntax are discussed below in Function Argument Labels and Parameter Names and Default Parameter Values.

The body of the <code>greet(person:)</code> function starts by defining a new <code>string</code> constant called <code>greeting</code> and setting it to a simple greeting message. This greeting is then passed back out of the function using the <code>return</code> keyword. In the line of code that says <code>return</code> <code>greeting</code>, the function finishes its execution and returns the current value of <code>greeting</code>.

You can call the <code>greet(person:)</code> function multiple times with different input values. The example above shows what happens if it is called with an input value of "Anna", and an input value of "Brian". The function returns a tailored greeting in each case.

To make the body of this function shorter, you can combine the message creation and the return statement into one line:

```
func greetAgain(person: String) -> String {
   return "Hello again, " + person + "!"
}

print(greetAgain(person: "Anna"))
// Prints "Hello again, Anna!"
```

Function Parameters and Return Values

Function parameters and return values are extremely flexible in Swift. You can define anything from a simple utility function with a single unnamed parameter to a complex function with expressive parameter names and different parameter options.

Functions Without Parameters

Functions are not required to define input parameters. Here's a function with no input parameters, which always returns the same String message whenever it is called:

```
func sayHelloWorld() -> String {
   return "hello, world"
}

print(sayHelloWorld())
// Prints "hello, world"
```

The function definition still needs parentheses after the function's name, even though it does not take any parameters. The function name is also followed by an empty pair of parentheses when the function is called.

Functions With Multiple Parameters

Functions can have multiple input parameters, which are written within the function's parentheses, separated by commas.

This function takes a person's name and whether they have already been greeted as input, and returns an appropriate greeting for that person:

```
1
   func greet(person: String, alreadyGreeted: Bool) -> String {
2
        if alreadyGreeted {
            return greetAgain(person: person)
3
        } else {
            return greet(person: person)
5
        }
6
   }
7
   print(greet(person: "Tim", alreadyGreeted: true))
8
   // Prints "Hello again, Tim!"
```

You call the <code>greet(person:alreadyGreeted:)</code> function by passing it both a <code>String</code> argument value labeled <code>person</code> and a <code>Bool</code> argument value labeled <code>alreadyGreeted</code> in parentheses, separated by commas. Note that this function is distinct from the <code>greet(person:)</code> function shown in an earlier section. Although both functions have names that begin with <code>greet</code>, the

greet(person:alreadyGreeted:) function takes two arguments but the
greet(person:) function takes only one.

Functions Without Return Values

Functions are not required to define a return type. Here's a version of the greet(person:) function, which prints its own String value rather than returning it:

```
func greet(person: String) {
   print("Hello, \(person)!")
}

greet(person: "Dave")
// Prints "Hello, Dave!"
```

Because it does not need to return a value, the function's definition does not include the return arrow (->) or a return type.

```
NOTE
```

Strictly speaking, this version of the <code>greet(person:)</code> function *does* still return a value, even though no return value is defined. Functions without a defined return type return a special value of type <code>Void</code>. This is simply an empty tuple, which is written as ().

The return value of a function can be ignored when it is called:

```
func printAndCount(string: String) -> Int {
   print(string)
   return string.count
}

func printWithoutCounting(string: String) {
   let _ = printAndCount(string: string)
}

printAndCount(string: "hello, world")
```

```
9  // prints "hello, world" and returns a value of 12
10 printWithoutCounting(string: "hello, world")
11  // prints "hello, world" but does not return a value
```

The first function, printAndCount(string:), prints a string, and then returns its character count as an Int. The second function, printWithoutCounting(string:), calls the first function, but ignores its return value. When the second function is called, the message is still printed by the first function, but the returned value is not used.

```
NOTE
```

Return values can be ignored, but a function that says it will return a value must always do so. A function with a defined return type cannot allow control to fall out of the bottom of the function without returning a value, and attempting to do so will result in a compile-time error.

Functions with Multiple Return Values

You can use a tuple type as the return type for a function to return multiple values as part of one compound return value.

The example below defines a function called minMax(array:), which finds the smallest and largest numbers in an array of Int values:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
1
2
         var currentMin = array[0]
         var currentMax = array[0]
3
         for value in array[1..<array.count] {</pre>
4
             if value < currentMin {</pre>
5
                  currentMin = value
6
             } else if value > currentMax {
7
                  currentMax = value
8
             }
9
10
         }
```

```
return (currentMin, currentMax)

return (currentMin, currentMax)
```

The minMax(array:) function returns a tuple containing two Int values. These values are labeled min and max so that they can be accessed by name when querying the function's return value.

The body of the minMax(array:) function starts by setting two working variables called currentMin and currentMax to the value of the first integer in the array. The function then iterates over the remaining values in the array and checks each value to see if it is smaller or larger than the values of currentMin and currentMax respectively. Finally, the overall minimum and maximum values are returned as a tuple of two Int values.

Because the tuple's member values are named as part of the function's return type, they can be accessed with dot syntax to retrieve the minimum and maximum found values:

```
1 let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
2 print("min is \((bounds.min)) and max is \((bounds.max)")
3 // Prints "min is -6 and max is 109"
```

Note that the tuple's members do not need to be named at the point that the tuple is returned from the function, because their names are already specified as part of the function's return type.

Optional Tuple Return Types

If the tuple type to be returned from a function has the potential to have "no value" for the entire tuple, you can use an *optional* tuple return type to reflect the fact that the entire tuple can be nil. You write an optional tuple return type by placing a question mark after the tuple type's closing parenthesis, such as (Int, Int)? Or (String, Int, Bool)?.

```
NOTE

An optional tuple type such as (Int, Int)? is different from a tuple that contains optional types
```

such as (Int?, Int?). With an optional tuple type, the entire tuple is optional, not just each individual value within the tuple.

The minMax(array:) function above returns a tuple containing two Int values. However, the function does not perform any safety checks on the array it is passed. If the array argument contains an empty array, the minMax(array:) function, as defined above, will trigger a runtime error when attempting to access array[0].

To handle an empty array safely, write the minMax(array:) function with an optional tuple return type and return a value of nil when the array is empty:

```
1
     func minMax(array: [Int]) -> (min: Int, max: Int)? {
         if array.isEmpty { return nil }
 2
         var currentMin = array[0]
 3
         var currentMax = array[0]
 4
         for value in array[1..<array.count] {</pre>
 5
             if value < currentMin {</pre>
 6
 7
                  currentMin = value
             } else if value > currentMax {
 8
 9
                  currentMax = value
             }
10
11
         return (currentMin, currentMax)
12
13
     }
```

You can use optional binding to check whether this version of the minMax(array:) function returns an actual tuple value or nil:

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
   print("min is \((bounds.min)) and max is \((bounds.max)\)")
}
// Prints "min is -6 and max is 109"
```

Functions With an Implicit Return

If the entire body of the function is a single expression, the function implicitly returns that expression. For example, both functions below have the same behavior:

```
1
    func greeting(for person: String) -> String {
         "Hello, " + person + "!"
2
3
    }
    print(greeting(for: "Dave"))
4
    // Prints "Hello, Dave!"
5
6
7
    func anotherGreeting(for person: String) -> String {
        return "Hello, " + person + "!"
8
    }
    print(anotherGreeting(for: "Dave"))
10
11
    // Prints "Hello, Dave!"
```

The entire definition of the <code>greeting(for:)</code> function is the greeting message that it returns, which means it can use this shorter form. The <code>anotherGreeting(for:)</code> function returns the same greeting message, using the <code>return</code> keyword like a longer function. Any function that you write as just one <code>return</code> line can omit the <code>return</code>.

As you'll see in <u>Shorthand Getter Declaration</u>, property getters can also use an implicit return.

Function Argument Labels and Parameter Names

Each function parameter has both an *argument label* and a *parameter name*. The argument label is used when calling the function; each argument is written in the function call with its argument label before it. The parameter name is used in the implementation of the function. By default, parameters use their parameter name

as their argument label.

All parameters must have unique names. Although it's possible for multiple parameters to have the same argument label, unique argument labels help make your code more readable.

Specifying Argument Labels

You write an argument label before the parameter name, separated by a space:

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the
        argument value
    // for that parameter.
}
```

Here's a variation of the greet(person:) function that takes a person's name and hometown and returns a greeting:

```
print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill! Glad you could visit from Cupertino."
```

The use of argument labels can allow a function to be called in an expressive, sentence-like manner, while still providing a function body that is readable and clear in intent.

Omitting Argument Labels

If you don't want an argument label for a parameter, write an underscore (_) instead of an explicit argument label for that parameter.

If a parameter has an argument label, the argument *must* be labeled when you call the function.

Default Parameter Values

You can define a *default value* for any parameter in a function by assigning a value to the parameter after that parameter's type. If a default value is defined, you can omit that parameter when calling the function.

```
// If you omit the second argument when calling this
function, then

// the value of parameterWithDefault is 12 inside the
function body.

SomeFunction(parameterWithoutDefault: 3, parameterWithDefault:
6) // parameterWithDefault is 6

someFunction(parameterWithoutDefault: 4) //
parameterWithDefault is 12
```

Place parameters that don't have default values at the beginning of a function's parameter list, before the parameters that have default values. Parameters that don't have default values are usually more important to the function's meaning —writing them first makes it easier to recognize that the same function is being called, regardless of whether any default parameters are omitted.

Variadic Parameters

A *variadic parameter* accepts zero or more values of a specified type. You use a variadic parameter to specify that the parameter can be passed a varying number of input values when the function is called. Write variadic parameters by inserting three period characters (...) after the parameter's type name.

The values passed to a variadic parameter are made available within the function's body as an array of the appropriate type. For example, a variadic parameter with a name of numbers and a type of <code>Double...</code> is made available within the function's body as a constant array called <code>numbers</code> of type <code>[Double]</code>.

The example below calculates the *arithmetic mean* (also known as the *average*) for a list of numbers of any length:

```
func arithmeticMean(_ numbers: Double...) -> Double {
  var total: Double = 0
```

```
3
        for number in numbers {
4
             total += number
5
        }
        return total / Double(numbers.count)
6
7
8
    arithmeticMean(1, 2, 3, 4, 5)
    // returns 3.0, which is the arithmetic mean of these five
          numbers
10
    arithmeticMean(3, 8.25, 18.75)
    // returns 10.0, which is the arithmetic mean of these three
11
          numbers
```

NOTE

A function may have at most one variadic parameter.

In-Out Parameters

Function parameters are constants by default. Trying to change the value of a function parameter from within the body of that function results in a compile-time error. This means that you can't change the value of a parameter by mistake. If you want a function to modify a parameter's value, and you want those changes to persist after the function call has ended, define that parameter as an *in-out parameter* instead.

You write an in-out parameter by placing the inout keyword right before a parameter's type. An in-out parameter has a value that is passed *in* to the function, is modified by the function, and is passed back *out* of the function to replace the original value. For a detailed discussion of the behavior of in-out parameters and associated compiler optimizations, see <u>In-Out Parameters</u>.

You can only pass a variable as the argument for an in-out parameter. You cannot pass a constant or a literal value as the argument, because constants and literals cannot be modified. You place an ampersand (&) directly before a variable's

name when you pass it as an argument to an in-out parameter, to indicate that it can be modified by the function.

```
NOTE

In-out parameters cannot have default values, and variadic parameters cannot be marked as inout.
```

Here's an example of a function called swapTwoInts(_:_:), which has two in-out integer parameters called a and b:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
   let temporaryA = a
   a = b
   b = temporaryA
}
```

The swapTwoInts(_:_:) function simply swaps the value of b into a, and the value of a into b. The function performs this swap by storing the value of a in a temporary constant called temporaryA, assigning the value of b to a, and then assigning temporaryA to b.

You can call the swapTwoInts(_:_:) function with two variables of type Int to swap their values. Note that the names of someInt and anotherInt are prefixed with an ampersand when they are passed to the swapTwoInts(_:_:) function:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

The example above shows that the original values of someInt and anotherInt are modified by the swapTwoInts(_:_:) function, even though they were originally defined outside of the function.

NOTE

In-out parameters are not the same as returning a value from a function. The SwapTwoInts example above does not define a return type or return a value, but it still modifies the values of someInt and anotherInt. In-out parameters are an alternative way for a function to have an effect outside of the scope of its function body.

Function Types

Every function has a specific *function type*, made up of the parameter types and the return type of the function.

For example:

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}

func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

This example defines two simple mathematical functions called addTwoInts and multiplyTwoInts. These functions each take two Int values, and return an Int value, which is the result of performing an appropriate mathematical operation.

The type of both of these functions is (Int, Int) -> Int. This can be read as:

"A function that has two parameters, both of type Int, and that returns a value of type Int."

Here's another example, for a function with no parameters or return value:

```
func printHelloWorld() {
print("hello, world")
```

```
3 }
```

The type of this function is () \rightarrow Void, or "a function that has no parameters, and returns Void."

Using Function Types

You use function types just like any other types in Swift. For example, you can define a constant or variable to be of a function type and assign an appropriate function to that variable:

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

This can be read as:

"Define a variable called mathFunction, which has a type of 'a function that takes two Int values, and returns an Int value.' Set this new variable to refer to the function called addTwoInts."

The addTwoInts(_:_:) function has the same type as the mathFunction variable, and so this assignment is allowed by Swift's type-checker.

You can now call the assigned function with the name mathFunction:

```
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"
```

A different function with the same matching type can be assigned to the same variable, in the same way as for nonfunction types:

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"
```

As with any other type, you can leave it to Swift to infer the function type when

you assign a function to a constant or variable:

Function Types as Parameter Types

You can use a function type such as (Int, Int) -> Int as a parameter type for another function. This enables you to leave some aspects of a function's implementation for the function's caller to provide when the function is called.

Here's an example to print the results of the math functions from above:

This example defines a function called printMathResult(_:_:_:), which has three parameters. The first parameter is called mathFunction, and is of type (Int, Int) -> Int. You can pass any function of that type as the argument for this first parameter. The second and third parameters are called a and b, and are both of type Int. These are used as the two input values for the provided math function.

When printMathResult(_:_:_:) is called, it is passed the addTwoInts(_:_:) function, and the integer values 3 and 5. It calls the provided function with the values 3 and 5, and prints the result of 8.

The role of printMathResult(_:_:_:) is to print the result of a call to a math function of an appropriate type. It doesn't matter what that function's implementation actually does—it matters only that the function is of the correct

type. This enables printMathResult(_:_:_:) to hand off some of its functionality to the caller of the function in a type-safe way.

Function Types as Return Types

You can use a function type as the return type of another function. You do this by writing a complete function type immediately after the return arrow (->) of the returning function.

The next example defines two simple functions called <code>stepForward(_:)</code> and <code>stepBackward(_:)</code>. The <code>stepForward(_:)</code> function returns a value one more than its input value, and the <code>stepBackward(_:)</code> function returns a value one less than its input value. Both functions have a type of <code>(Int) -> Int:</code>

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}

func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

Here's a function called chooseStepFunction(backward:), whose return type is (Int) -> Int. The chooseStepFunction(backward:) function returns the stepForward(_:) function or the stepBackward(_:) function based on a Boolean parameter called backward:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

You can now use chooseStepFunction(backward:) to obtain a function that will step in one direction or the other:

```
1 var currentValue = 3
```

The example above determines whether a positive or negative step is needed to move a variable called currentValue progressively closer to zero. currentValue has an initial value of 3, which means that currentValue > 0 returns true, causing chooseStepFunction(backward:) to return the stepBackward(_:) function. A reference to the returned function is stored in a constant called moveNearerToZero.

Now that moveNearerToZero refers to the correct function, it can be used to count to zero:

```
print("Counting to zero:")
1
2
    // Counting to zero:
3
    while currentValue != 0 {
        print("\(currentValue)...")
4
        currentValue = moveNearerToZero(currentValue)
5
6
    }
    print("zero!")
7
8
   // 3...
   // 2...
10 // 1...
11
    // zero!
```

Nested Functions

All of the functions you have encountered so far in this chapter have been examples of *global functions*, which are defined at a global scope. You can also define functions inside the bodies of other functions, known as *nested functions*.

Nested functions are hidden from the outside world by default, but can still be called and used by their enclosing function. An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope.

You can rewrite the chooseStepFunction(backward:) example above to use and return nested functions:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
1
        func stepForward(input: Int) -> Int { return input + 1 }
2
        func stepBackward(input: Int) -> Int { return input - 1 }
3
        return backward ? stepBackward : stepForward
5
    }
    var currentValue = -4
6
    let moveNearerToZero = chooseStepFunction(backward:
7
          currentValue > 0)
8
    // moveNearerToZero now refers to the nested stepForward()
          function
    while currentValue != 0 {
9
        print("\(currentValue)...")
10
        currentValue = moveNearerToZero(currentValue)
11
12
    }
    print("zero!")
13
14
    // -4...
15
    // -3...
    // -2...
16
17
    // -1...
    // zero!
18
```

Closures

Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.

Closures can capture and store references to any constants and variables from the context in which they are defined. This is known as *closing over* those constants and variables. Swift handles all of the memory management of capturing for you.

NOTE

Don't worry if you are not familiar with the concept of capturing. It is explained in detail below in <u>Capturing Values</u>.

Global and nested functions, as introduced in <u>Functions</u>, are actually special cases of closures. Closures take one of three forms:

- Global functions are closures that have a name and do not capture any values.
- Nested functions are closures that have a name and can capture values from their enclosing function.
- Closure expressions are unnamed closures written in a lightweight syntax that can capture values from their surrounding context.

Swift's closure expressions have a clean, clear style, with optimizations that encourage brief, clutter-free syntax in common scenarios. These optimizations include:

- Inferring parameter and return value types from context
- Implicit returns from single-expression closures
- Shorthand argument names

• Trailing closure syntax

Closure Expressions

Nested functions, as introduced in <u>Nested Functions</u>, are a convenient means of naming and defining self-contained blocks of code as part of a larger function. However, it is sometimes useful to write shorter versions of function-like constructs without a full declaration and name. This is particularly true when you work with functions or methods that take functions as one or more of their arguments.

Closure expressions are a way to write inline closures in a brief, focused syntax. Closure expressions provide several syntax optimizations for writing closures in a shortened form without loss of clarity or intent. The closure expression examples below illustrate these optimizations by refining a single example of the sorted(by:) method over several iterations, each of which expresses the same functionality in a more succinct way.

The Sorted Method

Swift's standard library provides a method called <code>sorted(by:)</code>, which sorts an array of values of a known type, based on the output of a sorting closure that you provide. Once it completes the sorting process, the <code>sorted(by:)</code> method returns a new array of the same type and size as the old one, with its elements in the correct sorted order. The original array is not modified by the <code>sorted(by:)</code> method.

The closure expression examples below use the sorted(by:) method to sort an array of String values in reverse alphabetical order. Here's the initial array to be sorted:

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

The sorted(by:) method accepts a closure that takes two arguments of the same

type as the array's contents, and returns a Bool value to say whether the first value should appear before or after the second value once the values are sorted. The sorting closure needs to return true if the first value should appear *before* the second value, and false otherwise.

This example is sorting an array of String values, and so the sorting closure needs to be a function of type (String, String) -> Bool.

One way to provide the sorting closure is to write a normal function of the correct type, and to pass it in as an argument to the sorted(by:) method:

If the first string (s1) is greater than the second string (s2), the backward(_:_:) function will return true, indicating that s1 should appear before s2 in the sorted array. For characters in strings, "greater than" means "appears later in the alphabet than". This means that the letter "B" is "greater than" the letter "A", and the string "Tom" is greater than the string "Tim". This gives a reverse alphabetical sort, with "Barry" being placed before "Alex", and so on.

However, this is a rather long-winded way to write what is essentially a single-expression function (a > b). In this example, it would be preferable to write the sorting closure inline, using closure expression syntax.

Closure Expression Syntax

Closure expression syntax has the following general form:

```
{ (parameters) -> return type in statements
```

The *parameters* in closure expression syntax can be in-out parameters, but they can't have a default value. Variadic parameters can be used if you name the variadic parameter. Tuples can also be used as parameter types and return types.

The example below shows a closure expression version of the backward(_:_:) function from above:

Note that the declaration of parameters and return type for this inline closure is identical to the declaration from the backward(_:_:) function. In both cases, it is written as (s1: String, s2: String) -> Bool. However, for the inline closure expression, the parameters and return type are written *inside* the curly braces, not outside of them.

The start of the closure's body is introduced by the in keyword. This keyword indicates that the definition of the closure's parameters and return type has finished, and the body of the closure is about to begin.

Because the body of the closure is so short, it can even be written on a single line:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) ->
    Bool in return s1 > s2 } )
```

This illustrates that the overall call to the <code>sorted(by:)</code> method has remained the same. A pair of parentheses still wrap the entire argument for the method. However, that argument is now an inline closure.

Inferring Type From Context

Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns. The <code>sorted(by:)</code> method is being called on an array of strings, so its argument must be a function of type (<code>String</code>, <code>String</code>) -> <code>Bool</code>. This means that the (<code>String</code>, <code>String</code>) and <code>Bool</code> types do not need to be written as part of the closure expression's definition. Because all of the types can be inferred, the return arrow (->) and the parentheses around the names of the parameters can also be omitted:

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

It is always possible to infer the parameter types and return type when passing a closure to a function or method as an inline closure expression. As a result, you never need to write an inline closure in its fullest form when the closure is used as a function or method argument.

Nonetheless, you can still make the types explicit if you wish, and doing so is encouraged if it avoids ambiguity for readers of your code. In the case of the sorted(by:) method, the purpose of the closure is clear from the fact that sorting is taking place, and it is safe for a reader to assume that the closure is likely to be working with String values, because it is assisting with the sorting of an array of strings.

Implicit Returns from Single-Expression Closures

Single-expression closures can implicitly return the result of their single expression by omitting the return keyword from their declaration, as in this version of the previous example:

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```

Here, the function type of the sorted(by:) method's argument makes it clear that a Bool value must be returned by the closure. Because the closure's body contains a single expression (s1 > s2) that returns a Bool value, there is no ambiguity, and the return keyword can be omitted.

Shorthand Argument Names

Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names \$0, \$1, \$2, and so on.

If you use these shorthand argument names within your closure expression, you can omit the closure's argument list from its definition, and the number and type of the shorthand argument names will be inferred from the expected function type. The in keyword can also be omitted, because the closure expression is made up entirely of its body:

```
reversedNames = names.sorted(by: \{ \$0 > \$1 \} )
```

Here, \$0 and \$1 refer to the closure's first and second String arguments.

Operator Methods

There's actually an even *shorter* way to write the closure expression above. Swift's string type defines its string-specific implementation of the greater-than operator (>) as a method that has two parameters of type String, and returns a value of type Bool. This exactly matches the method type needed by the sorted(by:) method. Therefore, you can simply pass in the greater-than operator, and Swift will infer that you want to use its string-specific implementation:

```
reversedNames = names.sorted(by: >)
```

For more about operator method, see **Operator Methods**.

Trailing Closures

If you need to pass a closure expression to a function as the function's final argument and the closure expression is long, it can be useful to write it as a

trailing closure instead. A trailing closure is written after the function call's parentheses, even though it is still an argument to the function. When you use the trailing closure syntax, you don't write the argument label for the closure as part of the function call.

```
func someFunctionThatTakesAClosure(closure: () -> Void) {
1
        // function body goes here
    }
3
4
    // Here's how you call this function without using a trailing
5
          closure:
6
7
    someFunctionThatTakesAClosure(closure: {
8
        // closure's body goes here
    })
9
10
    // Here's how you call this function with a trailing closure
11
          instead:
12
13
    someFunctionThatTakesAClosure() {
        // trailing closure's body goes here
14
15
    }
```

The string-sorting closure from the <u>Closure Expression Syntax</u> section above can be written outside of the <code>sorted(by:)</code> method's parentheses as a trailing closure:

```
reversedNames = names.sorted() { $0 > $1 }
```

If a closure expression is provided as the function or method's only argument and you provide that expression as a trailing closure, you do not need to write a pair of parentheses () after the function or method's name when you call the function:

```
reversedNames = names.sorted { $0 > $1 }
```

Trailing closures are most useful when the closure is sufficiently long that it is not possible to write it inline on a single line. As an example, Swift's Array type has a map(_:) method which takes a closure expression as its single argument. The closure is called once for each item in the array, and returns an alternative mapped value (possibly of some other type) for that item. The nature of the mapping and the type of the returned value is left up to the closure to specify.

After applying the provided closure to each array element, the map(_:) method returns a new array containing all of the new mapped values, in the same order as their corresponding values in the original array.

Here's how you can use the map(_:) method with a trailing closure to convert an array of Int values into an array of String values. The array [16, 58, 510] is used to create the new array ["OneSix", "FiveEight", "FiveOneZero"]:

```
1 let digitNames = [
2    0: "Zero", 1: "One", 2: "Two", 3: "Three", 4: "Four",
3    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
4 ]
5 let numbers = [16, 58, 510]
```

The code above creates a dictionary of mappings between the integer digits and English-language versions of their names. It also defines an array of integers, ready to be converted into strings.

You can now use the numbers array to create an array of string values, by passing a closure expression to the array's map(_:) method as a trailing closure:

```
1 let strings = numbers.map { (number) -> String in
2     var number = number
3     var output = ""
4     repeat {
5         output = digitNames[number % 10]! + output
```

```
6    number /= 10
7    } while number > 0
8    return output
9  }
10  // strings is inferred to be of type [String]
11  // its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

The map(_:) method calls the closure expression once for each item in the array. You do not need to specify the type of the closure's input parameter, number, because the type can be inferred from the values in the array to be mapped.

In this example, the variable number is initialized with the value of the closure's number parameter, so that the value can be modified within the closure body. (The parameters to functions and closures are always constants.) The closure expression also specifies a return type of String, to indicate the type that will be stored in the mapped output array.

The closure expression builds a string called output each time it is called. It calculates the last digit of number by using the remainder operator (number % 10), and uses this digit to look up an appropriate string in the digitNames dictionary. The closure can be used to create a string representation of any integer greater than zero.

NOTE

The call to the digitNames dictionary's subscript is followed by an exclamation mark (!), because dictionary subscripts return an optional value to indicate that the dictionary lookup can fail if the key does not exist. In the example above, it is guaranteed that number % 10 will always be a valid subscript key for the digitNames dictionary, and so an exclamation mark is used to force-unwrap the String value stored in the subscript's optional return value.

The string retrieved from the digitNames dictionary is added to the *front* of output, effectively building a string version of the number in reverse. (The expression number % 10 gives a value of 6 for 16, 8 for 58, and 0 for 510.)

The number variable is then divided by 10. Because it is an integer, it is rounded down during the division, so 16 becomes 1, 58 becomes 5, and 510 becomes 51.

The process is repeated until number is equal to 0, at which point the output string is returned by the closure, and is added to the output array by the map(_:) method.

The use of trailing closure syntax in the example above neatly encapsulates the closure's functionality immediately after the function that closure supports, without needing to wrap the entire closure within the $map(_:)$ method's outer parentheses.

Capturing Values

A closure can *capture* constants and variables from the surrounding context in which it is defined. The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists.

In Swift, the simplest form of a closure that can capture values is a nested function, written within the body of another function. A nested function can capture any of its outer function's arguments and can also capture any constants and variables defined within the outer function.

Here's an example of a function called makeIncrementer, which contains a nested function called incrementer. The nested incrementer() function captures two values, runningTotal and amount, from its surrounding context. After capturing these values, incrementer is returned by makeIncrementer as a closure that increments runningTotal by amount each time it is called.

```
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
   var runningTotal = 0
   func incrementer() -> Int {
      runningTotal += amount
      return runningTotal
   }
   return incrementer
```

```
8 }
```

The return type of makeIncrementer is () -> Int. This means that it returns a *function*, rather than a simple value. The function it returns has no parameters, and returns an Int value each time it is called. To learn how functions can return other functions, see <u>Function Types</u> as <u>Return Types</u>.

The makeIncrementer(forIncrement:) function defines an integer variable called runningTotal, to store the current running total of the incrementer that will be returned. This variable is initialized with a value of 0.

The makeIncrementer(forIncrement:) function has a single Int parameter with an argument label of forIncrement, and a parameter name of amount. The argument value passed to this parameter specifies how much runningTotal should be incremented by each time the returned incrementer function is called. The makeIncrementer function defines a nested function called incrementer, which performs the actual incrementing. This function simply adds amount to runningTotal, and returns the result.

When considered in isolation, the nested incrementer() function might seem unusual:

```
func incrementer() -> Int {
    runningTotal += amount
    return runningTotal
}
```

The incrementer() function doesn't have any parameters, and yet it refers to runningTotal and amount from within its function body. It does this by capturing a reference to runningTotal and amount from the surrounding function and using them within its own function body. Capturing by reference ensures that runningTotal and amount do not disappear when the call to makeIncrementer ends, and also ensures that runningTotal is available the next time the incrementer function is called.

As an optimization, Swift may instead capture and store a *copy* of a value if that value is not mutated by a closure, and if the value is not mutated after the closure is created.

Swift also handles all memory management involved in disposing of variables when they are no longer needed.

Here's an example of makeIncrementer in action:

```
let incrementByTen = makeIncrementer(forIncrement: 10)
```

This example sets a constant called incrementByTen to refer to an incrementer function that adds 10 to its runningTotal variable each time it is called. Calling the function multiple times shows this behavior in action:

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

If you create a second incrementer, it will have its own stored reference to a new, separate runningTotal variable:

```
1 let incrementBySeven = makeIncrementer(forIncrement: 7)
2 incrementBySeven()
3 // returns a value of 7
```

Calling the original incrementer (incrementByTen) again continues to increment its own runningTotal variable, and does not affect the variable captured by incrementBySeven:

```
incrementByTen()
// returns a value of 40
```

NOTE

If you assign a closure to a property of a class instance, and the closure captures that instance by referring to the instance or its members, you will create a strong reference cycle between the closure and the instance. Swift uses *capture lists* to break these strong reference cycles. For more information, see Strong Reference Cycles for Closures.

Closures Are Reference Types

In the example above, incrementBySeven and incrementByTen are constants, but the closures these constants refer to are still able to increment the runningTotal variables that they have captured. This is because functions and closures are reference types.

Whenever you assign a function or a closure to a constant or a variable, you are actually setting that constant or variable to be a *reference* to the function or closure. In the example above, it is the choice of closure that incrementByTen *refers to* that is constant, and not the contents of the closure itself.

This also means that if you assign a closure to two different constants or variables, both of those constants or variables refer to the same closure.

```
1 let alsoIncrementByTen = incrementByTen
2 alsoIncrementByTen()
3 // returns a value of 50
4
5 incrementByTen()
6 // returns a value of 60
```

The example above shows that calling alsoIncrementByTen is the same as calling incrementByTen. Because both of them refer to the same closure, they both increment and return the same running total.

Escaping Closures

A closure is said to *escape* a function when the closure is passed as an argument to the function, but is called after the function returns. When you declare a function that takes a closure as one of its parameters, you can write <code>@escaping</code> before the parameter's type to indicate that the closure is allowed to escape.

One way that a closure can escape is by being stored in a variable that is defined outside the function. As an example, many functions that start an asynchronous operation take a closure argument as a completion handler. The function returns after it starts the operation, but the closure isn't called until the operation is completed—the closure needs to escape, to be called later. For example:

```
var completionHandlers: [() -> Void] = []

func someFunctionWithEscapingClosure(completionHandler:
    @escaping () -> Void) {
    completionHandlers.append(completionHandler)
}
```

The someFunctionWithEscapingClosure(_:) function takes a closure as its argument and adds it to an array that's declared outside the function. If you didn't mark the parameter of this function with @escaping, you would get a compile-time error.

Marking a closure with @escaping means you have to refer to self explicitly within the closure. For example, in the code below, the closure passed to someFunctionWithEscapingClosure(_:) is an escaping closure, which means it needs to refer to self explicitly. In contrast, the closure passed to someFunctionWithNonescapingClosure(_:) is a nonescaping closure, which means it can refer to self implicitly.

```
func someFunctionWithNonescapingClosure(closure: () -> Void) {
   closure()
}
```

```
5
    class SomeClass {
6
         var x = 10
7
         func doSomething() {
             someFunctionWithEscapingClosure { self.x = 100 }
8
             someFunctionWithNonescapingClosure { x = 200 }
9
         }
10
11
     }
12
13
    let instance = SomeClass()
    instance.doSomething()
14
15
    print(instance.x)
16
    // Prints "200"
17
     completionHandlers.first?()
18
19
     print(instance.x)
     // Prints "100"
20
```

Autoclosures

An *autoclosure* is a closure that is automatically created to wrap an expression that's being passed as an argument to a function. It doesn't take any arguments, and when it's called, it returns the value of the expression that's wrapped inside of it. This syntactic convenience lets you omit braces around a function's parameter by writing a normal expression instead of an explicit closure.

It's common to *call* functions that take autoclosures, but it's not common to *implement* that kind of function. For example, the

assert(condition:message:file:line:) function takes an autoclosure for its condition and message parameters; its condition parameter is evaluated only in debug builds and its message parameter is evaluated only if condition is false.

An autoclosure lets you delay evaluation, because the code inside isn't run until you call the closure. Delaying evaluation is useful for code that has side effects or is computationally expensive, because it lets you control when that code is evaluated. The code below shows how a closure delays evaluation.

```
var customersInLine = ["Chris", "Alex", "Ewa", "Barry",
1
          "Daniella"]
    print(customersInLine.count)
2
    // Prints "5"
3
4
    let customerProvider = { customersInLine.remove(at: 0) }
5
    print(customersInLine.count)
6
7
    // Prints "5"
8
    print("Now serving \(customerProvider())!")
9
    // Prints "Now serving Chris!"
10
    print(customersInLine.count)
11
12
    // Prints "4"
```

Even though the first element of the customersInLine array is removed by the code inside the closure, the array element isn't removed until the closure is actually called. If the closure is never called, the expression inside the closure is never evaluated, which means the array element is never removed. Note that the type of customerProvider is not String but () -> String—a function with no parameters that returns a string.

You get the same behavior of delayed evaluation when you pass a closure as an argument to a function.

```
// customersInLine is ["Alex", "Ewa", "Barry", "Daniella"]
func serve(customer customerProvider: () -> String) {
   print("Now serving \((customerProvider())!")
}
```

```
5 serve(customer: { customersInLine.remove(at: 0) } )
6 // Prints "Now serving Alex!"
```

The serve(customer:) function in the listing above takes an explicit closure that returns a customer's name. The version of serve(customer:) below performs the same operation but, instead of taking an explicit closure, it takes an autoclosure by marking its parameter's type with the @autoclosure attribute. Now you can call the function as if it took a String argument instead of a closure. The argument is automatically converted to a closure, because the customerProvider parameter's type is marked with the @autoclosure attribute.

NOTE

Overusing autoclosures can make your code hard to understand. The context and function name should make it clear that evaluation is being deferred.

If you want an autoclosure that is allowed to escape, use both the <code>@autoclosure</code> and <code>@escaping</code> attributes. The <code>@escaping</code> attribute is described above in <code>Escaping</code> Closures.

```
// customersInLine is ["Barry", "Daniella"]
var customerProviders: [() -> String] = []

func collectCustomerProviders(_ customerProvider: @autoclosure
     @escaping () -> String) {
    customerProviders.append(customerProvider)
}
```

```
collectCustomerProviders(customersInLine.remove(at: 0))
7
    collectCustomerProviders(customersInLine.remove(at: 0))
8
    print("Collected \(customerProviders.count) closures.")
9
    // Prints "Collected 2 closures."
10
    for customerProvider in customerProviders {
11
         print("Now serving \(customerProvider())!")
12
13
    }
    // Prints "Now serving Barry!"
14
    // Prints "Now serving Daniella!"
15
```

In the code above, instead of calling the closure passed to it as its <code>customerProvider</code> argument, the <code>collectCustomerProviders(_:)</code> function appends the closure to the <code>customerProviders</code> array. The array is declared outside the scope of the function, which means the closures in the array can be executed after the function returns. As a result, the value of the <code>customerProvider</code> argument must be allowed to escape the function's scope.

Enumerations

An *enumeration* defines a common type for a group of related values and enables you to work with those values in a type-safe way within your code.

If you are familiar with C, you will know that C enumerations assign related names to a set of integer values. Enumerations in Swift are much more flexible, and don't have to provide a value for each case of the enumeration. If a value (known as a *raw* value) is provided for each enumeration case, the value can be a string, a character, or a value of any integer or floating-point type.

Alternatively, enumeration cases can specify associated values of *any* type to be stored along with each different case value, much as unions or variants do in other languages. You can define a common set of related cases as part of one enumeration, each of which has a different set of values of appropriate types associated with it.

Enumerations in Swift are first-class types in their own right. They adopt many features traditionally supported only by classes, such as computed properties to provide additional information about the enumeration's current value, and instance methods to provide functionality related to the values the enumeration represents. Enumerations can also define initializers to provide an initial case value; can be extended to expand their functionality beyond their original implementation; and can conform to protocols to provide standard functionality.

For more about these capabilities, see <u>Properties</u>, <u>Methods</u>, <u>Initialization</u>, <u>Extensions</u>, and <u>Protocols</u>.

Enumeration Syntax

You introduce enumerations with the enum keyword and place their entire definition within a pair of braces:

```
1  enum SomeEnumeration {
2    // enumeration definition goes here
3 }
```

Here's an example for the four main points of a compass:

```
1 enum CompassPoint {
2    case north
3    case south
4    case east
5    case west
6 }
```

The values defined in an enumeration (such as north, south, east, and west) are its *enumeration cases*. You use the case keyword to introduce new enumeration cases.

NOTE

Swift enumeration cases don't have an integer value set by default, unlike languages like C and Objective-C. In the CompassPoint example above, north, south, east and west don't implicitly equal 0, 1, 2 and 3. Instead, the different enumeration cases are values in their own right, with an explicitly defined type of CompassPoint.

Multiple cases can appear on a single line, separated by commas:

Each enumeration definition defines a new type. Like other types in Swift, their names (such as CompassPoint and Planet) start with a capital letter. Give enumeration types singular rather than plural names, so that they read as self-evident:

```
var directionToHead = CompassPoint.west
```

The type of directionToHead is inferred when it's initialized with one of the possible values of CompassPoint. Once directionToHead is declared as a CompassPoint, you can set it to a different CompassPoint value using a shorter dot syntax:

```
directionToHead = .east
```

The type of directionToHead is already known, and so you can drop the type when setting its value. This makes for highly readable code when working with explicitly typed enumeration values.

Matching Enumeration Values with a Switch Statement

You can match individual enumeration values with a switch statement:

```
directionToHead = .south
    switch directionToHead {
    case north:
3
        print("Lots of planets have a north")
4
    case south:
5
        print("Watch out for penguins")
6
    case east:
7
        print("Where the sun rises")
8
    case west:
9
         print("Where the skies are blue")
10
11
    }
    // Prints "Watch out for penguins"
12
```

You can read this code as:

"Consider the value of directionToHead. In the case where it equals .north, print "Lots of planets have a north". In the case where it equals .south, print "Watch out for penguins"."

```
...and so on.
```

As described in <u>Control Flow</u>, a switch statement must be exhaustive when considering an enumeration's cases. If the case for .west is omitted, this code doesn't compile, because it doesn't consider the complete list of <code>CompassPoint</code> cases. Requiring exhaustiveness ensures that enumeration cases aren't accidentally omitted.

When it isn't appropriate to provide a case for every enumeration case, you can provide a default case to cover any cases that aren't addressed explicitly:

```
1 let somePlanet = Planet.earth
2 switch somePlanet {
3  case .earth:
    print("Mostly harmless")
5  default:
    print("Not a safe place for humans")
7  }
8  // Prints "Mostly harmless"
```

Iterating over Enumeration Cases

For some enumerations, it's useful to have a collection of all of that enumeration's cases. You enable this by writing : CaseIterable after the enumeration's name. Swift exposes a collection of all the cases as an allCases property of the enumeration type. Here's an example:

```
1 enum Beverage: CaseIterable {
2    case coffee, tea, juice
```

```
3  }
4  let numberOfChoices = Beverage.allCases.count
5  print("\(numberOfChoices\) beverages available")
6  // Prints "3 beverages available"
```

In the example above, you write Beverage.allCases to access a collection that contains all of the cases of the Beverage enumeration. You can use allCases like any other collection—the collection's elements are instances of the enumeration type, so in this case they're Beverage values. The example above counts how many cases there are, and the example below uses a for loop to iterate over all the cases.

```
for beverage in Beverage.allCases {
   print(beverage)
}

// coffee
// tea
// juice
```

The syntax used in the examples above marks the enumeration as conforming to the <u>CaseIterable</u> protocol. For information about protocols, see <u>Protocols</u>.

Associated Values

The examples in the previous section show how the cases of an enumeration are a defined (and typed) value in their own right. You can set a constant or variable to Planet.earth, and check for this value later. However, it's sometimes useful to be able to store values of other types alongside these case values. This additional information is called an *associated value*, and it varies each time you use that case as a value in your code.

You can define Swift enumerations to store associated values of any given type,

and the value types can be different for each case of the enumeration if needed. Enumerations similar to these are known as *discriminated unions*, *tagged unions*, or *variants* in other programming languages.

For example, suppose an inventory tracking system needs to track products by two different types of barcode. Some products are labeled with 1D barcodes in UPC format, which uses the numbers 0 to 9. Each barcode has a number system digit, followed by five manufacturer code digits and five product code digits. These are followed by a check digit to verify that the code has been scanned correctly:



Other products are labeled with 2D barcodes in QR code format, which can use any ISO 8859-1 character and can encode a string up to 2,953 characters long:



It's convenient for an inventory tracking system to store UPC barcodes as a tuple of four integers, and QR code barcodes as a string of any length.

In Swift, an enumeration to define product barcodes of either type might look like this:

```
1 enum Barcode {
2    case upc(Int, Int, Int, Int)
3    case qrCode(String)
```

This can be read as:

"Define an enumeration type called Barcode, which can take either a value of upc with an associated value of type (Int, Int, Int, Int), or a value of grcode with an associated value of type String."

This definition doesn't provide any actual Int or String values—it just defines the *type* of associated values that Barcode constants and variables can store when they are equal to Barcode.upc or Barcode.qrCode.

You can then create new barcodes using either type:

```
var productBarcode = Barcode.upc(8, 85909, 51226, 3)
```

This example creates a new variable called productBarcode and assigns it a value of Barcode.upc with an associated tuple value of (8, 85909, 51226, 3).

You can assign the same product a different type of barcode:

```
productBarcode = .qrCode("ABCDEFGHIJKLMNOP")
```

At this point, the original Barcode upc and its integer values are replaced by the new Barcode and its string value. Constants and variables of type Barcode can store either a .upc or a .grcode (together with their associated values), but they can store only one of them at any given time.

You can check the different barcode types using a switch statement, similar to the example in Matching Enumeration Values with a Switch Statement. This time, however, the associated values are extracted as part of the switch statement. You extract each associated value as a constant (with the let prefix) or a variable (with the var prefix) for use within the switch case's body:

```
switch productBarcode {
1
   case .upc(let numberSystem, let manufacturer, let product, let
2
```

If all of the associated values for an enumeration case are extracted as constants, or if all are extracted as variables, you can place a single var or let annotation before the case name, for brevity:

```
switch productBarcode {
case let .upc(numberSystem, manufacturer, product, check):
    print("UPC : \(numberSystem), \(manufacturer), \(product),
        \((check).")

case let .qrCode(productCode):
    print("QR code: \((productCode)."))

}

// Prints "QR code: ABCDEFGHIJKLMNOP."
```

Raw Values

The barcode example in <u>Associated Values</u> shows how cases of an enumeration can declare that they store associated values of different types. As an alternative to associated values, enumeration cases can come prepopulated with default values (called *raw values*), which are all of the same type.

Here's an example that stores raw ASCII values alongside named enumeration cases:

```
enum ASCIIControlCharacter: Character {
    case tab = "\t"
    case lineFeed = "\n"
    case carriageReturn = "\r"
}
```

Here, the raw values for an enumeration called ASCIIControlCharacter are defined to be of type Character, and are set to some of the more common ASCII control characters. Character values are described in <u>Strings and Characters</u>.

Raw values can be strings, characters, or any of the integer or floating-point number types. Each raw value must be unique within its enumeration declaration.

NOTE

Raw values are *not* the same as associated values. Raw values are set to prepopulated values when you first define the enumeration in your code, like the three ASCII codes above. The raw value for a particular enumeration case is always the same. Associated values are set when you create a new constant or variable based on one of the enumeration's cases, and can be different each time you do so.

Implicitly Assigned Raw Values

When you're working with enumerations that store integer or string raw values, you don't have to explicitly assign a raw value for each case. When you don't, Swift automatically assigns the values for you.

For example, when integers are used for raw values, the implicit value for each case is one more than the previous case. If the first case doesn't have a value set, its value is 0.

The enumeration below is a refinement of the earlier Planet enumeration, with integer raw values to represent each planet's order from the sun:

```
1 enum Planet: Int {
```

```
case mercury = 1, venus, earth, mars, jupiter, saturn,
uranus, neptune
}
```

In the example above, Planet.mercury has an explicit raw value of 1, Planet.venus has an implicit raw value of 2, and so on.

When strings are used for raw values, the implicit value for each case is the text of that case's name.

The enumeration below is a refinement of the earlier CompassPoint enumeration, with string raw values to represent each direction's name:

```
1 enum CompassPoint: String {
2    case north, south, east, west
3 }
```

In the example above, CompassPoint.south has an implicit raw value of "south", and so on.

You access the raw value of an enumeration case with its rawValue property:

```
1 let earthsOrder = Planet.earth.rawValue
2 // earthsOrder is 3
3
4 let sunsetDirection = CompassPoint.west.rawValue
5 // sunsetDirection is "west"
```

Initializing from a Raw Value

If you define an enumeration with a raw-value type, the enumeration automatically receives an initializer that takes a value of the raw value's type (as a parameter called rawValue) and returns either an enumeration case or nil. You can use this initializer to try to create a new instance of the enumeration.

This example identifies Uranus from its raw value of 7:

```
1 let possiblePlanet = Planet(rawValue: 7)
2 // possiblePlanet is of type Planet? and equals Planet.uranus
```

Not all possible Int values will find a matching planet, however. Because of this, the raw value initializer always returns an *optional* enumeration case. In the example above, possiblePlanet is of type Planet?, or "optional Planet."

```
NOTE
```

The raw value initializer is a failable initializer, because not every raw value will return an enumeration case. For more information, see <u>Failable Initializers</u>.

If you try to find a planet with a position of 11, the optional Planet value returned by the raw value initializer will be nil:

```
1
    let positionToFind = 11
2
    if let somePlanet = Planet(rawValue: positionToFind) {
        switch somePlanet {
3
        case earth:
4
             print("Mostly harmless")
5
        default:
6
             print("Not a safe place for humans")
        }
    } else {
10
         print("There isn't a planet at position \
           (positionToFind)")
11
    }
    // Prints "There isn't a planet at position 11"
12
```

This example uses optional binding to try to access a planet with a raw value of 11. The statement if let somePlanet = Planet(rawValue: 11) creates an optional Planet, and sets somePlanet to the value of that optional Planet if it can

be retrieved. In this case, it isn't possible to retrieve a planet with a position of 11, and so the else branch is executed instead.

Recursive Enumerations

A *recursive enumeration* is an enumeration that has another instance of the enumeration as the associated value for one or more of the enumeration cases. You indicate that an enumeration case is recursive by writing indirect before it, which tells the compiler to insert the necessary layer of indirection.

For example, here is an enumeration that stores simple arithmetic expressions:

```
1  enum ArithmeticExpression {
2    case number(Int)
3    indirect case addition(ArithmeticExpression,
        ArithmeticExpression)
4    indirect case multiplication(ArithmeticExpression,
        ArithmeticExpression)
5  }
```

You can also write indirect before the beginning of the enumeration to enable indirection for all of the enumeration's cases that have an associated value:

This enumeration can store three kinds of arithmetic expressions: a plain number, the addition of two expressions, and the multiplication of two expressions. The addition and multiplication cases have associated values that are also arithmetic expressions—these associated values make it possible to nest expressions. For example, the expression (5 + 4) * 2 has a number on the right-hand side of the multiplication and another expression on the left-hand side of the multiplication. Because the data is nested, the enumeration used to store the data also needs to support nesting—this means the enumeration needs to be recursive. The code below shows the ArithmeticExpression recursive enumeration being created for (5 + 4) * 2:

A recursive function is a straightforward way to work with data that has a recursive structure. For example, here's a function that evaluates an arithmetic expression:

```
func evaluate(_ expression: ArithmeticExpression) -> Int {
1
         switch expression {
2
         case let .number(value):
3
             return value
5
         case let .addition(left, right):
             return evaluate(left) + evaluate(right)
6
         case let .multiplication(left, right):
7
             return evaluate(left) * evaluate(right)
8
9
         }
10
    }
11
    print(evaluate(product))
12
    // Prints "18"
13
```

This function evaluates a plain number by simply returning the associated value. It evaluates an addition or multiplication by evaluating the expression on the left-hand side, evaluating the expression on the right-hand side, and then adding them or multiplying them.

Structures and Classes

Structures and *classes* are general-purpose, flexible constructs that become the building blocks of your program's code. You define properties and methods to add functionality to your structures and classes using the same syntax you use to define constants, variables, and functions.

Unlike other programming languages, Swift doesn't require you to create separate interface and implementation files for custom structures and classes. In Swift, you define a structure or class in a single file, and the external interface to that class or structure is automatically made available for other code to use.

NOTE

An instance of a class is traditionally known as an *object*. However, Swift structures and classes are much closer in functionality than in other languages, and much of this chapter describes functionality that applies to instances of *either* a class or a structure type. Because of this, the more general term *instance* is used.

Comparing Structures and Classes

Structures and classes in Swift have many things in common. Both can:

- Define properties to store values
- Define methods to provide functionality
- Define subscripts to provide access to their values using subscript syntax
- Define initializers to set up their initial state
- Be extended to expand their functionality beyond a default implementation
- Conform to protocols to provide standard functionality of a certain kind

For more information, see <u>Properties</u>, <u>Methods</u>, <u>Subscripts</u>, <u>Initialization</u>, <u>Extensions</u>, and <u>Protocols</u>.

Classes have additional capabilities that structures don't have:

- Inheritance enables one class to inherit the characteristics of another.
- Type casting enables you to check and interpret the type of a class instance at runtime.
- Deinitializers enable an instance of a class to free up any resources it has assigned.
- Reference counting allows more than one reference to a class instance.

For more information, see <u>Inheritance</u>, <u>Type Casting</u>, <u>Deinitialization</u>, and <u>Automatic Reference Counting</u>.

The additional capabilities that classes support come at the cost of increased complexity. As a general guideline, prefer structures because they're easier to reason about, and use classes when they're appropriate or necessary. In practice, this means most of the custom data types you define will be structures and enumerations. For a more detailed comparison, see Choosing Between Structures and Classes.

Definition Syntax

Structures and classes have a similar definition syntax. You introduce structures with the struct keyword and classes with the class keyword. Both place their entire definition within a pair of braces:

```
struct SomeStructure {
    // structure definition goes here
}

class SomeClass {
    // class definition goes here
```

```
6 }
```

NOTE

Whenever you define a new structure or class, you define a new Swift type. Give types UpperCamelCase names (such as SomeStructure and SomeClass here) to match the capitalization of standard Swift types (such as String, Int, and Bool). Give properties and methods lowerCamelCase names (such as frameRate and incrementCount) to differentiate them from type names.

Here's an example of a structure definition and a class definition:

```
1
    struct Resolution {
2
         var width = 0
         var height = 0
3
    }
4
5
    class VideoMode {
         var resolution = Resolution()
6
         var interlaced = false
7
         var frameRate = 0.0
8
         var name: String?
10
     }
```

The example above defines a new structure called Resolution, to describe a pixel-based display resolution. This structure has two stored properties called width and height. Stored properties are constants or variables that are bundled up and stored as part of the structure or class. These two properties are inferred to be of type Int by setting them to an initial integer value of 0.

The example above also defines a new class called VideoMode, to describe a specific video mode for video display. This class has four variable stored properties. The first, resolution, is initialized with a new Resolution structure instance, which infers a property type of Resolution. For the other three properties, new VideoMode instances will be initialized with an interlaced setting of false (meaning "noninterlaced video"), a playback frame rate of 0.0, and an optional String value called name. The name property is automatically

given a default value of nil, or "no name value", because it's of an optional type.

Structure and Class Instances

The Resolution structure definition and the VideoMode class definition only describe what a Resolution or VideoMode will look like. They themselves don't describe a specific resolution or video mode. To do that, you need to create an instance of the structure or class.

The syntax for creating instances is very similar for both structures and classes:

```
1 let someResolution = Resolution()
2 let someVideoMode = VideoMode()
```

Structures and classes both use initializer syntax for new instances. The simplest form of initializer syntax uses the type name of the class or structure followed by empty parentheses, such as Resolution() or VideoMode(). This creates a new instance of the class or structure, with any properties initialized to their default values. Class and structure initialization is described in more detail in Initialization.

Accessing Properties

You can access the properties of an instance using *dot syntax*. In dot syntax, you write the property name immediately after the instance name, separated by a period (.), without any spaces:

```
print("The width of someResolution is \((someResolution.width)")
// Prints "The width of someResolution is 0"
```

In this example, someResolution.width refers to the width property of someResolution, and returns its default initial value of 0.

You can drill down into subproperties, such as the width property in the resolution property of a VideoMode:

You can also use dot syntax to assign a new value to a variable property:

Memberwise Initializers for Structure Types

All structures have an automatically generated *memberwise initializer*, which you can use to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name:

```
let vga = Resolution(width: 640, height: 480)
```

Unlike structures, class instances don't receive a default memberwise initializer. Initializers are described in more detail in <u>Initialization</u>.

Structures and Enumerations Are Value Types

A *value type* is a type whose value is *copied* when it's assigned to a variable or constant, or when it's passed to a function.

You've actually been using value types extensively throughout the previous chapters. In fact, all of the basic types in Swift—integers, floating-point numbers, Booleans, strings, arrays and dictionaries—are value types, and are implemented as structures behind the scenes.

All structures and enumerations are value types in Swift. This means that any structure and enumeration instances you create—and any value types they have as properties—are always copied when they are passed around in your code.

```
NOTE
```

Collections defined by the standard library like arrays, dictionaries, and strings use an optimization to reduce the performance cost of copying. Instead of making a copy immediately, these collections share the memory where the elements are stored between the original instance and any copies. If one of the copies of the collection is modified, the elements are copied just before the modification. The behavior you see in your code is always as if a copy took place immediately.

Consider this example, which uses the Resolution structure from the previous example:

```
1 let hd = Resolution(width: 1920, height: 1080)
2 var cinema = hd
```

This example declares a constant called hd and sets it to a Resolution instance initialized with the width and height of full HD video (1920 pixels wide by 1080 pixels high).

It then declares a variable called cinema and sets it to the current value of hd. Because Resolution is a structure, a *copy* of the existing instance is made, and this new copy is assigned to cinema. Even though hd and cinema now have the same width and height, they are two completely different instances behind the scenes.

Next, the width property of cinema is amended to be the width of the slightly wider 2K standard used for digital cinema projection (2048 pixels wide and 1080 pixels high):

```
cinema.width = 2048
```

Checking the width property of cinema shows that it has indeed changed to be 2048:

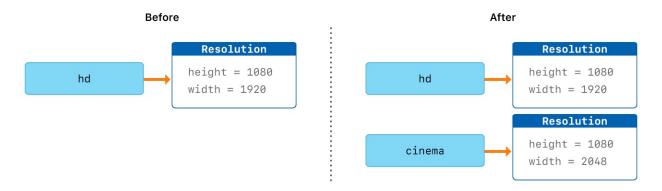
```
print("cinema is now \(cinema.width) pixels wide")
```

```
2 // Prints "cinema is now 2048 pixels wide"
```

However, the width property of the original hd instance still has the old value of 1920:

```
print("hd is still \(hd.width) pixels wide")
// Prints "hd is still 1920 pixels wide"
```

When cinema was given the current value of hd, the *values* stored in hd were copied into the new cinema instance. The end result was two completely separate instances that contained the same numeric values. However, because they are separate instances, setting the width of cinema to 2048 doesn't affect the width stored in hd, as shown in the figure below:



The same behavior applies to enumerations:

```
enum CompassPoint {
1
2
        case north, south, east, west
        mutating func turnNorth() {
3
            self = .north
4
        }
5
6
   }
7
   var currentDirection = CompassPoint.west
8
   let rememberedDirection = currentDirection
   currentDirection.turnNorth()
9
```

```
print("The current direction is \(currentDirection)")
print("The remembered direction is \(rememberedDirection)")
// Prints "The current direction is north"
// Prints "The remembered direction is west"
```

When rememberedDirection is assigned the value of currentDirection, it's actually set to a copy of that value. Changing the value of currentDirection thereafter doesn't affect the copy of the original value that was stored in rememberedDirection.

Classes Are Reference Types

Unlike value types, *reference types* are *not* copied when they are assigned to a variable or constant, or when they are passed to a function. Rather than a copy, a reference to the same existing instance is used.

Here's an example, using the VideoMode class defined above:

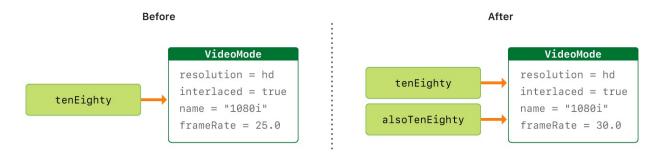
```
1 let tenEighty = VideoMode()
2 tenEighty.resolution = hd
3 tenEighty.interlaced = true
4 tenEighty.name = "1080i"
5 tenEighty.frameRate = 25.0
```

This example declares a new constant called tenEighty and sets it to refer to a new instance of the VideoMode class. The video mode is assigned a copy of the HD resolution of 1920 by 1080 from before. It's set to be interlaced, its name is set to "1080i", and its frame rate is set to 25.0 frames per second.

Next, tenEighty is assigned to a new constant, called alsoTenEighty, and the frame rate of alsoTenEighty is modified:

```
1 let alsoTenEighty = tenEighty
2 alsoTenEighty.frameRate = 30.0
```

Because classes are reference types, tenEighty and alsoTenEighty actually both refer to the *same* VideoMode instance. Effectively, they are just two different names for the same single instance, as shown in the figure below:



Checking the frameRate property of tenEighty shows that it correctly reports the new frame rate of 30.0 from the underlying VideoMode instance:

This example also shows how reference types can be harder to reason about. If tenEighty and alsoTenEighty were far apart in your program's code, it could be difficult to find all the ways that the video mode is changed. Wherever you use tenEighty, you also have to think about the code that uses alsoTenEighty, and vice versa. In contrast, value types are easier to reason about because all of the code that interacts with the same value is close together in your source files.

Note that tenEighty and alsoTenEighty are declared as *constants*, rather than variables. However, you can still change tenEighty.frameRate and alsoTenEighty.frameRate because the values of the tenEighty and alsoTenEighty constants themselves don't actually change. tenEighty and alsoTenEighty themselves don't "store" the VideoMode instance—instead, they both *refer* to a VideoMode instance behind the scenes. It's the frameRate property of the underlying VideoMode that is changed, not the values of the constant references to that VideoMode.

Identity Operators

Because classes are reference types, it's possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same isn't true for structures and enumerations, because they are always copied when they are assigned to a constant or variable, or passed to a function.)

It can sometimes be useful to find out whether two constants or variables refer to exactly the same instance of a class. To enable this, Swift provides two identity operators:

- Identical to (===)
- Not identical to (!==)

Use these operators to check whether two constants or variables refer to the same single instance:

Note that *identical to* (represented by three equals signs, or ===) doesn't mean the same thing as *equal to* (represented by two equals signs, or ==). *Identical to* means that two constants or variables of class type refer to exactly the same class instance. *Equal to* means that two instances are considered equal or equivalent in value, for some appropriate meaning of *equal*, as defined by the type's designer.

When you define your own custom structures and classes, it's your responsibility to decide what qualifies as two instances being equal. The process of defining your own implementations of the == and != operators is described in <u>Equivalence Operators</u>.

Pointers

If you have experience with C, C++, or Objective-C, you may know that these languages use *pointers* to refer to addresses in memory. A Swift constant or variable that refers to an instance of some reference type is similar to a pointer in C, but isn't a direct pointer to an address in memory, and doesn't require you to write an asterisk (*) to indicate that you are creating a reference. Instead, these references are defined like any other constant or variable in Swift. The standard library provides pointer and buffer types that you can use if you need to interact with pointers directly—see Manual Memory Management.

Properties

Properties associate values with a particular class, structure, or enumeration. Stored properties store constant and variable values as part of an instance, whereas computed properties calculate (rather than store) a value. Computed properties are provided by classes, structures, and enumerations. Stored properties are provided only by classes and structures.

Stored and computed properties are usually associated with instances of a particular type. However, properties can also be associated with the type itself. Such properties are known as type properties.

In addition, you can define property observers to monitor changes in a property's value, which you can respond to with custom actions. Property observers can be added to stored properties you define yourself, and also to properties that a subclass inherits from its superclass.

You can also use a property wrapper to reuse code in the getter and setter of multiple properties.

Stored Properties

In its simplest form, a stored property is a constant or variable that is stored as part of an instance of a particular class or structure. Stored properties can be either *variable stored properties* (introduced by the var keyword) or *constant stored properties* (introduced by the let keyword).

You can provide a default value for a stored property as part of its definition, as described in <u>Default Property Values</u>. You can also set and modify the initial value for a stored property during initialization. This is true even for constant stored properties, as described in <u>Assigning Constant Properties During Initialization</u>.

The example below defines a structure called ${\tt FixedLengthRange}$, which describes

a range of integers whose range length cannot be changed after it is created:

```
struct FixedLengthRange {
   var firstValue: Int
   let length: Int
}
var rangeOfThreeItems = FixedLengthRange(firstValue: 0, length:
        3)
// the range represents integer values 0, 1, and 2
rangeOfThreeItems.firstValue = 6
// the range now represents integer values 6, 7, and 8
```

Instances of FixedLengthRange have a variable stored property called firstValue and a constant stored property called length. In the example above, length is initialized when the new range is created and cannot be changed thereafter, because it is a constant property.

Stored Properties of Constant Structure Instances

If you create an instance of a structure and assign that instance to a constant, you cannot modify the instance's properties, even if they were declared as variable properties:

Because rangeOfFourItems is declared as a constant (with the let keyword), it is not possible to change its firstValue property, even though firstValue is a

variable property.

This behavior is due to structures being *value types*. When an instance of a value type is marked as a constant, so are all of its properties.

The same is not true for classes, which are *reference types*. If you assign an instance of a reference type to a constant, you can still change that instance's variable properties.

Lazy Stored Properties

A *lazy stored property* is a property whose initial value is not calculated until the first time it is used. You indicate a lazy stored property by writing the lazy modifier before its declaration.

```
NOTE
```

You must always declare a lazy property as a variable (with the var keyword), because its initial value might not be retrieved until after instance initialization completes. Constant properties must always have a value *before* initialization completes, and therefore cannot be declared as lazy.

Lazy properties are useful when the initial value for a property is dependent on outside factors whose values are not known until after an instance's initialization is complete. Lazy properties are also useful when the initial value for a property requires complex or computationally expensive setup that should not be performed unless or until it is needed.

The example below uses a lazy stored property to avoid unnecessary initialization of a complex class. This example defines two classes called <code>DataImporter</code> and <code>DataManager</code>, neither of which is shown in full:

```
class DataImporter {
/*
DataImporter is a class to import data from an external file.
The class is assumed to take a nontrivial amount of time to
```

```
initialize.
         */
5
        var filename = "data.txt"
6
7
        // the DataImporter class would provide data importing
          functionality here
    }
8
9
    class DataManager {
10
11
         lazy var importer = DataImporter()
        var data = [String]()
12
         // the DataManager class would provide data management
13
          functionality here
14
    }
15
16
    let manager = DataManager()
    manager.data.append("Some data")
17
18
    manager.data.append("Some more data")
    // the DataImporter instance for the importer property has not
19
          yet been created
```

The DataManager class has a stored property called data, which is initialized with a new, empty array of String values. Although the rest of its functionality is not shown, the purpose of this DataManager class is to manage and provide access to this array of String data.

Part of the functionality of the DataManager class is the ability to import data from a file. This functionality is provided by the DataImporter class, which is assumed to take a nontrivial amount of time to initialize. This might be because a DataImporter instance needs to open a file and read its contents into memory when the DataImporter instance is initialized.

It is possible for a DataManager instance to manage its data without ever importing data from a file, so there is no need to create a new DataImporter

instance when the DataManager itself is created. Instead, it makes more sense to create the DataImporter instance if and when it is first used.

Because it is marked with the lazy modifier, the DataImporter instance for the importer property is only created when the importer property is first accessed, such as when its filename property is queried:

```
print(manager.importer.filename)

// the DataImporter instance for the importer property has now
been created

// Prints "data.txt"
```

NOTE

If a property marked with the lazy modifier is accessed by multiple threads simultaneously and the property has not yet been initialized, there is no guarantee that the property will be initialized only once.

Stored Properties and Instance Variables

If you have experience with Objective-C, you may know that it provides *two* ways to store values and references as part of a class instance. In addition to properties, you can use instance variables as a backing store for the values stored in a property.

Swift unifies these concepts into a single property declaration. A Swift property does not have a corresponding instance variable, and the backing store for a property is not accessed directly. This approach avoids confusion about how the value is accessed in different contexts and simplifies the property's declaration into a single, definitive statement. All information about the property—including its name, type, and memory management characteristics—is defined in a single location as part of the type's definition.

Computed Properties

In addition to stored properties, classes, structures, and enumerations can define *computed properties*, which do not actually store a value. Instead, they provide a getter and an optional setter to retrieve and set other properties and values indirectly.

```
1
    struct Point {
2
         var \times = 0.0, y = 0.0
3
    }
    struct Size {
4
         var width = 0.0, height = 0.0
5
    }
6
7
    struct Rect {
         var origin = Point()
8
9
         var size = Size()
         var center: Point {
10
11
             get {
12
                 let centerX = origin.x + (size.width / 2)
13
                 let centerY = origin.y + (size.height / 2)
                 return Point(x: centerX, y: centerY)
14
             }
15
             set(newCenter) {
16
17
                 origin.x = newCenter.x - (size.width / 2)
                 origin.y = newCenter.y - (size.height / 2)
18
19
             }
         }
20
21
    }
    var square = Rect(origin: Point(x: 0.0, y: 0.0),
22
                        size: Size(width: 10.0, height: 10.0))
23
24
    let initialSquareCenter = square.center
25
     square.center = Point(x: 15.0, y: 15.0)
26
     print("square.origin is now at (\((square.origin.x), \)
```

```
(square.origin.y))")
27 // Prints "square.origin is now at (10.0, 10.0)"
```

This example defines three structures for working with geometric shapes:

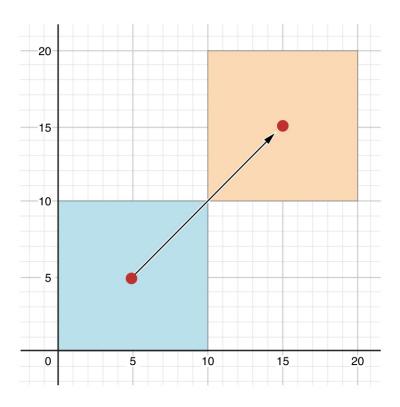
- Point encapsulates the x- and y-coordinate of a point.
- Size encapsulates a width and a height.
- Rect defines a rectangle by an origin point and a size.

The Rect structure also provides a computed property called center. The current center position of a Rect can always be determined from its origin and size, and so you don't need to store the center point as an explicit Point value. Instead, Rect defines a custom getter and setter for a computed variable called center, to enable you to work with the rectangle's center as if it were a real stored property.

The example above creates a new Rect variable called square. The square variable is initialized with an origin point of (0, 0), and a width and height of 10. This square is represented by the blue square in the diagram below.

The square variable's center property is then accessed through dot syntax (square.center), which causes the getter for center to be called, to retrieve the current property value. Rather than returning an existing value, the getter actually calculates and returns a new Point to represent the center of the square. As can be seen above, the getter correctly returns a center point of (5, 5).

The center property is then set to a new value of (15, 15), which moves the square up and to the right, to the new position shown by the orange square in the diagram below. Setting the center property calls the setter for center, which modifies the \times and y values of the stored origin property, and moves the square to its new position.



Shorthand Setter Declaration

If a computed property's setter doesn't define a name for the new value to be set, a default name of newValue is used. Here's an alternative version of the Rect structure that takes advantage of this shorthand notation:

```
1
    struct AlternativeRect {
2
        var origin = Point()
3
        var size = Size()
        var center: Point {
4
             get {
5
                 let centerX = origin.x + (size.width / 2)
6
                 let centerY = origin.y + (size.height / 2)
7
                 return Point(x: centerX, y: centerY)
8
9
             }
10
             set {
```

```
origin.x = newValue.x - (size.width / 2)
origin.y = newValue.y - (size.height / 2)

}

}

}

}
```

Shorthand Getter Declaration

If the entire body of a getter is a single expression, the getter implicitly returns that expression. Here's an another version of the Rect structure that takes advantage of this shorthand notation and the shorthand notation for setters:

```
struct CompactRect {
1
2
         var origin = Point()
         var size = Size()
3
         var center: Point {
4
             get {
5
                 Point(x: origin.x + (size.width / 2),
6
                       y: origin.y + (size.height / 2))
7
             }
8
             set {
9
                 origin.x = newValue.x - (size.width / 2)
10
11
                 origin.y = newValue.y - (size.height / 2)
12
             }
         }
13
14
     }
```

Omitting the return from a getter follows the same rules as omitting return from a function, as described in <u>Functions With an Implicit Return</u>.

Read-Only Computed Properties

A computed property with a getter but no setter is known as a *read-only computed property*. A read-only computed property always returns a value, and can be accessed through dot syntax, but cannot be set to a different value.

```
NOTE
```

You must declare computed properties—including read-only computed properties—as variable properties with the <code>var</code> keyword, because their value is not fixed. The <code>let</code> keyword is only used for constant properties, to indicate that their values cannot be changed once they are set as part of instance initialization.

You can simplify the declaration of a read-only computed property by removing the get keyword and its braces:

```
struct Cuboid {
1
        var width = 0.0, height = 0.0, depth = 0.0
2
        var volume: Double {
3
            return width * height * depth
4
        }
5
6
   }
   let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth:
7
          2.0)
   print("the volume of fourByFiveByTwo is \
          (fourByFiveByTwo.volume)")
   // Prints "the volume of fourByFiveByTwo is 40.0"
9
```

This example defines a new structure called Cuboid, which represents a 3D rectangular box with width, height, and depth properties. This structure also has a read-only computed property called volume, which calculates and returns the current volume of the cuboid. It doesn't make sense for volume to be settable, because it would be ambiguous as to which values of width, height, and depth should be used for a particular volume value. Nonetheless, it is useful for a Cuboid to provide a read-only computed property to enable external users to discover its current calculated volume.

Property Observers

Property observers observe and respond to changes in a property's value. Property observers are called every time a property's value is set, even if the new value is the same as the property's current value.

You can add property observers to any stored properties you define, except for lazy stored properties. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass. You don't need to define property observers for nonoverridden computed properties, because you can observe and respond to changes to their value in the computed property's setter. Property overriding is described in Overriding.

You have the option to define either or both of these observers on a property:

- willset is called just before the value is stored.
- didSet is called immediately after the new value is stored.

If you implement a willset observer, it's passed the new property value as a constant parameter. You can specify a name for this parameter as part of your willset implementation. If you don't write the parameter name and parentheses within your implementation, the parameter is made available with a default parameter name of newValue.

Similarly, if you implement a didSet observer, it's passed a constant parameter containing the old property value. You can name the parameter or use the default parameter name of oldValue. If you assign a value to a property within its own didSet observer, the new value that you assign replaces the one that was just set.

NOTE

The willset and didset observers of superclass properties are called when a property is set in a subclass initializer, after the superclass initializer has been called. They are not called while a class is setting its own properties, before the superclass initializer has been called.

For more information about initializer delegation, see <u>Initializer Delegation for Value Types</u> and <u>Initializer Delegation for Class Types</u>.

Here's an example of willset and didSet in action. The example below defines a new class called StepCounter, which tracks the total number of steps that a person takes while walking. This class might be used with input data from a pedometer or other step counter to keep track of a person's exercise during their daily routine.

```
1
    class StepCounter {
 2
         var totalSteps: Int = 0 {
             willSet(newTotalSteps) {
 3
                 print("About to set totalSteps to \
 4
           (newTotalSteps)")
             }
 5
             didSet {
 6
 7
                 if totalSteps > oldValue {
                     print("Added \(totalSteps - oldValue) steps")
 8
                 }
 9
             }
10
         }
11
12
     }
13
     let stepCounter = StepCounter()
     stepCounter.totalSteps = 200
14
15
     // About to set totalSteps to 200
16
    // Added 200 steps
     stepCounter.totalSteps = 360
17
     // About to set totalSteps to 360
18
    // Added 160 steps
19
20
     stepCounter.totalSteps = 896
21
     // About to set totalSteps to 896
22
     // Added 536 steps
```

The StepCounter class declares a totalSteps property of type Int. This is a

stored property with willSet and didSet observers.

The willset and didset observers for totalsteps are called whenever the property is assigned a new value. This is true even if the new value is the same as the current value.

This example's willset observer uses a custom parameter name of newTotalSteps for the upcoming new value. In this example, it simply prints out the value that is about to be set.

The didSet observer is called after the value of totalSteps is updated. It compares the new value of totalSteps against the old value. If the total number of steps has increased, a message is printed to indicate how many new steps have been taken. The didSet observer does not provide a custom parameter name for the old value, and the default name of oldValue is used instead.

NOTE

If you pass a property that has observers to a function as an in-out parameter, the willset and didSet observers are always called. This is because of the copy-in copy-out memory model for in-out parameters: The value is always written back to the property at the end of the function. For a detailed discussion of the behavior of in-out parameters, see In-Out Parameters.

Property Wrappers

A property wrapper adds a layer of separation between code that manages how a property is stored and the code that defines a property. For example, if you have properties that provide thread-safety checks or store their underlying data in a database, you have to write that code on every property. When you use a property wrapper, you write the management code once when you define the wrapper, and then reuse that management code by applying it to multiple properties.

To define a property wrapper, you make a structure, enumeration, or class that defines a wrappedValue property. In the code below, the TwelveOrLess structure ensures that the value it wraps always contains a number less than or equal to 12.

If you ask it to store a larger number, it stores 12 instead.

```
1
   @propertyWrapper
   struct TwelveOrLess {
2
        private var number = 0
3
        var wrappedValue: Int {
4
            get { return number }
5
6
            set { number = min(newValue, 12) }
7
        }
8
   }
```

The setter ensures that new values are less than 12, and the getter returns the stored value.

NOTE

The declaration for number in the example above marks the variable as private, which ensures number is used only in the implementation of TwelveOrLess. Code that's written anywhere else accesses the value using the getter and setter for wrappedValue, and can't use number directly. For information about private, see Access Control.

You apply a wrapper to a property by writing the wrapper's name before the property as an attribute. Here's a structure that stores a small rectangle, using the same (rather arbitrary) definition of "small" that's implemented by the TwelveOrLess property wrapper:

```
struct SmallRectangle {
    @TwelveOrLess var height: Int
    @TwelveOrLess var width: Int
}

var rectangle = SmallRectangle()
print(rectangle.height)
// Prints "0"
```

```
9
10 rectangle.height = 10
11 print(rectangle.height)
12 // Prints "10"
13
14 rectangle.height = 24
15 print(rectangle.height)
16 // Prints "12"
```

The height and width properties get their initial values from the definition of TwelveOrLess, which sets TwelveOrLess.number to zero. Storing the number 10 into rectangle.height succeeds because it's a small number. Trying to store 24 actually stores a value of 12 instead, because 24 is too large for the property setter's rule.

When you apply a wrapper to a property, the compiler synthesizes code that provides storage for the wrapper and code that provides access to the property through the wrapper. (The property wrapper is responsible for storing the wrapped value, so there's no synthesized code for that.) You could write code that uses the behavior of a property wrapper, without taking advantage of the special attribute syntax. For example, here's a version of SmallRectangle from the previous code listing that wraps its properties in the TwelveOrLess structure explicitly, instead of writing @TwelveOrLess as an attribute:

```
1
    struct SmallRectangle {
2
        private var _height = TwelveOrLess()
        private var _width = TwelveOrLess()
3
4
        var height: Int {
            get { return _height.wrappedValue }
5
            set { _height.wrappedValue = newValue }
6
7
        }
8
        var width: Int {
            get { return _width.wrappedValue }
9
```

```
set { _width.wrappedValue = newValue }

11  }

12 }
```

The _height and _width properties store an instance of the property wrapper, TwelveOrLess. The getter and setter for height and width wrap access to the wrappedValue property.

Setting Initial Values for Wrapped Properties

The code in the examples above sets the initial value for the wrapped property by giving number an initial value in the definition of TwelveOrLess. Code that uses this property wrapper, can't specify a different initial value for a property that's wrapped by TwelveOrLess—for example, the definition of SmallRectangle can't give height or width initial values. To support setting an initial value or other customization, the property wrapper needs to add an initializer. Here's an expanded version of TwelveOrLess called SmallNumber that defines initializers that set the wrapped and maximum value:

```
1
    @propertyWrapper
2
    struct SmallNumber {
3
         private var maximum: Int
         private var number: Int
4
5
6
         var wrappedValue: Int {
             get { return number }
7
             set { number = min(newValue, maximum) }
8
         }
9
10
         init() {
11
             maximum = 12
12
             number = 0
13
```

```
14
         }
         init(wrappedValue: Int) {
15
16
             maximum = 12
             number = min(wrappedValue, maximum)
17
         }
18
         init(wrappedValue: Int, maximum: Int) {
19
             self.maximum = maximum
20
             number = min(wrappedValue, maximum)
21
22
         }
23
    }
```

The definition of SmallNumber includes three initializers—init(), init(wrappedValue:), and init(wrappedValue:maximum:)—which the examples below use to set the wrapped value and the maximum value. For information about initialization and initializer syntax, see Initialization.

When you apply a wrapper to a property and you don't specify an initial value, Swift uses the <code>init()</code> initializer to set up the wrapper. For example:

```
struct ZeroRectangle {
    @SmallNumber var height: Int
    @SmallNumber var width: Int
}

var zeroRectangle = ZeroRectangle()
print(zeroRectangle.height, zeroRectangle.width)
// Prints "0 0"
```

The instances of SmallNumber that wrap height and width are created by calling SmallNumber(). The code inside that initializer sets the initial wrapped value and the initial maximum value, using the default values of zero and 12. The property wrapper still provides all of the initial values, like the earlier example that used TwelveOrLess in SmallRectangle. Unlike that example, SmallNumber also

supports writing those initial values as part of declaring the property.

When you specify an initial value for the property, Swift uses the init(wrappedValue:) initializer to set up the wrapper. For example:

```
struct UnitRectangle {
    @SmallNumber var height: Int = 1
    @SmallNumber var width: Int = 1
}

var unitRectangle = UnitRectangle()
print(unitRectangle.height, unitRectangle.width)
// Prints "1 1"
```

When you write = 1 on a property with a wrapper, that's translated into a call to the init(wrappedValue:) initializer. The instances of SmallNumber that wrap height and width are created by calling SmallNumber(wrappedValue: 1). The initializer uses the wrapped value that's specified here, and it uses the default maximum value of 12.

When you write arguments in parentheses after the custom attribute, Swift uses the initializer that accepts those arguments to set up the wrapper. For example, if you provide an initial value and a maximum value, Swift uses the init(wrappedValue:maximum:) initializer:

```
struct NarrowRectangle {
    @SmallNumber(wrappedValue: 2, maximum: 5) var height: Int
    @SmallNumber(wrappedValue: 3, maximum: 4) var width: Int
}

var narrowRectangle = NarrowRectangle()
print(narrowRectangle.height, narrowRectangle.width)
// Prints "2 3"
```

```
9
10 narrowRectangle.height = 100
11 narrowRectangle.width = 100
12 print(narrowRectangle.height, narrowRectangle.width)
13 // Prints "5 4"
```

The instance of SmallNumber that wraps height is created by calling SmallNumber(wrappedValue: 2, maximum: 5), and the instance that wraps width is created by calling SmallNumber(wrappedValue: 3, maximum: 4).

By including arguments to the property wrapper, you can set up the initial state in the wrapper or pass other options to the wrapper when it's created. This syntax is the most general way to use a property wrapper. You can provide whatever arguments you need to the attribute, and they're passed to the initializer.

When you include property wrapper arguments, you can also specify an initial value using assignment. Swift treats the assignment like a wrappedValue argument and uses the initializer that accepts the arguments you include. For example:

```
struct MixedRectangle {
        @SmallNumber var height: Int = 1
2
        @SmallNumber(maximum: 9) var width: Int = 2
3
4
    }
5
    var mixedRectangle = MixedRectangle()
6
7
    print(mixedRectangle.height)
    // Prints "1"
8
9
    mixedRectangle.height = 20
10
11
    print(mixedRectangle.height)
12
     // Prints "12"
```

The instance of SmallNumber that wraps height is created by calling SmallNumber(wrappedValue: 1), which uses the default maximum value of 12. The instance that wraps width is created by calling SmallNumber(wrappedValue: 2, maximum: 9).

Projecting a Value From a Property Wrapper

In addition to the wrapped value, a property wrapper can expose additional functionality by defining a *projected value*—for example, a property wrapper that manages access to a database can expose a flushDatabaseConnection() method on its projected value. The name of the projected value is the same as the wrapped value, except it begins with a dollar sign (\$). Because your code can't define properties that start with \$ the projected value never interferes with properties you define.

In the SmallNumber example above, if you try to set the property to a number that's too large, the property wrapper adjusts the number before storing it. The code below adds a projectedValue property to the SmallNumber structure to keep track of whether the property wrapper adjusted the new value for the property before storing that new value.

```
1
    @propertyWrapper
2
    struct SmallNumber {
         private var number = 0
3
         var projectedValue = false
         var wrappedValue: Int {
5
6
             get { return number }
7
             set {
                 if newValue > 12 {
8
                     number = 12
9
                     projectedValue = true
10
                 } else {
11
                     number = newValue
12
```

```
13
                     projectedValue = false
14
                 }
15
             }
         }
16
17
     }
     struct SomeStructure {
18
         @SmallNumber var someNumber: Int
19
20
    }
    var someStructure = SomeStructure()
21
22
23
     someStructure.someNumber = 4
24
     print(someStructure.$someNumber)
25
     // Prints "false"
26
27
    someStructure.someNumber = 55
28
     print(someStructure.$someNumber)
29
     // Prints "true"
```

Writing s.\$someNumber accesses the wrapper's projected value. After storing a small number like four, the value of s.\$someNumber is false. However, the projected value is true after trying to store a number that's too large, like 55.

A property wrapper can return a value of any type as its projected value. In this example, the property wrapper exposes only one piece of information—whether the number was adjusted—so it exposes that Boolean value as its projected value. A wrapper that needs to expose more information can return an instance of some other data type, or it can return self to expose the instance of the wrapper as its projected value.

When you access a projected value from code that's part of the type, like a property getter or an instance method, you can omit self. before the property name, just like accessing other properties. The code in the following example refers to the projected value of the wrapper around height and width as \$height

and \$width:

```
1
    enum Size {
2
         case small, large
3
    }
4
    struct SizedRectangle {
5
6
         @SmallNumber var height: Int
         @SmallNumber var width: Int
7
8
9
         mutating func resize(to size: Size) -> Bool {
             switch size {
10
             case .small:
11
                 height = 10
12
                 width = 20
13
             case large:
14
                 height = 100
15
                 width = 100
16
             }
17
18
             return $height || $width
19
         }
20
     }
```

Because property wrapper syntax is just syntactic sugar for a property with a getter and a setter, accessing height and width behaves the same as accessing any other property. For example, the code in resize(to:) accesses height and width using their property wrapper. If you call resize(to: .large), the switch case for .large sets the rectangle's height and width to 100. The wrapper prevents the value of those properties from being larger than 12, and it sets the projected value to true, to record the fact that it adjusted their values. At the end of resize(to:), the return statement checks \$height and \$width to determine whether the property wrapper adjusted either height or width.

Global and Local Variables

The capabilities described above for computing and observing properties are also available to *global variables* and *local variables*. Global variables are variables that are defined outside of any function, method, closure, or type context. Local variables are variables that are defined within a function, method, or closure context.

The global and local variables you have encountered in previous chapters have all been *stored variables*. Stored variables, like stored properties, provide storage for a value of a certain type and allow that value to be set and retrieved.

However, you can also define *computed variables* and define observers for stored variables, in either a global or local scope. Computed variables calculate their value, rather than storing it, and they are written in the same way as computed properties.

NOTE

Global constants and variables are always computed lazily, in a similar manner to <u>Lazy Stored</u> <u>Properties</u>. Unlike lazy stored properties, global constants and variables do not need to be marked with the <u>lazy</u> modifier.

Local constants and variables are never computed lazily.

Type Properties

Instance properties are properties that belong to an instance of a particular type. Every time you create a new instance of that type, it has its own set of property values, separate from any other instance.

You can also define properties that belong to the type itself, not to any one instance of that type. There will only ever be one copy of these properties, no matter how many instances of that type you create. These kinds of properties are called *type properties*.

Type properties are useful for defining values that are universal to *all* instances

of a particular type, such as a constant property that all instances can use (like a static constant in C), or a variable property that stores a value that is global to all instances of that type (like a static variable in C).

Stored type properties can be variables or constants. Computed type properties are always declared as variable properties, in the same way as computed instance properties.

NOTE

Unlike stored instance properties, you must always give stored type properties a default value. This is because the type itself does not have an initializer that can assign a value to a stored type property at initialization time.

Stored type properties are lazily initialized on their first access. They are guaranteed to be initialized only once, even when accessed by multiple threads simultaneously, and they do not need to be marked with the <code>lazy</code> modifier.

Type Property Syntax

In C and Objective-C, you define static constants and variables associated with a type as *global* static variables. In Swift, however, type properties are written as part of the type's definition, within the type's outer curly braces, and each type property is explicitly scoped to the type it supports.

You define type properties with the static keyword. For computed type properties for class types, you can use the class keyword instead to allow subclasses to override the superclass's implementation. The example below shows the syntax for stored and computed type properties:

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        return 1
    }
    enum SomeEnumeration {
```

```
8
         static var storedTypeProperty = "Some value."
         static var computedTypeProperty: Int {
9
10
             return 6
         }
11
12
     }
     class SomeClass {
13
         static var storedTypeProperty = "Some value."
14
         static var computedTypeProperty: Int {
15
             return 27
16
         }
17
         class var overrideableComputedTypeProperty: Int {
18
             return 107
19
20
         }
21
     }
```

NOTE

The computed type property examples above are for read-only computed type properties, but you can also define read-write computed type properties with the same syntax as for computed instance properties.

Querying and Setting Type Properties

Type properties are queried and set with dot syntax, just like instance properties. However, type properties are queried and set on the *type*, not on an instance of that type. For example:

```
print(SomeStructure.storedTypeProperty)
// Prints "Some value."

SomeStructure.storedTypeProperty = "Another value."
print(SomeStructure.storedTypeProperty)
// Prints "Another value."
```

```
print(SomeEnumeration.computedTypeProperty)

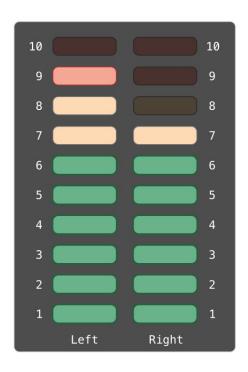
// Prints "6"

print(SomeClass.computedTypeProperty)

// Prints "27"
```

The examples that follow use two stored type properties as part of a structure that models an audio level meter for a number of audio channels. Each channel has an integer audio level between 0 and 10 inclusive.

The figure below illustrates how two of these audio channels can be combined to model a stereo audio level meter. When a channel's audio level is 0, none of the lights for that channel are lit. When the audio level is 10, all of the lights for that channel are lit. In this figure, the left channel has a current level of 9, and the right channel has a current level of 7:



The audio channels described above are represented by instances of the AudioChannel structure:

```
struct AudioChannel {
static let thresholdLevel = 10
```

```
3
         static var maxInputLevelForAllChannels = 0
        var currentLevel: Int = 0 {
4
             didSet {
5
                 if currentLevel > AudioChannel.thresholdLevel {
6
7
                     // cap the new audio level to the threshold
           leve1
                     currentLevel = AudioChannel.thresholdLevel
8
                 }
9
                 if currentLevel >
10
           AudioChannel.maxInputLevelForAllChannels {
                     // store this as the new overall maximum input
11
           level
                     AudioChannel.maxInputLevelForAllChannels =
12
           currentLevel
13
                 }
14
             }
15
         }
16
    }
```

The AudioChannel structure defines two stored type properties to support its functionality. The first, thresholdLevel, defines the maximum threshold value an audio level can take. This is a constant value of 10 for all AudioChannel instances. If an audio signal comes in with a higher value than 10, it will be capped to this threshold value (as described below).

The second type property is a variable stored property called maxInputLevelForAllChannels. This keeps track of the maximum input value that has been received by *any* AudioChannel instance. It starts with an initial value of 0.

The AudioChannel structure also defines a stored instance property called currentLevel, which represents the channel's current audio level on a scale of 0 to 10.

The currentLevel property has a didSet property observer to check the value of currentLevel whenever it is set. This observer performs two checks:

- If the new value of currentLevel is greater than the allowed thresholdLevel, the property observer caps currentLevel to thresholdLevel.
- If the new value of currentLevel (after any capping) is higher than any value previously received by *any* AudioChannel instance, the property observer stores the new currentLevel value in the maxInputLevelForAllChannels type property.

NOTE

In the first of these two checks, the didSet observer sets currentLevel to a different value. This does not, however, cause the observer to be called again.

You can use the AudioChannel structure to create two new audio channels called leftChannel and rightChannel, to represent the audio levels of a stereo sound system:

```
var leftChannel = AudioChannel()
var rightChannel = AudioChannel()
```

If you set the currentLevel of the *left* channel to 7, you can see that the maxInputLevelForAllChannels type property is updated to equal 7:

```
1 leftChannel.currentLevel = 7
2 print(leftChannel.currentLevel)
3 // Prints "7"
4 print(AudioChannel.maxInputLevelForAllChannels)
5 // Prints "7"
```

If you try to set the currentLevel of the *right* channel to 11, you can see that the right channel's currentLevel property is capped to the maximum value of 10, and the maxInputLevelForAllChannels type property is updated to equal 10:

```
rightChannel.currentLevel = 11
print(rightChannel.currentLevel)
// Prints "10"
print(AudioChannel.maxInputLevelForAllChannels)
// Prints "10"
```

Methods

Methods are functions that are associated with a particular type. Classes, structures, and enumerations can all define instance methods, which encapsulate specific tasks and functionality for working with an instance of a given type. Classes, structures, and enumerations can also define type methods, which are associated with the type itself. Type methods are similar to class methods in Objective-C.

The fact that structures and enumerations can define methods in Swift is a major difference from C and Objective-C. In Objective-C, classes are the only types that can define methods. In Swift, you can choose whether to define a class, structure, or enumeration, and still have the flexibility to define methods on the type you create.

Instance Methods

Instance methods are functions that belong to instances of a particular class, structure, or enumeration. They support the functionality of those instances, either by providing ways to access and modify instance properties, or by providing functionality related to the instance's purpose. Instance methods have exactly the same syntax as functions, as described in <u>Functions</u>.

You write an instance method within the opening and closing braces of the type it belongs to. An instance method has implicit access to all other instance methods and properties of that type. An instance method can be called only on a specific instance of the type it belongs to. It cannot be called in isolation without an existing instance.

Here's an example that defines a simple Counter class, which can be used to count the number of times an action occurs:

```
1 class Counter {
```

```
2
         var count = 0
         func increment() {
 3
             count += 1
 4
 5
         }
         func increment(by amount: Int) {
 6
 7
             count += amount
 8
         }
         func reset() {
9
             count = 0
10
         }
11
12
     }
```

The counter class defines three instance methods:

- increment() increments the counter by 1.
- increment(by: Int) increments the counter by a specified integer amount.
- reset() resets the counter to zero.

The Counter class also declares a variable property, count, to keep track of the current counter value.

You call instance methods with the same dot syntax as properties:

```
1 let counter = Counter()
2 // the initial counter value is 0
3 counter.increment()
4 // the counter's value is now 1
5 counter.increment(by: 5)
6 // the counter's value is now 6
7 counter.reset()
8 // the counter's value is now 0
```

Function parameters can have both a name (for use within the function's body) and an argument label (for use when calling the function), as described in Function Argument Labels and Parameter Names. The same is true for method parameters, because methods are just functions that are associated with a type.

The self Property

Every instance of a type has an implicit property called self, which is exactly equivalent to the instance itself. You use the self property to refer to the current instance within its own instance methods.

The increment() method in the example above could have been written like this:

```
func increment() {
    self.count += 1
}
```

In practice, you don't need to write self in your code very often. If you don't explicitly write self, Swift assumes that you are referring to a property or method of the current instance whenever you use a known property or method name within a method. This assumption is demonstrated by the use of count (rather than self.count) inside the three instance methods for counter.

The main exception to this rule occurs when a parameter name for an instance method has the same name as a property of that instance. In this situation, the parameter name takes precedence, and it becomes necessary to refer to the property in a more qualified way. You use the self property to distinguish between the parameter name and the property name.

Here, self disambiguates between a method parameter called \times and an instance property that is also called \times :

```
struct Point {
var x = 0.0, y = 0.0
func isToTheRightOf(x: Double) -> Bool {
```

```
4
             return self.x > x
5
        }
6
    }
7
    let somePoint = Point(x: 4.0, y: 5.0)
8
    if somePoint.isToTheRightOf(x: 1.0) {
        print("This point is to the right of the line where x ==
9
          1.0")
10
    }
    // Prints "This point is to the right of the line where x ==
11
          1.0"
```

Without the self prefix, Swift would assume that both uses of x referred to the method parameter called x.

Modifying Value Types from Within Instance Methods

Structures and enumerations are *value types*. By default, the properties of a value type cannot be modified from within its instance methods.

However, if you need to modify the properties of your structure or enumeration within a particular method, you can opt in to *mutating* behavior for that method. The method can then mutate (that is, change) its properties from within the method, and any changes that it makes are written back to the original structure when the method ends. The method can also assign a completely new instance to its implicit self property, and this new instance will replace the existing one when the method ends.

You can opt in to this behavior by placing the mutating keyword before the func keyword for that method:

```
struct Point {
var x = 0.0, y = 0.0
mutating func moveBy(x deltaX: Double, y deltaY: Double) {
```

The Point structure above defines a mutating moveBy(x:y:) method, which moves a Point instance by a certain amount. Instead of returning a new point, this method actually modifies the point on which it is called. The mutating keyword is added to its definition to enable it to modify its properties.

Note that you cannot call a mutating method on a constant of structure type, because its properties cannot be changed, even if they are variable properties, as described in <u>Stored Properties of Constant Structure Instances</u>:

```
1 let fixedPoint = Point(x: 3.0, y: 3.0)
2 fixedPoint.moveBy(x: 2.0, y: 3.0)
3 // this will report an error
```

Assigning to self Within a Mutating Method

Mutating methods can assign an entirely new instance to the implicit self property. The Point example shown above could have been written in the following way instead:

```
struct Point {
var x = 0.0, y = 0.0
mutating func moveBy(x deltaX: Double, y deltaY: Double) {
    self = Point(x: x + deltaX, y: y + deltaY)
```

```
5 }
6 }
```

This version of the mutating moveBy(x:y:) method creates a new structure whose x and y values are set to the target location. The end result of calling this alternative version of the method will be exactly the same as for calling the earlier version.

Mutating methods for enumerations can set the implicit self parameter to be a different case from the same enumeration:

```
1
    enum TriStateSwitch {
2
        case off, low, high
        mutating func next() {
3
             switch self {
4
             case off:
5
                 self = .low
6
7
             case low:
8
                 self = .high
9
             case high:
                 self = .off
10
11
             }
        }
12
13
    }
    var ovenLight = TriStateSwitch.low
14
    ovenLight.next()
15
    // ovenLight is now equal to .high
16
17
    ovenLight.next()
    // ovenLight is now equal to .off
18
```

This example defines an enumeration for a three-state switch. The switch cycles between three different power states (off, low and high) every time its next()

method is called.

Type Methods

Instance methods, as described above, are methods that you call on an instance of a particular type. You can also define methods that are called on the type itself. These kinds of methods are called *type methods*. You indicate type methods by writing the static keyword before the method's func keyword. Classes can use the class keyword instead, to allow subclasses to override the superclass's implementation of that method.

```
NOTE
```

In Objective-C, you can define type-level methods only for Objective-C classes. In Swift, you can define type-level methods for all classes, structures, and enumerations. Each type method is explicitly scoped to the type it supports.

Type methods are called with dot syntax, like instance methods. However, you call type methods on the type, not on an instance of that type. Here's how you call a type method on a class called SomeClass:

```
class SomeClass {
   class func someTypeMethod() {
      // type method implementation goes here
}
SomeClass.someTypeMethod()
```

Within the body of a type method, the implicit self property refers to the type itself, rather than an instance of that type. This means that you can use self to disambiguate between type properties and type method parameters, just as you do for instance properties and instance method parameters.

More generally, any unqualified method and property names that you use within

the body of a type method will refer to other type-level methods and properties. A type method can call another type method with the other method's name, without needing to prefix it with the type name. Similarly, type methods on structures and enumerations can access type properties by using the type property's name without a type name prefix.

The example below defines a structure called LevelTracker, which tracks a player's progress through the different levels or stages of a game. It is a single-player game, but can store information for multiple players on a single device.

All of the game's levels (apart from level one) are locked when the game is first played. Every time a player finishes a level, that level is unlocked for all players on the device. The LevelTracker structure uses type properties and methods to keep track of which levels of the game have been unlocked. It also tracks the current level for an individual player.

```
1
    struct LevelTracker {
         static var highestUnlockedLevel = 1
2
         var currentLevel = 1
4
5
         static func unlock(_ level: Int) {
             if level > highestUnlockedLevel { highestUnlockedLevel
6
           = level }
         }
7
8
9
         static func isUnlocked(_ level: Int) -> Bool {
             return level <= highestUnlockedLevel</pre>
10
         }
11
12
13
         @discardableResult
         mutating func advance(to level: Int) -> Bool {
14
             if LevelTracker.isUnlocked(level) {
15
                 currentLevel = level
16
```

```
17 return true

18 } else {

19 return false

20 }

21 }

22 }
```

The LevelTracker structure keeps track of the highest level that any player has unlocked. This value is stored in a type property called highestUnlockedLevel.

LevelTracker also defines two type functions to work with the highestUnlockedLevel property. The first is a type function called unlock(_:), which updates the value of highestUnlockedLevel whenever a new level is unlocked. The second is a convenience type function called isUnlocked(_:), which returns true if a particular level number is already unlocked. (Note that these type methods can access the highestUnlockedLevel type property without your needing to write it as LevelTracker.highestUnlockedLevel.)

In addition to its type property and type methods, LevelTracker tracks an individual player's progress through the game. It uses an instance property called currentLevel to track the level that a player is currently playing.

To help manage the currentLevel property, LevelTracker defines an instance method called advance(to:). Before updating currentLevel, this method checks whether the requested new level is already unlocked. The advance(to:) method returns a Boolean value to indicate whether or not it was actually able to set currentLevel. Because it's not necessarily a mistake for code that calls the advance(to:) method to ignore the return value, this function is marked with the @discardableResult attribute. For more information about this attribute, see Attributes.

The LevelTracker structure is used with the Player class, shown below, to track and update the progress of an individual player:

```
class Player {
   var tracker = LevelTracker()
```

```
3
         let playerName: String
         func complete(level: Int) {
4
             LevelTracker.unlock(level + 1)
5
             tracker.advance(to: level + 1)
6
7
         }
         init(name: String) {
             playerName = name
9
         }
10
11
     }
```

The Player class creates a new instance of LevelTracker to track that player's progress. It also provides a method called complete(level:), which is called whenever a player completes a particular level. This method unlocks the next level for all players and updates the player's progress to move them to the next level. (The Boolean return value of advance(to:) is ignored, because the level is known to have been unlocked by the call to LevelTracker.unlock(_:) on the previous line.)

You can create an instance of the Player class for a new player, and see what happens when the player completes level one:

If you create a second player, whom you try to move to a level that is not yet unlocked by any player in the game, the attempt to set the player's current level fails:

```
player = Player(name: "Beto")

if player.tracker.advance(to: 6) {
```

```
print("player is now on level 6")

print("level 6 has not yet been unlocked")

// Prints "level 6 has not yet been unlocked"
```

Subscripts

Classes, structures, and enumerations can define *subscripts*, which are shortcuts for accessing the member elements of a collection, list, or sequence. You use subscripts to set and retrieve values by index without needing separate methods for setting and retrieval. For example, you access elements in an Array instance as <code>someArray[index]</code> and elements in a <code>Dictionary</code> instance as <code>someDictionary[key]</code>.

You can define multiple subscripts for a single type, and the appropriate subscript overload to use is selected based on the type of index value you pass to the subscript. Subscripts are not limited to a single dimension, and you can define subscripts with multiple input parameters to suit your custom type's needs.

Subscript Syntax

Subscripts enable you to query instances of a type by writing one or more values in square brackets after the instance name. Their syntax is similar to both instance method syntax and computed property syntax. You write subscript definitions with the subscript keyword, and specify one or more input parameters and a return type, in the same way as instance methods. Unlike instance methods, subscripts can be read-write or read-only. This behavior is communicated by a getter and setter in the same way as for computed properties:

```
subscript(index: Int) -> Int {
   get {
        // Return an appropriate subscript value here.
}
set(newValue) {
        // Perform a suitable setting action here.
```

```
7 }
8 }
```

The type of newValue is the same as the return value of the subscript. As with computed properties, you can choose not to specify the setter's (newValue) parameter. A default parameter called newValue is provided to your setter if you do not provide one yourself.

As with read-only computed properties, you can simplify the declaration of a read-only subscript by removing the get keyword and its braces:

```
subscript(index: Int) -> Int {
// Return an appropriate subscript value here.
}
```

Here's an example of a read-only subscript implementation, which defines a <code>TimesTable</code> structure to represent an *n*-times-table of integers:

```
struct TimesTable {
1
2
        let multiplier: Int
        subscript(index: Int) -> Int {
3
            return multiplier * index
        }
5
6
   }
   let threeTimesTable = TimesTable(multiplier: 3)
7
   print("six times three is \((threeTimesTable[6])")
8
   // Prints "six times three is 18"
```

In this example, a new instance of TimesTable is created to represent the three-times-table. This is indicated by passing a value of 3 to the structure's initializer as the value to use for the instance's multiplier parameter.

You can query the threeTimesTable instance by calling its subscript, as shown in the call to threeTimesTable[6]. This requests the sixth entry in the three-times-

table, which returns a value of 18, or 3 times 6.

NOTE

An *n*-times-table is based on a fixed mathematical rule. It is not appropriate to set threeTimesTable[someIndex] to a new value, and so the subscript for TimesTable is defined as a read-only subscript.

Subscript Usage

The exact meaning of "subscript" depends on the context in which it is used. Subscripts are typically used as a shortcut for accessing the member elements in a collection, list, or sequence. You are free to implement subscripts in the most appropriate way for your particular class or structure's functionality.

For example, Swift's Dictionary type implements a subscript to set and retrieve the values stored in a Dictionary instance. You can set a value in a dictionary by providing a key of the dictionary's key type within subscript brackets, and assigning a value of the dictionary's value type to the subscript:

```
var numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
numberOfLegs["bird"] = 2
```

The example above defines a variable called numberOfLegs and initializes it with a dictionary literal containing three key-value pairs. The type of the numberOfLegs dictionary is inferred to be [String: Int]. After creating the dictionary, this example uses subscript assignment to add a String key of "bird" and an Int value of 2 to the dictionary.

For more information about Dictionary subscripting, see <u>Accessing and Modifying a Dictionary</u>.

NOTE

Swift's Dictionary type implements its key-value subscripting as a subscript that takes and returns an *optional* type. For the numberOfLegs dictionary above, the key-value subscript takes and returns a value of type Int?, or "optional int". The Dictionary type uses an optional

subscript type to model the fact that not every key will have a value, and to give a way to delete a value for a key by assigning a nil value for that key.

Subscript Options

Subscripts can take any number of input parameters, and these input parameters can be of any type. Subscripts can also return any type. Subscripts can use variadic parameters, but they can't use in-out parameters or provide default parameter values.

A class or structure can provide as many subscript implementations as it needs, and the appropriate subscript to be used will be inferred based on the types of the value or values that are contained within the subscript brackets at the point that the subscript is used. This definition of multiple subscripts is known as *subscript overloading*.

While it is most common for a subscript to take a single parameter, you can also define a subscript with multiple parameters if it is appropriate for your type. The following example defines a Matrix structure, which represents a two-dimensional matrix of Double values. The Matrix structure's subscript takes two integer parameters:

```
struct Matrix {
1
        let rows: Int, columns: Int
2
        var grid: [Double]
3
        init(rows: Int, columns: Int) {
             self.rows = rows
5
             self.columns = columns
6
             grid = Array(repeating: 0.0, count: rows * columns)
7
8
        func indexIsValid(row: Int, column: Int) -> Bool {
9
             return row >= 0 && row < rows && column >= 0 && column
10
```

```
< columns
11
         }
         subscript(row: Int, column: Int) -> Double {
12
             get {
13
                 assert(indexIsValid(row: row, column: column),
14
           "Index out of range")
                 return grid[(row * columns) + column]
15
             }
16
             set {
17
                 assert(indexIsValid(row: row, column: column),
18
           "Index out of range")
                 grid[(row * columns) + column] = newValue
19
20
             }
         }
21
22
     }
```

Matrix provides an initializer that takes two parameters called rows and columns, and creates an array that is large enough to store rows * columns values of type Double. Each position in the matrix is given an initial value of 0.0. To achieve this, the array's size, and an initial cell value of 0.0, are passed to an array initializer that creates and initializes a new array of the correct size. This initializer is described in more detail in Creating an Array with a Default Value.

You can construct a new Matrix instance by passing an appropriate row and column count to its initializer:

```
var matrix = Matrix(rows: 2, columns: 2)
```

The example above creates a new Matrix instance with two rows and two columns. The grid array for this Matrix instance is effectively a flattened version of the matrix, as read from top left to bottom right:

Values in the matrix can be set by passing row and column values into the subscript, separated by a comma:

```
1 matrix[0, 1] = 1.5
2 matrix[1, 0] = 3.2
```

These two statements call the subscript's setter to set a value of 1.5 in the top right position of the matrix (where row is 0 and column is 1), and 3.2 in the bottom left position (where row is 1 and column is 0):

$$\left[\begin{array}{cc} 0.0 & 1.5 \\ 3.2 & 0.0 \end{array}\right]$$

The Matrix subscript's getter and setter both contain an assertion to check that the subscript's row and column values are valid. To assist with these assertions, Matrix includes a convenience method called indexIsValid(row:column:), which checks whether the requested row and column are inside the bounds of the matrix:

```
func indexIsValid(row: Int, column: Int) -> Bool {
   return row >= 0 && row < rows && column >= 0 && column <
        columns
}</pre>
```

An assertion is triggered if you try to access a subscript that is outside of the matrix bounds:

```
1 let someValue = matrix[2, 2]
2 // This triggers an assert, because [2, 2] is outside of the
    matrix bounds.
```

Type Subscripts

Instance subscripts, as described above, are subscripts that you call on an instance of a particular type. You can also define subscripts that are called on the type itself. This kind of subscript is called a *type subscript*. You indicate a type subscript by writing the static keyword before the subscript keyword. Classes can use the class keyword instead, to allow subclasses to override the superclass's implementation of that subscript. The example below shows how you define and call a type subscript:

```
1
   enum Planet: Int {
2
        case mercury = 1, venus, earth, mars, jupiter, saturn,
          uranus, neptune
        static subscript(n: Int) -> Planet {
3
            return Planet(rawValue: n)!
4
5
        }
6
   }
7
   let mars = Planet[4]
8
    print(mars)
```

Inheritance

A class can *inherit* methods, properties, and other characteristics from another class. When one class inherits from another, the inheriting class is known as a *subclass*, and the class it inherits from is known as its *superclass*. Inheritance is a fundamental behavior that differentiates classes from other types in Swift.

Classes in Swift can call and access methods, properties, and subscripts belonging to their superclass and can provide their own overriding versions of those methods, properties, and subscripts to refine or modify their behavior. Swift helps to ensure your overrides are correct by checking that the override definition has a matching superclass definition.

Classes can also add property observers to inherited properties in order to be notified when the value of a property changes. Property observers can be added to any property, regardless of whether it was originally defined as a stored or computed property.

Defining a Base Class

Any class that does not inherit from another class is known as a *base class*.

NOTE

Swift classes do not inherit from a universal base class. Classes you define without specifying a superclass automatically become base classes for you to build upon.

The example below defines a base class called <code>Vehicle</code>. This base class defines a stored property called <code>currentSpeed</code>, with a default value of <code>0.0</code> (inferring a property type of <code>Double</code>). The <code>currentSpeed</code> property's value is used by a read-only computed <code>String</code> property called <code>description</code> to create a description of the vehicle.

The Vehicle base class also defines a method called makeNoise. This method

does not actually do anything for a base <code>Vehicle</code> instance, but will be customized by subclasses of <code>Vehicle</code> later on:

```
class Vehicle {
   var currentSpeed = 0.0
   var description: String {
      return "traveling at \(currentSpeed) miles per hour"
   }
   func makeNoise() {
      // do nothing - an arbitrary vehicle doesn't
      necessarily make a noise
   }
}
```

You create a new instance of Vehicle with *initializer syntax*, which is written as a type name followed by empty parentheses:

```
let someVehicle = Vehicle()
```

Having created a new Vehicle instance, you can access its description property to print a human-readable description of the vehicle's current speed:

```
print("Vehicle: \(someVehicle.description)")
// Vehicle: traveling at 0.0 miles per hour
```

The Vehicle class defines common characteristics for an arbitrary vehicle, but is not much use in itself. To make it more useful, you need to refine it to describe more specific kinds of vehicles.

Subclassing

Subclassing is the act of basing a new class on an existing class. The subclass

inherits characteristics from the existing class, which you can then refine. You can also add new characteristics to the subclass.

To indicate that a subclass has a superclass, write the subclass name before the superclass name, separated by a colon:

```
class SomeSubclass: SomeSuperclass {
    // subclass definition goes here
}
```

The following example defines a subclass called Bicycle, with a superclass of Vehicle:

```
class Bicycle: Vehicle {
   var hasBasket = false
}
```

The new Bicycle class automatically gains all of the characteristics of Vehicle, such as its currentSpeed and description properties and its makeNoise() method.

In addition to the characteristics it inherits, the Bicycle class defines a new stored property, hasBasket, with a default value of false (inferring a type of Bool for the property).

By default, any new Bicycle instance you create will not have a basket. You can set the hasBasket property to true for a particular Bicycle instance after that instance is created:

```
1 let bicycle = Bicycle()
2 bicycle.hasBasket = true
```

You can also modify the inherited currentSpeed property of a Bicycle instance, and query the instance's inherited description property:

```
bicycle.currentSpeed = 15.0
```

```
print("Bicycle: \(bicycle.description)")

// Bicycle: traveling at 15.0 miles per hour
```

Subclasses can themselves be subclassed. The next example creates a subclass of Bicycle for a two-seater bicycle known as a "tandem":

```
class Tandem: Bicycle {
   var currentNumberOfPassengers = 0
}
```

Tandem inherits all of the properties and methods from Bicycle, which in turn inherits all of the properties and methods from Vehicle. The Tandem subclass also adds a new stored property called currentNumberOfPassengers, with a default value of 0.

If you create an instance of Tandem, you can work with any of its new and inherited properties, and query the read-only description property it inherits from Vehicle:

```
1 let tandem = Tandem()
2 tandem.hasBasket = true
3 tandem.currentNumberOfPassengers = 2
4 tandem.currentSpeed = 22.0
5 print("Tandem: \((tandem.description)"))
6 // Tandem: traveling at 22.0 miles per hour
```

Overriding

A subclass can provide its own custom implementation of an instance method, type method, instance property, type property, or subscript that it would otherwise inherit from a superclass. This is known as *overriding*.

To override a characteristic that would otherwise be inherited, you prefix your overriding definition with the override keyword. Doing so clarifies that you intend to provide an override and have not provided a matching definition by mistake. Overriding by accident can cause unexpected behavior, and any overrides without the override keyword are diagnosed as an error when your code is compiled.

The override keyword also prompts the Swift compiler to check that your overriding class's superclass (or one of its parents) has a declaration that matches the one you provided for the override. This check ensures that your overriding definition is correct.

Accessing Superclass Methods, Properties, and Subscripts

When you provide a method, property, or subscript override for a subclass, it is sometimes useful to use the existing superclass implementation as part of your override. For example, you can refine the behavior of that existing implementation, or store a modified value in an existing inherited variable.

Where this is appropriate, you access the superclass version of a method, property, or subscript by using the super prefix:

- An overridden method named someMethod() can call the superclass version of someMethod() by calling super.someMethod() within the overriding method implementation.
- An overridden property called someProperty can access the superclass version of someProperty as super.someProperty within the overriding getter or setter implementation.
- An overridden subscript for someIndex can access the superclass version of the same subscript as super[someIndex] from within the overriding subscript implementation.

Overriding Methods

You can override an inherited instance or type method to provide a tailored or alternative implementation of the method within your subclass.

The following example defines a new subclass of Vehicle called Train, which overrides the makeNoise() method that Train inherits from Vehicle:

```
class Train: Vehicle {
    override func makeNoise() {
        print("Choo Choo")
    }
}
```

If you create a new instance of Train and call its makeNoise() method, you can see that the Train subclass version of the method is called:

```
1 let train = Train()
2 train.makeNoise()
3 // Prints "Choo Choo"
```

Overriding Properties

You can override an inherited instance or type property to provide your own custom getter and setter for that property, or to add property observers to enable the overriding property to observe when the underlying property value changes.

Overriding Property Getters and Setters

You can provide a custom getter (and setter, if appropriate) to override *any* inherited property, regardless of whether the inherited property is implemented as a stored or computed property at source. The stored or computed nature of an inherited property is not known by a subclass—it only knows that the inherited property has a certain name and type. You must always state both the name and the type of the property you are overriding, to enable the compiler to check that your override matches a superclass property with the same name and type.

You can present an inherited read-only property as a read-write property by providing both a getter and a setter in your subclass property override. You cannot, however, present an inherited read-write property as a read-only property.

NOTE

If you provide a setter as part of a property override, you must also provide a getter for that override. If you don't want to modify the inherited property's value within the overriding getter, you can simply pass through the inherited value by returning <code>super.someProperty</code> from the getter, where <code>someProperty</code> is the name of the property you are overriding.

The following example defines a new class called <code>car</code>, which is a subclass of <code>vehicle</code>. The <code>car</code> class introduces a new stored property called <code>gear</code>, with a default integer value of <code>1</code>. The <code>car</code> class also overrides the <code>description</code> property it inherits from <code>vehicle</code>, to provide a custom description that includes the current gear:

```
class Car: Vehicle {
   var gear = 1
   override var description: String {
      return super.description + " in gear \(gear)"
   }
}
```

The override of the description property starts by calling super description, which returns the Vehicle class's description property. The car class's version of description then adds some extra text onto the end of this description to provide information about the current gear.

If you create an instance of the Car class and set its gear and currentSpeed properties, you can see that its description property returns the tailored description defined within the Car class:

```
1 let car = Car()
2 car.currentSpeed = 25.0
```

```
3 car.gear = 3
4 print("Car: \(car.description\)")
5 // Car: traveling at 25.0 miles per hour in gear 3
```

Overriding Property Observers

You can use property overriding to add property observers to an inherited property. This enables you to be notified when the value of an inherited property changes, regardless of how that property was originally implemented. For more information on property observers, see <u>Property Observers</u>.

NOTE

You cannot add property observers to inherited constant stored properties or inherited read-only computed properties. The value of these properties cannot be set, and so it is not appropriate to provide a <code>willSet</code> or <code>didSet</code> implementation as part of an override.

Note also that you cannot provide both an overriding setter and an overriding property observer for the same property. If you want to observe changes to a property's value, and you are already providing a custom setter for that property, you can simply observe any value changes from within the custom setter.

The following example defines a new class called AutomaticCar, which is a subclass of Car. The AutomaticCar class represents a car with an automatic gearbox, which automatically selects an appropriate gear to use based on the current speed:

```
class AutomaticCar: Car {
    override var currentSpeed: Double {
        didSet {
            gear = Int(currentSpeed / 10.0) + 1
        }
    }
}
```

Whenever you set the currentSpeed property of an AutomaticCar instance, the property's didSet observer sets the instance's gear property to an appropriate

choice of gear for the new speed. Specifically, the property observer chooses a gear that is the new currentSpeed value divided by 10, rounded down to the nearest integer, plus 1. A speed of 35.0 produces a gear of 4:

```
1 let automatic = AutomaticCar()
2 automatic.currentSpeed = 35.0
3 print("AutomaticCar: \(automatic.description\)")
4 // AutomaticCar: traveling at 35.0 miles per hour in gear 4
```

Preventing Overrides

You can prevent a method, property, or subscript from being overridden by marking it as *final*. Do this by writing the final modifier before the method, property, or subscript's introducer keyword (such as final var, final func, final class func, and final subscript).

Any attempt to override a final method, property, or subscript in a subclass is reported as a compile-time error. Methods, properties, or subscripts that you add to a class in an extension can also be marked as final within the extension's definition.

You can mark an entire class as final by writing the final modifier before the class keyword in its class definition (final class). Any attempt to subclass a final class is reported as a compile-time error.

Initialization

Initialization is the process of preparing an instance of a class, structure, or enumeration for use. This process involves setting an initial value for each stored property on that instance and performing any other setup or initialization that is required before the new instance is ready for use.

You implement this initialization process by defining *initializers*, which are like special methods that can be called to create a new instance of a particular type. Unlike Objective-C initializers, Swift initializers do not return a value. Their primary role is to ensure that new instances of a type are correctly initialized before they are used for the first time.

Instances of class types can also implement a *deinitializer*, which performs any custom cleanup just before an instance of that class is deallocated. For more information about deinitializers, see <u>Deinitialization</u>.

Setting Initial Values for Stored Properties

Classes and structures *must* set all of their stored properties to an appropriate initial value by the time an instance of that class or structure is created. Stored properties cannot be left in an indeterminate state.

You can set an initial value for a stored property within an initializer, or by assigning a default property value as part of the property's definition. These actions are described in the following sections.

NOTE

When you assign a default value to a stored property, or set its initial value within an initializer, the value of that property is set directly, without calling any property observers.

Initializers

Initializers are called to create a new instance of a particular type. In its simplest form, an initializer is like an instance method with no parameters, written using the init keyword:

```
init() {
    // perform some initialization here
}
```

The example below defines a new structure called Fahrenheit to store temperatures expressed in the Fahrenheit scale. The Fahrenheit structure has one stored property, temperature, which is of type Double:

```
struct Fahrenheit {
1
2
        var temperature: Double
3
        init() {
4
            temperature = 32.0
        }
5
6
   }
7
   var f = Fahrenheit()
   print("The default temperature is \(f.temperature)°
          Fahrenheit")
   // Prints "The default temperature is 32.0° Fahrenheit"
```

The structure defines a single initializer, init, with no parameters, which initializes the stored temperature with a value of 32.0 (the freezing point of water in degrees Fahrenheit).

Default Property Values

You can set the initial value of a stored property from within an initializer, as shown above. Alternatively, specify a *default property value* as part of the property's declaration. You specify a default property value by assigning an

initial value to the property when it is defined.

```
NOTE
```

If a property always takes the same initial value, provide a default value rather than setting a value within an initializer. The end result is the same, but the default value ties the property's initialization more closely to its declaration. It makes for shorter, clearer initializers and enables you to infer the type of the property from its default value. The default value also makes it easier for you to take advantage of default initializers and initializer inheritance, as described later in this chapter.

You can write the Fahrenheit structure from above in a simpler form by providing a default value for its temperature property at the point that the property is declared:

```
struct Fahrenheit {
  var temperature = 32.0
}
```

Customizing Initialization

You can customize the initialization process with input parameters and optional property types, or by assigning constant properties during initialization, as described in the following sections.

Initialization Parameters

You can provide *initialization parameters* as part of an initializer's definition, to define the types and names of values that customize the initialization process. Initialization parameters have the same capabilities and syntax as function and method parameters.

The following example defines a structure called <code>celsius</code>, which stores temperatures expressed in degrees Celsius. The <code>celsius</code> structure implements two custom initializers called <code>init(fromFahrenheit:)</code> and <code>init(fromKelvin:)</code>, which initialize a new instance of the structure with a value from a different

temperature scale:

```
1
    struct Celsius {
        var temperatureInCelsius: Double
2
        init(fromFahrenheit fahrenheit: Double) {
3
             temperatureInCelsius = (fahrenheit - 32.0) / 1.8
 4
        }
5
6
        init(fromKelvin kelvin: Double) {
             temperatureInCelsius = kelvin - 273.15
7
8
        }
    }
9
10
    let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
    // boilingPointOfWater.temperatureInCelsius is 100.0
11
12
    let freezingPointOfWater = Celsius(fromKelvin: 273.15)
    // freezingPointOfWater.temperatureInCelsius is 0.0
13
```

The first initializer has a single initialization parameter with an argument label of fromFahrenheit and a parameter name of fahrenheit. The second initializer has a single initialization parameter with an argument label of fromKelvin and a parameter name of kelvin. Both initializers convert their single argument into the corresponding Celsius value and store this value in a property called temperatureInCelsius.

Parameter Names and Argument Labels

As with function and method parameters, initialization parameters can have both a parameter name for use within the initializer's body and an argument label for use when calling the initializer.

However, initializers do not have an identifying function name before their parentheses in the way that functions and methods do. Therefore, the names and types of an initializer's parameters play a particularly important role in identifying which initializer should be called. Because of this, Swift provides an

automatic argument label for *every* parameter in an initializer if you don't provide one.

The following example defines a structure called color, with three constant properties called red, green, and blue. These properties store a value between 0.0 and 1.0 to indicate the amount of red, green, and blue in the color.

Color provides an initializer with three appropriately named parameters of type Double for its red, green, and blue components. Color also provides a second initializer with a single white parameter, which is used to provide the same value for all three color components.

```
struct Color {
1
2
         let red, green, blue: Double
         init(red: Double, green: Double, blue: Double) {
3
             self.red = red
5
             self.green = green
             self.blue = blue
6
7
         }
         init(white: Double) {
8
             red
                   = white
             green = white
10
11
             blue = white
12
         }
13
    }
```

Both initializers can be used to create a new color instance, by providing named values for each initializer parameter:

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
let halfGray = Color(white: 0.5)
```

Note that it is not possible to call these initializers without using argument labels. Argument labels must always be used in an initializer if they are defined,

and omitting them is a compile-time error:

Initializer Parameters Without Argument Labels

If you do not want to use an argument label for an initializer parameter, write an underscore (_) instead of an explicit argument label for that parameter to override the default behavior.

Here's an expanded version of the Celsius example from <u>Initialization</u>

<u>Parameters</u> above, with an additional initializer to create a new Celsius instance from a Double value that is already in the Celsius scale:

```
1
    struct Celsius {
        var temperatureInCelsius: Double
        init(fromFahrenheit fahrenheit: Double) {
3
4
             temperatureInCelsius = (fahrenheit - 32.0) / 1.8
        }
5
        init(fromKelvin kelvin: Double) {
6
             temperatureInCelsius = kelvin - 273.15
7
        }
        init(_ celsius: Double) {
             temperatureInCelsius = celsius
10
        }
11
12
    }
    let bodyTemperature = Celsius(37.0)
13
14
    // bodyTemperature.temperatureInCelsius is 37.0
```

The initializer call Celsius(37.0) is clear in its intent without the need for an

argument label. It is therefore appropriate to write this initializer as init(_ celsius: Double) so that it can be called by providing an unnamed Double value.

Optional Property Types

If your custom type has a stored property that is logically allowed to have "no value"—perhaps because its value cannot be set during initialization, or because it is allowed to have "no value" at some later point—declare the property with an *optional* type. Properties of optional type are automatically initialized with a value of nil, indicating that the property is deliberately intended to have "no value yet" during initialization.

The following example defines a class called SurveyQuestion, with an optional String property called response:

```
1
    class SurveyQuestion {
         var text: String
2
3
         var response: String?
         init(text: String) {
4
             self.text = text
5
         }
6
7
         func ask() {
             print(text)
8
         }
9
10
     }
11
    let cheeseQuestion = SurveyQuestion(text: "Do you like
           cheese?")
    cheeseQuestion.ask()
12
    // Prints "Do you like cheese?"
13
     cheeseQuestion.response = "Yes, I do like cheese."
14
```

The response to a survey question cannot be known until it is asked, and so the response property is declared with a type of String?, or "optional String". It is

automatically assigned a default value of nil, meaning "no string yet", when a new instance of SurveyQuestion is initialized.

Assigning Constant Properties During Initialization

You can assign a value to a constant property at any point during initialization, as long as it is set to a definite value by the time initialization finishes. Once a constant property is assigned a value, it can't be further modified.

NOTE

For class instances, a constant property can be modified during initialization only by the class that introduces it. It cannot be modified by a subclass.

You can revise the SurveyQuestion example from above to use a constant property rather than a variable property for the text property of the question, to indicate that the question does not change once an instance of SurveyQuestion is created. Even though the text property is now a constant, it can still be set within the class's initializer:

```
class SurveyQuestion {
1
2
         let text: String
         var response: String?
         init(text: String) {
             self.text = text
5
6
         func ask() {
7
             print(text)
         }
10
     }
    let beetsQuestion = SurveyQuestion(text: "How about beets?")
11
    beetsQuestion.ask()
12
    // Prints "How about beets?"
13
```

Default Initializers

Swift provides a *default initializer* for any structure or class that provides default values for all of its properties and does not provide at least one initializer itself. The default initializer simply creates a new instance with all of its properties set to their default values.

This example defines a class called ShoppingListItem, which encapsulates the name, quantity, and purchase state of an item in a shopping list:

```
class ShoppingListItem {
  var name: String?
  var quantity = 1
  var purchased = false
}
var item = ShoppingListItem()
```

Because all properties of the ShoppingListItem class have default values, and because it is a base class with no superclass, ShoppingListItem automatically gains a default initializer implementation that creates a new instance with all of its properties set to their default values. (The name property is an optional String property, and so it automatically receives a default value of nil, even though this value is not written in the code.) The example above uses the default initializer for the ShoppingListItem class to create a new instance of the class with initializer syntax, written as ShoppingListItem(), and assigns this new instance to a variable called item.

Memberwise Initializers for Structure Types

Structure types automatically receive a *memberwise initializer* if they don't define any of their own custom initializers. Unlike a default initializer, the structure receives a memberwise initializer even if it has stored properties that don't have default values.

The memberwise initializer is a shorthand way to initialize the member properties of new structure instances. Initial values for the properties of the new instance can be passed to the memberwise initializer by name.

The example below defines a structure called Size with two properties called width and height. Both properties are inferred to be of type Double by assigning a default value of 0.0.

The Size structure automatically receives an init(width:height:) memberwise initializer, which you can use to initialize a new Size instance:

```
struct Size {
    var width = 0.0, height = 0.0
}

let twoByTwo = Size(width: 2.0, height: 2.0)
```

When you call a memberwise initializer, you can omit values for any properties that have default values. In the example above, the Size structure has a default value for both its height and width properties. You can omit either property or both properties, and the initializer uses the default value for anything you omit—for example:

```
1 let zeroByTwo = Size(height: 2.0)
2 print(zeroByTwo.width, zeroByTwo.height)
3 // Prints "0.0 2.0"
4
5 let zeroByZero = Size()
6 print(zeroByZero.width, zeroByZero.height)
7 // Prints "0.0 0.0"
```

Initializer Delegation for Value Types

Initializers can call other initializers to perform part of an instance's initialization. This process, known as *initializer delegation*, avoids duplicating code across multiple initializers.

The rules for how initializer delegation works, and for what forms of delegation are allowed, are different for value types and class types. Value types (structures and enumerations) do not support inheritance, and so their initializer delegation process is relatively simple, because they can only delegate to another initializer that they provide themselves. Classes, however, can inherit from other classes, as described in Inheritance. This means that classes have additional responsibilities for ensuring that all stored properties they inherit are assigned a suitable value during initialization. These responsibilities are described in Class Inheritance and Initialization below.

For value types, you use self.init to refer to other initializers from the same value type when writing your own custom initializers. You can call self.init only from within an initializer.

Note that if you define a custom initializer for a value type, you will no longer have access to the default initializer (or the memberwise initializer, if it is a structure) for that type. This constraint prevents a situation in which additional essential setup provided in a more complex initializer is accidentally circumvented by someone using one of the automatic initializers.

NOTE

If you want your custom value type to be initializable with the default initializer and memberwise initializer, and also with your own custom initializers, write your custom initializers in an extension rather than as part of the value type's original implementation. For more information, see Extensions.

The following example defines a custom Rect structure to represent a geometric rectangle. The example requires two supporting structures called Size and Point, both of which provide default values of 0.0 for all of their properties:

```
struct Size {

var width = 0.0, height = 0.0
```

```
3  }
4  struct Point {
5    var x = 0.0, y = 0.0
6  }
```

You can initialize the Rect structure below in one of three ways—by using its default zero-initialized origin and size property values, by providing a specific origin point and size, or by providing a specific center point and size. These initialization options are represented by three custom initializers that are part of the Rect structure's definition:

```
1
    struct Rect {
2
         var origin = Point()
3
         var size = Size()
         init() {}
4
         init(origin: Point, size: Size) {
5
             self.origin = origin
6
             self.size = size
7
8
9
         init(center: Point, size: Size) {
             let originX = center.x - (size.width / 2)
10
             let originY = center.y - (size.height / 2)
11
             self.init(origin: Point(x: originX, y: originY), size:
12
           size)
         }
13
14
    }
```

The first Rect initializer, init(), is functionally the same as the default initializer that the structure would have received if it did not have its own custom initializers. This initializer has an empty body, represented by an empty pair of curly braces {}. Calling this initializer returns a Rect instance whose origin and size properties are both initialized with the default values of Point(x: 0.0, y:

0.0) and Size(width: 0.0, height: 0.0) from their property definitions:

```
1 let basicRect = Rect()
2 // basicRect's origin is (0.0, 0.0) and its size is (0.0, 0.0)
```

The second Rect initializer, init(origin:size:), is functionally the same as the memberwise initializer that the structure would have received if it did not have its own custom initializers. This initializer simply assigns the origin and size argument values to the appropriate stored properties:

The third Rect initializer, init(center:size:), is slightly more complex. It starts by calculating an appropriate origin point based on a center point and a size value. It then calls (or *delegates*) to the init(origin:size:) initializer, which stores the new origin and size values in the appropriate properties:

The init(center:size:) initializer could have assigned the new values of origin and size to the appropriate properties itself. However, it is more convenient (and clearer in intent) for the init(center:size:) initializer to take advantage of an existing initializer that already provides exactly that functionality.

```
NOTE

For an alternative way to write this example without defining the init() and init(origin:size:) initializers yourself, see <a href="Extensions">Extensions</a>.
```

Class Inheritance and Initialization

All of a class's stored properties—including any properties the class inherits from its superclass—*must* be assigned an initial value during initialization.

Swift defines two kinds of initializers for class types to help ensure all stored properties receive an initial value. These are known as designated initializers and convenience initializers.

Designated Initializers and Convenience Initializers

Designated initializers are the primary initializers for a class. A designated initializer fully initializes all properties introduced by that class and calls an appropriate superclass initializer to continue the initialization process up the superclass chain.

Classes tend to have very few designated initializers, and it is quite common for a class to have only one. Designated initializers are "funnel" points through which initialization takes place, and through which the initialization process continues up the superclass chain.

Every class must have at least one designated initializer. In some cases, this requirement is satisfied by inheriting one or more designated initializers from a superclass, as described in <u>Automatic Initializer Inheritance</u> below.

Convenience initializers are secondary, supporting initializers for a class. You can define a convenience initializer to call a designated initializer from the same class as the convenience initializer with some of the designated initializer's parameters set to default values. You can also define a convenience initializer to create an instance of that class for a specific use case or input value type.

You do not have to provide convenience initializers if your class does not require them. Create convenience initializers whenever a shortcut to a common initialization pattern will save time or make initialization of the class clearer in intent.

Syntax for Designated and Convenience Initializers

Designated initializers for classes are written in the same way as simple initializers for value types:

```
init( parameters ) {
    statements
}
```

Convenience initializers are written in the same style, but with the convenience modifier placed before the init keyword, separated by a space:

```
convenience init( parameters ) {
    statements
}
```

Initializer Delegation for Class Types

To simplify the relationships between designated and convenience initializers, Swift applies the following three rules for delegation calls between initializers:

Rule 1

A designated initializer must call a designated initializer from its immediate superclass.

Rule 2

A convenience initializer must call another initializer from the *same* class.

Rule 3

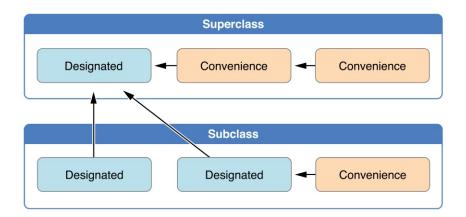
A convenience initializer must ultimately call a designated initializer.

A simple way to remember this is:

• Designated initializers must always delegate *up*.

• Convenience initializers must always delegate *across*.

These rules are illustrated in the figure below:



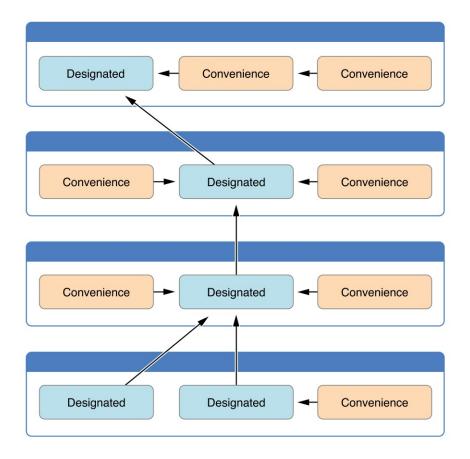
Here, the superclass has a single designated initializer and two convenience initializers. One convenience initializer calls another convenience initializer, which in turn calls the single designated initializer. This satisfies rules 2 and 3 from above. The superclass does not itself have a further superclass, and so rule 1 does not apply.

The subclass in this figure has two designated initializers and one convenience initializer. The convenience initializer must call one of the two designated initializers, because it can only call another initializer from the same class. This satisfies rules 2 and 3 from above. Both designated initializers must call the single designated initializer from the superclass, to satisfy rule 1 from above.

NOTE

These rules don't affect how users of your classes *create* instances of each class. Any initializer in the diagram above can be used to create a fully initialized instance of the class they belong to. The rules only affect how you write the implementation of the class's initializers.

The figure below shows a more complex class hierarchy for four classes. It illustrates how the designated initializers in this hierarchy act as "funnel" points for class initialization, simplifying the interrelationships among classes in the chain:



Two-Phase Initialization

Class initialization in Swift is a two-phase process. In the first phase, each stored property is assigned an initial value by the class that introduced it. Once the initial state for every stored property has been determined, the second phase begins, and each class is given the opportunity to customize its stored properties further before the new instance is considered ready for use.

The use of a two-phase initialization process makes initialization safe, while still giving complete flexibility to each class in a class hierarchy. Two-phase initialization prevents property values from being accessed before they are initialized, and prevents property values from being set to a different value by another initializer unexpectedly.

NOTE

Swift's two-phase initialization process is similar to initialization in Objective-C. The main difference is that during phase 1, Objective-C assigns zero or null values (such as 0 or nil) to every

property. Swift's initialization flow is more flexible in that it lets you set custom initial values, and can cope with types for which 0 or nil is not a valid default value.

Swift's compiler performs four helpful safety-checks to make sure that twophase initialization is completed without error:

Safety check 1

A designated initializer must ensure that all of the properties introduced by its class are initialized before it delegates up to a superclass initializer.

As mentioned above, the memory for an object is only considered fully initialized once the initial state of all of its stored properties is known. In order for this rule to be satisfied, a designated initializer must make sure that all of its own properties are initialized before it hands off up the chain.

Safety check 2

A designated initializer must delegate up to a superclass initializer before assigning a value to an inherited property. If it doesn't, the new value the designated initializer assigns will be overwritten by the superclass as part of its own initialization.

Safety check 3

A convenience initializer must delegate to another initializer before assigning a value to *any* property (including properties defined by the same class). If it doesn't, the new value the convenience initializer assigns will be overwritten by its own class's designated initializer.

Safety check 4

An initializer cannot call any instance methods, read the values of any instance properties, or refer to self as a value until after the first phase of initialization is complete.

The class instance is not fully valid until the first phase ends. Properties can only be accessed, and methods can only be called, once the class instance is known to be valid at the end of the first phase.

Here's how two-phase initialization plays out, based on the four safety checks above:

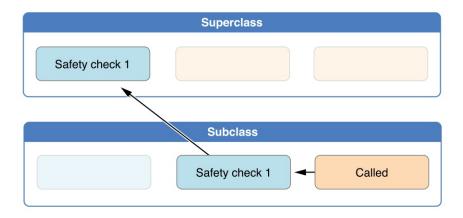
Phase 1

- A designated or convenience initializer is called on a class.
- Memory for a new instance of that class is allocated. The memory is not yet initialized.
- A designated initializer for that class confirms that all stored properties introduced by that class have a value. The memory for these stored properties is now initialized.
- The designated initializer hands off to a superclass initializer to perform the same task for its own stored properties.
- This continues up the class inheritance chain until the top of the chain is reached.
- Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance's memory is considered to be fully initialized, and phase 1 is complete.

Phase 2

- Working back down from the top of the chain, each designated initializer in the chain has the option to customize the instance further. Initializers are now able to access self and can modify its properties, call its instance methods, and so on.
- Finally, any convenience initializers in the chain have the option to customize the instance and to work with self.

Here's how phase 1 looks for an initialization call for a hypothetical subclass and superclass:



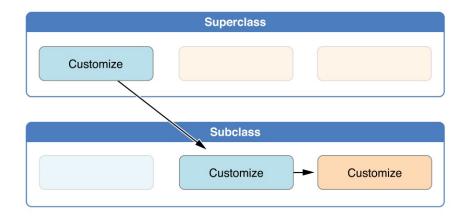
In this example, initialization begins with a call to a convenience initializer on the subclass. This convenience initializer cannot yet modify any properties. It delegates across to a designated initializer from the same class.

The designated initializer makes sure that all of the subclass's properties have a value, as per safety check 1. It then calls a designated initializer on its superclass to continue the initialization up the chain.

The superclass's designated initializer makes sure that all of the superclass properties have a value. There are no further superclasses to initialize, and so no further delegation is needed.

As soon as all properties of the superclass have an initial value, its memory is considered fully initialized, and phase 1 is complete.

Here's how phase 2 looks for the same initialization call:



The superclass's designated initializer now has an opportunity to customize the

instance further (although it does not have to).

Once the superclass's designated initializer is finished, the subclass's designated initializer can perform additional customization (although again, it does not have to).

Finally, once the subclass's designated initializer is finished, the convenience initializer that was originally called can perform additional customization.

Initializer Inheritance and Overriding

Unlike subclasses in Objective-C, Swift subclasses do not inherit their superclass initializers by default. Swift's approach prevents a situation in which a simple initializer from a superclass is inherited by a more specialized subclass and is used to create a new instance of the subclass that is not fully or correctly initialized.

NOTE

Superclass initializers *are* inherited in certain circumstances, but only when it is safe and appropriate to do so. For more information, see <u>Automatic Initializer Inheritance</u> below.

If you want a custom subclass to present one or more of the same initializers as its superclass, you can provide a custom implementation of those initializers within the subclass.

When you write a subclass initializer that matches a superclass *designated* initializer, you are effectively providing an override of that designated initializer. Therefore, you must write the override modifier before the subclass's initializer definition. This is true even if you are overriding an automatically provided default initializer, as described in <u>Default Initializers</u>.

As with an overridden property, method or subscript, the presence of the override modifier prompts Swift to check that the superclass has a matching designated initializer to be overridden, and validates that the parameters for your overriding initializer have been specified as intended.

You always write the override modifier when overriding a superclass designated initializer, even if your subclass's implementation of the initializer is a convenience initializer.

Conversely, if you write a subclass initializer that matches a superclass *convenience* initializer, that superclass convenience initializer can never be called directly by your subclass, as per the rules described above in Initializer Delegation for Class Types. Therefore, your subclass is not (strictly speaking) providing an override of the superclass initializer. As a result, you do not write the override modifier when providing a matching implementation of a superclass convenience initializer.

The example below defines a base class called <code>Vehicle</code>. This base class declares a stored property called <code>numberOfWheels</code>, with a default <code>Int</code> value of <code>0</code>. The <code>numberOfWheels</code> property is used by a computed property called <code>description</code> to create a <code>String</code> description of the vehicle's characteristics:

```
class Vehicle {
   var numberOfWheels = 0
   var description: String {
      return "\(numberOfWheels) wheel(s)"
   }
}
```

The <code>Vehicle</code> class provides a default value for its only stored property, and does not provide any custom initializers itself. As a result, it automatically receives a default initializer, as described in Default Initializers. The default initializer (when available) is always a designated initializer for a class, and can be used to create a new <code>Vehicle</code> instance with a <code>numberOfWheels</code> of <code>0</code>:

```
1 let vehicle = Vehicle()
2 print("Vehicle: \(vehicle.description)")
3 // Vehicle: 0 wheel(s)
```

The next example defines a subclass of Vehicle called Bicycle:

```
class Bicycle: Vehicle {
   override init() {
      super.init()
      numberOfWheels = 2
   }
}
```

The Bicycle subclass defines a custom designated initializer, init(). This designated initializer matches a designated initializer from the superclass of Bicycle, and so the Bicycle version of this initializer is marked with the override modifier.

The init() initializer for Bicycle starts by calling super.init(), which calls the default initializer for the Bicycle class's superclass, Vehicle. This ensures that the numberOfWheels inherited property is initialized by Vehicle before Bicycle has the opportunity to modify the property. After calling super.init(), the original value of numberOfWheels is replaced with a new value of 2.

If you create an instance of Bicycle, you can call its inherited description computed property to see how its number of wheels property has been updated:

```
1 let bicycle = Bicycle()
2 print("Bicycle: \((bicycle.description)"))
3 // Bicycle: 2 wheel(s)
```

If a subclass initializer performs no customization in phase 2 of the initialization process, and the superclass has a zero-argument designated initializer, you can omit a call to <code>super.init()</code> after assigning values to all of the subclass's stored properties.

This example defines another subclass of <code>Vehicle</code>, called <code>Hoverboard</code>. In its initializer, the <code>Hoverboard</code> class sets only its <code>color</code> property. Instead of making an explicit call to <code>super.init()</code>, this initializer relies on an implicit call to its

superclass's initializer to complete the process.

```
1
    class Hoverboard: Vehicle {
2
         var color: String
         init(color: String) {
3
             self.color = color
4
             // super.init() implicitly called here
5
6
         }
         override var description: String {
7
             return "\(super.description) in a beautiful \(color)"
8
9
         }
10
     }
```

An instance of Hoverboard uses the default number of wheels supplied by the Vehicle initializer.

```
1 let hoverboard = Hoverboard(color: "silver")
2 print("Hoverboard: \((hoverboard.description)")
3 // Hoverboard: 0 wheel(s) in a beautiful silver
```

NOTE

Subclasses can modify inherited variable properties during initialization, but can not modify inherited constant properties.

Automatic Initializer Inheritance

As mentioned above, subclasses do not inherit their superclass initializers by default. However, superclass initializers *are* automatically inherited if certain conditions are met. In practice, this means that you do not need to write initializer overrides in many common scenarios, and can inherit your superclass initializers with minimal effort whenever it is safe to do so.

Assuming that you provide default values for any new properties you introduce

in a subclass, the following two rules apply:

Rule 1

If your subclass doesn't define any designated initializers, it automatically inherits all of its superclass designated initializers.

Rule 2

If your subclass provides an implementation of *all* of its superclass designated initializers—either by inheriting them as per rule 1, or by providing a custom implementation as part of its definition—then it automatically inherits all of the superclass convenience initializers.

These rules apply even if your subclass adds further convenience initializers.

NOTE

A subclass can implement a superclass designated initializer as a subclass convenience initializer as part of satisfying rule 2.

Designated and Convenience Initializers in Action

The following example shows designated initializers, convenience initializers, and automatic initializer inheritance in action. This example defines a hierarchy of three classes called <code>Food</code>, <code>RecipeIngredient</code>, and <code>ShoppingListItem</code>, and demonstrates how their initializers interact.

The base class in the hierarchy is called Food, which is a simple class to encapsulate the name of a foodstuff. The Food class introduces a single String property called name and provides two initializers for creating Food instances:

```
class Food {
var name: String
init(name: String) {
self.name = name
```

```
5  }
6  convenience init() {
7  self.init(name: "[Unnamed]")
8  }
9 }
```

The figure below shows the initializer chain for the Food class:



Classes do not have a default memberwise initializer, and so the Food class provides a designated initializer that takes a single argument called name. This initializer can be used to create a new Food instance with a specific name:

```
1 let namedMeat = Food(name: "Bacon")
2 // namedMeat's name is "Bacon"
```

The init(name: String) initializer from the Food class is provided as a *designated* initializer, because it ensures that all stored properties of a new Food instance are fully initialized. The Food class does not have a superclass, and so the init(name: String) initializer does not need to call super.init() to complete its initialization.

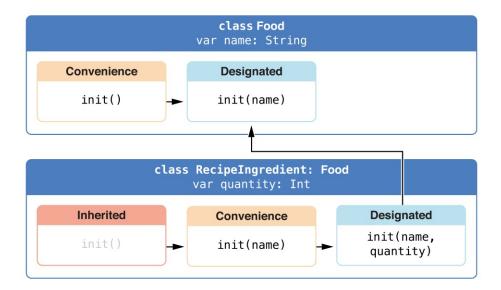
The Food class also provides a *convenience* initializer, init(), with no arguments. The init() initializer provides a default placeholder name for a new food by delegating across to the Food class's init(name: String) with a name value of [Unnamed]:

```
1 let mysteryMeat = Food()
2 // mysteryMeat's name is "[Unnamed]"
```

The second class in the hierarchy is a subclass of Food called RecipeIngredient. The RecipeIngredient class models an ingredient in a cooking recipe. It introduces an Int property called quantity (in addition to the name property it inherits from Food) and defines two initializers for creating RecipeIngredient instances:

```
class RecipeIngredient: Food {
1
2
         var quantity: Int
         init(name: String, quantity: Int) {
3
             self.quantity = quantity
4
             super init(name: name)
5
         }
6
7
         override convenience init(name: String) {
             self.init(name: name, quantity: 1)
8
         }
9
10
     }
```

The figure below shows the initializer chain for the RecipeIngredient class:



The RecipeIngredient class has a single designated initializer, init(name: String, quantity: Int), which can be used to populate all of the properties of a new RecipeIngredient instance. This initializer starts by assigning the passed

quantity argument to the quantity property, which is the only new property introduced by RecipeIngredient. After doing so, the initializer delegates up to the init(name: String) initializer of the Food class. This process satisfies safety check 1 from Two-Phase Initialization above.

RecipeIngredient also defines a convenience initializer, init(name: String), which is used to create a RecipeIngredient instance by name alone. This convenience initializer assumes a quantity of 1 for any RecipeIngredient instance that is created without an explicit quantity. The definition of this convenience initializer makes RecipeIngredient instances quicker and more convenient to create, and avoids code duplication when creating several single-quantity RecipeIngredient instances. This convenience initializer simply delegates across to the class's designated initializer, passing in a quantity value of 1.

The init(name: String) convenience initializer provided by RecipeIngredient takes the same parameters as the init(name: String) designated initializer from Food. Because this convenience initializer overrides a designated initializer from its superclass, it must be marked with the override modifier (as described in Initializer Inheritance and Overriding).

Even though RecipeIngredient provides the init(name: String) initializer as a convenience initializer, RecipeIngredient has nonetheless provided an implementation of all of its superclass's designated initializers. Therefore, RecipeIngredient automatically inherits all of its superclass's convenience initializers too.

In this example, the superclass for RecipeIngredient is Food, which has a single convenience initializer called init(). This initializer is therefore inherited by RecipeIngredient. The inherited version of init() functions in exactly the same way as the Food version, except that it delegates to the RecipeIngredient version of init(name: String) rather than the Food version.

All three of these initializers can be used to create new RecipeIngredient instances:

```
1 let oneMysteryItem = RecipeIngredient()
```

```
2 let oneBacon = RecipeIngredient(name: "Bacon")
3 let sixEggs = RecipeIngredient(name: "Eggs", quantity: 6)
```

The third and final class in the hierarchy is a subclass of RecipeIngredient called ShoppingListItem. The ShoppingListItem class models a recipe ingredient as it appears in a shopping list.

Every item in the shopping list starts out as "unpurchased". To represent this fact, ShoppingListItem introduces a Boolean property called purchased, with a default value of false. ShoppingListItem also adds a computed description property, which provides a textual description of a ShoppingListItem instance:

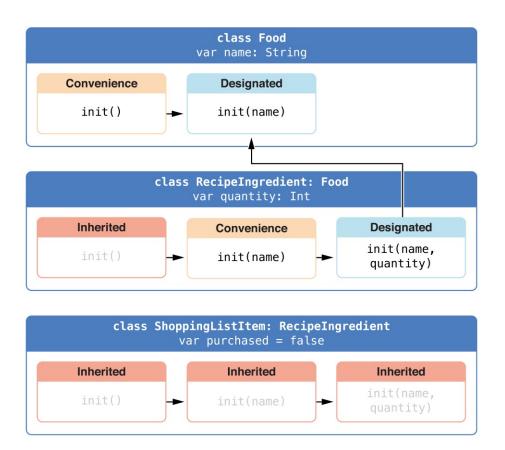
```
class ShoppingListItem: RecipeIngredient {
1
2
        var purchased = false
3
        var description: String {
            var output = "\(quantity) x \(name)"
4
            output += purchased ? " ✓" : " X"
5
            return output
6
7
        }
8
   }
```

NOTE

ShoppingListItem does not define an initializer to provide an initial value for purchased, because items in a shopping list (as modeled here) always start out unpurchased.

Because it provides a default value for all of the properties it introduces and does not define any initializers itself, ShoppingListItem automatically inherits *all* of the designated and convenience initializers from its superclass.

The figure below shows the overall initializer chain for all three classes:



You can use all three of the inherited initializers to create a new ShoppingListItem instance:

```
1
    var breakfastList = [
 2
         ShoppingListItem(),
 3
         ShoppingListItem(name: "Bacon"),
         ShoppingListItem(name: "Eggs", quantity: 6),
 4
 5
    ]
    breakfastList[0].name = "Orange juice"
 6
    breakfastList[0].purchased = true
 7
    for item in breakfastList {
8
         print(item.description)
 9
10
     }
    // 1 x Orange juice ✔
11
    // 1 x Bacon X
12
```

Here, a new array called breakfastList is created from an array literal containing three new ShoppingListItem instances. The type of the array is inferred to be [ShoppingListItem]. After the array is created, the name of the ShoppingListItem at the start of the array is changed from "[Unnamed]" to "Orange juice" and it is marked as having been purchased. Printing the description of each item in the array shows that their default states have been set as expected.

Failable Initializers

It is sometimes useful to define a class, structure, or enumeration for which initialization can fail. This failure might be triggered by invalid initialization parameter values, the absence of a required external resource, or some other condition that prevents initialization from succeeding.

To cope with initialization conditions that can fail, define one or more failable initializers as part of a class, structure, or enumeration definition. You write a failable initializer by placing a question mark after the init keyword (init?).

NOTE

You cannot define a failable and a nonfailable initializer with the same parameter types and names.

A failable initializer creates an *optional* value of the type it initializes. You write return nil within a failable initializer to indicate a point at which initialization failure can be triggered.

NOTE

Strictly speaking, initializers do not return a value. Rather, their role is to ensure that self is fully and correctly initialized by the time that initialization ends. Although you write return nil to trigger an initialization failure, you do not use the return keyword to indicate initialization success.

For instance, failable initializers are implemented for numeric type conversions.

To ensure conversion between numeric types maintains the value exactly, use the <code>init(exactly:)</code> initializer. If the type conversion cannot maintain the value, the initializer fails.

```
1
    let wholeNumber: Double = 12345.0
2
    let pi = 3.14159
3
    if let valueMaintained = Int(exactly: wholeNumber) {
4
        print("\(wholeNumber) conversion to Int maintains value of
5
          \(valueMaintained)")
6
    }
7
    // Prints "12345.0 conversion to Int maintains value of 12345"
8
    let valueChanged = Int(exactly: pi)
10
    // valueChanged is of type Int?, not Int
11
    if valueChanged == nil {
12
        print("\(pi) conversion to Int does not maintain value")
13
    }
14
    // Prints "3.14159 conversion to Int does not maintain value"
15
```

The example below defines a structure called Animal, with a constant String property called species. The Animal structure also defines a failable initializer with a single parameter called species. This initializer checks if the species value passed to the initializer is an empty string. If an empty string is found, an initialization failure is triggered. Otherwise, the species property's value is set, and initialization succeeds:

```
struct Animal {
let species: String
init?(species: String) {
   if species.isEmpty { return nil }
}
```

You can use this failable initializer to try to initialize a new Animal instance and to check if initialization succeeded:

If you pass an empty string value to the failable initializer's species parameter, the initializer triggers an initialization failure:

```
1 let anonymousCreature = Animal(species: "")
2 // anonymousCreature is of type Animal?, not Animal
3
4 if anonymousCreature == nil {
5    print("The anonymous creature could not be initialized")
6 }
7 // Prints "The anonymous creature could not be initialized"
```

NOTE

Checking for an empty string value (such as "" rather than "Giraffe") is not the same as checking for nil to indicate the absence of an *optional* String value. In the example above, an empty string ("") is a valid, non-optional String. However, it is not appropriate for an animal to have an empty string as the value of its species property. To model this restriction, the failable initializer triggers an initialization failure if an empty string is found.

Failable Initializers for Enumerations

You can use a failable initializer to select an appropriate enumeration case based on one or more parameters. The initializer can then fail if the provided parameters do not match an appropriate enumeration case.

The example below defines an enumeration called TemperatureUnit, with three possible states (kelvin, celsius, and fahrenheit). A failable initializer is used to find an appropriate enumeration case for a Character value representing a temperature symbol:

```
1
    enum TemperatureUnit {
         case kelvin, celsius, fahrenheit
 2
         init?(symbol: Character) {
 3
             switch symbol {
 4
             case "K":
 5
                  self = .kelvin
 6
             case "C":
 7
                  self = .celsius
 8
             case "F":
9
                  self = .fahrenheit
10
             default:
11
                  return nil
12
13
             }
14
         }
15
     }
```

You can use this failable initializer to choose an appropriate enumeration case for the three possible states and to cause initialization to fail if the parameter does not match one of these states:

```
1 let fahrenheitUnit = TemperatureUnit(symbol: "F")
2 if fahrenheitUnit != nil {
    print("This is a defined temperature unit, so
```

```
initialization succeeded.")
4
    }
    // Prints "This is a defined temperature unit, so
5
          initialization succeeded."
6
    let unknownUnit = TemperatureUnit(symbol: "X")
7
    if unknownUnit == nil {
8
        print("This is not a defined temperature unit, so
9
          initialization failed.")
10
    }
    // Prints "This is not a defined temperature unit, so
11
          initialization failed."
```

Failable Initializers for Enumerations with Raw Values

Enumerations with raw values automatically receive a failable initializer, init? (rawValue:), that takes a parameter called rawValue of the appropriate raw-value type and selects a matching enumeration case if one is found, or triggers an initialization failure if no matching value exists.

You can rewrite the TemperatureUnit example from above to use raw values of type Character and to take advantage of the init?(rawValue:) initializer:

```
1 enum TemperatureUnit: Character {
2    case kelvin = "K", celsius = "C", fahrenheit = "F"
3  }
4  
5  let fahrenheitUnit = TemperatureUnit(rawValue: "F")
6  if fahrenheitUnit != nil {
7    print("This is a defined temperature unit, so
    initialization succeeded.")
```

```
8
    }
    // Prints "This is a defined temperature unit, so
          initialization succeeded."
10
    let unknownUnit = TemperatureUnit(rawValue: "X")
11
    if unknownUnit == nil {
12
        print("This is not a defined temperature unit, so
13
          initialization failed.")
14
    }
    // Prints "This is not a defined temperature unit, so
15
          initialization failed."
```

Propagation of Initialization Failure

A failable initializer of a class, structure, or enumeration can delegate across to another failable initializer from the same class, structure, or enumeration. Similarly, a subclass failable initializer can delegate up to a superclass failable initializer.

In either case, if you delegate to another initializer that causes initialization to fail, the entire initialization process fails immediately, and no further initialization code is executed.

```
NOTE
```

A failable initializer can also delegate to a nonfailable initializer. Use this approach if you need to add a potential failure state to an existing initialization process that does not otherwise fail.

The example below defines a subclass of Product called CartItem. The CartItem class models an item in an online shopping cart. CartItem introduces a stored constant property called quantity and ensures that this property always has a value of at least 1:

```
class Product {
```

```
2
         let name: String
         init?(name: String) {
 3
             if name.isEmpty { return nil }
 4
             self.name = name
 5
         }
 6
 7
    }
 8
    class CartItem: Product {
9
         let quantity: Int
10
         init?(name: String, quantity: Int) {
11
             if quantity < 1 { return nil }</pre>
12
             self.quantity = quantity
13
14
             super.init(name: name)
15
         }
16
     }
```

The failable initializer for CartItem starts by validating that it has received a quantity value of 1 or more. If the quantity is invalid, the entire initialization process fails immediately and no further initialization code is executed. Likewise, the failable initializer for Product checks the name value, and the initializer process fails immediately if name is the empty string.

If you create a CartItem instance with a nonempty name and a quantity of ${\tt 1}$ or more, initialization succeeds:

```
if let twoSocks = CartItem(name: "sock", quantity: 2) {
   print("Item: \((twoSocks.name), quantity: \\
        (twoSocks.quantity)")
}
// Prints "Item: sock, quantity: 2"
```

If you try to create a CartItem instance with a quantity value of 0, the CartItem

initializer causes initialization to fail:

```
if let zeroShirts = CartItem(name: "shirt", quantity: 0) {
   print("Item: \(zeroShirts.name), quantity: \\
        (zeroShirts.quantity)")
} else {
   print("Unable to initialize zero shirts")
}
// Prints "Unable to initialize zero shirts"
```

Similarly, if you try to create a CartItem instance with an empty name value, the superclass Product initializer causes initialization to fail:

```
if let oneUnnamed = CartItem(name: "", quantity: 1) {
   print("Item: \(oneUnnamed.name), quantity: \\
        (oneUnnamed.quantity)")
} else {
   print("Unable to initialize one unnamed product")
}
// Prints "Unable to initialize one unnamed product"
```

Overriding a Failable Initializer

You can override a superclass failable initializer in a subclass, just like any other initializer. Alternatively, you can override a superclass failable initializer with a subclass *nonfailable* initializer. This enables you to define a subclass for which initialization cannot fail, even though initialization of the superclass is allowed to fail.

Note that if you override a failable superclass initializer with a nonfailable subclass initializer, the only way to delegate up to the superclass initializer is to force-unwrap the result of the failable superclass initializer.

The example below defines a class called <code>Document</code>. This class models a document that can be initialized with a <code>name</code> property that is either a nonempty string value or <code>nil</code>, but cannot be an empty string:

```
1
    class Document {
         var name: String?
2
         // this initializer creates a document with a nil name
3
           value
         init() {}
4
5
         // this initializer creates a document with a nonempty name
           value
         init?(name: String) {
6
7
             if name.isEmpty { return nil }
8
             self.name = name
9
         }
10
     }
```

The next example defines a subclass of Document called

AutomaticallyNamedDocument. The AutomaticallyNamedDocument subclass overrides both of the designated initializers introduced by Document. These overrides ensure that an AutomaticallyNamedDocument instance has an initial name value of "[Untitled]" if the instance is initialized without a name, or if an empty string is passed to the init(name:) initializer:

```
class AutomaticallyNamedDocument: Document {
    override init() {
        super.init()
        self.name = "[Untitled]"
    }
}
```

```
6
         override init(name: String) {
             super.init()
 7
 8
             if name isEmpty {
                  self.name = "[Untitled]"
 9
             } else {
10
                  self.name = name
11
12
             }
         }
13
14
     }
```

The AutomaticallyNamedDocument overrides its superclass's failable init? (name:) initializer with a nonfailable init(name:) initializer. Because AutomaticallyNamedDocument copes with the empty string case in a different way than its superclass, its initializer does not need to fail, and so it provides a nonfailable version of the initializer instead.

You can use forced unwrapping in an initializer to call a failable initializer from the superclass as part of the implementation of a subclass's nonfailable initializer. For example, the UntitledDocument subclass below is always named "[Untitled]", and it uses the failable init(name:) initializer from its superclass during initialization.

```
class UntitledDocument: Document {
   override init() {
       super.init(name: "[Untitled]")!
   }
}
```

In this case, if the init(name:) initializer of the superclass were ever called with an empty string as the name, the forced unwrapping operation would result in a runtime error. However, because it's called with a string constant, you can see that the initializer won't fail, so no runtime error can occur in this case.

The init! Failable Initializer

You typically define a failable initializer that creates an optional instance of the appropriate type by placing a question mark after the init keyword (init?). Alternatively, you can define a failable initializer that creates an implicitly unwrapped optional instance of the appropriate type. Do this by placing an exclamation mark after the init keyword (init!) instead of a question mark.

You can delegate from init? to init! and vice versa, and you can override init? with init! and vice versa. You can also delegate from init to init!, although doing so will trigger an assertion if the init! initializer causes initialization to fail.

Required Initializers

Write the required modifier before the definition of a class initializer to indicate that every subclass of the class must implement that initializer:

```
class SomeClass {
   required init() {
        // initializer implementation goes here
}
```

You must also write the required modifier before every subclass implementation of a required initializer, to indicate that the initializer requirement applies to further subclasses in the chain. You do not write the override modifier when overriding a required designated initializer:

```
class SomeSubclass: SomeClass {
   required init() {
      // subclass implementation of the required initializer
      goes here
```

```
4 }
5 }
```

NOTE

You do not have to provide an explicit implementation of a required initializer if you can satisfy the requirement with an inherited initializer.

Setting a Default Property Value with a Closure or Function

If a stored property's default value requires some customization or setup, you can use a closure or global function to provide a customized default value for that property. Whenever a new instance of the type that the property belongs to is initialized, the closure or function is called, and its return value is assigned as the property's default value.

These kinds of closures or functions typically create a temporary value of the same type as the property, tailor that value to represent the desired initial state, and then return that temporary value to be used as the property's default value.

Here's a skeleton outline of how a closure can be used to provide a default property value:

```
class SomeClass {
    let someProperty: SomeType = {
        // create a default value for someProperty inside this closure
        // someValue must be of the same type as SomeType
        return someValue
    }()
}
```

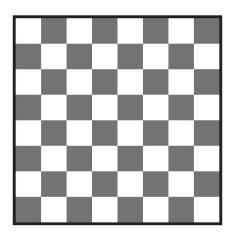
Note that the closure's end curly brace is followed by an empty pair of

parentheses. This tells Swift to execute the closure immediately. If you omit these parentheses, you are trying to assign the closure itself to the property, and not the return value of the closure.

NOTE

If you use a closure to initialize a property, remember that the rest of the instance has not yet been initialized at the point that the closure is executed. This means that you cannot access any other property values from within your closure, even if those properties have default values. You also cannot use the implicit Self property, or call any of the instance's methods.

The example below defines a structure called Chessboard, which models a board for the game of chess. Chess is played on an 8×8 board, with alternating black and white squares.



To represent this game board, the Chessboard structure has a single property called boardColors, which is an array of 64 Bool values. A value of true in the array represents a black square and a value of false represents a white square. The first item in the array represents the top left square on the board and the last item in the array represents the bottom right square on the board.

The boardColors array is initialized with a closure to set up its color values:

```
struct Chessboard {

let boardColors: [Bool] = {

var temporaryBoard = [Bool]()

var isBlack = false
```

```
5
             for i in 1...8 {
                 for j in 1...8 {
6
                     temporaryBoard.append(isBlack)
7
                     isBlack = !isBlack
8
9
                 isBlack = !isBlack
10
             }
11
             return temporaryBoard
12
         }()
13
         func squareIsBlackAt(row: Int, column: Int) -> Bool {
14
             return boardColors[(row * 8) + column]
15
16
         }
17
     }
```

Whenever a new Chessboard instance is created, the closure is executed, and the default value of boardColors is calculated and returned. The closure in the example above calculates and sets the appropriate color for each square on the board in a temporary array called temporaryBoard, and returns this temporary array as the closure's return value once its setup is complete. The returned array value is stored in boardColors and can be queried with the squareIsBlackAt(row:column:) utility function:

```
1 let board = Chessboard()
2 print(board.squareIsBlackAt(row: 0, column: 1))
3 // Prints "true"
4 print(board.squareIsBlackAt(row: 7, column: 7))
5 // Prints "false"
```

Deinitialization

A *deinitializer* is called immediately before a class instance is deallocated. You write deinitializers with the deinit keyword, similar to how initializers are written with the init keyword. Deinitializers are only available on class types.

How Deinitialization Works

Swift automatically deallocates your instances when they are no longer needed, to free up resources. Swift handles the memory management of instances through *automatic reference counting (ARC)*, as described in <u>Automatic Reference Counting</u>. Typically you don't need to perform manual cleanup when your instances are deallocated. However, when you are working with your own resources, you might need to perform some additional cleanup yourself. For example, if you create a custom class to open a file and write some data to it, you might need to close the file before the class instance is deallocated.

Class definitions can have at most one deinitializer per class. The deinitializer does not take any parameters and is written without parentheses:

```
deinit {
    // perform the deinitialization
}
```

Deinitializers are called automatically, just before instance deallocation takes place. You are not allowed to call a deinitializer yourself. Superclass deinitializers are inherited by their subclasses, and the superclass deinitializer is called automatically at the end of a subclass deinitializer implementation. Superclass deinitializers are always called, even if a subclass does not provide its own deinitializer.

Because an instance is not deallocated until after its deinitializer is called, a deinitializer can access all properties of the instance it is called on and can

modify its behavior based on those properties (such as looking up the name of a file that needs to be closed).

Deinitializers in Action

Here's an example of a deinitializer in action. This example defines two new types, Bank and Player, for a simple game. The Bank class manages a made-up currency, which can never have more than 10,000 coins in circulation. There can only ever be one Bank in the game, and so the Bank is implemented as a class with type properties and methods to store and manage its current state:

```
1
    class Bank {
2
         static var coinsInBank = 10 000
         static func distribute(coins numberOfCoinsRequested: Int) -
3
           > Int {
             let numberOfCoinsToVend = min(numberOfCoinsRequested,
4
           coinsInBank)
             coinsInBank -= numberOfCoinsToVend
5
             return numberOfCoinsToVend
6
7
         }
8
         static func receive(coins: Int) {
             coinsInBank += coins
         }
10
11
    }
```

Bank keeps track of the current number of coins it holds with its coinsInBank property. It also offers two methods—distribute(coins:) and receive(coins:)—to handle the distribution and collection of coins.

The distribute(coins:) method checks that there are enough coins in the bank before distributing them. If there are not enough coins, Bank returns a smaller number than the number that was requested (and returns zero if no coins are left

in the bank). It returns an integer value to indicate the actual number of coins that were provided.

The receive(coins:) method simply adds the received number of coins back into the bank's coin store.

The Player class describes a player in the game. Each player has a certain number of coins stored in their purse at any time. This is represented by the player's coinsInPurse property:

```
1
    class Player {
2
         var coinsInPurse: Int
3
         init(coins: Int) {
             coinsInPurse = Bank.distribute(coins: coins)
         }
5
         func win(coins: Int) {
6
7
             coinsInPurse += Bank.distribute(coins: coins)
         }
8
         deinit {
             Bank.receive(coins: coinsInPurse)
10
         }
11
12
    }
```

Each Player instance is initialized with a starting allowance of a specified number of coins from the bank during initialization, although a Player instance may receive fewer than that number if not enough coins are available.

The Player class defines a win(coins:) method, which retrieves a certain number of coins from the bank and adds them to the player's purse. The Player class also implements a deinitializer, which is called just before a Player instance is deallocated. Here, the deinitializer simply returns all of the player's coins to the bank:

```
var playerOne: Player? = Player(coins: 100)
```

A new Player instance is created, with a request for 100 coins if they are available. This Player instance is stored in an optional Player variable called player one. An optional variable is used here, because players can leave the game at any point. The optional lets you track whether there is currently a player in the game.

Because playerOne is an optional, it is qualified with an exclamation mark (!) when its coinsInPurse property is accessed to print its default number of coins, and whenever its win(coins:) method is called:

Here, the player has won 2,000 coins. The player's purse now contains 2,100 coins, and the bank has only 7,900 coins left.

```
playerOne = nil
print("PlayerOne has left the game")

// Prints "PlayerOne has left the game"
print("The bank now has \(Bank.coinsInBank) coins")

// Prints "The bank now has 10000 coins"
```

The player has now left the game. This is indicated by setting the optional player one variable to nil, meaning "no Player instance." At the point that this happens, the player one variable's reference to the Player instance is broken. No other properties or variables are still referring to the Player instance, and so it is deallocated in order to free up its memory. Just before this happens, its deinitializer is called automatically, and its coins are returned to the bank.

Optional Chaining

Optional chaining is a process for querying and calling properties, methods, and subscripts on an optional that might currently be nil. If the optional contains a value, the property, method, or subscript call succeeds; if the optional is nil, the property, method, or subscript call returns nil. Multiple queries can be chained together, and the entire chain fails gracefully if any link in the chain is nil.

NOTE

Optional chaining in Swift is similar to messaging nil in Objective-C, but in a way that works for any type, and that can be checked for success or failure.

Optional Chaining as an Alternative to Forced Unwrapping

You specify optional chaining by placing a question mark (?) after the optional value on which you wish to call a property, method or subscript if the optional is non-nil. This is very similar to placing an exclamation mark (!) after an optional value to force the unwrapping of its value. The main difference is that optional chaining fails gracefully when the optional is nil, whereas forced unwrapping triggers a runtime error when the optional is nil.

To reflect the fact that optional chaining can be called on a nil value, the result of an optional chaining call is always an optional value, even if the property, method, or subscript you are querying returns a non-optional value. You can use this optional return value to check whether the optional chaining call was successful (the returned optional contains a value), or did not succeed due to a nil value in the chain (the returned optional value is nil).

Specifically, the result of an optional chaining call is of the same type as the expected return value, but wrapped in an optional. A property that normally returns an Int will return an Int? when accessed through optional chaining.

The next several code snippets demonstrate how optional chaining differs from

forced unwrapping and enables you to check for success.

First, two classes called Person and Residence are defined:

```
class Person {
  var residence: Residence?
}

class Residence {
  var numberOfRooms = 1
}
```

Residence instances have a single Int property called numberOfRooms, with a default value of 1. Person instances have an optional residence property of type Residence?.

If you create a new Person instance, its residence property is default initialized to nil, by virtue of being optional. In the code below, john has a residence property value of nil:

```
let john = Person()
```

If you try to access the numberOfRooms property of this person's residence, by placing an exclamation mark after residence to force the unwrapping of its value, you trigger a runtime error, because there is no residence value to unwrap:

```
1 let roomCount = john.residence!.numberOfRooms
2 // this triggers a runtime error
```

The code above succeeds when john.residence has a non-nil value and will set roomCount to an Int value containing the appropriate number of rooms. However, this code always triggers a runtime error when residence is nil, as illustrated above.

Optional chaining provides an alternative way to access the value of numberOfRooms. To use optional chaining, use a question mark in place of the exclamation mark:

```
if let roomCount = john.residence?.numberOfRooms {
   print("John's residence has \(roomCount) room(s).")
} else {
   print("Unable to retrieve the number of rooms.")
}
// Prints "Unable to retrieve the number of rooms."
```

This tells Swift to "chain" on the optional residence property and to retrieve the value of numberOfRooms if residence exists.

Because the attempt to access numberOfRooms has the potential to fail, the optional chaining attempt returns a value of type Int?, or "optional Int". When residence is nil, as in the example above, this optional Int will also be nil, to reflect the fact that it was not possible to access numberOfRooms. The optional Int is accessed through optional binding to unwrap the integer and assign the non-optional value to the roomCount variable.

Note that this is true even though numberOfRooms is a non-optional Int. The fact that it is queried through an optional chain means that the call to numberOfRooms will always return an Int? instead of an Int.

You can assign a Residence instance to john.residence, so that it no longer has a nil value:

```
john.residence = Residence()
```

john.residence now contains an actual Residence instance, rather than nil. If you try to access numberOfRooms with the same optional chaining as before, it will now return an Int? that contains the default numberOfRooms value of 1:

```
if let roomCount = john.residence?.numberOfRooms {
```

```
print("John's residence has \(roomCount) room(s).")
} else {
print("Unable to retrieve the number of rooms.")
}
// Prints "John's residence has 1 room(s)."
```

Defining Model Classes for Optional Chaining

You can use optional chaining with calls to properties, methods, and subscripts that are more than one level deep. This enables you to drill down into subproperties within complex models of interrelated types, and to check whether it is possible to access properties, methods, and subscripts on those subproperties.

The code snippets below define four model classes for use in several subsequent examples, including examples of multilevel optional chaining. These classes expand upon the Person and Residence model from above by adding a Room and Address class, with associated properties, methods, and subscripts.

The Person class is defined in the same way as before:

```
class Person {
  var residence: Residence?
}
```

The Residence class is more complex than before. This time, the Residence class defines a variable property called rooms, which is initialized with an empty array of type [Room]:

```
class Residence {
  var rooms = [Room]()
  var numberOfRooms: Int {
```

```
4
             return rooms.count
 5
         }
         subscript(i: Int) -> Room {
 6
 7
             get {
                  return rooms[i]
 8
             }
 9
             set {
10
                  rooms[i] = newValue
11
12
             }
         }
13
         func printNumberOfRooms() {
14
             print("The number of rooms is \(numberOfRooms)")
15
16
         }
17
         var address: Address?
18
     }
```

Because this version of Residence stores an array of Room instances, its numberOfRooms property is implemented as a computed property, not a stored property. The computed numberOfRooms property simply returns the value of the count property from the rooms array.

As a shortcut to accessing its rooms array, this version of Residence provides a read-write subscript that provides access to the room at the requested index in the rooms array.

This version of Residence also provides a method called printNumberOfRooms, which simply prints the number of rooms in the residence.

Finally, Residence defines an optional property called address, with a type of Address?. The Address class type for this property is defined below.

The Room class used for the rooms array is a simple class with one property called name, and an initializer to set that property to a suitable room name:

```
class Room {
   let name: String
   init(name: String) { self.name = name }
}
```

The final class in this model is called Address. This class has three optional properties of type String?. The first two properties, buildingName and buildingNumber, are alternative ways to identify a particular building as part of an address. The third property, street, is used to name the street for that address:

```
class Address {
1
2
         var buildingName: String?
3
         var buildingNumber: String?
         var street: String?
4
         func buildingIdentifier() -> String? {
5
             if let buildingNumber = buildingNumber, let street =
6
           street {
                 return "\(buildingNumber) \(street)"
7
             } else if buildingName != nil {
8
                 return buildingName
9
             } else {
10
11
                 return nil
12
             }
         }
13
14
     }
```

The Address class also provides a method called buildingIdentifier(), which has a return type of String?. This method checks the properties of the address and returns buildingName if it has a value, or buildingNumber concatenated with street if both have values, or nil otherwise.

Accessing Properties Through Optional Chaining

As demonstrated in <u>Optional Chaining as an Alternative to Forced Unwrapping</u>, you can use optional chaining to access a property on an optional value, and to check if that property access is successful.

Use the classes defined above to create a new Person instance, and try to access its numberOfRooms property as before:

```
1 let john = Person()
2 if let roomCount = john.residence?.numberOfRooms {
3    print("John's residence has \((roomCount) room(s).")
4 } else {
5    print("Unable to retrieve the number of rooms.")
6 }
7 // Prints "Unable to retrieve the number of rooms."
```

Because john.residence is nil, this optional chaining call fails in the same way as before.

You can also attempt to set a property's value through optional chaining:

```
1 let someAddress = Address()
2 someAddress.buildingNumber = "29"
3 someAddress.street = "Acacia Road"
4 john.residence?.address = someAddress
```

In this example, the attempt to set the address property of john.residence will fail, because john.residence is currently nil.

The assignment is part of the optional chaining, which means none of the code on the right-hand side of the = operator is evaluated. In the previous example, it's not easy to see that someAddress is never evaluated, because accessing a constant doesn't have any side effects. The listing below does the same assignment, but it uses a function to create the address. The function prints "Function was called"

before returning a value, which lets you see whether the right-hand side of the = operator was evaluated.

```
func createAddress() -> Address {
1
        print("Function was called.")
2
3
        let someAddress = Address()
4
        someAddress.buildingNumber = "29"
5
        someAddress.street = "Acacia Road"
7
        return someAddress
9
    }
    john.residence?.address = createAddress()
10
```

You can tell that the createAddress() function isn't called, because nothing is printed.

Calling Methods Through Optional Chaining

You can use optional chaining to call a method on an optional value, and to check whether that method call is successful. You can do this even if that method does not define a return value.

The printNumberOfRooms() method on the Residence class prints the current value of numberOfRooms. Here's how the method looks:

```
func printNumberOfRooms() {
   print("The number of rooms is \((numberOfRooms)\)")
}
```

This method does not specify a return type. However, functions and methods with no return type have an implicit return type of void, as described in

<u>Functions Without Return Values</u>. This means that they return a value of (), or an empty tuple.

If you call this method on an optional value with optional chaining, the method's return type will be <code>Void?</code>, not <code>Void</code>, because return values are always of an optional type when called through optional chaining. This enables you to use an <code>if</code> statement to check whether it was possible to call the <code>printNumberOfRooms()</code> method, even though the method does not itself define a return value. Compare the return value from the <code>printNumberOfRooms</code> call against <code>nil</code> to see if the method call was successful:

```
if john.residence?.printNumberOfRooms() != nil {
   print("It was possible to print the number of rooms.")
} else {
   print("It was not possible to print the number of rooms.")
}
// Prints "It was not possible to print the number of rooms."
```

The same is true if you attempt to set a property through optional chaining. The example above in <u>Accessing Properties Through Optional Chaining</u> attempts to set an address value for john.residence, even though the residence property is nil. Any attempt to set a property through optional chaining returns a value of type <code>Void?</code>, which enables you to compare against nil to see if the property was set successfully:

```
if (john.residence?.address = someAddress) != nil {
   print("It was possible to set the address.")
} else {
   print("It was not possible to set the address.")
}
// Prints "It was not possible to set the address."
```

Accessing Subscripts Through Optional Chaining

You can use optional chaining to try to retrieve and set a value from a subscript on an optional value, and to check whether that subscript call is successful.

```
NOTE
```

When you access a subscript on an optional value through optional chaining, you place the question mark *before* the subscript's brackets, not after. The optional chaining question mark always follows immediately after the part of the expression that is optional.

The example below tries to retrieve the name of the first room in the rooms array of the john.residence property using the subscript defined on the Residence class. Because john.residence is currently nil, the subscript call fails:

```
if let firstRoomName = john.residence?[0].name {
   print("The first room name is \(firstRoomName\).")
} else {
   print("Unable to retrieve the first room name.")
}
// Prints "Unable to retrieve the first room name."
```

The optional chaining question mark in this subscript call is placed immediately after john.residence, before the subscript brackets, because john.residence is the optional value on which optional chaining is being attempted.

Similarly, you can try to set a new value through a subscript with optional chaining:

```
john.residence?[0] = Room(name: "Bathroom")
```

This subscript setting attempt also fails, because residence is currently nil.

If you create and assign an actual Residence instance to john.residence, with one or more Room instances in its rooms array, you can use the Residence subscript to access the actual items in the rooms array through optional chaining:

```
let johnsHouse = Residence()
1
2
    johnsHouse.rooms.append(Room(name: "Living Room"))
3
    johnsHouse.rooms.append(Room(name: "Kitchen"))
    john.residence = johnsHouse
4
5
    if let firstRoomName = john.residence?[0].name {
6
7
        print("The first room name is \((firstRoomName).")
    } else {
8
        print("Unable to retrieve the first room name.")
9
10
    }
    // Prints "The first room name is Living Room."
11
```

Accessing Subscripts of Optional Type

If a subscript returns a value of optional type—such as the key subscript of Swift's Dictionary type—place a question mark *after* the subscript's closing bracket to chain on its optional return value:

```
var testScores = ["Dave": [86, 82, 84], "Bev": [79, 94, 81]]

testScores["Dave"]?[0] = 91

testScores["Bev"]?[0] += 1

testScores["Brian"]?[0] = 72

// the "Dave" array is now [91, 82, 84] and the "Bev" array is now [80, 94, 81]
```

The example above defines a dictionary called testScores, which contains two key-value pairs that map a String key to an array of Int values. The example uses optional chaining to set the first item in the "Dave" array to 91; to increment the first item in the "Bev" array by 1; and to try to set the first item in an array for a key of "Brian". The first two calls succeed, because the testScores dictionary contains keys for "Dave" and "Bev". The third call fails, because the testScores dictionary does not contain a key for "Brian".

Linking Multiple Levels of Chaining

You can link together multiple levels of optional chaining to drill down to properties, methods, and subscripts deeper within a model. However, multiple levels of optional chaining do not add more levels of optionality to the returned value.

To put it another way:

- If the type you are trying to retrieve is not optional, it will become optional because of the optional chaining.
- If the type you are trying to retrieve is *already* optional, it will not become *more* optional because of the chaining.

Therefore:

- If you try to retrieve an Int value through optional chaining, an Int? is always returned, no matter how many levels of chaining are used.
- Similarly, if you try to retrieve an Int? value through optional chaining, an Int? is always returned, no matter how many levels of chaining are used.

The example below tries to access the street property of the address property of the residence property of john. There are *two* levels of optional chaining in use here, to chain through the residence and address properties, both of which are of optional type:

```
if let johnsStreet = john.residence?.address?.street {
   print("John's street name is \(johnsStreet).")
} else {
   print("Unable to retrieve the address.")
}
// Prints "Unable to retrieve the address."
```

The value of john.residence currently contains a valid Residence instance. However, the value of john.residence.address is currently nil. Because of this,

the call to john.residence?.address?.street fails.

Note that in the example above, you are trying to retrieve the value of the street property. The type of this property is String?. The return value of john.residence?.address?.street is therefore also String?, even though two levels of optional chaining are applied in addition to the underlying optional type of the property.

If you set an actual Address instance as the value for john.residence.address, and set an actual value for the address's street property, you can access the value of the street property through multilevel optional chaining:

```
1
    let johnsAddress = Address()
    johnsAddress.buildingName = "The Larches"
2
    johnsAddress.street = "Laurel Street"
3
    john.residence?.address = johnsAddress
5
    if let johnsStreet = john.residence?.address?.street {
6
7
        print("John's street name is \(johnsStreet).")
    } else {
8
        print("Unable to retrieve the address.")
9
10
    // Prints "John's street name is Laurel Street."
11
```

In this example, the attempt to set the address property of john.residence will succeed, because the value of john.residence currently contains a valid Residence instance.

Chaining on Methods with Optional Return Values

The previous example shows how to retrieve the value of a property of optional type through optional chaining. You can also use optional chaining to call a

method that returns a value of optional type, and to chain on that method's return value if needed.

The example below calls the Address class's buildingIdentifier() method through optional chaining. This method returns a value of type String?. As described above, the ultimate return type of this method call after optional chaining is also String?:

If you want to perform further optional chaining on this method's return value, place the optional chaining question mark *after* the method's parentheses:

```
if let beginsWithThe =
1
2
          john.residence?.address?.buildingIdentifier()?.hasPrefix("The
        if beginsWithThe {
3
            print("John's building identifier begins with
4
          \"The\".")
        } else {
5
            print("John's building identifier does not begin with
6
          \"The\".")
        }
7
8
   }
   // Prints "John's building identifier begins with "The"."
9
```

NOTE

In the example above, you place the optional chaining question mark after the parentheses, because the optional value you are chaining on is the buildingIdentifier() method's return value, and not the buildingIdentifier() method itself.

Error Handling

Error handling is the process of responding to and recovering from error conditions in your program. Swift provides first-class support for throwing, catching, propagating, and manipulating recoverable errors at runtime.

Some operations aren't guaranteed to always complete execution or produce a useful output. Optionals are used to represent the absence of a value, but when an operation fails, it's often useful to understand what caused the failure, so that your code can respond accordingly.

As an example, consider the task of reading and processing data from a file on disk. There are a number of ways this task can fail, including the file not existing at the specified path, the file not having read permissions, or the file not being encoded in a compatible format. Distinguishing among these different situations allows a program to resolve some errors and to communicate to the user any errors it can't resolve.

NOTE

Error handling in Swift interoperates with error handling patterns that use the NSError class in Cocoa and Objective-C. For more information about this class, see <u>Handling Cocoa Errors in Swift</u>.

Representing and Throwing Errors

In Swift, errors are represented by values of types that conform to the Error protocol. This empty protocol indicates that a type can be used for error handling.

Swift enumerations are particularly well suited to modeling a group of related error conditions, with associated values allowing for additional information about the nature of an error to be communicated. For example, here's how you might represent the error conditions of operating a vending machine inside a game:

```
enum VendingMachineError: Error {
    case invalidSelection
    case insufficientFunds(coinsNeeded: Int)
    case outOfStock
}
```

Throwing an error lets you indicate that something unexpected happened and the normal flow of execution can't continue. You use a throw statement to throw an error. For example, the following code throws an error to indicate that five additional coins are needed by the vending machine:

```
throw VendingMachineError.insufficientFunds(coinsNeeded: 5)
```

Handling Errors

When an error is thrown, some surrounding piece of code must be responsible for handling the error—for example, by correcting the problem, trying an alternative approach, or informing the user of the failure.

There are four ways to handle errors in Swift. You can propagate the error from a function to the code that calls that function, handle the error using a do-catch statement, handle the error as an optional value, or assert that the error will not occur. Each approach is described in a section below.

When a function throws an error, it changes the flow of your program, so it's important that you can quickly identify places in your code that can throw errors. To identify these places in your code, write the try keyword—or the try? or try! variation—before a piece of code that calls a function, method, or initializer that can throw an error. These keywords are described in the sections below.

NOTE

Error handling in Swift resembles exception handling in other languages, with the use of the try, catch and throw keywords. Unlike exception handling in many languages—including Objective-C—error handling in Swift does not involve unwinding the call stack, a process that can be

computationally expensive. As such, the performance characteristics of a throw statement are comparable to those of a return statement.

Propagating Errors Using Throwing Functions

To indicate that a function, method, or initializer can throw an error, you write the throws keyword in the function's declaration after its parameters. A function marked with throws is called a *throwing function*. If the function specifies a return type, you write the throws keyword before the return arrow (->).

```
func canThrowErrors() throws -> String

func cannotThrowErrors() -> String
```

A throwing function propagates errors that are thrown inside of it to the scope from which it's called.

NOTE

Only throwing functions can propagate errors. Any errors thrown inside a nonthrowing function must be handled inside the function.

In the example below, the VendingMachine class has a <code>vend(itemNamed:)</code> method that throws an appropriate <code>VendingMachineError</code> if the requested item is not available, is out of stock, or has a cost that exceeds the current deposited amount:

```
struct Item {
var price: Int
var count: Int
}

class VendingMachine {
var inventory = [
```

```
8
             "Candy Bar": Item(price: 12, count: 7),
             "Chips": Item(price: 10, count: 4),
9
             "Pretzels": Item(price: 7, count: 11)
10
11
         1
12
         var coinsDeposited = 0
13
         func vend(itemNamed name: String) throws {
14
15
             guard let item = inventory[name] else {
                 throw VendingMachineError.invalidSelection
16
17
             }
18
             guard item.count > 0 else {
19
20
                 throw VendingMachineError.outOfStock
21
             }
22
23
             guard item.price <= coinsDeposited else {</pre>
24
                 throw
          VendingMachineError.insufficientFunds(coinsNeeded:
          item.price - coinsDeposited)
25
             }
26
             coinsDeposited -= item.price
27
28
             var newItem = item
29
30
             newItem.count -= 1
31
             inventory[name] = newItem
32
33
             print("Dispensing \(name)")
34
        }
35
    }
```

The implementation of the <code>vend(itemNamed:)</code> method uses <code>guard</code> statements to exit the method early and throw appropriate errors if any of the requirements for purchasing a snack aren't met. Because a <code>throw</code> statement immediately transfers program control, an item will be vended only if all of these requirements are met.

Because the <code>vend(itemNamed:)</code> method propagates any errors it throws, any code that calls this method must either handle the errors—using a <code>do-catch</code> statement, <code>try?</code>, or <code>try!</code>—or continue to propagate them. For example, the <code>buyFavoriteSnack(person:vendingMachine:)</code> in the example below is also a throwing function, and any errors that the <code>vend(itemNamed:)</code> method throws will propagate up to the point where the <code>buyFavoriteSnack(person:vendingMachine:)</code> function is called.

```
1
   let favoriteSnacks = [
        "Alice": "Chips",
2
        "Bob": "Licorice",
3
        "Eve": "Pretzels",
4
5
   ]
   func buyFavoriteSnack(person: String, vendingMachine:
6
          VendingMachine) throws {
7
        let snackName = favoriteSnacks[person] ?? "Candy Bar"
8
        try vendingMachine.vend(itemNamed: snackName)
9
   }
```

In this example, the buyFavoriteSnack(person: vendingMachine:) function looks up a given person's favorite snack and tries to buy it for them by calling the vend(itemNamed:) method. Because the vend(itemNamed:) method can throw an error, it's called with the try keyword in front of it.

Throwing initializers can propagate errors in the same way as throwing functions. For example, the initializer for the PurchasedSnack structure in the listing below calls a throwing function as part of the initialization process, and it handles any errors that it encounters by propagating them to its caller.

```
struct PurchasedSnack {
   let name: String
   init(name: String, vendingMachine: VendingMachine) throws {
      try vendingMachine.vend(itemNamed: name)
      self.name = name
   }
}
```

Handling Errors Using Do-Catch

You use a do-catch statement to handle errors by running a block of code. If an error is thrown by the code in the do clause, it is matched against the catch clauses to determine which one of them can handle the error.

Here is the general form of a do-catch statement:

```
do {
    try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
} catch {
    statements
}
```

You write a pattern after catch to indicate what errors that clause can handle. If a catch clause doesn't have a pattern, the clause matches any error and binds the error to a local constant named error. For more information about pattern matching, see Patterns.

For example, the following code matches against all three cases of the

VendingMachineError enumeration.

```
1
    var vendingMachine = VendingMachine()
    vendingMachine.coinsDeposited = 8
2
3
    do {
        try buyFavoriteSnack(person: "Alice", vendingMachine:
4
          vendingMachine)
        print("Success! Yum.")
5
    } catch VendingMachineError invalidSelection {
6
        print("Invalid Selection.")
7
8
    } catch VendingMachineError.outOfStock {
        print("Out of Stock.")
9
     } catch VendingMachineError.insufficientFunds(let coinsNeeded)
10
          {
11
         print("Insufficient funds. Please insert an additional \
           (coinsNeeded) coins.")
    } catch {
12
        print("Unexpected error: \(error).")
13
14
    }
15
    // Prints "Insufficient funds. Please insert an additional 2
          coins."
```

In the above example, the <code>buyFavoriteSnack(person:vendingMachine:)</code> function is called in a try expression, because it can throw an error. If an error is thrown, execution immediately transfers to the <code>catch</code> clauses, which decide whether to allow propagation to continue. If no pattern is matched, the error gets caught by the final <code>catch</code> clause and is bound to a local error constant. If no error is thrown, the remaining statements in the <code>do</code> statement are executed.

The catch clauses don't have to handle every possible error that the code in the do clause can throw. If none of the catch clauses handle the error, the error propagates to the surrounding scope. However, the propagated error must be

handled by *some* surrounding scope. In a nonthrowing function, an enclosing docatch clause must handle the error. In a throwing function, either an enclosing docatch clause or the caller must handle the error. If the error propagates to the top-level scope without being handled, you'll get a runtime error.

For example, the above example can be written so any error that isn't a <code>VendingMachineError</code> is instead caught by the calling function:

```
func nourish(with item: String) throws {
1
        do {
2
             try vendingMachine.vend(itemNamed: item)
3
         } catch is VendingMachineError {
             print("Invalid selection, out of stock, or not enough
5
          money.")
        }
6
7
    }
8
    do {
         try nourish(with: "Beet-Flavored Chips")
10
     } catch {
11
         print("Unexpected non-vending-machine-related error: \
12
           (error)")
13
     // Prints "Invalid selection, out of stock, or not enough
14
           money."
```

In the nourish(with:) function, if vend(itemNamed:) throws an error that's one of the cases of the VendingMachineError enumeration, nourish(with:) handles the error by printing a message. Otherwise, nourish(with:) propagates the error to its call site. The error is then caught by the general catch clause.

Converting Errors to Optional Values

You use try? to handle an error by converting it to an optional value. If an error is thrown while evaluating the try? expression, the value of the expression is nil. For example, in the following code x and y have the same value and behavior:

```
1
    func someThrowingFunction() throws -> Int {
         // ...
2
3
    }
4
5
    let x = try? someThrowingFunction()
6
    let y: Int?
7
    do {
8
         y = try someThrowingFunction()
9
10
     } catch {
         y = nil
11
12
     }
```

If someThrowingFunction() throws an error, the value of x and y is nil. Otherwise, the value of x and y is the value that the function returned. Note that x and y are an optional of whatever type someThrowingFunction() returns. Here the function returns an integer, so x and y are optional integers.

Using try? lets you write concise error handling code when you want to handle all errors in the same way. For example, the following code uses several approaches to fetch data, or returns nil if all of the approaches fail.

```
func fetchData() -> Data? {
   if let data = try? fetchDataFromDisk() { return data }
   if let data = try? fetchDataFromServer() { return data }
   return nil
}
```

Disabling Error Propagation

Sometimes you know a throwing function or method won't, in fact, throw an error at runtime. On those occasions, you can write try! before the expression to disable error propagation and wrap the call in a runtime assertion that no error will be thrown. If an error actually is thrown, you'll get a runtime error.

For example, the following code uses a <code>loadImage(atPath:)</code> function, which loads the image resource at a given path or throws an error if the image can't be loaded. In this case, because the image is shipped with the application, no error will be thrown at runtime, so it is appropriate to disable error propagation.

Specifying Cleanup Actions

You use a defer statement to execute a set of statements just before code execution leaves the current block of code. This statement lets you do any necessary cleanup that should be performed regardless of *how* execution leaves the current block of code—whether it leaves because an error was thrown or because of a statement such as return or break. For example, you can use a defer statement to ensure that file descriptors are closed and manually allocated memory is freed.

A defer statement defers execution until the current scope is exited. This statement consists of the defer keyword and the statements to be executed later. The deferred statements may not contain any code that would transfer control out of the statements, such as a break or a return statement, or by throwing an error. Deferred actions are executed in the reverse of the order that they're written in your source code. That is, the code in the first defer statement executes last, the code in the second defer statement executes second to last, and so on. The last defer statement in source code order executes first.

```
func processFile(filename: String) throws {
```

```
if exists(filename) {
 2
             let file = open(filename)
 3
             defer {
 4
                 close(file)
5
             }
 6
             while let line = try file.readline() {
 7
                 // Work with the file.
 8
9
             }
             // close(file) is called here, at the end of the
10
           scope.
         }
11
12
     }
```

The above example uses a defer statement to ensure that the $open(_:)$ function has a corresponding call to $close(_:)$.

NOTE

You can use a defer statement even when no error handling code is involved.

Type Casting

Type casting is a way to check the type of an instance, or to treat that instance as a different superclass or subclass from somewhere else in its own class hierarchy.

Type casting in Swift is implemented with the is and as operators. These two operators provide a simple and expressive way to check the type of a value or cast a value to a different type.

You can also use type casting to check whether a type conforms to a protocol, as described in Checking for Protocol Conformance.

Defining a Class Hierarchy for Type Casting

You can use type casting with a hierarchy of classes and subclasses to check the type of a particular class instance and to cast that instance to another class within the same hierarchy. The three code snippets below define a hierarchy of classes and an array containing instances of those classes, for use in an example of type casting.

The first snippet defines a new base class called MediaItem. This class provides basic functionality for any kind of item that appears in a digital media library. Specifically, it declares a name property of type String, and an init name initializer. (It is assumed that all media items, including all movies and songs, will have a name.)

```
class MediaItem {
  var name: String
  init(name: String) {
    self.name = name
}
```

```
6 }
```

The next snippet defines two subclasses of MediaItem. The first subclass, Movie, encapsulates additional information about a movie or film. It adds a director property on top of the base MediaItem class, with a corresponding initializer. The second subclass, Song, adds an artist property and initializer on top of the base class:

```
class Movie: MediaItem {
1
2
         var director: String
         init(name: String, director: String) {
3
             self.director = director
4
             super init(name: name)
5
         }
6
7
    }
8
    class Song: MediaItem {
9
10
         var artist: String
         init(name: String, artist: String) {
11
12
             self.artist = artist
13
             super.init(name: name)
14
         }
15
     }
```

The final snippet creates a constant array called library, which contains two Movie instances and three Song instances. The type of the library array is inferred by initializing it with the contents of an array literal. Swift's type checker is able to deduce that Movie and Song have a common superclass of MediaItem, and so it infers a type of [MediaItem] for the library array:

```
1 let library = [
2 Movie(name: "Casablanca", director: "Michael Curtiz"),
```

```
Song(name: "Blue Suede Shoes", artist: "Elvis Presley"),

Movie(name: "Citizen Kane", director: "Orson Welles"),

Song(name: "The One And Only", artist: "Chesney Hawkes"),

Song(name: "Never Gonna Give You Up", artist: "Rick

Astley")

// the type of "library" is inferred to be [MediaItem]
```

The items stored in library are still Movie and Song instances behind the scenes. However, if you iterate over the contents of this array, the items you receive back are typed as MediaItem, and not as Movie or Song. In order to work with them as their native type, you need to *check* their type, or *downcast* them to a different type, as described below.

Checking Type

Use the *type check operator* (is) to check whether an instance is of a certain subclass type. The type check operator returns true if the instance is of that subclass type and false if it is not.

The example below defines two variables, movieCount and songCount, which count the number of Movie and Song instances in the library array:

```
var movieCount = 0
var songCount = 0

for item in library {
   if item is Movie {
       movieCount += 1
   } else if item is Song {
       songCount += 1
```

```
9  }
10 }
11
12 print("Media library contains \((movieCount) movies and \((songCount) songs")\)
13 // Prints "Media library contains 2 movies and 3 songs"
```

This example iterates through all items in the library array. On each pass, the for-in loop sets the item constant to the next MediaItem in the array.

item is Movie returns true if the current MediaItem is a Movie instance and false if it is not. Similarly, item is Song checks whether the item is a Song instance. At the end of the for-in loop, the values of movieCount and songCount contain a count of how many MediaItem instances were found of each type.

Downcasting

A constant or variable of a certain class type may actually refer to an instance of a subclass behind the scenes. Where you believe this is the case, you can try to *downcast* to the subclass type with a *type cast operator* (as? or as!).

Because downcasting can fail, the type cast operator comes in two different forms. The conditional form, as?, returns an optional value of the type you are trying to downcast to. The forced form, as!, attempts the downcast and force-unwraps the result as a single compound action.

Use the conditional form of the type cast operator (as?) when you are not sure if the downcast will succeed. This form of the operator will always return an optional value, and the value will be nil if the downcast was not possible. This enables you to check for a successful downcast.

Use the forced form of the type cast operator (as!) only when you are sure that the downcast will always succeed. This form of the operator will trigger a runtime error if you try to downcast to an incorrect class type.

The example below iterates over each MediaItem in library, and prints an appropriate description for each item. To do this, it needs to access each item as a true Movie or Song, and not just as a MediaItem. This is necessary in order for it to be able to access the director or artist property of a Movie or Song for use in the description.

In this example, each item in the array might be a Movie, or it might be a Song. You don't know in advance which actual class to use for each item, and so it is appropriate to use the conditional form of the type cast operator (as?) to check the downcast each time through the loop:

```
1
    for item in library {
2
        if let movie = item as? Movie {
             print("Movie: \(movie.name), dir. \(movie.director)")
3
        } else if let song = item as? Song {
4
             print("Song: \(song.name), by \(song.artist)")
5
        }
6
7
    }
8
    // Movie: Casablanca, dir. Michael Curtiz
9
    // Song: Blue Suede Shoes, by Elvis Presley
10
    // Movie: Citizen Kane, dir. Orson Welles
11
12
    // Song: The One And Only, by Chesney Hawkes
    // Song: Never Gonna Give You Up, by Rick Astley
13
```

The example starts by trying to downcast the current item as a Movie. Because item is a MediaItem instance, it's possible that it *might* be a Movie; equally, it's also possible that it might be a Song, or even just a base MediaItem. Because of this uncertainty, the as? form of the type cast operator returns an *optional* value when attempting to downcast to a subclass type. The result of item as? Movie is of type Movie?, or "optional Movie".

Downcasting to Movie fails when applied to the Song instances in the library array. To cope with this, the example above uses optional binding to check

whether the optional Movie actually contains a value (that is, to find out whether the downcast succeeded.) This optional binding is written "if let movie = item as? Movie", which can be read as:

"Try to access item as a Movie. If this is successful, set a new temporary constant called movie to the value stored in the returned optional Movie."

If the downcasting succeeds, the properties of movie are then used to print a description for that Movie instance, including the name of its director. A similar principle is used to check for Song instances, and to print an appropriate description (including artist name) whenever a Song is found in the library.

NOTE

Casting does not actually modify the instance or change its values. The underlying instance remains the same; it is simply treated and accessed as an instance of the type to which it has been cast.

Type Casting for Any and AnyObject

Swift provides two special types for working with nonspecific types:

- Any can represent an instance of any type at all, including function types.
- AnyObject can represent an instance of any class type.

Use Any and AnyObject only when you explicitly need the behavior and capabilities they provide. It is always better to be specific about the types you expect to work with in your code.

Here's an example of using Any to work with a mix of different types, including function types and nonclass types. The example creates an array called things, which can store values of type Any:

```
var things = [Any]()
things.append(0)
```

```
4
    things.append(0.0)
    things.append(42)
5
    things.append(3.14159)
6
7
    things.append("hello")
    things.append((3.0, 5.0))
8
    things.append(Movie(name: "Ghostbusters", director: "Ivan
9
          Reitman"))
10
     things.append({ (name: String) -> String in "Hello, \(name)"
          })
```

The things array contains two Int values, two Double values, a String value, a tuple of type (Double, Double), the movie "Ghostbusters", and a closure expression that takes a String value and returns another String value.

To discover the specific type of a constant or variable that is known only to be of type Any or AnyObject, you can use an is or as pattern in a switch statement's cases. The example below iterates over the items in the things array and queries the type of each item with a switch statement. Several of the switch statement's cases bind their matched value to a constant of the specified type to enable its value to be printed:

```
1
    for thing in things {
         switch thing {
         case 0 as Int:
3
             print("zero as an Int")
4
5
         case 0 as Double:
             print("zero as a Double")
6
         case let someInt as Int:
7
             print("an integer value of \((someInt)")
8
         case let someDouble as Double where someDouble > 0:
9
             print("a positive double value of \((someDouble))")
10
         case is Double:
11
```

```
12
             print("some other double value that I don't want to
          print")
        case let someString as String:
13
             print("a string value of \"\(someString)\"")
14
15
        case let (x, y) as (Double, Double):
             print("an (x, y) point at (x), (y)")
16
         case let movie as Movie:
17
             print("a movie called \(movie.name), dir. \
18
          (movie.director)")
        case let stringConverter as (String) -> String:
19
             print(stringConverter("Michael"))
20
         default:
21
22
             print("something else")
23
        }
24
    }
25
26
    // zero as an Int
27
    // zero as a Double
28
    // an integer value of 42
29
    // a positive double value of 3.14159
    // a string value of "hello"
30
31
    // an (x, y) point at 3.0, 5.0
    // a movie called Ghostbusters, dir. Ivan Reitman
32
    // Hello, Michael
33
```

NOTE

The Any type represents values of any type, including optional types. Swift gives you a warning if you use an optional value where a value of type Any is expected. If you really do need to use an optional value as an Any value, you can use the as operator to explicitly cast the optional to Any, as shown below.

```
1 let optionalNumber: Int? = 3
2 things.append(optionalNumber)  // Warning
3 things.append(optionalNumber as Any) // No warning
```

Nested Types

Enumerations are often created to support a specific class or structure's functionality. Similarly, it can be convenient to define utility classes and structures purely for use within the context of a more complex type. To accomplish this, Swift enables you to define *nested types*, whereby you nest supporting enumerations, classes, and structures within the definition of the type they support.

To nest a type within another type, write its definition within the outer braces of the type it supports. Types can be nested to as many levels as are required.

Nested Types in Action

The example below defines a structure called BlackjackCard, which models a playing card as used in the game of Blackjack. The BlackjackCard structure contains two nested enumeration types called Suit and Rank.

In Blackjack, the Ace cards have a value of either one or eleven. This feature is represented by a structure called <code>values</code>, which is nested within the <code>Rank</code> enumeration:

```
struct BlackjackCard {

// nested Suit enumeration
enum Suit: Character {
    case spades = "*", hearts = "\o", diamonds = "\o", clubs
    = "*"
}

// nested Rank enumeration
```

```
9
        enum Rank: Int {
             case two = 2, three, four, five, six, seven, eight,
10
          nine, ten
             case jack, queen, king, ace
11
12
             struct Values {
                 let first: Int, second: Int?
13
14
             }
             var values: Values {
15
                 switch self {
16
                 case .ace:
17
                     return Values(first: 1, second: 11)
18
                 case jack, queen, king:
19
20
                     return Values(first: 10, second: nil)
                 default:
21
22
                     return Values(first: self.rawValue, second:
          nil)
23
                 }
24
             }
25
         }
26
27
        // BlackjackCard properties and methods
         let rank: Rank, suit: Suit
28
         var description: String {
29
             var output = "suit is \(suit.rawValue),"
30
31
             output += " value is \((rank.values.first)"
32
             if let second = rank.values.second {
                 output += " or \(second)"
33
             }
34
35
             return output
36
        }
```

The Suit enumeration describes the four common playing card suits, together with a raw Character value to represent their symbol.

The Rank enumeration describes the thirteen possible playing card ranks, together with a raw Int value to represent their face value. (This raw Int value is not used for the Jack, Queen, King, and Ace cards.)

As mentioned above, the Rank enumeration defines a further nested structure of its own, called Values. This structure encapsulates the fact that most cards have one value, but the Ace card has two values. The Values structure defines two properties to represent this:

- first, of type Int
- second, of type Int?, or "optional Int"

Rank also defines a computed property, values, which returns an instance of the Values structure. This computed property considers the rank of the card and initializes a new Values instance with appropriate values based on its rank. It uses special values for jack, queen, king, and ace. For the numeric cards, it uses the rank's raw Int value.

The BlackjackCard structure itself has two properties—rank and suit. It also defines a computed property called description, which uses the values stored in rank and suit to build a description of the name and value of the card. The description property uses optional binding to check whether there is a second value to display, and if so, inserts additional description detail for that second value.

Because BlackjackCard is a structure with no custom initializers, it has an implicit memberwise initializer, as described in Memberwise Initializers for Structure Types. You can use this initializer to initialize a new constant called theAceOfSpades:

```
let theAceOfSpades = BlackjackCard(rank: .ace, suit: .spades)
```

```
print("theAceOfSpades: \(theAceOfSpades.description)")

// Prints "theAceOfSpades: suit is *, value is 1 or 11"
```

Even though Rank and Suit are nested within BlackjackCard, their type can be inferred from context, and so the initialization of this instance is able to refer to the enumeration cases by their case names (.ace and .spades) alone. In the example above, the description property correctly reports that the Ace of Spades has a value of 1 or 11.

Referring to Nested Types

To use a nested type outside of its definition context, prefix its name with the name of the type it is nested within:

```
1 let heartsSymbol = BlackjackCard.Suit.hearts.rawValue
2 // heartsSymbol is "♡"
```

For the example above, this enables the names of Suit, Rank, and Values to be kept deliberately short, because their names are naturally qualified by the context in which they are defined.

Extensions

Extensions add new functionality to an existing class, structure, enumeration, or protocol type. This includes the ability to extend types for which you do not have access to the original source code (known as *retroactive modeling*). Extensions are similar to categories in Objective-C. (Unlike Objective-C categories, Swift extensions do not have names.)

Extensions in Swift can:

- Add computed instance properties and computed type properties
- Define instance methods and type methods
- Provide new initializers
- Define subscripts
- Define and use new nested types
- Make an existing type conform to a protocol

In Swift, you can even extend a protocol to provide implementations of its requirements or add additional functionality that conforming types can take advantage of. For more details, see Protocol Extensions.

NOTE

Extensions can add new functionality to a type, but they cannot override existing functionality.

Extension Syntax

Declare extensions with the extension keyword:

```
1 extension SomeType {
2    // new functionality to add to SomeType goes here
3 }
```

An extension can extend an existing type to make it adopt one or more protocols. To add protocol conformance, you write the protocol names the same way as you write them for a class or structure:

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

Adding protocol conformance in this way is described in <u>Adding Protocol</u> Conformance with an Extension.

An extension can be used to extend an existing generic type, as described in Extending a Generic Type. You can also extend a generic type to conditionally add functionality, as described in Extensions with a Generic Where Clause.

```
NOTE
```

If you define an extension to add new functionality to an existing type, the new functionality will be available on all existing instances of that type, even if they were created before the extension was defined.

Computed Properties

Extensions can add computed instance properties and computed type properties to existing types. This example adds five computed instance properties to Swift's built-in <code>Double</code> type, to provide basic support for working with distance units:

```
1 extension Double {
2  var km: Double { return self * 1_000.0 }
3  var m: Double { return self }
```

```
4
        var cm: Double { return self / 100.0 }
        var mm: Double { return self / 1_000.0 }
5
        var ft: Double { return self / 3.28084 }
6
7
    }
8
    let oneInch = 25.4.mm
    print("One inch is \((oneInch)\) meters")
    // Prints "One inch is 0.0254 meters"
10
    let threeFeet = 3.ft
11
    print("Three feet is \((threeFeet) meters")
12
    // Prints "Three feet is 0.914399970739201 meters"
13
```

These computed properties express that a <code>Double</code> value should be considered as a certain unit of length. Although they are implemented as computed properties, the names of these properties can be appended to a floating-point literal value with dot syntax, as a way to use that literal value to perform distance conversions.

In this example, a Double value of 1.0 is considered to represent "one meter". This is why the m computed property returns self—the expression 1.m is considered to calculate a Double value of 1.0.

Other units require some conversion to be expressed as a value measured in meters. One kilometer is the same as 1,000 meters, so the km computed property multiplies the value by 1_000.00 to convert into a number expressed in meters. Similarly, there are 3.28084 feet in a meter, and so the ft computed property divides the underlying Double value by 3.28084, to convert it from feet to meters.

These properties are read-only computed properties, and so they are expressed without the get keyword, for brevity. Their return value is of type Double, and can be used within mathematical calculations wherever a Double is accepted:

```
1 let aMarathon = 42.km + 195.m
2 print("A marathon is \((aMarathon)\) meters long")
3 // Prints "A marathon is 42195.0 meters long"
```

NOTE

Extensions can add new computed properties, but they cannot add stored properties, or add property observers to existing properties.

Initializers

Extensions can add new initializers to existing types. This enables you to extend other types to accept your own custom types as initializer parameters, or to provide additional initialization options that were not included as part of the type's original implementation.

Extensions can add new convenience initializers to a class, but they cannot add new designated initializers or deinitializers to a class. Designated initializers and deinitializers must always be provided by the original class implementation.

If you use an extension to add an initializer to a value type that provides default values for all of its stored properties and does not define any custom initializers, you can call the default initializer and memberwise initializer for that value type from within your extension's initializer. This wouldn't be the case if you had written the initializer as part of the value type's original implementation, as described in Initializer Delegation for Value Types.

If you use an extension to add an initializer to a structure that was declared in another module, the new initializer can't access self until it calls an initializer from the defining module.

The example below defines a custom Rect structure to represent a geometric rectangle. The example also defines two supporting structures called Size and Point, both of which provide default values of 0.0 for all of their properties:

```
struct Size {
   var width = 0.0, height = 0.0
}

struct Point {
```

Because the Rect structure provides default values for all of its properties, it receives a default initializer and a memberwise initializer automatically, as described in <u>Default Initializers</u>. These initializers can be used to create new Rect instances:

```
1 let defaultRect = Rect()
2 let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
3 size: Size(width: 5.0, height: 5.0))
```

You can extend the Rect structure to provide an additional initializer that takes a specific center point and size:

```
1  extension Rect {
2    init(center: Point, size: Size) {
3        let originX = center.x - (size.width / 2)
4        let originY = center.y - (size.height / 2)
5        self.init(origin: Point(x: originX, y: originY), size: size)
6    }
7 }
```

This new initializer starts by calculating an appropriate origin point based on the provided center point and size value. The initializer then calls the structure's automatic memberwise initializer init(origin:size:), which stores the new origin and size values in the appropriate properties:

NOTE

If you provide a new initializer with an extension, you are still responsible for making sure that each instance is fully initialized once the initializer completes.

Methods

Extensions can add new instance methods and type methods to existing types. The following example adds a new instance method called repetitions to the Int type:

```
1  extension Int {
2    func repetitions(task: () -> Void) {
3       for _ in 0..<self {
4         task()
5       }
6    }
7  }</pre>
```

The repetitions(task:) method takes a single argument of type () -> Void, which indicates a function that has no parameters and does not return a value.

After defining this extension, you can call the repetitions(task:) method on any integer to perform a task that many number of times:

```
1 3.repetitions {
2 print("Hello!")
3 }
```

```
4 // Hello!
5 // Hello!
6 // Hello!
```

Mutating Instance Methods

Instance methods added with an extension can also modify (or *mutate*) the instance itself. Structure and enumeration methods that modify self or its properties must mark the instance method as mutating, just like mutating methods from an original implementation.

The example below adds a new mutating method called square to Swift's Int type, which squares the original value:

```
1  extension Int {
2    mutating func square() {
3       self = self * self
4    }
5  }
6  var someInt = 3
7  someInt.square()
8  // someInt is now 9
```

Subscripts

Extensions can add new subscripts to an existing type. This example adds an integer subscript to Swift's built-in Int type. This subscript [n] returns the decimal digit n places in from the right of the number:

- 123456789[0] returns 9
- 123456789[1] returns 8

...and so on:

```
1
     extension Int {
         subscript(digitIndex: Int) -> Int {
 2
             var decimalBase = 1
 3
             for _ in 0..<digitIndex {</pre>
 4
                 decimalBase *= 10
 5
 6
             }
             return (self / decimalBase) % 10
 7
 8
         }
 9
     }
     746381295[0]
10
11
     // returns 5
12
    746381295[1]
13
    // returns 9
14
    746381295[2]
    // returns 2
15
16
    746381295[8]
17
     // returns 7
```

If the Int value does not have enough digits for the requested index, the subscript implementation returns 0, as if the number had been padded with zeros to the left:

```
1 746381295[9]
2 // returns 0, as if you had requested:
3 0746381295[9]
```

Nested Types

Extensions can add new nested types to existing classes, structures, and enumerations:

```
extension Int {
1
 2
         enum Kind {
             case negative, zero, positive
 3
 4
         }
         var kind: Kind {
 5
             switch self {
 6
 7
             case 0:
 8
                  return .zero
             case let x where x > 0:
 9
                  return .positive
10
             default:
11
12
                  return .negative
13
             }
14
         }
15
     }
```

This example adds a new nested enumeration to Int. This enumeration, called Kind, expresses the kind of number that a particular integer represents. Specifically, it expresses whether the number is negative, zero, or positive.

This example also adds a new computed instance property to Int, called kind, which returns the appropriate Kind enumeration case for that integer.

The nested enumeration can now be used with any Int value:

```
func printIntegerKinds(_ numbers: [Int]) {
  for number in numbers {
    switch number.kind {
    case .negative:
```

```
print("- ", terminator: "")
5
             case zero:
 6
                 print("0 ", terminator: "")
 7
             case positive:
8
                 print("+ ", terminator: "")
9
             }
10
         }
11
         print("")
12
13
    }
    printIntegerKinds([3, 19, -27, 0, -6, 0, 7])
14
15
     // Prints "+ + - 0 - 0 + "
```

This function, printIntegerKinds(_:), takes an input array of Int values and iterates over those values in turn. For each integer in the array, the function considers the kind computed property for that integer, and prints an appropriate description.

NOTE

number.kind is already known to be of type Int.Kind. Because of this, all of the Int.Kind case values can be written in shorthand form inside the Switch statement, such as .negative rather than Int.Kind.negative.

Protocols

A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol can then be *adopted* by a class, structure, or enumeration to provide an actual implementation of those requirements. Any type that satisfies the requirements of a protocol is said to *conform* to that protocol.

In addition to specifying requirements that conforming types must implement, you can extend a protocol to implement some of these requirements or to implement additional functionality that conforming types can take advantage of.

Protocol Syntax

You define protocols in a very similar way to classes, structures, and enumerations:

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

Custom types state that they adopt a particular protocol by placing the protocol's name after the type's name, separated by a colon, as part of their definition. Multiple protocols can be listed, and are separated by commas:

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}
```

If a class has a superclass, list the superclass name before any protocols it adopts, followed by a comma:

```
class SomeClass: SomeSuperclass, FirstProtocol, AnotherProtocol
{
    // class definition goes here
}
```

Property Requirements

A protocol can require any conforming type to provide an instance property or type property with a particular name and type. The protocol doesn't specify whether the property should be a stored property or a computed property—it only specifies the required property name and type. The protocol also specifies whether each property must be gettable or gettable *and* settable.

If a protocol requires a property to be gettable and settable, that property requirement can't be fulfilled by a constant stored property or a read-only computed property. If the protocol only requires a property to be gettable, the requirement can be satisfied by any kind of property, and it's valid for the property to be also settable if this is useful for your own code.

Property requirements are always declared as variable properties, prefixed with the var keyword. Gettable and settable properties are indicated by writing { get } after their type declaration, and gettable properties are indicated by writing { get }.

```
protocol SomeProtocol {
   var mustBeSettable: Int { get set }
   var doesNotNeedToBeSettable: Int { get }
}
```

Always prefix type property requirements with the static keyword when you define them in a protocol. This rule pertains even though type property requirements can be prefixed with the class or static keyword when implemented by a class:

```
protocol AnotherProtocol {
    static var someTypeProperty: Int { get set }
}
```

Here's an example of a protocol with a single instance property requirement:

```
protocol FullyNamed {
   var fullName: String { get }
}
```

The FullyNamed protocol requires a conforming type to provide a fully qualified name. The protocol doesn't specify anything else about the nature of the conforming type—it only specifies that the type must be able to provide a full name for itself. The protocol states that any FullyNamed type must have a gettable instance property called fullName, which is of type String.

Here's an example of a simple structure that adopts and conforms to the FullyNamed protocol:

```
struct Person: FullyNamed {
   var fullName: String
}

let john = Person(fullName: "John Appleseed")
// john.fullName is "John Appleseed"
```

This example defines a structure called Person, which represents a specific named person. It states that it adopts the FullyNamed protocol as part of the first line of its definition.

Each instance of Person has a single stored property called fullName, which is of type String. This matches the single requirement of the FullyNamed protocol, and means that Person has correctly conformed to the protocol. (Swift reports an error at compile-time if a protocol requirement is not fulfilled.)

Here's a more complex class, which also adopts and conforms to the FullyNamed

protocol:

```
1
    class Starship: FullyNamed {
2
        var prefix: String?
        var name: String
3
        init(name: String, prefix: String? = nil) {
4
             self.name = name
5
6
             self.prefix = prefix
7
        }
        var fullName: String {
8
             return (prefix != nil ? prefix! + " " : "") + name
10
        }
11
    }
    var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
12
    // ncc1701.fullName is "USS Enterprise"
13
```

This class implements the fullName property requirement as a computed readonly property for a starship. Each Starship class instance stores a mandatory name and an optional prefix. The fullName property uses the prefix value if it exists, and prepends it to the beginning of name to create a full name for the starship.

Method Requirements

Protocols can require specific instance methods and type methods to be implemented by conforming types. These methods are written as part of the protocol's definition in exactly the same way as for normal instance and type methods, but without curly braces or a method body. Variadic parameters are allowed, subject to the same rules as for normal methods. Default values, however, can't be specified for method parameters within a protocol's definition.

As with type property requirements, you always prefix type method

requirements with the static keyword when they're defined in a protocol. This is true even though type method requirements are prefixed with the class or static keyword when implemented by a class:

```
protocol SomeProtocol {
    static func someTypeMethod()
}
```

The following example defines a protocol with a single instance method requirement:

```
protocol RandomNumberGenerator {
  func random() -> Double
}
```

This protocol, RandomNumberGenerator, requires any conforming type to have an instance method called random, which returns a Double value whenever it's called. Although it's not specified as part of the protocol, it's assumed that this value will be a number from 0.0 up to (but not including) 1.0.

The RandomNumberGenerator protocol doesn't make any assumptions about how each random number will be generated—it simply requires the generator to provide a standard way to generate a new random number.

Here's an implementation of a class that adopts and conforms to the RandomNumberGenerator protocol. This class implements a pseudorandom number generator algorithm known as a *linear congruential generator*:

```
class LinearCongruentialGenerator: RandomNumberGenerator {
   var lastRandom = 42.0
   let m = 139968.0
   let a = 3877.0
   let c = 29573.0
   func random() -> Double {
```

```
7
             lastRandom = ((lastRandom * a + c)
                 .truncatingRemainder(dividingBy:m))
8
             return lastRandom / m
9
        }
10
11
    }
    let generator = LinearCongruentialGenerator()
12
    print("Here's a random number: \((generator.random())")
13
    // Prints "Here's a random number: 0.3746499199817101"
14
    print("And another one: \((generator.random())")
15
    // Prints "And another one: 0.729023776863283"
16
```

Mutating Method Requirements

It's sometimes necessary for a method to modify (or *mutate*) the instance it belongs to. For instance methods on value types (that is, structures and enumerations) you place the mutating keyword before a method's func keyword to indicate that the method is allowed to modify the instance it belongs to and any properties of that instance. This process is described in <u>Modifying Value Types from Within Instance Methods</u>.

If you define a protocol instance method requirement that is intended to mutate instances of any type that adopts the protocol, mark the method with the mutating keyword as part of the protocol's definition. This enables structures and enumerations to adopt the protocol and satisfy that method requirement.

NOTE

If you mark a protocol instance method requirement as mutating, you don't need to write the mutating keyword when writing an implementation of that method for a class. The mutating keyword is only used by structures and enumerations.

The example below defines a protocol called Togglable, which defines a single instance method requirement called toggle. As its name suggests, the toggle()

method is intended to toggle or invert the state of any conforming type, typically by modifying a property of that type.

The toggle() method is marked with the mutating keyword as part of the Togglable protocol definition, to indicate that the method is expected to mutate the state of a conforming instance when it's called:

```
protocol Togglable {
    mutating func toggle()
}
```

If you implement the Togglable protocol for a structure or enumeration, that structure or enumeration can conform to the protocol by providing an implementation of the toggle() method that is also marked as mutating.

The example below defines an enumeration called <code>OnOffSwitch</code>. This enumeration toggles between two states, indicated by the enumeration cases on and <code>off</code>. The enumeration's toggle implementation is marked as <code>mutating</code>, to match the <code>Togglable</code> protocol's requirements:

```
1
    enum OnOffSwitch: Togglable {
2
         case off, on
         mutating func toggle() {
3
             switch self {
5
             case off:
                 self = .on
6
7
             case on:
                 self = .off
8
9
             }
         }
10
     }
11
12
    var lightSwitch = OnOffSwitch.off
     lightSwitch.toggle()
13
```

Initializer Requirements

Protocols can require specific initializers to be implemented by conforming types. You write these initializers as part of the protocol's definition in exactly the same way as for normal initializers, but without curly braces or an initializer body:

```
protocol SomeProtocol {
2
        init(someParameter: Int)
3
   }
```

Class Implementations of Protocol Initializer Requirements

You can implement a protocol initializer requirement on a conforming class as either a designated initializer or a convenience initializer. In both cases, you must mark the initializer implementation with the required modifier:

```
class SomeClass: SomeProtocol {
1
2
        required init(someParameter: Int) {
            // initializer implementation goes here
3
        }
4
5
   }
```

The use of the required modifier ensures that you provide an explicit or inherited implementation of the initializer requirement on all subclasses of the conforming class, such that they also conform to the protocol.

For more information on required initializers, see **Required Initializers**.

You don't need to mark protocol initializer implementations with the required modifier on classes that are marked with the final modifier, because final classes can't subclassed. For more about the final modifier, see Preventing Overrides.

If a subclass overrides a designated initializer from a superclass, and also implements a matching initializer requirement from a protocol, mark the initializer implementation with both the required and override modifiers:

```
protocol SomeProtocol {
1
         init()
2
3
    }
4
    class SomeSuperClass {
5
         init() {
6
             // initializer implementation goes here
         }
8
    }
9
10
     class SomeSubClass: SomeSuperClass, SomeProtocol {
11
         // "required" from SomeProtocol conformance; "override"
12
           from SomeSuperClass
13
         required override init() {
14
             // initializer implementation goes here
15
         }
16
     }
```

Failable Initializer Requirements

Protocols can define failable initializer requirements for conforming types, as defined in <u>Failable Initializers</u>.

A failable initializer requirement can be satisfied by a failable or nonfailable

initializer on a conforming type. A nonfailable initializer requirement can be satisfied by a nonfailable initializer or an implicitly unwrapped failable initializer.

Protocols as Types

Protocols don't actually implement any functionality themselves. Nonetheless, you can use protocols as a fully fledged types in your code. Using a protocol as a type is sometimes called an *existential type*, which comes from the phrase "there exists a type *T* such that *T* conforms to the protocol".

You can use a protocol in many places where other types are allowed, including:

- As a parameter type or return type in a function, method, or initializer
- As the type of a constant, variable, or property
- As the type of items in an array, dictionary, or other container

NOTE

Because protocols are types, begin their names with a capital letter (such as FullyNamed and RandomNumberGenerator) to match the names of other types in Swift (such as Int, String, and Double).

Here's an example of a protocol used as a type:

```
class Dice {
  let sides: Int
  let generator: RandomNumberGenerator
  init(sides: Int, generator: RandomNumberGenerator) {
    self.sides = sides
    self.generator = generator
}
```

```
func roll() -> Int {
    return Int(generator.random() * Double(sides)) + 1
}
```

This example defines a new class called <code>Dice</code>, which represents an *n*-sided dice for use in a board game. <code>Dice</code> instances have an integer property called <code>sides</code>, which represents how many sides they have, and a property called <code>generator</code>, which provides a random number generator from which to create dice roll values.

The generator property is of type RandomNumberGenerator. Therefore, you can set it to an instance of *any* type that adopts the RandomNumberGenerator protocol. Nothing else is required of the instance you assign to this property, except that the instance must adopt the RandomNumberGenerator protocol. Because its type is RandomNumberGenerator, code inside the Dice class can only interact with generator in ways that apply to all generators that conform to this protocol. That means it can't use any methods or properties that are defined by the underlying type of the generator. However, you can downcast from a protocol type to an underlying type in the same way you can downcast from a superclass to a subclass, as discussed in Downcasting.

parameter called generator, which is also of type RandomNumberGenerator. You can pass a value of any conforming type in to this parameter when initializing a new Dice instance.

Dice provides one instance method, roll, which returns an integer value between 1 and the number of sides on the dice. This method calls the generator's random() method to create a new random number between 0.0 and 1.0, and uses this random number to create a dice roll value within the correct range. Because generator is known to adopt RandomNumberGenerator, it's guaranteed to have a random() method to call.

Here's how the Dice class can be used to create a six-sided dice with a LinearCongruentialGenerator instance as its random number generator:

```
1
   var d6 = Dice(sides: 6, generator:
          LinearCongruentialGenerator())
2
    for _ in 1 . . 5 {
3
        print("Random dice roll is \(d6.roll())")
4
    }
    // Random dice roll is 3
5
    // Random dice roll is 5
6
    // Random dice roll is 4
7
   // Random dice roll is 5
8
    // Random dice roll is 4
```

Delegation

Delegation is a design pattern that enables a class or structure to hand off (or *delegate*) some of its responsibilities to an instance of another type. This design pattern is implemented by defining a protocol that encapsulates the delegated responsibilities, such that a conforming type (known as a delegate) is guaranteed to provide the functionality that has been delegated. Delegation can be used to respond to a particular action, or to retrieve data from an external source without needing to know the underlying type of that source.

The example below defines two protocols for use with dice-based board games:

```
protocol DiceGame {
    var dice: Dice { get }
    func play()

protocol DiceGameDelegate: AnyObject {
    func gameDidStart(_ game: DiceGame)
    func game(_ game: DiceGame, didStartNewTurnWithDiceRoll
```

```
diceRoll: Int)

func gameDidEnd(_ game: DiceGame)

}
```

The DiceGame protocol is a protocol that can be adopted by any game that involves dice.

The DiceGameDelegate protocol can be adopted to track the progress of a DiceGame. To prevent strong reference cycles, delegates are declared as weak references. For information about weak references, see Strong Reference Cycles
Between Class Instances. Marking the protocol as class-only lets the SnakesAndLadders class later in this chapter declare that its delegate must use a weak reference. A class-only protocol is marked by its inheritance from AnyObject, as discussed in Class-Only Protocols.

Here's a version of the *Snakes and Ladders* game originally introduced in <u>Control Flow</u>. This version is adapted to use a Dice instance for its dice-rolls; to adopt the DiceGame protocol; and to notify a DiceGameDelegate about its progress:

```
1
    class SnakesAndLadders: DiceGame {
2
        let finalSquare = 25
        let dice = Dice(sides: 6, generator:
3
           LinearCongruentialGenerator())
        var square = 0
4
5
        var board: [Int]
        init() {
6
             board = Array(repeating: 0, count: finalSquare + 1)
7
             board[03] = +08; board[06] = +11; board[09] = +09;
8
           board[10] = +02
             board[14] = -10; board[19] = -11; board[22] = -02;
9
           board[24] = -08
         }
10
11
        weak var delegate: DiceGameDelegate?
```

```
12
         func play() {
13
             square = 0
             delegate?.gameDidStart(self)
14
             gameLoop: while square != finalSquare {
15
16
                 let diceRoll = dice.roll()
                 delegate?.game(self, didStartNewTurnWithDiceRoll:
17
           diceRoll)
                 switch square + diceRoll {
18
19
                 case finalSquare:
                     break gameLoop
20
                 case let newSquare where newSquare > finalSquare:
21
                      continue gameLoop
22
23
                 default:
                      square += diceRoll
24
                      square += board[square]
25
26
                 }
27
             }
             delegate?.gameDidEnd(self)
28
29
         }
30
     }
```

For a description of the *Snakes and Ladders* gameplay, see <u>Break</u>.

This version of the game is wrapped up as a class called SnakesAndLadders, which adopts the DiceGame protocol. It provides a gettable dice property and a play() method in order to conform to the protocol. (The dice property is declared as a constant property because it doesn't need to change after initialization, and the protocol only requires that it must be gettable.)

The *Snakes and Ladders* game board setup takes place within the class's <code>init()</code> initializer. All game logic is moved into the protocol's <code>play</code> method, which uses the protocol's required <code>dice</code> property to provide its dice roll values.

Note that the delegate property is defined as an *optional* DiceGameDelegate, because a delegate isn't required in order to play the game. Because it's of an optional type, the delegate property is automatically set to an initial value of nil. Thereafter, the game instantiator has the option to set the property to a suitable delegate. Because the DiceGameDelegate protocol is class-only, you can declare the delegate to be weak to prevent reference cycles.

DiceGameDelegate provides three methods for tracking the progress of a game. These three methods have been incorporated into the game logic within the play() method above, and are called when a new game starts, a new turn begins, or the game ends.

Because the delegate property is an *optional* DiceGameDelegate, the play() method uses optional chaining each time it calls a method on the delegate. If the delegate property is nil, these delegate calls fail gracefully and without error. If the delegate property is non-nil, the delegate methods are called, and are passed the SnakesAndLadders instance as a parameter.

This next example shows a class called DiceGameTracker, which adopts the DiceGameDelegate protocol:

```
1
    class DiceGameTracker: DiceGameDelegate {
2
         var numberOfTurns = 0
         func gameDidStart(_ game: DiceGame) {
3
             numberOfTurns = 0
4
5
             if game is SnakesAndLadders {
                 print("Started a new game of Snakes and Ladders")
6
7
             }
             print("The game is using a \((game.dice.sides)-sided)
8
           dice")
         }
9
         func game(_ game: DiceGame, didStartNewTurnWithDiceRoll
10
           diceRoll: Int) {
             numberOfTurns += 1
11
```

```
print("Rolled a \(diceRoll)")

print("Rolled a \(diceRoll)")

func gameDidEnd(_ game: DiceGame) {
    print("The game lasted for \(numberOfTurns) turns")
}

}
```

DiceGameTracker implements all three methods required by DiceGameDelegate. It uses these methods to keep track of the number of turns a game has taken. It resets a numberOfTurns property to zero when the game starts, increments it each time a new turn begins, and prints out the total number of turns once the game has ended.

The implementation of <code>gameDidStart(_:)</code> shown above uses the <code>game</code> parameter to print some introductory information about the game that is about to be played. The <code>game</code> parameter has a type of <code>DiceGame</code>, not <code>SnakesAndLadders</code>, and so <code>gameDidStart(_:)</code> can access and use only methods and properties that are implemented as part of the <code>DiceGame</code> protocol. However, the method is still able to use type casting to query the type of the underlying instance. In this example, it checks whether <code>game</code> is actually an instance of <code>SnakesAndLadders</code> behind the scenes, and prints an appropriate message if so.

The gameDidStart(_:) method also accesses the dice property of the passed game parameter. Because game is known to conform to the DiceGame protocol, it's guaranteed to have a dice property, and so the gameDidStart(_:) method is able to access and print the dice's sides property, regardless of what kind of game is being played.

Here's how DiceGameTracker looks in action:

```
1 let tracker = DiceGameTracker()
2 let game = SnakesAndLadders()
3 game.delegate = tracker
4 game.play()
5 // Started a new game of Snakes and Ladders
```

```
6  // The game is using a 6-sided dice
7  // Rolled a 3
8  // Rolled a 5
9  // Rolled a 4
10  // Rolled a 5
11  // The game lasted for 4 turns
```

Adding Protocol Conformance with an Extension

You can extend an existing type to adopt and conform to a new protocol, even if you don't have access to the source code for the existing type. Extensions can add new properties, methods, and subscripts to an existing type, and are therefore able to add any requirements that a protocol may demand. For more about extensions, see Extensions.

```
NOTE
```

Existing instances of a type automatically adopt and conform to a protocol when that conformance is added to the instance's type in an extension.

For example, this protocol, called <code>TextRepresentable</code>, can be implemented by any type that has a way to be represented as text. This might be a description of itself, or a text version of its current state:

```
protocol TextRepresentable {
    var textualDescription: String { get }
}
```

The Dice class from above can be extended to adopt and conform to TextRepresentable:

```
1 extension Dice: TextRepresentable {
2  var textualDescription: String {
```

```
return "A \(sides)-sided dice"

}

}
```

This extension adopts the new protocol in exactly the same way as if Dice had provided it in its original implementation. The protocol name is provided after the type name, separated by a colon, and an implementation of all requirements of the protocol is provided within the extension's curly braces.

Any Dice instance can now be treated as TextRepresentable:

Similarly, the SnakesAndLadders game class can be extended to adopt and conform to the TextRepresentable protocol:

```
extension SnakesAndLadders: TextRepresentable {
   var textualDescription: String {
      return "A game of Snakes and Ladders with \
        (finalSquare) squares"
   }
}
print(game.textualDescription)
// Prints "A game of Snakes and Ladders with 25 squares"
```

Conditionally Conforming to a Protocol

A generic type may be able to satisfy the requirements of a protocol only under certain conditions, such as when the type's generic parameter conforms to the protocol. You can make a generic type conditionally conform to a protocol by

listing constraints when extending the type. Write these constraints after the name of the protocol you're adopting by writing a generic where clause. For more about generic where clauses, see <u>Generic Where Clauses</u>.

The following extension makes Array instances conform to the TextRepresentable protocol whenever they store elements of a type that conforms to TextRepresentable.

```
1
   extension Array: TextRepresentable where Element:
          TextRepresentable {
       var textualDescription: String {
2
            let itemsAsText = self.map { $0.textualDescription }
3
            return "[" + itemsAsText.joined(separator: ", ") + "]"
       }
5
6
   }
7
   let myDice = [d6, d12]
8
   print(myDice.textualDescription)
   // Prints "[A 6-sided dice, A 12-sided dice]"
```

Declaring Protocol Adoption with an Extension

If a type already conforms to all of the requirements of a protocol, but has not yet stated that it adopts that protocol, you can make it adopt the protocol with an empty extension:

```
struct Hamster {
var name: String
var textualDescription: String {
    return "A hamster named \(name\)"
}

extension Hamster: TextRepresentable {}
```

Instances of Hamster can now be used wherever TextRepresentable is the required type:

NOTE

Types don't automatically adopt a protocol just by satisfying its requirements. They must always explicitly declare their adoption of the protocol.

Collections of Protocol Types

A protocol can be used as the type to be stored in a collection such as an array or a dictionary, as mentioned in <u>Protocols as Types</u>. This example creates an array of TextRepresentable things:

```
let things: [TextRepresentable] = [game, d12, simonTheHamster]
```

It's now possible to iterate over the items in the array, and print each item's textual description:

```
for thing in things {
   print(thing.textualDescription)
}

// A game of Snakes and Ladders with 25 squares
// A 12-sided dice
// A hamster named Simon
```

Note that the thing constant is of type TextRepresentable. It's not of type Dice, or DiceGame, or Hamster, even if the actual instance behind the scenes is of one of those types. Nonetheless, because it's of type TextRepresentable, and anything that is TextRepresentable is known to have a textualDescription property, it's safe to access thing.textualDescription each time through the loop.

Protocol Inheritance

A protocol can *inherit* one or more other protocols and can add further requirements on top of the requirements it inherits. The syntax for protocol inheritance is similar to the syntax for class inheritance, but with the option to list multiple inherited protocols, separated by commas:

```
protocol InheritingProtocol: SomeProtocol, AnotherProtocol {
    // protocol definition goes here
}
```

Here's an example of a protocol that inherits the TextRepresentable protocol from above:

```
protocol PrettyTextRepresentable: TextRepresentable {
   var prettyTextualDescription: String { get }
}
```

This example defines a new protocol, PrettyTextRepresentable, which inherits from TextRepresentable. Anything that adopts PrettyTextRepresentable must satisfy all of the requirements enforced by TextRepresentable, *plus* the additional requirements enforced by PrettyTextRepresentable. In this example, PrettyTextRepresentable adds a single requirement to provide a gettable property called prettyTextualDescription that returns a String.

The SnakesAndLadders class can be extended to adopt and conform to PrettyTextRepresentable:

```
1
    extension SnakesAndLadders: PrettyTextRepresentable {
2
         var prettyTextualDescription: String {
             var output = textualDescription + ":\n"
3
             for index in 1...finalSquare {
4
                 switch board[index] {
5
                 case let ladder where ladder > 0:
6
                      output += "▲ "
7
                 case let snake where snake < 0:
8
                      output += "▼ "
9
                 default:
10
                      output += "o "
11
12
                 }
13
             }
             return output
14
15
         }
16
     }
```

This extension states that it adopts the PrettyTextRepresentable protocol and provides an implementation of the prettyTextualDescription property for the SnakesAndLadders type. Anything that is PrettyTextRepresentable must also be TextRepresentable, and so the implementation of prettyTextualDescription starts by accessing the textualDescription property from the TextRepresentable protocol to begin an output string. It appends a colon and a line break, and uses this as the start of its pretty text representation. It then iterates through the array of board squares, and appends a geometric shape to represent the contents of each square:

- If the square's value is greater than 0, it's the base of a ladder, and is represented by ▲.
- If the square's value is less than ⊙, it's the head of a snake, and is represented by ▼.

• Otherwise, the square's value is 0, and it's a "free" square, represented by o.

The prettyTextualDescription property can now be used to print a pretty text description of any SnakesAndLadders instance:

```
print(game.prettyTextualDescription)

// A game of Snakes and Ladders with 25 squares:

// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ △ ○ ○ ▼ ○ ○ ○ ▼ ○ ○ ▼ ○
```

Class-Only Protocols

You can limit protocol adoption to class types (and not structures or enumerations) by adding the AnyObject protocol to a protocol's inheritance list.

In the example above, <code>SomeClassOnlyProtocol</code> can only be adopted by class types. It's a compile-time error to write a structure or enumeration definition that tries to adopt <code>SomeClassOnlyProtocol</code>.

NOTE

Use a class-only protocol when the behavior defined by that protocol's requirements assumes or requires that a conforming type has reference semantics rather than value semantics. For more about reference and value semantics, see Structures and Enumerations Are Value Types and Classes Are Reference Types.

Protocol Composition

It can be useful to require a type to conform to multiple protocols at the same time. You can combine multiple protocols into a single requirement with a *protocol composition*. Protocol compositions behave as if you defined a temporary local protocol that has the combined requirements of all protocols in the composition. Protocol compositions don't define any new protocol types.

Protocol compositions have the form SomeProtocol & AnotherProtocol. You can list as many protocols as you need, separating them with ampersands (&). In addition to its list of protocols, a protocol composition can also contain one class type, which you can use to specify a required superclass.

Here's an example that combines two protocols called Named and Aged into a single protocol composition requirement on a function parameter:

```
1
    protocol Named {
2
        var name: String { get }
3
    }
    protocol Aged {
4
5
        var age: Int { get }
6
    }
7
    struct Person: Named, Aged {
        var name: String
8
        var age: Int
10
    }
    func wishHappyBirthday(to celebrator: Named & Aged) {
11
12
         print("Happy birthday, \(celebrator.name), you're \
           (celebrator.age)!")
    }
13
14
    let birthdayPerson = Person(name: "Malcolm", age: 21)
    wishHappyBirthday(to: birthdayPerson)
15
    // Prints "Happy birthday, Malcolm, you're 21!"
16
```

In this example, the Named protocol has a single requirement for a gettable String

property called name. The Aged protocol has a single requirement for a gettable Int property called age. Both protocols are adopted by a structure called Person.

The example also defines a wishHappyBirthday(to:) function. The type of the celebrator parameter is Named & Aged, which means "any type that conforms to both the Named and Aged protocols." It doesn't matter which specific type is passed to the function, as long as it conforms to both of the required protocols.

The example then creates a new Person instance called birthdayPerson and passes this new instance to the wishHappyBirthday(to:) function. Because Person conforms to both protocols, this call is valid, and the wishHappyBirthday(to:) function can print its birthday greeting.

Here's an example that combines the Named protocol from the previous example with a Location class:

```
1
    class Location {
2
        var latitude: Double
3
        var longitude: Double
        init(latitude: Double, longitude: Double) {
4
             self.latitude = latitude
5
             self.longitude = longitude
6
7
        }
8
    }
    class City: Location, Named {
         var name: String
10
11
         init(name: String, latitude: Double, longitude: Double) {
12
             self.name = name
             super.init(latitude: latitude, longitude: longitude)
13
         }
14
15
    }
    func beginConcert(in location: Location & Named) {
16
         print("Hello, \(location.name)!")
17
```

The beginConcert(in:) function takes a parameter of type Location & Named, which means "any type that's a subclass of Location and that conforms to the Named protocol." In this case, City satisfies both requirements.

Passing birthdayPerson to the beginConcert(in:) function is invalid because Person isn't a subclass of Location. Likewise, if you made a subclass of Location that didn't conform to the Named protocol, calling beginConcert(in:) with an instance of that type is also invalid.

Checking for Protocol Conformance

You can use the is and as operators described in <u>Type Casting</u> to check for protocol conformance, and to cast to a specific protocol. Checking for and casting to a protocol follows exactly the same syntax as checking for and casting to a type:

- The is operator returns true if an instance conforms to a protocol and returns false if it doesn't.
- The as? version of the downcast operator returns an optional value of the protocol's type, and this value is nil if the instance doesn't conform to that protocol.
- The as! version of the downcast operator forces the downcast to the protocol type and triggers a runtime error if the downcast doesn't succeed.

This example defines a protocol called HasArea, with a single property

requirement of a gettable Double property called area:

```
protocol HasArea {
   var area: Double { get }
}
```

Here are two classes, Circle and Country, both of which conform to the HasArea protocol:

```
class Circle: HasArea {
1
2
        let pi = 3.1415927
3
        var radius: Double
        var area: Double { return pi * radius * radius }
        init(radius: Double) { self.radius = radius }
5
6
    }
7
    class Country: HasArea {
8
        var area: Double
        init(area: Double) { self.area = area }
10
    }
```

The Circle class implements the area property requirement as a computed property, based on a stored radius property. The Country class implements the area requirement directly as a stored property. Both classes correctly conform to the HasArea protocol.

Here's a class called Animal, which doesn't conform to the HasArea protocol:

```
class Animal {
  var legs: Int
  init(legs: Int) { self.legs = legs }
}
```

The Circle, Country and Animal classes don't have a shared base class.

Nonetheless, they're all classes, and so instances of all three types can be used to initialize an array that stores values of type AnyObject:

```
1 let objects: [AnyObject] = [
2    Circle(radius: 2.0),
3    Country(area: 243_610),
4    Animal(legs: 4)
5 ]
```

The objects array is initialized with an array literal containing a Circle instance with a radius of 2 units; a Country instance initialized with the surface area of the United Kingdom in square kilometers; and an Animal instance with four legs.

The objects array can now be iterated, and each object in the array can be checked to see if it conforms to the HasArea protocol:

```
for object in objects {
1
2
        if let objectWithArea = object as? HasArea {
             print("Area is \(objectWithArea.area)")
3
        } else {
4
             print("Something that doesn't have an area")
5
        }
6
7
    }
    // Area is 12.5663708
    // Area is 243610.0
    // Something that doesn't have an area
10
```

Whenever an object in the array conforms to the HasArea protocol, the optional value returned by the as? operator is unwrapped with optional binding into a constant called <code>objectWithArea</code>. The <code>objectWithArea</code> constant is known to be of type <code>HasArea</code>, and so its area property can be accessed and printed in a type-safe way.

Note that the underlying objects aren't changed by the casting process. They

continue to be a Circle, a Country and an Animal. However, at the point that they're stored in the objectWithArea constant, they're only known to be of type HasArea, and so only their area property can be accessed.

Optional Protocol Requirements

You can define *optional requirements* for protocols. These requirements don't have to be implemented by types that conform to the protocol. Optional requirements are prefixed by the <code>optional</code> modifier as part of the protocol's definition. Optional requirements are available so that you can write code that interoperates with Objective-C. Both the protocol and the optional requirement must be marked with the <code>@objc</code> attribute. Note that <code>@objc</code> protocols can be adopted only by classes that inherit from Objective-C classes or other <code>@objc</code> classes. They can't be adopted by structures or enumerations.

When you use a method or property in an optional requirement, its type automatically becomes an optional. For example, a method of type (Int) -> String becomes ((Int) -> String)?. Note that the entire function type is wrapped in the optional, not the method's return value.

An optional protocol requirement can be called with optional chaining, to account for the possibility that the requirement was not implemented by a type that conforms to the protocol. You check for an implementation of an optional method by writing a question mark after the name of the method when it's called, such as <code>someOptionalMethod?(someArgument)</code>. For information on optional chaining, see Optional Chaining.

The following example defines an integer-counting class called Counter, which uses an external data source to provide its increment amount. This data source is defined by the CounterDataSource protocol, which has two optional requirements:

```
1 @objc protocol CounterDataSource {
2    @objc optional func increment(forCount count: Int) -> Int
3    @objc optional var fixedIncrement: Int { get }
```

```
4 }
```

The CounterDataSource protocol defines an optional method requirement called increment(forCount:) and an optional property requirement called fixedIncrement. These requirements define two different ways for data sources to provide an appropriate increment amount for a Counter instance.

NOTE

Strictly speaking, you can write a custom class that conforms to CounterDataSource without implementing *either* protocol requirement. They're both optional, after all. Although technically allowed, this wouldn't make for a very good data source.

The Counter class, defined below, has an optional dataSource property of type CounterDataSource?:

```
class Counter {
1
2
         var count = 0
3
         var dataSource: CounterDataSource?
         func increment() {
             if let amount = dataSource?.increment?(forCount: count)
5
           {
6
                 count += amount
             } else if let amount = dataSource?.fixedIncrement {
7
                 count += amount
             }
         }
10
11
     }
```

The Counter class stores its current value in a variable property called count. The Counter class also defines a method called increment, which increments the count property every time the method is called.

The increment() method first tries to retrieve an increment amount by looking for an implementation of the increment(forCount:) method on its data source.

The increment() method uses optional chaining to try to call increment(forCount:), and passes the current count value as the method's single argument.

Note that *two* levels of optional chaining are at play here. First, it's possible that dataSource may be nil, and so dataSource has a question mark after its name to indicate that increment(forCount:) should be called only if dataSource isn't nil. Second, even if dataSource *does* exist, there's no guarantee that it implements increment(forCount:), because it's an optional requirement. Here, the possibility that increment(forCount:) might not be implemented is also handled by optional chaining. The call to increment(forCount:) happens only if increment(forCount:) exists—that is, if it isn't nil. This is why increment(forCount:) is also written with a question mark after its name.

Because the call to increment(forCount:) can fail for either of these two reasons, the call returns an *optional* Int value. This is true even though increment(forCount:) is defined as returning a non-optional Int value in the definition of CounterDataSource. Even though there are two optional chaining operations, one after another, the result is still wrapped in a single optional. For more information about using multiple optional chaining operations, see <u>Linking Multiple Levels of Chaining</u>.

After calling increment(forCount:), the optional Int that it returns is unwrapped into a constant called amount, using optional binding. If the optional Int does contain a value—that is, if the delegate and method both exist, and the method returned a value—the unwrapped amount is added onto the stored count property, and incrementation is complete.

If it's not possible to retrieve a value from the increment(forCount:) method—either because dataSource is nil, or because the data source doesn't implement increment(forCount:)—then the increment() method tries to retrieve a value from the data source's fixedIncrement property instead. The fixedIncrement property is also an optional requirement, so its value is an optional Int value, even though fixedIncrement is defined as a non-optional Int property as part of the CounterDataSource protocol definition.

Here's a simple CounterDataSource implementation where the data source returns a constant value of 3 every time it's queried. It does this by implementing

the optional fixedIncrement property requirement:

```
class ThreeSource: NSObject, CounterDataSource {
   let fixedIncrement = 3
}
```

You can use an instance of ThreeSource as the data source for a new Counter instance:

```
1
    var counter = Counter()
    counter.dataSource = ThreeSource()
2
3
    for _ in 1...4 {
        counter.increment()
4
        print(counter.count)
5
6
    }
7
    // 3
   // 6
8
    // 9
10
    // 12
```

The code above creates a new Counter instance; sets its data source to be a new ThreeSource instance; and calls the counter's increment() method four times. As expected, the counter's count property increases by three each time increment() is called.

Here's a more complex data source called TowardsZeroSource, which makes a Counter instance count up or down towards zero from its current count value:

```
class TowardsZeroSource: NSObject, CounterDataSource {
  func increment(forCount count: Int) -> Int {
    if count == 0 {
      return 0
    } else if count < 0 {</pre>
```

The TowardsZeroSource class implements the optional increment(forCount:) method from the CounterDataSource protocol and uses the count argument value to work out which direction to count in. If count is already zero, the method returns 0 to indicate that no further counting should take place.

You can use an instance of TowardsZeroSource with the existing Counter instance to count from -4 to zero. Once the counter reaches zero, no more counting takes place:

```
counter.count = -4
1
    counter.dataSource = TowardsZeroSource()
2
    for _ in 1...5 {
3
         counter.increment()
4
         print(counter.count)
6
    }
7
    // -3
    // -2
8
    // -1
10
    // 0
     // 0
11
```

Protocol Extensions

Protocols can be extended to provide method, initializer, subscript, and

computed property implementations to conforming types. This allows you to define behavior on protocols themselves, rather than in each type's individual conformance or in a global function.

For example, the RandomNumberGenerator protocol can be extended to provide a randomBool() method, which uses the result of the required random() method to return a random Bool value:

```
1 extension RandomNumberGenerator {
2  func randomBool() -> Bool {
3  return random() > 0.5
4  }
5 }
```

By creating an extension on the protocol, all conforming types automatically gain this method implementation without any additional modification.

```
let generator = LinearCongruentialGenerator()
print("Here's a random number: \((generator.random())")

// Prints "Here's a random number: 0.3746499199817101"
print("And here's a random Boolean: \((generator.randomBool())"))
// Prints "And here's a random Boolean: true"
```

Protocol extensions can add implementations to conforming types but can't make a protocol extend or inherit from another protocol. Protocol inheritance is always specified in the protocol declaration itself.

Providing Default Implementations

You can use protocol extensions to provide a default implementation to any method or computed property requirement of that protocol. If a conforming type provides its own implementation of a required method or property, that implementation will be used instead of the one provided by the extension.

Protocol requirements with default implementations provided by extensions are distinct from optional protocol requirements. Although conforming types don't have to provide their own implementation of either, requirements with default implementations can be called without optional chaining.

For example, the PrettyTextRepresentable protocol, which inherits the TextRepresentable protocol can provide a default implementation of its required prettyTextualDescription property to simply return the result of accessing the textualDescription property:

```
1  extension PrettyTextRepresentable {
2    var prettyTextualDescription: String {
3       return textualDescription
4    }
5 }
```

Adding Constraints to Protocol Extensions

When you define a protocol extension, you can specify constraints that conforming types must satisfy before the methods and properties of the extension are available. You write these constraints after the name of the protocol you're extending by writing a generic where clause. For more about generic where clauses, see <u>Generic Where Clauses</u>.

For example, you can define an extension to the Collection protocol that applies to any collection whose elements conform to the Equatable protocol. By constraining a collection's elements to the Equatable protocol, a part of the standard library, you can use the == and != operators to check for equality and inequality between two elements.

```
1 extension Collection where Element: Equatable {
2  func allEqual() -> Bool {
3  for element in self {
4  if element != self.first {
```

The allEqual() method returns true only if all the elements in the collection are equal.

Consider two arrays of integers, one where all the elements are the same, and one where they aren't:

```
1 let equalNumbers = [100, 100, 100, 100, 100]
2 let differentNumbers = [100, 100, 200, 100, 200]
```

Because arrays conform to Collection and integers conform to Equatable, equalNumbers and differentNumbers can use the allEqual() method:

```
print(equalNumbers.allEqual())

// Prints "true"

print(differentNumbers.allEqual())

// Prints "false"
```

NOTE

If a conforming type satisfies the requirements for multiple constrained extensions that provide implementations for the same method or property, Swift uses the implementation corresponding to the most specialized constraints.

Generics

Generic code enables you to write flexible, reusable functions and types that can work with any type, subject to requirements that you define. You can write code that avoids duplication and expresses its intent in a clear, abstracted manner.

Generics are one of the most powerful features of Swift, and much of the Swift standard library is built with generic code. In fact, you've been using generics throughout the *Language Guide*, even if you didn't realize it. For example, Swift's Array and Dictionary types are both generic collections. You can create an array that holds Int values, or an array that holds String values, or indeed an array for any other type that can be created in Swift. Similarly, you can create a dictionary to store values of any specified type, and there are no limitations on what that type can be.

The Problem That Generics Solve

Here's a standard, nongeneric function called swapTwoInts(_:_:), which swaps two Int values:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
   let temporaryA = a
   a = b
   b = temporaryA
}
```

This function makes use of in-out parameters to swap the values of a and b, as described in In-Out Parameters.

The swapTwoInts(_:_:) function swaps the original value of b into a, and the original value of a into b. You can call this function to swap the values in two Int variables:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \((someInt)\), and anotherInt is now \((anotherInt)\)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

The swapTwoInts(_:_:) function is useful, but it can only be used with Int values. If you want to swap two String values, or two Double values, you have to write more functions, such as the swapTwoStrings(_:_:) and swapTwoDoubles(_:_:) functions shown below:

```
1
    func swapTwoStrings(_ a: inout String, _ b: inout String) {
         let temporaryA = a
2
         a = b
3
         b = temporaryA
4
5
    }
6
7
    func swapTwoDoubles(_ a: inout Double, _ b: inout Double) {
         let temporaryA = a
8
         a = b
9
10
         b = temporaryA
11
     }
```

You may have noticed that the bodies of the <code>swapTwoInts(_:_:)</code>, <code>swapTwoStrings(_:_:)</code>, and <code>swapTwoDoubles(_:_:)</code> functions are identical. The only difference is the type of the values that they accept (<code>Int</code>, <code>String</code>, and <code>Double</code>).

It's more useful, and considerably more flexible, to write a single function that swaps two values of *any* type. Generic code enables you to write such a function. (A generic version of these functions is defined below.)

NOTE

In all three functions, the types of a and b must be the same. If a and b aren't of the same type, it isn't possible to swap their values. Swift is a type-safe language, and doesn't allow (for example) a variable of type String and a variable of type Double to swap values with each other. Attempting to do so results in a compile-time error.

Generic Functions

Generic functions can work with any type. Here's a generic version of the swapTwoInts(_:_:) function from above, called swapTwoValues(_:_:):

```
func swapTwoValues<T>(_ a: inout T, _ b: inout T) {
   let temporaryA = a
   a = b
   b = temporaryA
}
```

The body of the swaptwovalues(_:_:) function is identical to the body of the swaptwoints(_:_:) function. However, the first line of swaptwovalues(_:_:) is slightly different from swaptwoints(_:_:). Here's how the first lines compare:

```
func swapTwoInts(_ a: inout Int, _ b: inout Int)
func swapTwoValues<T>(_ a: inout T, _ b: inout T)
```

The generic version of the function uses a *placeholder* type name (called \top , in this case) instead of an *actual* type name (such as Int, String, or Double). The placeholder type name doesn't say anything about what \top must be, but it *does* say that both a and b must be of the same type \top , whatever \top represents. The actual type to use in place of \top is determined each time the swapTwoValues(_:_:) function is called.

The other difference between a generic function and a nongeneric function is that the generic function's name (swapTwoValues(_:_:)) is followed by the

placeholder type name (\top) inside angle brackets (< \top >). The brackets tell Swift that \top is a placeholder type name within the <code>swapTwoValues(_:_:)</code> function definition. Because \top is a placeholder, Swift doesn't look for an actual type called \top .

The swapTwoValues(_:_:) function can now be called in the same way as swapTwoInts, except that it can be passed two values of *any* type, as long as both of those values are of the same type as each other. Each time swapTwoValues(_:_:) is called, the type to use for T is inferred from the types of values passed to the function.

In the two examples below, T is inferred to be Int and String respectively:

```
1
   var someInt = 3
   var anotherInt = 107
   swapTwoValues(&someInt, &anotherInt)
3
   // someInt is now 107, and anotherInt is now 3
4
5
   var someString = "hello"
6
   var anotherString = "world"
7
   swapTwoValues(&someString, &anotherString)
8
   // someString is now "world", and anotherString is now "hello"
9
```

NOTE

The swapTwoValues(_:_:) function defined above is inspired by a generic function called swap, which is part of the Swift standard library, and is automatically made available for you to use in your apps. If you need the behavior of the swapTwoValues(_:_:) function in your own code, you can use Swift's existing swap(_:_:) function rather than providing your own implementation.

Type Parameters

In the swapTwoValues(_:_:) example above, the placeholder type T is an example of a *type parameter*. Type parameters specify and name a placeholder type, and

are written immediately after the function's name, between a pair of matching angle brackets (such as <T>).

Once you specify a type parameter, you can use it to define the type of a function's parameters (such as the a and b parameters of the swapTwoValues(_:_:) function), or as the function's return type, or as a type annotation within the body of the function. In each case, the type parameter is replaced with an *actual* type whenever the function is called. (In the swapTwoValues(_:_:) example above, T was replaced with Int the first time the function was called, and was replaced with String the second time it was called.)

You can provide more than one type parameter by writing multiple type parameter names within the angle brackets, separated by commas.

Naming Type Parameters

In most cases, type parameters have descriptive names, such as Key and Value in Dictionary<Key, Value> and Element in Array<Element>, which tells the reader about the relationship between the type parameter and the generic type or function it's used in. However, when there isn't a meaningful relationship between them, it's traditional to name them using single letters such as \top , \cup , and \vee , such as \top in the swapTwoValues(_:_:) function above.

NOTE

Always give type parameters upper camel case names (such as T and MyTypeParameter) to indicate that they're a placeholder for a *type*, not a value.

Generic Types

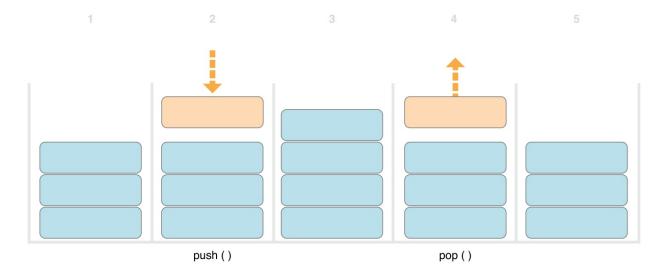
In addition to generic functions, Swift enables you to define your own *generic types*. These are custom classes, structures, and enumerations that can work with *any* type, in a similar way to Array and Dictionary.

This section shows you how to write a generic collection type called Stack. A stack is an ordered set of values, similar to an array, but with a more restricted set of operations than Swift's Array type. An array allows new items to be inserted and removed at any location in the array. A stack, however, allows new items to be appended only to the end of the collection (known as *pushing* a new value on to the stack). Similarly, a stack allows items to be removed only from the end of the collection (known as *popping* a value off the stack).

NOTE

The concept of a stack is used by the UINavigationController class to model the view controllers in its navigation hierarchy. You call the UINavigationController class pushViewController(_:animated:) method to add (or push) a view controller on to the navigation stack, and its popViewControllerAnimated(_:) method to remove (or pop) a view controller from the navigation stack. A stack is a useful collection model whenever you need a strict "last in, first out" approach to managing a collection.

The illustration below shows the push and pop behavior for a stack:



- 1. There are currently three values on the stack.
- 2. A fourth value is pushed onto the top of the stack.
- 3. The stack now holds four values, with the most recent one at the top.
- 4. The top item in the stack is popped.

5. After popping a value, the stack once again holds three values.

Here's how to write a nongeneric version of a stack, in this case for a stack of Int values:

```
1
    struct IntStack {
2
        var items = [Int]()
        mutating func push(_ item: Int) {
3
            items.append(item)
4
5
        }
6
        mutating func pop() -> Int {
7
            return items.removeLast()
        }
8
9
   }
```

This structure uses an Array property called items to store the values in the stack. Stack provides two methods, push and pop, to push and pop values on and off the stack. These methods are marked as mutating, because they need to modify (or *mutate*) the structure's items array.

The IntStack type shown above can only be used with Int values, however. It would be much more useful to define a *generic* Stack class, that can manage a stack of *any* type of value.

Here's a generic version of the same code:

```
struct Stack<Element> {
   var items = [Element]()
   mutating func push(_ item: Element) {
      items.append(item)
   }
   mutating func pop() -> Element {
      return items.removeLast()
```

```
8 }
9 }
```

Note how the generic version of Stack is essentially the same as the nongeneric version, but with a type parameter called Element instead of an actual type of Int. This type parameter is written within a pair of angle brackets (<Element>) immediately after the structure's name.

Element defines a placeholder name for a type to be provided later. This future type can be referred to as Element anywhere within the structure's definition. In this case, Element is used as a placeholder in three places:

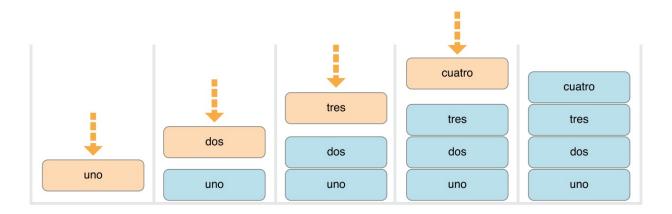
- To create a property called items, which is initialized with an empty array of values of type Element
- To specify that the push(_:) method has a single parameter called item, which must be of type Element
- To specify that the value returned by the pop() method will be a value of type Element

Because it's a generic type, Stack can be used to create a stack of *any* valid type in Swift, in a similar manner to Array and Dictionary.

You create a new Stack instance by writing the type to be stored in the stack within angle brackets. For example, to create a new stack of strings, you write Stack<String>():

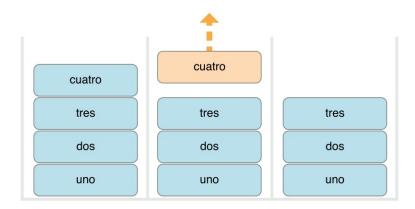
```
var stackOfStrings = Stack<String>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")
stackOfStrings.push("cuatro")
// the stack now contains 4 strings
```

Here's how stackOfStrings looks after pushing these four values on to the stack:



Popping a value from the stack removes and returns the top value, "cuatro":

Here's how the stack looks after popping its top value:



Extending a Generic Type

When you extend a generic type, you don't provide a type parameter list as part of the extension's definition. Instead, the type parameter list from the *original* type definition is available within the body of the extension, and the original type parameter names are used to refer to the type parameters from the original definition.

The following example extends the generic Stack type to add a read-only computed property called topItem, which returns the top item on the stack without popping it from the stack:

```
1  extension Stack {
2    var topItem: Element? {
3       return items.isEmpty ? nil : items[items.count - 1]
4    }
5 }
```

The topItem property returns an optional value of type Element. If the stack is empty, topItem returns nil; if the stack isn't empty, topItem returns the final item in the items array.

Note that this extension doesn't define a type parameter list. Instead, the Stack type's existing type parameter name, Element, is used within the extension to indicate the optional type of the topItem computed property.

The topItem computed property can now be used with any Stack instance to access and query its top item without removing it.

```
if let topItem = stackOfStrings.topItem {
    print("The top item on the stack is \(topItem).")
}

// Prints "The top item on the stack is tres."
```

Extensions of a generic type can also include requirements that instances of the extended type must satisfy in order to gain the new functionality, as discussed in Extensions with a Generic Where Clause below.

Type Constraints

The swapTwoValues(_:_:) function and the Stack type can work with any type.

However, it's sometimes useful to enforce certain *type constraints* on the types that can be used with generic functions and generic types. Type constraints specify that a type parameter must inherit from a specific class, or conform to a particular protocol or protocol composition.

For example, Swift's Dictionary type places a limitation on the types that can be used as keys for a dictionary. As described in Dictionaries, the type of a dictionary's keys must be *hashable*. That is, it must provide a way to make itself uniquely representable. Dictionary needs its keys to be hashable so that it can check whether it already contains a value for a particular key. Without this requirement, Dictionary could not tell whether it should insert or replace a value for a particular key, nor would it be able to find a value for a given key that is already in the dictionary.

This requirement is enforced by a type constraint on the key type for Dictionary, which specifies that the key type must conform to the Hashable protocol, a special protocol defined in the Swift standard library. All of Swift's basic types (such as String, Int, Double, and Bool) are hashable by default.

You can define your own type constraints when creating custom generic types, and these constraints provide much of the power of generic programming. Abstract concepts like Hashable characterize types in terms of their conceptual characteristics, rather than their concrete type.

Type Constraint Syntax

You write type constraints by placing a single class or protocol constraint after a type parameter's name, separated by a colon, as part of the type parameter list. The basic syntax for type constraints on a generic function is shown below (although the syntax is the same for generic types):

The hypothetical function above has two type parameters. The first type parameter, \top , has a type constraint that requires \top to be a subclass of SomeClass. The second type parameter, \cup , has a type constraint that requires \cup to conform to the protocol SomeProtocol.

Type Constraints in Action

Here's a nongeneric function called findIndex(ofString:in:), which is given a String value to find and an array of String values within which to find it. The findIndex(ofString:in:) function returns an optional Int value, which will be the index of the first matching string in the array if it's found, or nil if the string can't be found:

```
func findIndex(ofString valueToFind: String, in array:
          [String]) -> Int? {
        for (index, value) in array.enumerated() {
2
            if value == valueToFind {
3
                return index
4
5
            }
        }
6
        return nil
7
8
   }
```

The findIndex(ofString:in:) function can be used to find a string value in an array of strings:

```
1 let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
2 if let foundIndex = findIndex(ofString: "llama", in: strings) {
3    print("The index of llama is \((foundIndex)\)")
4 }
5 // Prints "The index of llama is 2"
```

The principle of finding the index of a value in an array isn't useful only for

strings, however. You can write the same functionality as a generic function by replacing any mention of strings with values of some type \top instead.

Here's how you might expect a generic version of findIndex(ofString:in:), called findIndex(of:in:), to be written. Note that the return type of this function is still Int?, because the function returns an optional index number, not an optional value from the array. Be warned, though—this function doesn't compile, for reasons explained after the example:

```
func findIndex<T>(of valueToFind: T, in array:[T]) -> Int? {
   for (index, value) in array.enumerated() {
      if value == valueToFind {
          return index
      }
   }
   return nil
}
```

This function doesn't compile as written above. The problem lies with the equality check, "if value == valueToFind". Not every type in Swift can be compared with the equal to operator (==). If you create your own class or structure to represent a complex data model, for example, then the meaning of "equal to" for that class or structure isn't something that Swift can guess for you. Because of this, it isn't possible to guarantee that this code will work for *every* possible type T, and an appropriate error is reported when you try to compile the code.

All is not lost, however. The Swift standard library defines a protocol called Equatable, which requires any conforming type to implement the equal to operator (==) and the not equal to operator (!=) to compare any two values of that type. All of Swift's standard types automatically support the Equatable protocol.

Any type that is Equatable can be used safely with the findIndex(of:in:) function, because it's guaranteed to support the equal to operator. To express this fact, you write a type constraint of Equatable as part of the type parameter's definition when you define the function:

```
func findIndex<T: Equatable>(of valueToFind: T, in array:[T]) -
1
          > Int? {
2
        for (index, value) in array.enumerated() {
3
            if value == valueToFind {
                return index
4
5
            }
        }
6
7
        return nil
8
    }
```

The single type parameter for findIndex(of:in:) is written as T: Equatable, which means "any type T that conforms to the Equatable protocol."

The findIndex(of:in:) function now compiles successfully and can be used with any type that is Equatable, such as Double or String:

Associated Types

When defining a protocol, it's sometimes useful to declare one or more associated types as part of the protocol's definition. An *associated type* gives a placeholder name to a type that is used as part of the protocol. The actual type to use for that associated type isn't specified until the protocol is adopted. Associated types are specified with the associated type keyword.

Associated Types in Action

Here's an example of a protocol called Container, which declares an associated type called Item:

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
```

The Container protocol defines three required capabilities that any container must provide:

- It must be possible to add a new item to the container with an append(_:) method.
- It must be possible to access a count of the items in the container through a count property that returns an Int value.
- It must be possible to retrieve each item in the container with a subscript that takes an Int index value.

This protocol doesn't specify how the items in the container should be stored or what type they're allowed to be. The protocol only specifies the three bits of functionality that any type must provide in order to be considered a Container. A conforming type can provide additional functionality, as long as it satisfies these three requirements.

Any type that conforms to the Container protocol must be able to specify the type of values it stores. Specifically, it must ensure that only items of the right type are added to the container, and it must be clear about the type of the items returned by its subscript.

To define these requirements, the Container protocol needs a way to refer to the type of the elements that a container will hold, without knowing what that type is

for a specific container. The Container protocol needs to specify that any value passed to the append(_:) method must have the same type as the container's element type, and that the value returned by the container's subscript will be of the same type as the container's element type.

To achieve this, the <code>Container</code> protocol declares an associated type called <code>Item</code>, written as <code>associatedtype Item</code>. The protocol doesn't define what <code>Item</code> is—that information is left for any conforming type to provide. Nonetheless, the <code>Item</code> alias provides a way to refer to the type of the items in a <code>Container</code>, and to define a type for use with the <code>append(_:)</code> method and subscript, to ensure that the expected behavior of any <code>Container</code> is enforced.

Here's a version of the nongeneric IntStack type from <u>Generic Types</u> above, adapted to conform to the Container protocol:

```
1
    struct IntStack: Container {
         // original IntStack implementation
2
3
         var items = [Int]()
         mutating func push(_ item: Int) {
4
             items.append(item)
5
         }
6
7
         mutating func pop() -> Int {
             return items.removeLast()
8
         }
9
         // conformance to the Container protocol
10
         typealias Item = Int
11
         mutating func append(_ item: Int) {
12
             self.push(item)
13
14
         }
15
         var count: Int {
             return items.count
16
17
         }
18
         subscript(i: Int) -> Int {
```

The IntStack type implements all three of the Container protocol's requirements, and in each case wraps part of the IntStack type's existing functionality to satisfy these requirements.

Moreover, IntStack specifies that for this implementation of Container, the appropriate Item to use is a type of Int. The definition of typealias Item = Int turns the abstract type of Item into a concrete type of Int for this implementation of the Container protocol.

Thanks to Swift's type inference, you don't actually need to declare a concrete Item of Int as part of the definition of IntStack. Because IntStack conforms to all of the requirements of the Container protocol, Swift can infer the appropriate Item to use, simply by looking at the type of the append(_:) method's item parameter and the return type of the subscript. Indeed, if you delete the typealias Item = Int line from the code above, everything still works, because it's clear what type should be used for Item.

You can also make the generic Stack type conform to the Container protocol:

```
struct Stack<Element>: Container {
1
2
        // original Stack<Element> implementation
        var items = [Element]()
3
        mutating func push(_ item: Element) {
4
             items.append(item)
5
        }
6
        mutating func pop() -> Element {
7
             return items.removeLast()
8
9
        }
         // conformance to the Container protocol
10
         mutating func append(_ item: Element) {
11
```

```
12
              self.push(item)
13
         }
14
         var count: Int {
15
              return items.count
         }
16
         subscript(i: Int) -> Element {
17
              return items[i]
18
         }
19
20
     }
```

This time, the type parameter Element is used as the type of the append(_:) method's item parameter and the return type of the subscript. Swift can therefore infer that Element is the appropriate type to use as the Item for this particular container.

Extending an Existing Type to Specify an Associated Type

You can extend an existing type to add conformance to a protocol, as described in <u>Adding Protocol Conformance with an Extension</u>. This includes a protocol with an associated type.

Swift's Array type already provides an append(_:) method, a count property, and a subscript with an Int index to retrieve its elements. These three capabilities match the requirements of the Container protocol. This means that you can extend Array to conform to the Container protocol simply by declaring that Array adopts the protocol. You do this with an empty extension, as described in Declaring Protocol Adoption with an Extension:

```
extension Array: Container {}
```

Array's existing append(_:) method and subscript enable Swift to infer the appropriate type to use for Item, just as for the generic Stack type above. After defining this extension, you can use any Array as a Container.

Adding Constraints to an Associated Type

You can add type constraints to an associated type in a protocol to require that conforming types satisfy those constraints. For example, the following code defines a version of Container that requires the items in the container to be equatable.

```
protocol Container {
   associatedtype Item: Equatable
   mutating func append(_ item: Item)
   var count: Int { get }
   subscript(i: Int) -> Item { get }
}
```

To conform to this version of Container, the container's Item type has to conform to the Equatable protocol.

Using a Protocol in Its Associated Type's Constraints

A protocol can appear as part of its own requirements. For example, here's a protocol that refines the Container protocol, adding the requirement of a suffix(_:) method. The suffix(_:) method returns a given number of elements from the end of the container, storing them in an instance of the Suffix type.

```
protocol SuffixableContainer: Container {
    associatedtype Suffix: SuffixableContainer where
    Suffix.Item == Item
    func suffix(_ size: Int) -> Suffix
}
```

In this protocol, Suffix is an associated type, like the Item type in the Container example above. Suffix has two constraints: It must conform to the SuffixableContainer protocol (the protocol currently being defined), and its Item type must be the same as the container's Item type. The constraint on Item

is a generic where clause, which is discussed in <u>Associated Types with a Generic</u> Where Clause below.

Here's an extension of the Stack type from <u>Generic Types</u> above that adds conformance to the SuffixableContainer protocol:

```
1
    extension Stack: SuffixableContainer {
         func suffix(_ size: Int) -> Stack {
2
             var result = Stack()
3
             for index in (count-size)..<count {</pre>
4
                 result.append(self[index])
5
             }
6
             return result
7
         }
8
        // Inferred that Suffix is Stack.
9
    }
10
    var stackOfInts = Stack<Int>()
11
    stackOfInts.append(10)
12
    stackOfInts.append(20)
13
    stackOfInts.append(30)
14
15
    let suffix = stackOfInts.suffix(2)
16
     // suffix contains 20 and 30
```

In the example above, the Suffix associated type for Stack is also Stack, so the suffix operation on Stack returns another Stack. Alternatively, a type that conforms to SuffixableContainer can have a Suffix type that's different from itself—meaning the suffix operation can return a different type. For example, here's an extension to the nongeneric IntStack type that adds
SuffixableContainer conformance, using Stack<Int> as its suffix type instead of IntStack:

```
extension IntStack: SuffixableContainer {
func suffix(_ size: Int) -> Stack<Int> {
```

```
var result = Stack<Int>()
for index in (count-size)..<count {
    result.append(self[index])
}
return result
}
// Inferred that Suffix is Stack<Int>.
```

Generic Where Clauses

Type constraints, as described in <u>Type Constraints</u>, enable you to define requirements on the type parameters associated with a generic function, subscript, or type.

It can also be useful to define requirements for associated types. You do this by defining a *generic where clause*. A generic where clause enables you to require that an associated type must conform to a certain protocol, or that certain type parameters and associated types must be the same. A generic where clause starts with the where keyword, followed by constraints for associated types or equality relationships between types and associated types. You write a generic where clause right before the opening curly brace of a type or function's body.

The example below defines a generic function called allItemsMatch, which checks to see if two Container instances contain the same items in the same order. The function returns a Boolean value of true if all items match and a value of false if they don't.

The two containers to be checked don't have to be the same type of container (although they can be), but they do have to hold the same type of items. This requirement is expressed through a combination of type constraints and a generic where clause:

```
func allItemsMatch<C1: Container, C2: Container>
1
2
         (_ someContainer: C1, _ anotherContainer: C2) -> Bool
3
         where C1.Item == C2.Item, C1.Item: Equatable {
4
             // Check that both containers contain the same number
5
           of items.
             if someContainer.count != anotherContainer.count {
6
                 return false
7
             }
8
9
             // Check each pair of items to see if they're
10
           equivalent.
11
             for i in 0..<someContainer.count {</pre>
                 if someContainer[i] != anotherContainer[i] {
12
                      return false
13
14
                 }
15
             }
16
17
             // All items match, so return true.
18
             return true
19
     }
```

This function takes two arguments called someContainer and anotherContainer. The someContainer argument is of type C1, and the anotherContainer argument is of type C2. Both C1 and C2 are type parameters for two container types to be determined when the function is called.

The following requirements are placed on the function's two type parameters:

- C1 must conform to the Container protocol (written as C1: Container).
- C2 must also conform to the Container protocol (written as C2: Container).

- The Item for C1 must be the same as the Item for C2 (written as C1.Item == C2.Item).
- The Item for C1 must conform to the Equatable protocol (written as C1.Item: Equatable).

The first and second requirements are defined in the function's type parameter list, and the third and fourth requirements are defined in the function's generic where clause.

These requirements mean:

- someContainer is a container of type C1.
- anotherContainer is a container of type C2.
- someContainer and anotherContainer contain the same type of items.
- The items in someContainer can be checked with the not equal operator (!=) to see if they're different from each other.

The third and fourth requirements combine to mean that the items in anotherContainer can *also* be checked with the != operator, because they're exactly the same type as the items in someContainer.

These requirements enable the allItemsMatch(_:_:) function to compare the two containers, even if they're of a different container type.

The allItemsMatch(_:_:) function starts by checking that both containers contain the same number of items. If they contain a different number of items, there's no way that they can match, and the function returns false.

After making this check, the function iterates over all of the items in someContainer with a for-in loop and the half-open range operator (..<). For each item, the function checks whether the item from someContainer isn't equal to the corresponding item in anotherContainer. If the two items aren't equal, then the two containers don't match, and the function returns false.

If the loop finishes without finding a mismatch, the two containers match, and

the function returns true.

Here's how the allItemsMatch(_:_:) function looks in action:

```
var stackOfStrings = Stack<String>()
2
    stackOfStrings.push("uno")
3
    stackOfStrings.push("dos")
    stackOfStrings.push("tres")
4
5
    var arrayOfStrings = ["uno", "dos", "tres"]
6
7
    if allItemsMatch(stackOfStrings, arrayOfStrings) {
8
        print("All items match.")
9
    } else {
10
         print("Not all items match.")
11
12
    }
    // Prints "All items match."
13
```

The example above creates a Stack instance to store String values, and pushes three strings onto the stack. The example also creates an Array instance initialized with an array literal containing the same three strings as the stack. Even though the stack and the array are of a different type, they both conform to the Container protocol, and both contain the same type of values. You can therefore call the allitemsMatch(_:_:) function with these two containers as its arguments. In the example above, the allitemsMatch(_:_:) function correctly reports that all of the items in the two containers match.

Extensions with a Generic Where Clause

You can also use a generic where clause as part of an extension. The example below extends the generic Stack structure from the previous examples to add an <code>isTop(_:)</code> method.

```
1
    extension Stack where Element: Equatable {
2
        func isTop(_ item: Element) -> Bool {
3
            guard let topItem = items.last else {
                return false
4
            }
5
            return topItem == item
6
7
        }
8
   }
```

This new <code>isTop(_:)</code> method first checks that the stack isn't empty, and then compares the given item against the stack's topmost item. If you tried to do this without a generic <code>where</code> clause, you would have a problem: The implementation of <code>isTop(_:)</code> uses the <code>==</code> operator, but the definition of <code>Stack</code> doesn't require its items to be equatable, so using the <code>==</code> operator results in a compile-time error. Using a generic <code>where</code> clause lets you add a new requirement to the extension, so that the extension adds the <code>isTop(_:)</code> method only when the items in the stack are equatable.

Here's how the <code>isTop(_:)</code> method looks in action:

```
if stackOfStrings.isTop("tres") {
   print("Top element is tres.")
} else {
   print("Top element is something else.")
}
// Prints "Top element is tres."
```

If you try to call the <code>isTop(_:)</code> method on a stack whose elements aren't equatable, you'll get a compile-time error.

```
struct NotEquatable { }

var notEquatableStack = Stack<NotEquatable>()

let notEquatableValue = NotEquatable()
```

```
notEquatableStack.push(notEquatableValue)
notEquatableStack.isTop(notEquatableValue) // Error
```

You can use a generic where clause with extensions to a protocol. The example below extends the Container protocol from the previous examples to add a startsWith(_:) method.

```
1 extension Container where Item: Equatable {
2  func startsWith(_ item: Item) -> Bool {
3  return count >= 1 && self[0] == item
4  }
5 }
```

The startswith(_:) method first makes sure that the container has at least one item, and then it checks whether the first item in the container matches the given item. This new startswith(_:) method can be used with any type that conforms to the Container protocol, including the stacks and arrays used above, as long as the container's items are equatable.

```
if [9, 9, 9].startsWith(42) {
   print("Starts with 42.")
} else {
   print("Starts with something else.")
}
// Prints "Starts with something else."
```

The generic where clause in the example above requires Item to conform to a protocol, but you can also write a generic where clauses that require Item to be a specific type. For example:

```
1 extension Container where Item == Double {
2 func average() -> Double {
3 var sum = 0.0
```

```
4
             for index in 0..<count {</pre>
 5
                  sum += self[index]
 6
             }
 7
             return sum / Double(count)
         }
 8
     }
 9
10
     print([1260.0, 1200.0, 98.6, 37.0].average())
     // Prints "648.9"
11
```

This example adds an average() method to containers whose Item type is Double. It iterates over the items in the container to add them up, and divides by the container's count to compute the average. It explicitly converts the count from Int to Double to be able to do floating-point division.

You can include multiple requirements in a generic where clause that is part of an extension, just like you can for a generic where clause that you write elsewhere. Separate each requirement in the list with a comma.

Associated Types with a Generic Where Clause

You can include a generic where clause on an associated type. For example, suppose you want to make a version of container that includes an iterator, like what the Sequence protocol uses in the standard library. Here's how you write that:

```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item)
    var count: Int { get }
    subscript(i: Int) -> Item { get }
```

The generic where clause on Iterator requires that the iterator must traverse over elements of the same item type as the container's items, regardless of the iterator's type. The makeIterator() function provides access to a container's iterator.

For a protocol that inherits from another protocol, you add a constraint to an inherited associated type by including the generic where clause in the protocol declaration. For example, the following code declares a ComparableContainer protocol that requires Item to conform to Comparable:

Generic Subscripts

Subscripts can be generic, and they can include generic where clauses. You write the placeholder type name inside angle brackets after subscript, and you write a generic where clause right before the opening curly brace of the subscript's body. For example:

```
1  extension Container {
2   subscript<Indices: Sequence>(indices: Indices) -> [Item]
3   where Indices.Iterator.Element == Int {
4     var result = [Item]()
5     for index in indices {
6         result.append(self[index])
7   }
```

```
8 return result
9 }
10 }
```

This extension to the Container protocol adds a subscript that takes a sequence of indices and returns an array containing the items at each given index. This generic subscript is constrained as follows:

- The generic parameter Indices in angle brackets has to be a type that conforms to the Sequence protocol from the standard library.
- The subscript takes a single parameter, indices, which is an instance of that Indices type.
- The generic where clause requires that the iterator for the sequence must traverse over elements of type Int. This ensures that the indices in the sequence are the same type as the indices used for a container.

Taken together, these constraints mean that the value passed for the indices parameter is a sequence of integers.

Opaque Types

A function or method with an opaque return type hides its return value's type information. Instead of providing a concrete type as the function's return type, the return value is described in terms of the protocols it supports. Hiding type information is useful at boundaries between a module and code that calls into the module, because the underlying type of the return value can remain private. Unlike returning a value whose type is a protocol type, opaque types preserve type identity—the compiler has access to the type information, but clients of the module don't.

The Problem That Opaque Types Solve

For example, suppose you're writing a module that draws ASCII art shapes. The basic characteristic of an ASCII art shape is a draw() function that returns the string representation of that shape, which you can use as the requirement for the Shape protocol:

```
1
    protocol Shape {
2
         func draw() -> String
3
    }
4
    struct Triangle: Shape {
5
         var size: Int
6
7
         func draw() -> String {
             var result = [String]()
8
             for length in 1...size {
9
                 result.append(String(repeating: "*", count:
10
           length))
             }
11
```

```
return result.joined(separator: "\n")
}

let smallTriangle = Triangle(size: 3)
print(smallTriangle.draw())
// *
// *
// **
```

You could use generics to implement operations like flipping a shape vertically, as shown in the code below. However, there's an important limitation to this approach: The flipped result exposes the exact generic types that were used to create it.

```
struct FlippedShape<T: Shape>: Shape {
1
 2
         var shape: T
 3
         func draw() -> String {
             let lines = shape.draw().split(separator: "\n")
 4
             return lines.reversed().joined(separator: "\n")
 5
         }
 6
 7
    }
    let flippedTriangle = FlippedShape(shape: smallTriangle)
8
    print(flippedTriangle.draw())
10
11
     // *
12
```

This approach to defining a JoinedShape<T: Shape, U: Shape> structure that joins two shapes together vertically, like the code below shows, results in types like JoinedShape<FlippedShape<Triangle>, Triangle> from joining a flipped triangle with another triangle.

```
1
    struct JoinedShape<T: Shape, U: Shape>: Shape {
2
         var top: T
3
         var bottom: U
         func draw() -> String {
             return top.draw() + "\n" + bottom.draw()
5
         }
6
7
    }
    let joinedTriangles = JoinedShape(top: smallTriangle, bottom:
8
           flippedTriangle)
9
    print(joinedTriangles.draw())
10
11
12
13
14
15
```

Exposing detailed information about the creation of a shape allows types that aren't meant to be part of the ASCII art module's public interface to leak out because of the need to state the full return type. The code inside the module could build up the same shape in a variety of ways, and other code outside the module that uses the shape shouldn't have to account for the implementation details about the list of transformations. Wrapper types like JoinedShape and FlippedShape don't matter to the module's users, and they shouldn't be visible. The module's public interface consists of operations like joining and flipping a shape, and those operations return another Shape value.

Returning an Opaque Type

You can think of an opaque type like being the reverse of a generic type. Generic types let the code that calls a function pick the type for that function's

parameters and return value in a way that's abstracted away from the function implementation. For example, the function in the following code returns a type that depends on its caller:

```
func max<T>(\_ x: T, \_ y: T) -> T where T: Comparable { ... }
```

The code that calls $\max(_:_:)$ chooses the values for x and y, and the type of those values determines the concrete type of T. The calling code can use any type that conforms to the Comparable protocol. The code inside the function is written in a general way so it can handle whatever type the caller provides. The implementation of $\max(_:_:)$ uses only functionality that all Comparable types share.

Those roles are reversed for a function with an opaque return type. An opaque type lets the function implementation pick the type for the value it returns in a way that's abstracted away from the code that calls the function. For example, the function in the following example returns a trapezoid without exposing the underlying type of that shape.

```
struct Square: Shape {
1
         var size: Int
         func draw() -> String {
3
             let line = String(repeating: "*", count: size)
4
             let result = Array<String>(repeating: line, count:
5
           size)
             return result.joined(separator: "\n")
6
7
         }
    }
8
9
    func makeTrapezoid() -> some Shape {
10
11
         let top = Triangle(size: 2)
         let middle = Square(size: 2)
12
13
         let bottom = FlippedShape(shape: top)
```

```
14
         let trapezoid = JoinedShape(
15
             top: top,
             bottom: JoinedShape(top: middle, bottom: bottom)
16
         )
17
         return trapezoid
18
     }
19
    let trapezoid = makeTrapezoid()
20
21
    print(trapezoid.draw())
22
23
24
26
27
```

The makeTrapezoid() function in this example declares its return type as some Shape; as a result, the function returns a value of some given type that conforms to the Shape protocol, without specifying any particular concrete type. Writing makeTrapezoid() this way lets it express the fundamental aspect of its public interface—the value it returns is a shape—without making the specific types that the shape is made from a part of its public interface. This implementation uses two triangles and a square, but the function could be rewritten to draw a trapezoid in a variety of other ways without changing its return type.

This example highlights the way that an opaque return type is like the reverse of a generic type. The code inside makeTrapezoid() can return any type it needs to, as long as that type conforms to the Shape protocol, like the calling code does for a generic function. The code that calls the function needs to be written in a general way, like the implementation of a generic function, so that it can work with any Shape value that's returned by makeTrapezoid().

You can also combine opaque return types with generics. The functions in the following code both return a value of some type that conforms to the Shape protocol.

```
func flip<T: Shape>(_ shape: T) -> some Shape {
1
         return FlippedShape(shape: shape)
2
3
    }
    func join<T: Shape, U: Shape>(_ top: T, _ bottom: U) -> some
4
           Shape {
         JoinedShape(top: top, bottom: bottom)
5
    }
6
7
    let opaqueJoinedTriangles = join(smallTriangle,
8
           flip(smallTriangle))
    print(opaqueJoinedTriangles.draw())
9
10
    // *
11
12
13
14
15
```

The value of opaqueJoinedTriangles in this example is the same as joinedTriangles in the generics example in the The Problem That Opaque Types Solve section earlier in this chapter. However, unlike the value in that example, flip(_:) and join(_:_:) wrap the underlying types that the generic shape operations return in an opaque return type, which prevents those types from being visible. Both functions are generic because the types they rely on are generic, and the type parameters to the function pass along the type information needed by FlippedShape and JoinedShape.

If a function with an opaque return type returns from multiple places, all of the possible return values must have the same type. For a generic function, that return type can use the function's generic type parameters, but it must still be a single type. For example, here's an *invalid* version of the shape-flipping function that includes a special case for squares:

```
func invalidFlip<T: Shape>(_ shape: T) -> some Shape {
   if shape is Square {
      return shape // Error: return types don't match
   }
   return FlippedShape(shape: shape) // Error: return types
   don't match
}
```

If you call this function with a Square, it returns a Square; otherwise, it returns a FlippedShape. This violates the requirement to return values of only one type and makes invalidFlip(_:) invalid code. One way to fix invalidFlip(_:) is to move the special case for squares into the implementation of FlippedShape, which lets this function always return a FlippedShape value:

```
1
    struct FlippedShape<T: Shape>: Shape {
         var shape: T
2
         func draw() -> String {
3
             if shape is Square {
4
                 return shape.draw()
5
             }
6
7
             let lines = shape.draw().split(separator: "\n")
8
             return lines.reversed().joined(separator: "\n")
9
         }
10
     }
```

The requirement to always return a single type doesn't prevent you from using generics in an opaque return type. Here's an example of a function that incorporates its type parameter into the underlying type of the value it returns:

```
3 }
```

In this case, the underlying type of the return value varies depending on \top : Whatever shape is passed it, repeat(shape:count:) creates and returns an array of that shape. Nevertheless, the return value always has the same underlying type of $[\top]$, so it follows the requirement that functions with opaque return types must return values of only a single type.

Differences Between Opaque Types and Protocol Types

Returning an opaque type looks very similar to using a protocol type as the return type of a function, but these two kinds of return type differ in whether they preserve type identity. An opaque type refers to one specific type, although the caller of the function isn't able to see which type; a protocol type can refer to any type that conforms to the protocol. Generally speaking, protocol types give you more flexibility about the underlying types of the values they store, and opaque types let you make stronger guarantees about those underlying types.

For example, here's a version of flip(_:) that returns a value of protocol type instead of using an opaque return type:

```
func protoFlip<T: Shape>(_ shape: T) -> Shape {
   return FlippedShape(shape: shape)
}
```

This version of $protoFlip(_:)$ has the same body as $flip(_:)$, and it always returns a value of the same type. Unlike $flip(_:)$, the value that $protoFlip(_:)$ returns isn't required to always have the same type—it just has to conform to the Shape protocol. Put another way, $protoFlip(_:)$ makes a much looser API contract with its caller than $flip(_:)$ makes. It reserves the flexibility to return values of multiple types:

```
func protoFlip<T: Shape>(_ shape: T) -> Shape {
   if shape is Square {
```

```
return shape

return shape

return FlippedShape(shape: shape)

}
```

The revised version of the code returns an instance of square or an instance of FlippedShape, depending on what shape is passed in. Two flipped shapes returned by this function might have completely different types. Other valid versions of this function could return values of different types when flipping multiple instances of the same shape. The less specific return type information from protoFlip(_:) means that many operations that depend on type information aren't available on the returned value. For example, it's not possible to write an == operator comparing results returned by this function.

```
1 let protoFlippedTriangle = protoFlip(smallTriangle)
2 let sameThing = protoFlip(smallTriangle)
3 protoFlippedTriangle == sameThing // Error
```

The error on the last line of the example occurs for several reasons. The immediate issue is that the Shape doesn't include an == operator as part of its protocol requirements. If you try adding one, the next issue you'll encounter is that the == operator needs to know the types of its left-hand and right-hand arguments. This sort of operator usually takes arguments of type Self, matching whatever concrete type adopts the protocol, but adding a Self requirement to the protocol doesn't allow for the type erasure that happens when you use the protocol as a type.

Using a protocol type as the return type for a function gives you the flexibility to return any type that conforms to the protocol. However, the cost of that flexibility is that some operations aren't possible on the returned values. The example shows how the == operator isn't available—it depends on specific type information that isn't preserved by using a protocol type.

Another problem with this approach is that the shape transformations don't nest.

The result of flipping a triangle is a value of type <code>Shape</code>, and the <code>protoFlip(_:)</code> function takes an argument of some type that conforms to the <code>Shape</code> protocol. However, a value of a protocol type doesn't conform to that protocol; the value returned by <code>protoFlip(_:)</code> doesn't conform to <code>Shape</code>. This means code like <code>protoFlip(protoFlip(smallTriange))</code> that applies multiple transformations is invalid because the flipped shape isn't a valid argument to <code>protoFlip(_:)</code>.

In contrast, opaque types preserve the identity of the underlying type. Swift can infer associated types, which lets you use an opaque return value in places where a protocol type can't be used as a return value. For example, here's a version of the Container protocol from Generics:

```
protocol Container {
    associatedtype Item
    var count: Int { get }
    subscript(i: Int) -> Item { get }
}
extension Array: Container { }
```

You can't use Container as the return type of a function because that protocol has an associated type. You also can't use it as constraint a generic return type because there isn't enough information outside the function body to infer what the generic type needs to be.

```
1
   // Error: Protocol with associated types can't be used as a
          return type.
   func makeProtocolContainer<T>(item: T) -> Container {
2
        return [item]
3
4
   }
5
6
   // Error: Not enough information to infer C.
7
   func makeProtocolContainer<T, C: Container>(item: T) -> C {
8
        return [item]
```

```
9 }
```

Using the opaque type some Container as a return type expresses the desired API contract—the function returns a container, but declines to specify the container's type:

```
func makeOpaqueContainer<T>(item: T) -> some Container {
   return [item]
}

let opaqueContainer = makeOpaqueContainer(item: 12)

let twelve = opaqueContainer[0]

print(type(of: twelve))

// Prints "Int"
```

The type of twelve is inferred to be Int, which illustrates the fact that type inference works with opaque types. In the implementation of makeOpaqueContainer(item:), the underlying type of the opaque container is [T]. In this case, T is Int, so the return value is an array of integers and the Item associated type is inferred to be Int. The subscript on Container returns Item, which means that the type of twelve is also inferred to be Int.

Automatic Reference Counting

Swift uses *Automatic Reference Counting* (ARC) to track and manage your app's memory usage. In most cases, this means that memory management "just works" in Swift, and you do not need to think about memory management yourself. ARC automatically frees up the memory used by class instances when those instances are no longer needed.

However, in a few cases ARC requires more information about the relationships between parts of your code in order to manage memory for you. This chapter describes those situations and shows how you enable ARC to manage all of your app's memory. Using ARC in Swift is very similar to the approach described in Transitioning to ARC Release Notes for using ARC with Objective-C.

Reference counting applies only to instances of classes. Structures and enumerations are value types, not reference types, and are not stored and passed by reference.

How ARC Works

Every time you create a new instance of a class, ARC allocates a chunk of memory to store information about that instance. This memory holds information about the type of the instance, together with the values of any stored properties associated with that instance.

Additionally, when an instance is no longer needed, ARC frees up the memory used by that instance so that the memory can be used for other purposes instead. This ensures that class instances do not take up space in memory when they are no longer needed.

However, if ARC were to deallocate an instance that was still in use, it would no longer be possible to access that instance's properties, or call that instance's methods. Indeed, if you tried to access the instance, your app would most likely crash.

To make sure that instances don't disappear while they are still needed, ARC tracks how many properties, constants, and variables are currently referring to each class instance. ARC will not deallocate an instance as long as at least one active reference to that instance still exists.

To make this possible, whenever you assign a class instance to a property, constant, or variable, that property, constant, or variable makes a *strong reference* to the instance. The reference is called a "strong" reference because it keeps a firm hold on that instance, and does not allow it to be deallocated for as long as that strong reference remains.

ARC in Action

Here's an example of how Automatic Reference Counting works. This example starts with a simple class called Person, which defines a stored constant property called name:

```
class Person {
1
         let name: String
         init(name: String) {
             self.name = name
4
             print("\(name) is being initialized")
5
         }
6
         deinit {
             print("\(name) is being deinitialized")
9
         }
10
     }
```

The Person class has an initializer that sets the instance's name property and prints a message to indicate that initialization is underway. The Person class also has a deinitializer that prints a message when an instance of the class is deallocated.

The next code snippet defines three variables of type Person?, which are used to set up multiple references to a new Person instance in subsequent code snippets. Because these variables are of an optional type (Person?, not Person), they are automatically initialized with a value of nil, and do not currently reference a Person instance.

```
var reference1: Person?
var reference2: Person?
var reference3: Person?
```

You can now create a new Person instance and assign it to one of these three variables:

```
reference1 = Person(name: "John Appleseed")
// Prints "John Appleseed is being initialized"
```

Note that the message "John Appleseed is being initialized" is printed at the point that you call the Person class's initializer. This confirms that initialization has taken place.

Because the new Person instance has been assigned to the reference1 variable, there is now a strong reference from reference1 to the new Person instance. Because there is at least one strong reference, ARC makes sure that this Person is kept in memory and is not deallocated.

If you assign the same Person instance to two more variables, two more strong references to that instance are established:

```
reference2 = reference1
reference3 = reference1
```

There are now *three* strong references to this single Person instance.

If you break two of these strong references (including the original reference) by assigning nil to two of the variables, a single strong reference remains, and the Person instance is not deallocated:

```
1 reference1 = nil
2 reference2 = nil
```

ARC does not deallocate the Person instance until the third and final strong reference is broken, at which point it's clear that you are no longer using the Person instance:

```
1 reference3 = nil
2 // Prints "John Appleseed is being deinitialized"
```

Strong Reference Cycles Between Class Instances

In the examples above, ARC is able to track the number of references to the new Person instance you create and to deallocate that Person instance when it's no longer needed.

However, it's possible to write code in which an instance of a class *never* gets to a point where it has zero strong references. This can happen if two class instances hold a strong reference to each other, such that each instance keeps the other alive. This is known as a *strong reference cycle*.

You resolve strong reference cycles by defining some of the relationships between classes as weak or unowned references instead of as strong references. This process is described in Resolving Strong Reference Cycles Between Class Instances. However, before you learn how to resolve a strong reference cycle, it's useful to understand how such a cycle is caused.

Here's an example of how a strong reference cycle can be created by accident. This example defines two classes called Person and Apartment, which model a block of apartments and its residents:

```
class Person {
let name: String
```

```
3
        init(name: String) { self.name = name }
        var apartment: Apartment?
4
        deinit { print("\(name\) is being deinitialized") }
5
6
    }
7
    class Apartment {
8
        let unit: String
        init(unit: String) { self.unit = unit }
10
         var tenant: Person?
11
        deinit { print("Apartment \(unit) is being deinitialized")
12
           }
13
    }
```

Every Person instance has a name property of type String and an optional apartment property that is initially nil. The apartment property is optional, because a person may not always have an apartment.

Similarly, every Apartment instance has a unit property of type String and has an optional tenant property that is initially nil. The tenant property is optional because an apartment may not always have a tenant.

Both of these classes also define a deinitializer, which prints the fact that an instance of that class is being deinitialized. This enables you to see whether instances of Person and Apartment are being deallocated as expected.

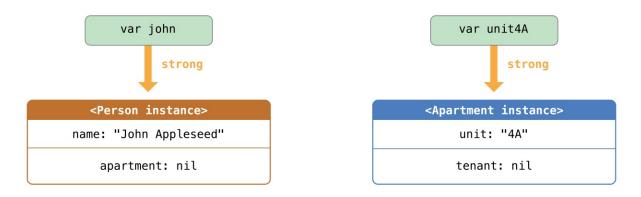
This next code snippet defines two variables of optional type called john and unit4A, which will be set to a specific Apartment and Person instance below. Both of these variables have an initial value of nil, by virtue of being optional:

```
var john: Person?
var unit4A: Apartment?
```

You can now create a specific Person instance and Apartment instance and assign these new instances to the john and unit4A variables:

```
john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")
```

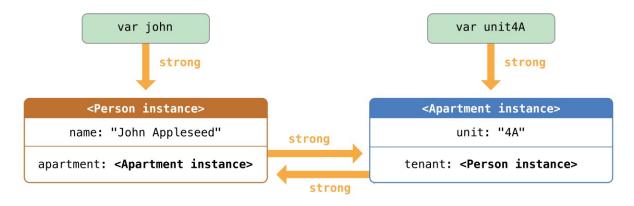
Here's how the strong references look after creating and assigning these two instances. The john variable now has a strong reference to the new Person instance, and the unit4A variable has a strong reference to the new Apartment instance:



You can now link the two instances together so that the person has an apartment, and the apartment has a tenant. Note that an exclamation mark (!) is used to unwrap and access the instances stored inside the john and unit4A optional variables, so that the properties of those instances can be set:

```
john!.apartment = unit4A
unit4A!.tenant = john
```

Here's how the strong references look after you link the two instances together:



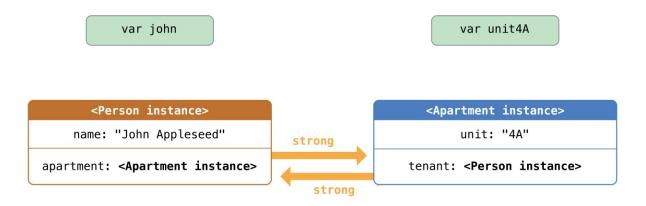
Unfortunately, linking these two instances creates a strong reference cycle

between them. The Person instance now has a strong reference to the Apartment instance, and the Apartment instance has a strong reference to the Person instance. Therefore, when you break the strong references held by the john and unit4A variables, the reference counts do not drop to zero, and the instances are not deallocated by ARC:

```
john = nil
unit4A = nil
```

Note that neither deinitializer was called when you set these two variables to nil. The strong reference cycle prevents the Person and Apartment instances from ever being deallocated, causing a memory leak in your app.

Here's how the strong references look after you set the john and unit4A variables to nil:



The strong references between the Person instance and the Apartment instance remain and cannot be broken.

Resolving Strong Reference Cycles Between Class Instances

Swift provides two ways to resolve strong reference cycles when you work with properties of class type: weak references and unowned references.

Weak and unowned references enable one instance in a reference cycle to refer to the other instance *without* keeping a strong hold on it. The instances can then

refer to each other without creating a strong reference cycle.

Use a weak reference when the other instance has a shorter lifetime—that is, when the other instance can be deallocated first. In the Apartment example above, it's appropriate for an apartment to be able to have no tenant at some point in its lifetime, and so a weak reference is an appropriate way to break the reference cycle in this case. In contrast, use an unowned reference when the other instance has the same lifetime or a longer lifetime.

Weak References

A *weak reference* is a reference that does not keep a strong hold on the instance it refers to, and so does not stop ARC from disposing of the referenced instance. This behavior prevents the reference from becoming part of a strong reference cycle. You indicate a weak reference by placing the weak keyword before a property or variable declaration.

Because a weak reference does not keep a strong hold on the instance it refers to, it's possible for that instance to be deallocated while the weak reference is still referring to it. Therefore, ARC automatically sets a weak reference to nil when the instance that it refers to is deallocated. And, because weak references need to allow their value to be changed to nil at runtime, they are always declared as variables, rather than constants, of an optional type.

You can check for the existence of a value in the weak reference, just like any other optional value, and you will never end up with a reference to an invalid instance that no longer exists.

```
NOTE
```

Property observers aren't called when ARC sets a weak reference to nil.

The example below is identical to the Person and Apartment example from above, with one important difference. This time around, the Apartment type's tenant property is declared as a weak reference:

```
1 class Person {
```

```
2
        let name: String
        init(name: String) { self.name = name }
3
        var apartment: Apartment?
4
5
        deinit { print("\(name\) is being deinitialized") }
6
    }
7
    class Apartment {
8
        let unit: String
9
        init(unit: String) { self.unit = unit }
10
        weak var tenant: Person?
11
        deinit { print("Apartment \(unit\) is being deinitialized")
12
           }
13
    }
```

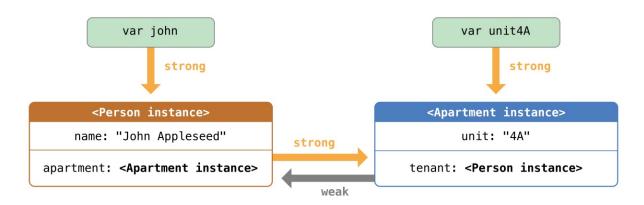
The strong references from the two variables (john and unit4A) and the links between the two instances are created as before:

```
var john: Person?
var unit4A: Apartment?

john = Person(name: "John Appleseed")
unit4A = Apartment(unit: "4A")

john!.apartment = unit4A
unit4A!.tenant = john
```

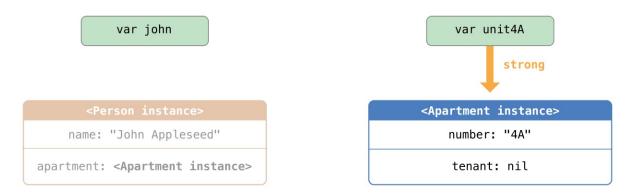
Here's how the references look now that you've linked the two instances together:



The Person instance still has a strong reference to the Apartment instance, but the Apartment instance now has a *weak* reference to the Person instance. This means that when you break the strong reference held by the john variable by setting it to nil, there are no more strong references to the Person instance:

```
john = nil
// Prints "John Appleseed is being deinitialized"
```

Because there are no more strong references to the Person instance, it's deallocated and the tenant property is set to nil:



The only remaining strong reference to the Apartment instance is from the unit4A variable. If you break *that* strong reference, there are no more strong references to the Apartment instance:

```
unit4A = nil
// Prints "Apartment 4A is being deinitialized"
```

Because there are no more strong references to the Apartment instance, it too is

deallocated:

var john

var unit4A

<Person instance>

name: "John Appleseed"

apartment: <Apartment instance>

<Apartment instance>

unit: "4A"

tenant: nil

NOTE

In systems that use garbage collection, weak pointers are sometimes used to implement a simple caching mechanism because objects with no strong references are deallocated only when memory pressure triggers garbage collection. However, with ARC, values are deallocated as soon as their last strong reference is removed, making weak references unsuitable for such a purpose.

Unowned References

Like a weak reference, an *unowned reference* does not keep a strong hold on the instance it refers to. Unlike a weak reference, however, an unowned reference is used when the other instance has the same lifetime or a longer lifetime. You indicate an unowned reference by placing the unowned keyword before a property or variable declaration.

An unowned reference is expected to always have a value. As a result, ARC never sets an unowned reference's value to nil, which means that unowned references are defined using non-optional types.

IMPORTANT

Use an unowned reference only when you are sure that the reference *always* refers to an instance that has not been deallocated.

If you try to access the value of an unowned reference after that instance has been deallocated, you'll get a runtime error.

The following example defines two classes, Customer and CreditCard, which model a bank customer and a possible credit card for that customer. These two

classes each store an instance of the other class as a property. This relationship has the potential to create a strong reference cycle.

The relationship between Customer and CreditCard is slightly different from the relationship between Apartment and Person seen in the weak reference example above. In this data model, a customer may or may not have a credit card, but a credit card will *always* be associated with a customer. A CreditCard instance never outlives the Customer that it refers to. To represent this, the Customer class has an optional card property, but the CreditCard class has an unowned (and non-optional) customer property.

Furthermore, a new CreditCard instance can *only* be created by passing a number value and a customer instance to a custom CreditCard initializer. This ensures that a CreditCard instance always has a customer instance associated with it when the CreditCard instance is created.

Because a credit card will always have a customer, you define its customer property as an unowned reference, to avoid a strong reference cycle:

```
1
    class Customer {
2
         let name: String
         var card: CreditCard?
3
         init(name: String) {
4
             self.name = name
5
6
         }
         deinit { print("\(name\) is being deinitialized") }
7
8
    }
9
10
    class CreditCard {
         let number: UInt64
11
         unowned let customer: Customer
12
         init(number: UInt64, customer: Customer) {
13
             self.number = number
14
             self.customer = customer
15
```

```
16  }
17  deinit { print("Card #\(number) is being deinitialized") }
18 }
```

NOTE

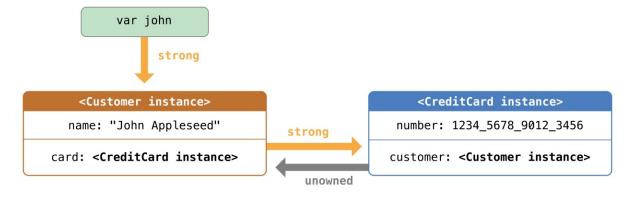
The number property of the CreditCard class is defined with a type of UInt64 rather than Int, to ensure that the number property's capacity is large enough to store a 16-digit card number on both 32-bit and 64-bit systems.

This next code snippet defines an optional Customer variable called john, which will be used to store a reference to a specific customer. This variable has an initial value of nil, by virtue of being optional:

```
var john: Customer?
```

You can now create a Customer instance, and use it to initialize and assign a new CreditCard instance as that customer's card property:

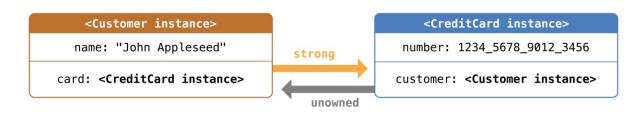
Here's how the references look, now that you've linked the two instances:



The Customer instance now has a strong reference to the CreditCard instance, and the CreditCard instance has an unowned reference to the Customer instance.

Because of the unowned customer reference, when you break the strong reference held by the john variable, there are no more strong references to the Customer instance:

var john



Because there are no more strong references to the Customer instance, it's deallocated. After this happens, there are no more strong references to the CreditCard instance, and it too is deallocated:

```
john = nil
// Prints "John Appleseed is being deinitialized"
// Prints "Card #1234567890123456 is being deinitialized"
```

The final code snippet above shows that the deinitializers for the Customer instance and CreditCard instance both print their "deinitialized" messages after the john variable is set to nil.

NOTE

The examples above show how to use *safe* unowned references. Swift also provides *unsafe* unowned references for cases where you need to disable runtime safety checks—for example, for performance reasons. As with all unsafe operations, you take on the responsibility for checking that code for safety.

You indicate an unsafe unowned reference by writing unowned (unsafe). If you try to access an unsafe unowned reference after the instance that it refers to is deallocated, your program will try to access the memory location where the instance used to be, which is an unsafe operation.

Unowned References and Implicitly Unwrapped Optional

Properties

The examples for weak and unowned references above cover two of the more common scenarios in which it's necessary to break a strong reference cycle.

The Person and Apartment example shows a situation where two properties, both of which are allowed to be nil, have the potential to cause a strong reference cycle. This scenario is best resolved with a weak reference.

The Customer and CreditCard example shows a situation where one property that is allowed to be nil and another property that cannot be nil have the potential to cause a strong reference cycle. This scenario is best resolved with an unowned reference.

However, there is a third scenario, in which *both* properties should always have a value, and neither property should ever be nil once initialization is complete. In this scenario, it's useful to combine an unowned property on one class with an implicitly unwrapped optional property on the other class.

This enables both properties to be accessed directly (without optional unwrapping) once initialization is complete, while still avoiding a reference cycle. This section shows you how to set up such a relationship.

The example below defines two classes, <code>Country</code> and <code>City</code>, each of which stores an instance of the other class as a property. In this data model, every country must always have a capital city, and every city must always belong to a country. To represent this, the <code>Country</code> class has a <code>capitalCity</code> property, and the <code>City</code> class has a <code>country</code> property:

```
class Country {
  let name: String
  var capitalCity: City!
  init(name: String, capitalName: String) {
    self.name = name
    self.capitalCity = City(name: capitalName, country: self)
}
```

```
8
    }
9
    class City {
10
11
         let name: String
12
         unowned let country: Country
         init(name: String, country: Country) {
13
             self.name = name
14
             self.country = country
15
16
         }
17
    }
```

To set up the interdependency between the two classes, the initializer for City takes a Country instance, and stores this instance in its country property.

The initializer for City is called from within the initializer for Country. However, the initializer for Country cannot pass self to the City initializer until a new Country instance is fully initialized, as described in Two-Phase Initialization.

To cope with this requirement, you declare the capitalCity property of Country as an implicitly unwrapped optional property, indicated by the exclamation mark at the end of its type annotation (City!). This means that the capitalCity property has a default value of nil, like any other optional, but can be accessed without the need to unwrap its value as described in Implicitly Unwrapped
Optionals.

Because capitalcity has a default nil value, a new Country instance is considered fully initialized as soon as the Country instance sets its name property within its initializer. This means that the Country initializer can start to reference and pass around the implicit self property as soon as the name property is set. The Country initializer can therefore pass self as one of the parameters for the City initializer when the Country initializer is setting its own capitalCity property.

All of this means that you can create the Country and City instances in a single statement, without creating a strong reference cycle, and the capitalCity

property can be accessed directly, without needing to use an exclamation mark to unwrap its optional value:

In the example above, the use of an implicitly unwrapped optional means that all of the two-phase class initializer requirements are satisfied. The capitalCity property can be used and accessed like a non-optional value once initialization is complete, while still avoiding a strong reference cycle.

Strong Reference Cycles for Closures

You saw above how a strong reference cycle can be created when two class instance properties hold a strong reference to each other. You also saw how to use weak and unowned references to break these strong reference cycles.

A strong reference cycle can also occur if you assign a closure to a property of a class instance, and the body of that closure captures the instance. This capture might occur because the closure's body accesses a property of the instance, such as self.someProperty, or because the closure calls a method on the instance, such as self.someMethod(). In either case, these accesses cause the closure to "capture" self, creating a strong reference cycle.

This strong reference cycle occurs because closures, like classes, are *reference types*. When you assign a closure to a property, you are assigning a *reference* to that closure. In essence, it's the same problem as above—two strong references are keeping each other alive. However, rather than two class instances, this time it's a class instance and a closure that are keeping each other alive.

Swift provides an elegant solution to this problem, known as a *closure capture list*. However, before you learn how to break a strong reference cycle with a

closure capture list, it's useful to understand how such a cycle can be caused.

The example below shows how you can create a strong reference cycle when using a closure that references self. This example defines a class called HTMLElement, which provides a simple model for an individual element within an HTML document:

```
class HTMLElement {
1
 2
 3
         let name: String
         let text: String?
 4
 5
         lazy var asHTML: () -> String = {
 6
             if let text = self.text {
 7
                 return "<\(self.name)>\(text)</\(self.name)>"
 8
9
             } else {
                 return "<\(self.name) />"
10
11
             }
12
         }
13
         init(name: String, text: String? = nil) {
14
15
             self.name = name
             self.text = text
16
         }
17
18
19
         deinit {
             print("\(name) is being deinitialized")
20
21
         }
22
23
     }
```

The HTMLElement class defines a name property, which indicates the name of the element, such as "h1" for a heading element, "p" for a paragraph element, or "br" for a line break element. HTMLElement also defines an optional text property, which you can set to a string that represents the text to be rendered within that HTML element.

In addition to these two simple properties, the HTMLElement class defines a lazy property called asHTML. This property references a closure that combines name and text into an HTML string fragment. The asHTML property is of type () -> string, or "a function that takes no parameters, and returns a String value".

By default, the ashtml property is assigned a closure that returns a string representation of an HTML tag. This tag contains the optional text value if it exists, or no text content if text does not exist. For a paragraph element, the closure would return "some text" or "", depending on whether the text property equals "some text" or nil.

The ashtml property is named and used somewhat like an instance method. However, because ashtml is a closure property rather than an instance method, you can replace the default value of the ashtml property with a custom closure, if you want to change the HTML rendering for a particular HTML element.

For example, the ashtml property could be set to a closure that defaults to some text if the text property is nil, in order to prevent the representation from returning an empty HTML tag:

```
1 let heading = HTMLElement(name: "h1")
2 let defaultText = "some default text"
3 heading.asHTML = {
4    return "<\(heading.name)>\(heading.text ?? defaultText)</\( (heading.name)>"
5 }
6 print(heading.asHTML())
7 // Prints "<h1>some default text</h1>"
```

The ashTML property is declared as a lazy property, because it's only needed if and when the element actually needs to be rendered as a string value for some HTML output target. The fact that ashTML is a lazy property means that you can refer to self within the default closure, because the lazy property will not be accessed until after initialization has been completed and self is known to exist.

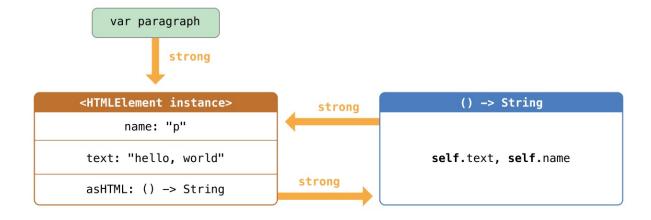
The HTMLElement class provides a single initializer, which takes a name argument and (if desired) a text argument to initialize a new element. The class also defines a deinitializer, which prints a message to show when an HTMLElement instance is deallocated.

Here's how you use the HTMLElement class to create and print a new instance:

NOTE

The paragraph variable above is defined as an *optional* HTMLElement, so that it can be set to nil below to demonstrate the presence of a strong reference cycle.

Unfortunately, the HTMLElement class, as written above, creates a strong reference cycle between an HTMLElement instance and the closure used for its default asHTML value. Here's how the cycle looks:



The instance's ashtml property holds a strong reference to its closure. However, because the closure refers to self within its body (as a way to reference self.name and self.text), the closure *captures* self, which means that it holds a strong reference back to the htmlelement instance. A strong reference cycle is created between the two. (For more information about capturing values in a closure, see <u>Capturing Values</u>.)

NOTE

Even though the closure refers to Self multiple times, it only captures one strong reference to the HTMLElement instance.

If you set the paragraph variable to nil and break its strong reference to the HTMLElement instance, neither the HTMLElement instance nor its closure are deallocated, because of the strong reference cycle:

```
paragraph = nil
```

Note that the message in the HTMLElement deinitializer is not printed, which shows that the HTMLElement instance is not deallocated.

Resolving Strong Reference Cycles for Closures

You resolve a strong reference cycle between a closure and a class instance by defining a *capture list* as part of the closure's definition. A capture list defines the rules to use when capturing one or more reference types within the closure's body. As with strong reference cycles between two class instances, you declare each captured reference to be a weak or unowned reference rather than a strong reference. The appropriate choice of weak or unowned depends on the relationships between the different parts of your code.

NOTE

Swift requires you to write self.someProperty or self.someMethod() (rather than just someProperty or someMethod()) whenever you refer to a member of self within a closure. This helps you remember that it's possible to capture self by accident.

Defining a Capture List

Each item in a capture list is a pairing of the weak or unowned keyword with a reference to a class instance (such as self) or a variable initialized with some value (such as delegate = self.delegate). These pairings are written within a pair of square braces, separated by commas.

Place the capture list before a closure's parameter list and return type if they are provided:

```
1 lazy var someClosure = {
2     [unowned self, weak delegate = self.delegate]
3     (index: Int, stringToProcess: String) -> String in
4     // closure body goes here
5 }
```

If a closure does not specify a parameter list or return type because they can be inferred from context, place the capture list at the very start of the closure, followed by the in keyword:

```
1 lazy var someClosure = {
2    [unowned self, weak delegate = self.delegate] in
3    // closure body goes here
4 }
```

Weak and Unowned References

Define a capture in a closure as an unowned reference when the closure and the instance it captures will always refer to each other, and will always be deallocated at the same time.

Conversely, define a capture as a weak reference when the captured reference may become nil at some point in the future. Weak references are always of an optional type, and automatically become nil when the instance they reference is deallocated. This enables you to check for their existence within the closure's

body.

```
NOTE
```

If the captured reference will never become nil, it should always be captured as an unowned reference, rather than a weak reference.

An unowned reference is the appropriate capture method to use to resolve the strong reference cycle in the HTMLElement example from Strong Reference
Cycles for Closures above. Here's how you write the HTMLElement class to avoid the cycle:

```
1
    class HTMLElement {
2
3
         let name: String
         let text: String?
4
5
         lazy var asHTML: () -> String = {
6
             [unowned self] in
7
             if let text = self.text {
8
                 return "<\(self.name)>\(text)</\(self.name)>"
9
10
             } else {
11
                 return "<\(self.name) />"
12
             }
         }
13
14
         init(name: String, text: String? = nil) {
15
             self.name = name
16
             self.text = text
17
18
         }
19
         deinit {
20
```

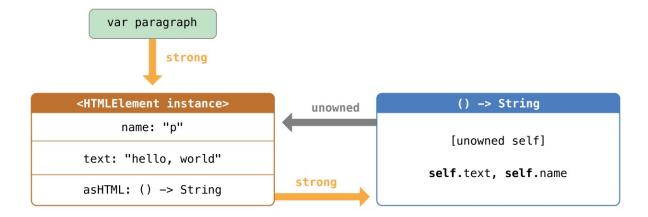
```
print("\(name) is being deinitialized")

print("\(name) is being deinitia
```

This implementation of HTMLElement is identical to the previous implementation, apart from the addition of a capture list within the asHTML closure. In this case, the capture list is [unowned self], which means "capture self as an unowned reference rather than a strong reference".

You can create and print an HTMLElement instance as before:

Here's how the references look with the capture list in place:



This time, the capture of self by the closure is an unowned reference, and does not keep a strong hold on the HTMLElement instance it has captured. If you set the strong reference from the paragraph variable to nil, the HTMLElement instance is deallocated, as can be seen from the printing of its deinitializer message in the example below:

```
paragraph = nil
// Prints "p is being deinitialized"
```

For more information about capture lists, see <u>Capture Lists</u>.

Memory Safety

By default, Swift prevents unsafe behavior from happening in your code. For example, Swift ensures that variables are initialized before they're used, memory isn't accessed after it's been deallocated, and array indices are checked for out-of-bounds errors.

Swift also makes sure that multiple accesses to the same area of memory don't conflict, by requiring code that modifies a location in memory to have exclusive access to that memory. Because Swift manages memory automatically, most of the time you don't have to think about accessing memory at all. However, it's important to understand where potential conflicts can occur, so you can avoid writing code that has conflicting access to memory. If your code does contain conflicts, you'll get a compile-time or runtime error.

Understanding Conflicting Access to Memory

Access to memory happens in your code when you do things like set the value of a variable or pass an argument to a function. For example, the following code contains both a read access and a write access:

```
// A write access to the memory where one is stored.
var one = 1

// A read access from the memory where one is stored.
print("We're number \((one)!")
```

A conflicting access to memory can occur when different parts of your code are trying to access the same location in memory at the same time. Multiple accesses to a location in memory at the same time can produce unpredictable or inconsistent behavior. In Swift, there are ways to modify a value that span several lines of code, making it possible to attempt to access a value in the

middle of its own modification.

You can see a similar problem by thinking about how you update a budget that's written on a piece of paper. Updating the budget is a two-step process: First you add the items' names and prices, and then you change the total amount to reflect the items currently on the list. Before and after the update, you can read any information from the budget and get a correct answer, as shown in the figure below.



While you're adding items to the budget, it's in a temporary, invalid state because the total amount hasn't been updated to reflect the newly added items. Reading the total amount during the process of adding an item gives you incorrect information.

This example also demonstrates a challenge you may encounter when fixing conflicting access to memory: There are sometimes multiple ways to fix the conflict that produce different answers, and it's not always obvious which answer is correct. In this example, depending on whether you wanted the original total amount or the updated total amount, either \$5 or \$320 could be the correct answer. Before you can fix the conflicting access, you have to determine what it was intended to do.

NOTE

If you've written concurrent or multithreaded code, conflicting access to memory might be a familiar problem. However, the conflicting access discussed here can happen on a single thread and *doesn't* involve concurrent or multithreaded code.

If you have conflicting access to memory from within a single thread, Swift guarantees that you'll get an error at either compile time or runtime. For multithreaded code, use <u>Thread Sanitizer</u> to help detect conflicting access across threads.

Characteristics of Memory Access

There are three characteristics of memory access to consider in the context of conflicting access: whether the access is a read or a write, the duration of the access, and the location in memory being accessed. Specifically, a conflict occurs if you have two accesses that meet all of the following conditions:

- At least one is a write access.
- They access the same location in memory.
- Their durations overlap.

The difference between a read and write access is usually obvious: a write access changes the location in memory, but a read access doesn't. The location in memory refers to what is being accessed—for example, a variable, constant, or property. The duration of a memory access is either instantaneous or long-term.

An access is *instantaneous* if it's not possible for other code to run after that access starts but before it ends. By their nature, two instantaneous accesses can't happen at the same time. Most memory access is instantaneous. For example, all the read and write accesses in the code listing below are instantaneous:

```
func oneMore(than number: Int) -> Int {
    return number + 1
}

var myNumber = 1
myNumber = oneMore(than: myNumber)
print(myNumber)
// Prints "2"
```

However, there are several ways to access memory, called *long-term* accesses, that span the execution of other code. The difference between instantaneous access and long-term access is that it's possible for other code to run after a long-term access starts but before it ends, which is called *overlap*. A long-term access can overlap with other long-term accesses and instantaneous accesses.

Overlapping accesses appear primarily in code that uses in-out parameters in functions and methods or mutating methods of a structure. The specific kinds of Swift code that use long-term accesses are discussed in the sections below.

Conflicting Access to In-Out Parameters

A function has long-term write access to all of its in-out parameters. The write access for an in-out parameter starts after all of the non-in-out parameters have been evaluated and lasts for the entire duration of that function call. If there are multiple in-out parameters, the write accesses start in the same order as the parameters appear.

One consequence of this long-term write access is that you can't access the original variable that was passed as in-out, even if scoping rules and access control would otherwise permit it—any access to the original creates a conflict. For example:

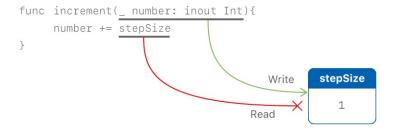
```
var stepSize = 1

func increment(_ number: inout Int) {
   number += stepSize
}

increment(&stepSize)

// Error: conflicting accesses to stepSize
```

In the code above, stepSize is a global variable, and it is normally accessible from within increment(_:). However, the read access to stepSize overlaps with the write access to number. As shown in the figure below, both number and stepSize refer to the same location in memory. The read and write accesses refer to the same memory and they overlap, producing a conflict.



One way to solve this conflict is to make an explicit copy of stepSize:

```
// Make an explicit copy.
var copyOfStepSize = stepSize
increment(&copyOfStepSize)

// Update the original.
stepSize = copyOfStepSize
// stepSize is now 2
```

When you make a copy of stepSize before calling increment(_:), it's clear that the value of copyOfStepSize is incremented by the current step size. The read access ends before the write access starts, so there isn't a conflict.

Another consequence of long-term write access to in-out parameters is that passing a single variable as the argument for multiple in-out parameters of the same function produces a conflict. For example:

```
func balance(_ x: inout Int, _ y: inout Int) {
1
        let sum = x + y
2
3
        x = sum / 2
4
        y = sum - x
5
   }
   var playerOneScore = 42
6
7
   var playerTwoScore = 30
   balance(&playerOneScore, &playerTwoScore) // OK
8
   balance(&playerOneScore, &playerOneScore)
9
```

The balance(_:_:) function above modifies its two parameters to divide the total value evenly between them. Calling it with playerOneScore and playerTwoScore as arguments doesn't produce a conflict—there are two write accesses that overlap in time, but they access different locations in memory. In contrast, passing playerOneScore as the value for both parameters produces a conflict because it tries to perform two write accesses to the same location in memory at the same time.

NOTE

Because operators are functions, they can also have long-term accesses to their in-out parameters. For example, if balance(_:_:) was an operator function named <^>, writing playerOneScore playerOneScore would result in the same conflict as balance(&playerOneScore, &playerOneScore).

Conflicting Access to self in Methods

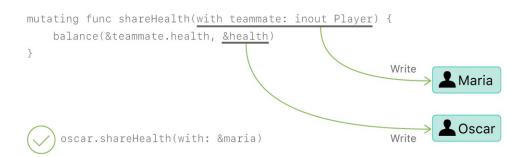
A mutating method on a structure has write access to self for the duration of the method call. For example, consider a game where each player has a health amount, which decreases when taking damage, and an energy amount, which decreases when using special abilities.

```
1
   struct Player {
        var name: String
3
        var health: Int
        var energy: Int
4
5
        static let maxHealth = 10
6
        mutating func restoreHealth() {
7
            health = Player.maxHealth
8
9
        }
```

In the restoreHealth() method above, a write access to self starts at the beginning of the method and lasts until the method returns. In this case, there's no other code inside restoreHealth() that could have an overlapping access to the properties of a Player instance. The shareHealth(with:) method below takes another Player instance as an in-out parameter, creating the possibility of overlapping accesses.

```
1
   extension Player {
2
        mutating func shareHealth(with teammate: inout Player) {
            balance(&teammate.health, &health)
3
        }
4
   }
5
6
   var oscar = Player(name: "Oscar", health: 10, energy: 10)
7
   var maria = Player(name: "Maria", health: 5, energy: 10)
8
   oscar shareHealth(with: &maria) // OK
9
```

In the example above, calling the shareHealth(with:) method for Oscar's player to share health with Maria's player doesn't cause a conflict. There's a write access to oscar during the method call because oscar is the value of self in a mutating method, and there's a write access to maria for the same duration because maria was passed as an in-out parameter. As shown in the figure below, they access different locations in memory. Even though the two write accesses overlap in time, they don't conflict.

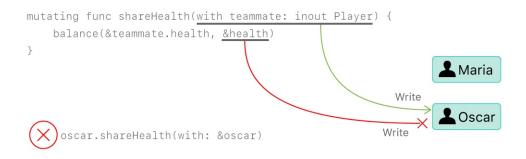


However, if you pass oscar as the argument to shareHealth(with:), there's a conflict:

```
oscar.shareHealth(with: &oscar)

// Error: conflicting accesses to oscar
```

The mutating method needs write access to self for the duration of the method, and the in-out parameter needs write access to teammate for the same duration. Within the method, both self and teammate refer to the same location in memory —as shown in the figure below. The two write accesses refer to the same memory and they overlap, producing a conflict.



Conflicting Access to Properties

Types like structures, tuples, and enumerations are made up of individual constituent values, such as the properties of a structure or the elements of a tuple. Because these are value types, mutating any piece of the value mutates the whole value, meaning read or write access to one of the properties requires read or write access to the whole value. For example, overlapping write accesses to the elements of a tuple produces a conflict:

```
var playerInformation = (health: 10, energy: 20)
balance(&playerInformation.health, &playerInformation.energy)
// Error: conflicting access to properties of playerInformation
```

In the example above, calling balance(_:_:) on the elements of a tuple produces

a conflict because there are overlapping write accesses to playerInformation. Both playerInformation.health and playerInformation.energy are passed as in-out parameters, which means balance(_:_:) needs write access to them for the duration of the function call. In both cases, a write access to the tuple element requires a write access to the entire tuple. This means there are two write accesses to playerInformation with durations that overlap, causing a conflict.

The code below shows that the same error appears for overlapping write accesses to the properties of a structure that's stored in a global variable.

```
var holly = Player(name: "Holly", health: 10, energy: 10)
balance(&holly.health, &holly.energy) // Error
```

In practice, most access to the properties of a structure can overlap safely. For example, if the variable holly in the example above is changed to a local variable instead of a global variable, the compiler can prove that overlapping access to stored properties of the structure is safe:

```
func someFunction() {
   var oscar = Player(name: "Oscar", health: 10, energy: 10)
   balance(&oscar.health, &oscar.energy) // OK
}
```

In the example above, Oscar's health and energy are passed as the two in-out parameters to balance(_:_:). The compiler can prove that memory safety is preserved because the two stored properties don't interact in any way.

The restriction against overlapping access to properties of a structure isn't always necessary to preserve memory safety. Memory safety is the desired guarantee, but exclusive access is a stricter requirement than memory safety—which means some code preserves memory safety, even though it violates exclusive access to memory. Swift allows this memory-safe code if the compiler can prove that the nonexclusive access to memory is still safe. Specifically, it can prove that overlapping access to properties of a structure is safe if the following conditions apply:

- You're accessing only stored properties of an instance, not computed properties or class properties.
- The structure is the value of a local variable, not a global variable.
- The structure is either not captured by any closures, or it's captured only by nonescaping closures.

If the compiler can't prove the access is safe, it doesn't allow the access.

Access Control

Access control restricts access to parts of your code from code in other source files and modules. This feature enables you to hide the implementation details of your code, and to specify a preferred interface through which that code can be accessed and used.

You can assign specific access levels to individual types (classes, structures, and enumerations), as well as to properties, methods, initializers, and subscripts belonging to those types. Protocols can be restricted to a certain context, as can global constants, variables, and functions.

In addition to offering various levels of access control, Swift reduces the need to specify explicit access control levels by providing default access levels for typical scenarios. Indeed, if you are writing a single-target app, you may not need to specify explicit access control levels at all.

NOTE

The various aspects of your code that can have access control applied to them (properties, types, functions, and so on) are referred to as "entities" in the sections below, for brevity.

Modules and Source Files

Swift's access control model is based on the concept of modules and source files.

A *module* is a single unit of code distribution—a framework or application that is built and shipped as a single unit and that can be imported by another module with Swift's import keyword.

Each build target (such as an app bundle or framework) in Xcode is treated as a separate module in Swift. If you group together aspects of your app's code as a stand-alone framework—perhaps to encapsulate and reuse that code across multiple applications—then everything you define within that framework will be

part of a separate module when it's imported and used within an app, or when it's used within another framework.

A *source file* is a single Swift source code file within a module (in effect, a single file within an app or framework). Although it's common to define individual types in separate source files, a single source file can contain definitions for multiple types, functions, and so on.

Access Levels

Swift provides five different *access levels* for entities within your code. These access levels are relative to the source file in which an entity is defined, and also relative to the module that source file belongs to.

- *Open access* and *public access* enable entities to be used within any source file from their defining module, and also in a source file from another module that imports the defining module. You typically use open or public access when specifying the public interface to a framework. The difference between open and public access is described below.
- *Internal access* enables entities to be used within any source file from their defining module, but not in any source file outside of that module. You typically use internal access when defining an app's or a framework's internal structure.
- *File-private access* restricts the use of an entity to its own defining source file. Use file-private access to hide the implementation details of a specific piece of functionality when those details are used within an entire file.
- *Private access* restricts the use of an entity to the enclosing declaration, and to extensions of that declaration that are in the same file. Use private access to hide the implementation details of a specific piece of functionality when those details are used only within a single declaration.

Open access is the highest (least restrictive) access level and private access is the lowest (most restrictive) access level.

Open access applies only to classes and class members, and it differs from public access by allowing code outside the module to subclass and override, as discussed below in <u>Subclassing</u>. Marking a class as open explicitly indicates that you've considered the impact of code from other modules using that class as a superclass, and that you've designed your class's code accordingly.

Guiding Principle of Access Levels

Access levels in Swift follow an overall guiding principle: *No entity can be defined in terms of another entity that has a lower (more restrictive) access level.*

For example:

- A public variable can't be defined as having an internal, file-private, or private type, because the type might not be available everywhere that the public variable is used.
- A function can't have a higher access level than its parameter types and return type, because the function could be used in situations where its constituent types are unavailable to the surrounding code.

The specific implications of this guiding principle for different aspects of the language are covered in detail below.

Default Access Levels

All entities in your code (with a few specific exceptions, as described later in this chapter) have a default access level of internal if you don't specify an explicit access level yourself. As a result, in many cases you don't need to specify an explicit access level in your code.

Access Levels for Single-Target Apps

When you write a simple single-target app, the code in your app is typically self-contained within the app and doesn't need to be made available outside of the

app's module. The default access level of internal already matches this requirement. Therefore, you don't need to specify a custom access level. You may, however, want to mark some parts of your code as file private or private in order to hide their implementation details from other code within the app's module.

Access Levels for Frameworks

When you develop a framework, mark the public-facing interface to that framework as open or public so that it can be viewed and accessed by other modules, such as an app that imports the framework. This public-facing interface is the application programming interface (or API) for the framework.

NOTE

Any internal implementation details of your framework can still use the default access level of internal, or can be marked as private or file private if you want to hide them from other parts of the framework's internal code. You need to mark an entity as open or public only if you want it to become part of your framework's API.

Access Levels for Unit Test Targets

When you write an app with a unit test target, the code in your app needs to be made available to that module in order to be tested. By default, only entities marked as open or public are accessible to other modules. However, a unit test target can access any internal entity, if you mark the import declaration for a product module with the <code>@testable</code> attribute and compile that product module with testing enabled.

Access Control Syntax

Define the access level for an entity by placing one of the open, public, internal, fileprivate, or private modifiers at the beginning of the entity's declaration.

```
1
   public class SomePublicClass {}
2
   internal class SomeInternalClass {}
3
   fileprivate class SomeFilePrivateClass {}
   private class SomePrivateClass {}
4
5
   public var somePublicVariable = 0
6
   internal let someInternalConstant = 0
7
   fileprivate func someFilePrivateFunction() {}
8
   private func somePrivateFunction() {}
9
```

Unless otherwise specified, the default access level is internal, as described in Default Access Levels. This means that SomeInternalClass and someInternalConstant can be written without an explicit access-level modifier, and will still have an access level of internal:

```
class SomeInternalClass {} // implicitly internal
let someInternalConstant = 0 // implicitly internal
```

Custom Types

If you want to specify an explicit access level for a custom type, do so at the point that you define the type. The new type can then be used wherever its access level permits. For example, if you define a file-private class, that class can only be used as the type of a property, or as a function parameter or return type, in the source file in which the file-private class is defined.

The access control level of a type also affects the default access level of that type's *members* (its properties, methods, initializers, and subscripts). If you define a type's access level as private or file private, the default access level of its members will also be private or file private. If you define a type's access level as internal or public (or use the default access level of internal without specifying an access level explicitly), the default access level of the type's

members will be internal.

IMPORTANT

A public type defaults to having internal members, not public members. If you want a type member to be public, you must explicitly mark it as such. This requirement ensures that the public-facing API for a type is something you opt in to publishing, and avoids presenting the internal workings of a type as public API by mistake.

```
1
    public class SomePublicClass {
                                                    // explicitly
          public class
        public var somePublicProperty = 0
                                                     // explicitly
2
          public class member
        var someInternalProperty = 0
                                                     // implicitly
3
          internal class member
        fileprivate func someFilePrivateMethod() {} // explicitly
4
          file-private class member
        private func somePrivateMethod() {}
                                                     // explicitly
5
          private class member
6
    }
7
    class SomeInternalClass {
                                                    // implicitly
8
          internal class
9
        var someInternalProperty = 0
                                                     // implicitly
          internal class member
10
        fileprivate func someFilePrivateMethod() {} // explicitly
          file-private class member
                                                     // explicitly
11
        private func somePrivateMethod() {}
          private class member
12
    }
13
14
    fileprivate class SomeFilePrivateClass { // explicitly
```

```
file-private class
                                                        // implicitly
15
         func someFilePrivateMethod() {}
           file-private class member
16
         private func somePrivateMethod() {}
                                                        // explicitly
           private class member
    }
17
18
    private class SomePrivateClass {
                                                       // explicitly
19
           private class
         func somePrivateMethod() {}
                                                        // implicitly
20
           private class member
21
    }
```

Tuple Types

The access level for a tuple type is the most restrictive access level of all types used in that tuple. For example, if you compose a tuple from two different types, one with internal access and one with private access, the access level for that compound tuple type will be private.

NOTE

Tuple types don't have a standalone definition in the way that classes, structures, enumerations, and functions do. A tuple type's access level is determined automatically from the types that make up the tuple type, and can't be specified explicitly.

Function Types

The access level for a function type is calculated as the most restrictive access level of the function's parameter types and return type. You must specify the access level explicitly as part of the function's definition if the function's calculated access level doesn't match the contextual default.

The example below defines a global function called <code>someFunction()</code>, without providing a specific access-level modifier for the function itself. You might expect this function to have the default access level of "internal", but this isn't the case. In fact, <code>someFunction()</code> won't compile as written below:

```
func someFunction() -> (SomeInternalClass, SomePrivateClass) {
    // function implementation goes here
}
```

The function's return type is a tuple type composed from two of the custom classes defined above in <u>Custom Types</u>. One of these classes is defined as internal, and the other is defined as private. Therefore, the overall access level of the compound tuple type is private (the minimum access level of the tuple's constituent types).

Because the function's return type is private, you must mark the function's overall access level with the private modifier for the function declaration to be valid:

It's not valid to mark the definition of someFunction() with the public or internal modifiers, or to use the default setting of internal, because public or internal users of the function might not have appropriate access to the private class used in the function's return type.

Enumeration Types

The individual cases of an enumeration automatically receive the same access level as the enumeration they belong to. You can't specify a different access level for individual enumeration cases.

In the example below, the CompassPoint enumeration has an explicit access level of public. The enumeration cases north, south, east, and west therefore also have an access level of public:

```
public enum CompassPoint {
    case north
    case south
    case east
    case west
}
```

Raw Values and Associated Values

The types used for any raw values or associated values in an enumeration definition must have an access level at least as high as the enumeration's access level. For example, you can't use a private type as the raw-value type of an enumeration with an internal access level.

Nested Types

The access level of a nested type is the same as its containing type, unless the containing type is public. Nested types defined within a public type have an automatic access level of internal. If you want a nested type within a public type to be publicly available, you must explicitly declare the nested type as public.

Subclassing

You can subclass any class that can be accessed in the current access context and that's defined in the same module as the subclass. You can also subclass any open class that's defined in a different module. A subclass can't have a higher access level than its superclass—for example, you can't write a public subclass of an internal superclass.

In addition, for classes that are defined in the same module, you can override any class member (method, property, initializer, or subscript) that's visible in a certain access context. For classes that are defined in another module, you can override any open class member.

An override can make an inherited class member more accessible than its superclass version. In the example below, class A is a public class with a file-private method called <code>someMethod()</code>. Class B is a subclass of A, with a reduced access level of "internal". Nonetheless, class B provides an override of <code>someMethod()</code> with an access level of "internal", which is *higher* than the original implementation of <code>someMethod()</code>:

```
public class A {
   fileprivate func someMethod() {}
}

internal class B: A {
   override internal func someMethod() {}
}
```

It's even valid for a subclass member to call a superclass member that has lower access permissions than the subclass member, as long as the call to the superclass's member takes place within an allowed access level context (that is, within the same source file as the superclass for a file-private member call, or within the same module as the superclass for an internal member call):

```
public class A {
   fileprivate func someMethod() {}
}

internal class B: A {
   override internal func someMethod() {
      super.someMethod()
```

```
8 }
9 }
```

Because superclass A and subclass B are defined in the same source file, it's valid for the B implementation of someMethod() to call super.someMethod().

Constants, Variables, Properties, and Subscripts

A constant, variable, or property can't be more public than its type. It's not valid to write a public property with a private type, for example. Similarly, a subscript can't be more public than either its index type or return type.

If a constant, variable, property, or subscript makes use of a private type, the constant, variable, property, or subscript must also be marked as private:

```
private var privateInstance = SomePrivateClass()
```

Getters and Setters

Getters and setters for constants, variables, properties, and subscripts automatically receive the same access level as the constant, variable, property, or subscript they belong to.

You can give a setter a *lower* access level than its corresponding getter, to restrict the read-write scope of that variable, property, or subscript. You assign a lower access level by writing fileprivate(set), private(set), or internal(set) before the var or subscript introducer.

NOTE

This rule applies to stored properties as well as computed properties. Even though you don't write an explicit getter and setter for a stored property, Swift still synthesizes an implicit getter and setter for you to provide access to the stored property's backing storage. Use fileprivate(set), private(set), and internal(set) to change the access level of this synthesized setter in exactly the same way as for an explicit setter in a computed property.

The example below defines a structure called TrackedString, which keeps track of the number of times a string property is modified:

```
struct TrackedString {
   private(set) var numberOfEdits = 0

var value: String = "" {
   didSet {
      numberOfEdits += 1
   }
}
```

The TrackedString structure defines a stored string property called value, with an initial value of "" (an empty string). The structure also defines a stored integer property called numberOfEdits, which is used to track the number of times that value is modified. This modification tracking is implemented with a didSet property observer on the value property, which increments numberOfEdits every time the value property is set to a new value.

The TrackedString structure and the value property don't provide an explicit access-level modifier, and so they both receive the default access level of internal. However, the access level for the numberOfEdits property is marked with a private(set) modifier to indicate that the property's getter still has the default access level of internal, but the property is settable only from within code that's part of the TrackedString structure. This enables TrackedString to modify the numberOfEdits property internally, but to present the property as a read-only property when it's used outside the structure's definition.

If you create a TrackedString instance and modify its string value a few times, you can see the numberOfEdits property value update to match the number of modifications:

```
var stringToEdit = TrackedString()
stringToEdit.value = "This string will be tracked."
stringToEdit.value += " This edit will increment
```

```
numberOfEdits."

4 stringToEdit.value += " So will this one."

5 print("The number of edits is \(stringToEdit.numberOfEdits)")

6 // Prints "The number of edits is 3"
```

Although you can query the current value of the numberOfEdits property from within another source file, you can't *modify* the property from another source file. This restriction protects the implementation details of the TrackedString edit-tracking functionality, while still providing convenient access to an aspect of that functionality.

Note that you can assign an explicit access level for both a getter and a setter if required. The example below shows a version of the TrackedString structure in which the structure is defined with an explicit access level of public. The structure's members (including the numberOfEdits property) therefore have an internal access level by default. You can make the structure's numberOfEdits property getter public, and its property setter private, by combining the public and private(set) access-level modifiers:

```
public struct TrackedString {
1
        public private(set) var numberOfEdits = 0
2
        public var value: String = "" {
3
            didSet {
4
5
                numberOfEdits += 1
6
            }
7
        }
8
        public init() {}
9
   }
```

Initializers

Custom initializers can be assigned an access level less than or equal to the type

that they initialize. The only exception is for required initializers (as defined in Required Initializers). A required initializer must have the same access level as the class it belongs to.

As with function and method parameters, the types of an initializer's parameters can't be more private than the initializer's own access level.

Default Initializers

As described in <u>Default Initializers</u>, Swift automatically provides a *default initializer* without any arguments for any structure or base class that provides default values for all of its properties and doesn't provide at least one initializer itself.

A default initializer has the same access level as the type it initializes, unless that type is defined as public. For a type that is defined as public, the default initializer is considered internal. If you want a public type to be initializable with a no-argument initializer when used in another module, you must explicitly provide a public no-argument initializer yourself as part of the type's definition.

Default Memberwise Initializers for Structure Types

The default memberwise initializer for a structure type is considered private if any of the structure's stored properties are private. Likewise, if any of the structure's stored properties are file private, the initializer is file private. Otherwise, the initializer has an access level of internal.

As with the default initializer above, if you want a public structure type to be initializable with a memberwise initializer when used in another module, you must provide a public memberwise initializer yourself as part of the type's definition.

Protocols

If you want to assign an explicit access level to a protocol type, do so at the point that you define the protocol. This enables you to create protocols that can only be adopted within a certain access context.

The access level of each requirement within a protocol definition is automatically set to the same access level as the protocol. You can't set a protocol requirement to a different access level than the protocol it supports. This ensures that all of the protocol's requirements will be visible on any type that adopts the protocol.

NOTE

If you define a public protocol, the protocol's requirements require a public access level for those requirements when they're implemented. This behavior is different from other types, where a public type definition implies an access level of internal for the type's members.

Protocol Inheritance

If you define a new protocol that inherits from an existing protocol, the new protocol can have at most the same access level as the protocol it inherits from. For example, you can't write a public protocol that inherits from an internal protocol.

Protocol Conformance

A type can conform to a protocol with a lower access level than the type itself. For example, you can define a public type that can be used in other modules, but whose conformance to an internal protocol can only be used within the internal protocol's defining module.

The context in which a type conforms to a particular protocol is the minimum of the type's access level and the protocol's access level. For example, if a type is public, but a protocol it conforms to is internal, the type's conformance to that protocol is also internal.

When you write or extend a type to conform to a protocol, you must ensure that

the type's implementation of each protocol requirement has at least the same access level as the type's conformance to that protocol. For example, if a public type conforms to an internal protocol, the type's implementation of each protocol requirement must be at least internal.

NOTE

In Swift, as in Objective-C, protocol conformance is global—it isn't possible for a type to conform to a protocol in two different ways within the same program.

Extensions

You can extend a class, structure, or enumeration in any access context in which the class, structure, or enumeration is available. Any type members added in an extension have the same default access level as type members declared in the original type being extended. If you extend a public or internal type, any new type members you add have a default access level of internal. If you extend a file-private type, any new type members you add have a default access level of file private. If you extend a private type, any new type members you add have a default access level of private.

Alternatively, you can mark an extension with an explicit access-level modifier (for example, private) to set a new default access level for all members defined within the extension. This new default can still be overridden within the extension for individual type members.

You can't provide an explicit access-level modifier for an extension if you're using that extension to add protocol conformance. Instead, the protocol's own access level is used to provide the default access level for each protocol requirement implementation within the extension.

Private Members in Extensions

Extensions that are in the same file as the class, structure, or enumeration that they extend behave as if the code in the extension had been written as part of the

original type's declaration. As a result, you can:

- Declare a private member in the original declaration, and access that member from extensions in the same file.
- Declare a private member in one extension, and access that member from another extension in the same file.
- Declare a private member in an extension, and access that member from the original declaration in the same file.

This behavior means you can use extensions in the same way to organize your code, whether or not your types have private entities. For example, given the following simple protocol:

```
protocol SomeProtocol {
   func doSomething()
}
```

You can use an extension to add protocol conformance, like this:

```
1
    struct SomeStruct {
        private var privateVariable = 12
2
3
   }
4
   extension SomeStruct: SomeProtocol {
5
        func doSomething() {
6
            print(privateVariable)
8
        }
9
    }
```

Generics

The access level for a generic type or generic function is the minimum of the access level of the generic type or function itself and the access level of any type constraints on its type parameters.

Type Aliases

Any type aliases you define are treated as distinct types for the purposes of access control. A type alias can have an access level less than or equal to the access level of the type it aliases. For example, a private type alias can alias a private, file-private, internal, public, or open type, but a public type alias can't alias an internal, file-private, or private type.

NOTE

This rule also applies to type aliases for associated types used to satisfy protocol conformances.

Advanced Operators

In addition to the operators described in <u>Basic Operators</u>, Swift provides several advanced operators that perform more complex value manipulation. These include all of the bitwise and bit shifting operators you will be familiar with from C and Objective-C.

Unlike arithmetic operators in C, arithmetic operators in Swift do not overflow by default. Overflow behavior is trapped and reported as an error. To opt in to overflow behavior, use Swift's second set of arithmetic operators that overflow by default, such as the overflow addition operator (&+). All of these overflow operators begin with an ampersand (&).

When you define your own structures, classes, and enumerations, it can be useful to provide your own implementations of the standard Swift operators for these custom types. Swift makes it easy to provide tailored implementations of these operators and to determine exactly what their behavior should be for each type you create.

You're not limited to the predefined operators. Swift gives you the freedom to define your own custom infix, prefix, postfix, and assignment operators, with custom precedence and associativity values. These operators can be used and adopted in your code like any of the predefined operators, and you can even extend existing types to support the custom operators you define.

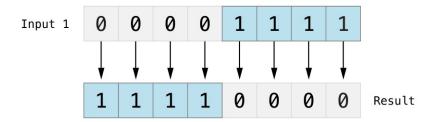
Bitwise Operators

Bitwise operators enable you to manipulate the individual raw data bits within a data structure. They are often used in low-level programming, such as graphics programming and device driver creation. Bitwise operators can also be useful when you work with raw data from external sources, such as encoding and decoding data for communication over a custom protocol.

Swift supports all of the bitwise operators found in C, as described below.

Bitwise NOT Operator

The *bitwise NOT operator* (~) inverts all bits in a number:



The bitwise NOT operator is a prefix operator, and appears immediately before the value it operates on, without any white space:

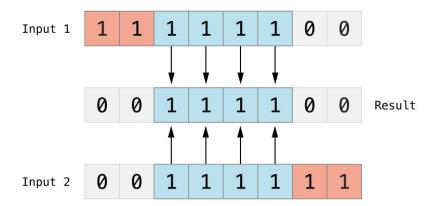
```
1 let initialBits: UInt8 = 0b00001111
2 let invertedBits = ~initialBits // equals 11110000
```

UInt8 integers have eight bits and can store any value between 0 and 255. This example initializes a UInt8 integer with the binary value 00001111, which has its first four bits set to 0, and its second four bits set to 1. This is equivalent to a decimal value of 15.

The bitwise NOT operator is then used to create a new constant called invertedBits, which is equal to initialBits, but with all of the bits inverted. Zeros become ones, and ones become zeros. The value of invertedBits is 11110000, which is equal to an unsigned decimal value of 240.

Bitwise AND Operator

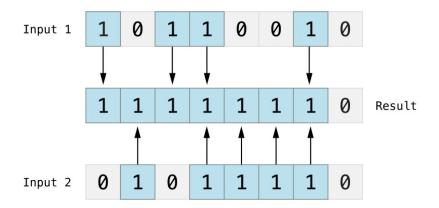
The *bitwise AND operator* (&) combines the bits of two numbers. It returns a new number whose bits are set to 1 only if the bits were equal to 1 in *both* input numbers:



In the example below, the values of firstSixBits and lastSixBits both have four middle bits equal to 1. The bitwise AND operator combines them to make the number <code>00111100</code>, which is equal to an unsigned decimal value of <code>60</code>:

Bitwise OR Operator

The *bitwise OR operator* (\mid) compares the bits of two numbers. The operator returns a new number whose bits are set to 1 if the bits are equal to 1 in *either* input number:



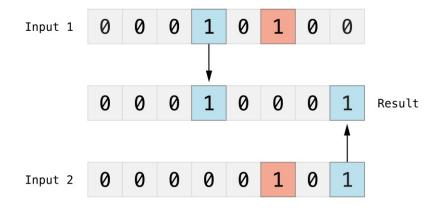
In the example below, the values of someBits and moreBits have different bits set

to 1. The bitwise OR operator combines them to make the number 111111110, which equals an unsigned decimal of 254:

```
1 let someBits: UInt8 = 0b10110010
2 let moreBits: UInt8 = 0b01011110
3 let combinedbits = someBits | moreBits // equals 11111110
```

Bitwise XOR Operator

The *bitwise XOR operator*, or "exclusive OR operator" (^), compares the bits of two numbers. The operator returns a new number whose bits are set to 1 where the input bits are different and are set to 0 where the input bits are the same:



In the example below, the values of firstBits and otherBits each have a bit set to 1 in a location that the other does not. The bitwise XOR operator sets both of these bits to 1 in its output value. All of the other bits in firstBits and otherBits match and are set to 0 in the output value:

```
1 let firstBits: UInt8 = 0b00010100
2 let otherBits: UInt8 = 0b00000101
3 let outputBits = firstBits ^ otherBits // equals 00010001
```

Bitwise Left and Right Shift Operators

The bitwise left shift operator (<<) and bitwise right shift operator (>>) move all bits in a number to the left or the right by a certain number of places, according to the rules defined below.

Bitwise left and right shifts have the effect of multiplying or dividing an integer by a factor of two. Shifting an integer's bits to the left by one position doubles its value, whereas shifting it to the right by one position halves its value.

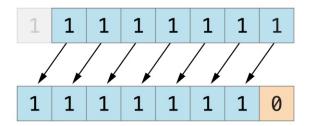
Shifting Behavior for Unsigned Integers

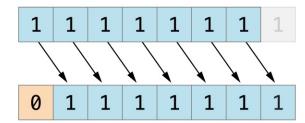
The bit-shifting behavior for unsigned integers is as follows:

- 1. Existing bits are moved to the left or right by the requested number of places.
- 2. Any bits that are moved beyond the bounds of the integer's storage are discarded.
- 3. Zeros are inserted in the spaces left behind after the original bits are moved to the left or right.

This approach is known as a logical shift.

The illustration below shows the results of 11111111 << 1 (which is 11111111 shifted to the left by 1 place), and 11111111 >> 1 (which is 111111111 shifted to the right by 1 place). Blue numbers are shifted, gray numbers are discarded, and orange zeros are inserted:





Here's how bit shifting looks in Swift code:

```
let shiftBits: UInt8 = 4 // 00000100 in binary
```

You can use bit shifting to encode and decode values within other data types:

```
1 let pink: UInt32 = 0xCC6699
2 let redComponent = (pink & 0xFF0000) >> 16  // redComponent
        is 0xCC, or 204
3 let greenComponent = (pink & 0x00FF00) >> 8  // greenComponent
        is 0x66, or 102
4 let blueComponent = pink & 0x0000FF  // blueComponent
        is 0x99, or 153
```

This example uses a UInt32 constant called pink to store a Cascading Style Sheets color value for the color pink. The CSS color value #CC6699 is written as 0xCC6699 in Swift's hexadecimal number representation. This color is then decomposed into its red (CC), green (66), and blue (99) components by the bitwise AND operator (&) and the bitwise right shift operator (>>).

The red component is obtained by performing a bitwise AND between the numbers 0xcc6699 and 0xff0000. The zeros in 0xff0000 effectively "mask" the second and third bytes of 0xcc6699, causing the 6699 to be ignored and leaving 0xcc0000 as the result.

This number is then shifted 16 places to the right (>> 16). Each pair of characters in a hexadecimal number uses 8 bits, so a move 16 places to the right will convert 0xcc0000 into 0x0000cc. This is the same as 0xcc, which has a decimal value of 204.

Similarly, the green component is obtained by performing a bitwise AND between the numbers 0xcc6699 and 0x00FF00, which gives an output value of 0x006600. This output value is then shifted eight places to the right, giving a

value of 0x66, which has a decimal value of 102.

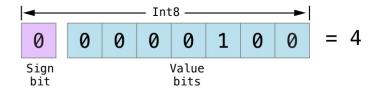
Finally, the blue component is obtained by performing a bitwise AND between the numbers 0xcc6699 and 0x00000FF, which gives an output value of 0x000099. There's no need to shift this to the right, as 0x000099 already equals 0x99, which has a decimal value of 153.

Shifting Behavior for Signed Integers

The shifting behavior is more complex for signed integers than for unsigned integers, because of the way signed integers are represented in binary. (The examples below are based on 8-bit signed integers for simplicity, but the same principles apply for signed integers of any size.)

Signed integers use their first bit (known as the *sign bit*) to indicate whether the integer is positive or negative. A sign bit of 0 means positive, and a sign bit of 1 means negative.

The remaining bits (known as the *value bits*) store the actual value. Positive numbers are stored in exactly the same way as for unsigned integers, counting upwards from 0. Here's how the bits inside an Int8 look for the number 4:



The sign bit is 0 (meaning "positive"), and the seven value bits are just the number 4, written in binary notation.

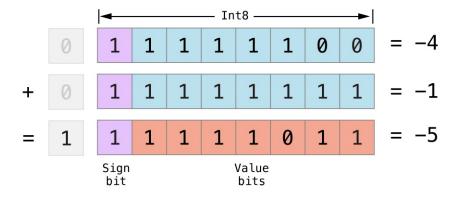
Negative numbers, however, are stored differently. They are stored by subtracting their absolute value from 2 to the power of n, where n is the number of value bits. An eight-bit number has seven value bits, so this means 2 to the power of 7, or 128.

Here's how the bits inside an Int8 look for the number -4:

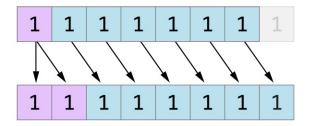
This time, the sign bit is 1 (meaning "negative"), and the seven value bits have a binary value of 124 (which is 128 - 4):

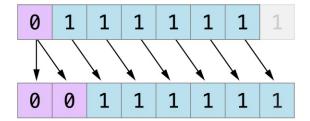
This encoding for negative numbers is known as a *two's complement* representation. It may seem an unusual way to represent negative numbers, but it has several advantages.

First, you can add -1 to -4, simply by performing a standard binary addition of all eight bits (including the sign bit), and discarding anything that doesn't fit in the eight bits once you're done:



Second, the two's complement representation also lets you shift the bits of negative numbers to the left and right like positive numbers, and still end up doubling them for every shift you make to the left, or halving them for every shift you make to the right. To achieve this, an extra rule is used when signed integers are shifted to the right: When you shift signed integers to the right, apply the same rules as for unsigned integers, but fill any empty bits on the left with the *sign bit*, rather than with a zero.





This action ensures that signed integers have the same sign after they are shifted to the right, and is known as an *arithmetic shift*.

Because of the special way that positive and negative numbers are stored, shifting either of them to the right moves them closer to zero. Keeping the sign bit the same during this shift means that negative integers remain negative as their value moves closer to zero.

Overflow Operators

If you try to insert a number into an integer constant or variable that cannot hold that value, by default Swift reports an error rather than allowing an invalid value to be created. This behavior gives extra safety when you work with numbers that are too large or too small.

For example, the Int16 integer type can hold any signed integer between -32768 and 32767. Trying to set an Int16 constant or variable to a number outside of this range causes an error:

```
var potentialOverflow = Int16.max
// potentialOverflow equals 32767, which is the maximum value
          an Int16 can hold
potentialOverflow += 1
// this causes an error
```

Providing error handling when values get too large or too small gives you much more flexibility when coding for boundary value conditions.

However, when you specifically want an overflow condition to truncate the number of available bits, you can opt in to this behavior rather than triggering an error. Swift provides three arithmetic *overflow operators* that opt in to the overflow behavior for integer calculations. These operators all begin with an ampersand (&):

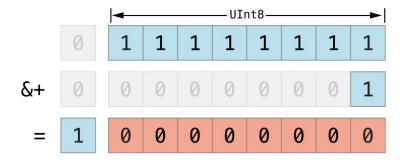
- Overflow addition (&+)
- Overflow subtraction (&-)
- Overflow multiplication (&*)

Value Overflow

Numbers can overflow in both the positive and negative direction.

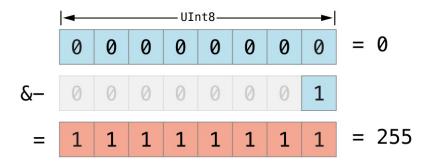
Here's an example of what happens when an unsigned integer is allowed to overflow in the positive direction, using the overflow addition operator (&+):

The variable unsignedOverflow is initialized with the maximum value a UInt8 can hold (255, or 11111111 in binary). It is then incremented by 1 using the overflow addition operator (&+). This pushes its binary representation just over the size that a UInt8 can hold, causing it to overflow beyond its bounds, as shown in the diagram below. The value that remains within the bounds of the UInt8 after the overflow addition is 000000000, or zero.



Something similar happens when an unsigned integer is allowed to overflow in the negative direction. Here's an example using the overflow subtraction operator (&-):

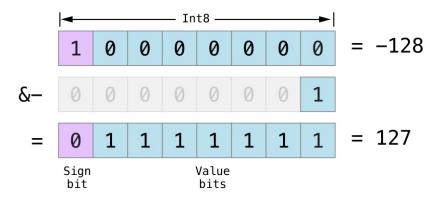
The minimum value that a UInt8 can hold is zero, or 00000000 in binary. If you subtract 1 from 00000000 using the overflow subtraction operator (&-), the number will overflow and wrap around to 11111111, or 255 in decimal.



Overflow also occurs for signed integers. All addition and subtraction for signed integers is performed in bitwise fashion, with the sign bit included as part of the numbers being added or subtracted, as described in Bitwise Left and Right Shift Operators.

```
var signedOverflow = Int8.min
```

The minimum value that an Int8 can hold is -128, or 10000000 in binary. Subtracting 1 from this binary number with the overflow operator gives a binary value of 01111111, which toggles the sign bit and gives positive 127, the maximum positive value that an Int8 can hold.



For both signed and unsigned integers, overflow in the positive direction wraps around from the maximum valid integer value back to the minimum, and overflow in the negative direction wraps around from the minimum value to the maximum.

Precedence and Associativity

Operator *precedence* gives some operators higher priority than others; these operators are applied first.

Operator *associativity* defines how operators of the same precedence are grouped together—either grouped from the left, or grouped from the right. Think of it as meaning "they associate with the expression to their left," or "they associate with the expression to their right."

It is important to consider each operator's precedence and associativity when working out the order in which a compound expression will be calculated. For example, operator precedence explains why the following expression equals 17.

```
1 2 + 3 % 4 * 5
2 // this equals 17
```

If you read strictly from left to right, you might expect the expression to be calculated as follows:

- 2 plus 3 equals 5
- 5 remainder 4 equals 1
- 1 times 5 equals 5

However, the actual answer is 17, not 5. Higher-precedence operators are evaluated before lower-precedence ones. In Swift, as in C, the remainder operator (%) and the multiplication operator (*) have a higher precedence than the addition operator (+). As a result, they are both evaluated before the addition is considered.

However, remainder and multiplication have the *same* precedence as each other. To work out the exact evaluation order to use, you also need to consider their associativity. Remainder and multiplication both associate with the expression to their left. Think of this as adding implicit parentheses around these parts of the expression, starting from their left:

```
2 + ((3 % 4) * 5)
```

(3 % 4) is 3, so this is equivalent to:

```
2 + (3 * 5)
```

(3 * 5) is 15, so this is equivalent to:

```
2 + 15
```

This calculation yields the final answer of 17.

For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see Operator Declarations.

```
NOTE
```

Swift's operator precedences and associativity rules are simpler and more predictable than those found in C and Objective-C. However, this means that they are not exactly the same as in C-based languages. Be careful to ensure that operator interactions still behave in the way you intend when porting existing code to Swift.

Operator Methods

Classes and structures can provide their own implementations of existing operators. This is known as *overloading* the existing operators.

The example below shows how to implement the arithmetic addition operator (+) for a custom structure. The arithmetic addition operator is a *binary operator* because it operates on two targets and is said to be *infix* because it appears in between those two targets.

The example defines a Vector2D structure for a two-dimensional position vector (x, y), followed by a definition of an *operator method* to add together instances of the Vector2D structure:

```
struct Vector2D {
   var x = 0.0, y = 0.0
}

extension Vector2D {
   static func + (left: Vector2D, right: Vector2D) -> Vector2D
   {
```

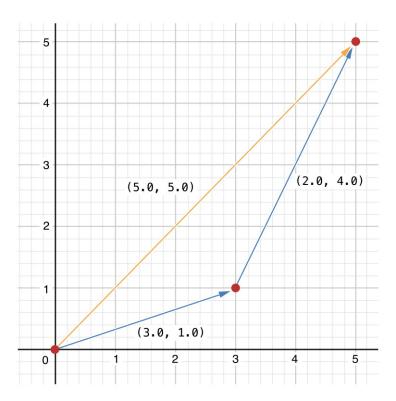
The operator method is defined as a type method on Vector2D, with a method name that matches the operator to be overloaded (+). Because addition isn't part of the essential behavior for a vector, the type method is defined in an extension of Vector2D rather than in the main structure declaration of Vector2D. Because the arithmetic addition operator is a binary operator, this operator method takes two input parameters of type Vector2D and returns a single output value, also of type Vector2D.

In this implementation, the input parameters are named left and right to represent the Vector2D instances that will be on the left side and right side of the + operator. The method returns a new Vector2D instance, whose x and y properties are initialized with the sum of the x and y properties from the two Vector2D instances that are added together.

The type method can be used as an infix operator between existing Vector2D instances:

```
1 let vector = Vector2D(x: 3.0, y: 1.0)
2 let anotherVector = Vector2D(x: 2.0, y: 4.0)
3 let combinedVector = vector + anotherVector
4 // combinedVector is a Vector2D instance with values of (5.0, 5.0)
```

This example adds together the vectors (3.0, 1.0) and (2.0, 4.0) to make the vector (5.0, 5.0), as illustrated below.



Prefix and Postfix Operators

The example shown above demonstrates a custom implementation of a binary infix operator. Classes and structures can also provide implementations of the standard *unary operators*. Unary operators operate on a single target. They are *prefix* if they precede their target (such as -a) and *postfix* operators if they follow their target (such as b!).

You implement a prefix or postfix unary operator by writing the prefix or postfix modifier before the func keyword when declaring the operator method:

```
1  extension Vector2D {
2    static prefix func - (vector: Vector2D) -> Vector2D {
3       return Vector2D(x: -vector.x, y: -vector.y)
4    }
5 }
```

The example above implements the unary minus operator (-a) for Vector2D

instances. The unary minus operator is a prefix operator, and so this method has to be qualified with the prefix modifier.

For simple numeric values, the unary minus operator converts positive numbers into their negative equivalent and vice versa. The corresponding implementation for Vector2D instances performs this operation on both the x and y properties:

```
1 let positive = Vector2D(x: 3.0, y: 4.0)
2 let negative = -positive
3 // negative is a Vector2D instance with values of (-3.0, -4.0)
4 let alsoPositive = -negative
5 // alsoPositive is a Vector2D instance with values of (3.0, 4.0)
```

Compound Assignment Operators

Compound assignment operators combine assignment (=) with another operation. For example, the addition assignment operator (+=) combines addition and assignment into a single operation. You mark a compound assignment operator's left input parameter type as inout, because the parameter's value will be modified directly from within the operator method.

The example below implements an addition assignment operator method for Vector 2D instances:

```
1  extension Vector2D {
2    static func += (left: inout Vector2D, right: Vector2D) {
3        left = left + right
4    }
5  }
```

Because an addition operator was defined earlier, you don't need to reimplement the addition process here. Instead, the addition assignment operator method takes advantage of the existing addition operator method, and uses it to set the left value to be the left value plus the right value:

```
var original = Vector2D(x: 1.0, y: 2.0)
let vectorToAdd = Vector2D(x: 3.0, y: 4.0)
original += vectorToAdd
// original now has values of (4.0, 6.0)
```

NOTE

It isn't possible to overload the default assignment operator (=). Only the compound assignment operators can be overloaded. Similarly, the ternary conditional operator (a ? b : c) can't be overloaded.

Equivalence Operators

By default, custom classes and structures don't have an implementation of the *equivalence operators*, known as the *equal to* operator (==) and *not equal to* operator (!=). You usually implement the == operator, and use the standard library's default implementation of the != operator that negates the result of the == operator. There are two ways to implement the == operator: You can implement it yourself, or for many types, you can ask Swift to synthesize an implementation for you. In both cases, you add conformance to the standard library's Equatable protocol.

You provide an implementation of the == operator in the same way as you implement other infix operators:

```
1  extension Vector2D: Equatable {
2    static func == (left: Vector2D, right: Vector2D) -> Bool {
3       return (left.x == right.x) && (left.y == right.y)
4    }
5  }
```

The example above implements an == operator to check whether two Vector2D instances have equivalent values. In the context of Vector2D, it makes sense to

consider "equal" as meaning "both instances have the same x values and y values", and so this is the logic used by the operator implementation.

You can now use this operator to check whether two Vector2D instances are equivalent:

```
1 let twoThree = Vector2D(x: 2.0, y: 3.0)
2 let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
3 if twoThree == anotherTwoThree {
4    print("These two vectors are equivalent.")
5 }
6 // Prints "These two vectors are equivalent."
```

In many simple cases, you can ask Swift to provide synthesized implementations of the equivalence operators for you. Swift provides synthesized implementations for the following kinds of custom types:

- Structures that have only stored properties that conform to the Equatable protocol
- Enumerations that have only associated types that conform to the Equatable protocol
- Enumerations that have no associated types

To receive a synthesized implementation of ==, declare Equatable conformance in the file that contains the original declaration, without implementing an == operator yourself.

The example below defines a Vector3D structure for a three-dimensional position vector (x, y, z), similar to the Vector2D structure. Because the x, y, and z properties are all of an Equatable type, Vector3D receives synthesized implementations of the equivalence operators.

```
struct Vector3D: Equatable {
   var x = 0.0, y = 0.0, z = 0.0
```

```
3  }
4
5  let twoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
6  let anotherTwoThreeFour = Vector3D(x: 2.0, y: 3.0, z: 4.0)
7  if twoThreeFour == anotherTwoThreeFour {
8    print("These two vectors are also equivalent.")
9  }
10  // Prints "These two vectors are also equivalent."
```

Custom Operators

You can declare and implement your own *custom operators* in addition to the standard operators provided by Swift. For a list of characters that can be used to define custom operators, see Operators.

New operators are declared at a global level using the operator keyword, and are marked with the prefix, infix or postfix modifiers:

```
prefix operator +++
```

The example above defines a new prefix operator called +++. This operator does not have an existing meaning in Swift, and so it is given its own custom meaning below in the specific context of working with Vector2D instances. For the purposes of this example, +++ is treated as a new "prefix doubling" operator. It doubles the x and y values of a Vector2D instance, by adding the vector to itself with the addition assignment operator defined earlier. To implement the +++ operator, you add a type method called +++ to Vector2D as follows:

```
1 extension Vector2D {
2   static prefix func +++ (vector: inout Vector2D) -> Vector2D
   {
3     vector += vector
```

```
4    return vector
5   }
6 }
7 
8  var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
9  let afterDoubling = +++toBeDoubled
10  // toBeDoubled now has values of (2.0, 8.0)
11  // afterDoubling also has values of (2.0, 8.0)
```

Precedence for Custom Infix Operators

Custom infix operators each belong to a precedence group. A precedence group specifies an operator's precedence relative to other infix operators, as well as the operator's associativity. See Precedence and Associativity for an explanation of how these characteristics affect an infix operator's interaction with other infix operators.

A custom infix operator that is not explicitly placed into a precedence group is given a default precedence group with a precedence immediately higher than the precedence of the ternary conditional operator.

The following example defines a new custom infix operator called +-, which belongs to the precedence group AdditionPrecedence:

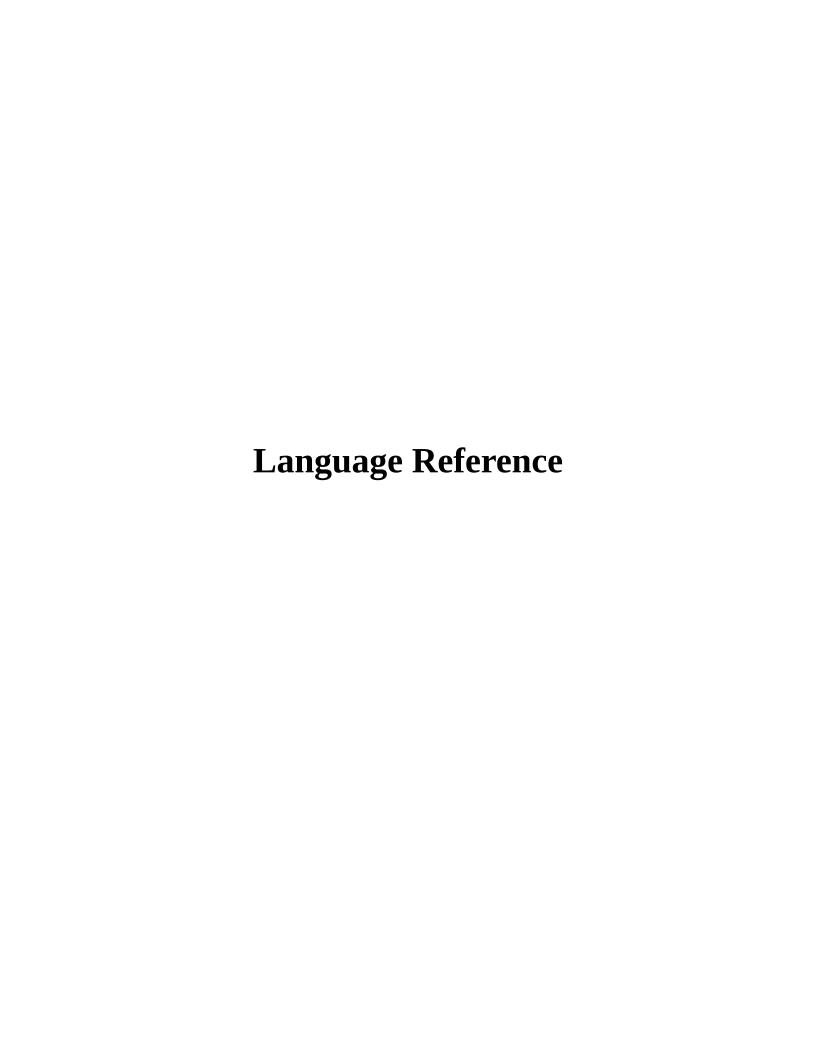
```
infix operator +-: AdditionPrecedence
extension Vector2D {
    static func +- (left: Vector2D, right: Vector2D) ->
        Vector2D {
        return Vector2D(x: left.x + right.x, y: left.y -
        right.y)
    }
}
```

```
7 let firstVector = Vector2D(x: 1.0, y: 2.0)
8 let secondVector = Vector2D(x: 3.0, y: 4.0)
9 let plusMinusVector = firstVector +- secondVector
10 // plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

This operator adds together the × values of two vectors, and subtracts the y value of the second vector from the first. Because it is in essence an "additive" operator, it has been given the same precedence group as additive infix operators such as + and -. For information about the operators provided by the Swift standard library, including a complete list of the operator precedence groups and associativity settings, see Operator Declarations. For more information about precedence groups and to see the syntax for defining your own operators and precedence groups, see Operator Declaration.

NOTE

You do not specify a precedence when defining a prefix or postfix operator. However, if you apply both a prefix and a postfix operator to the same operand, the postfix operator is applied first.



About the Language Reference

This part of the book describes the formal grammar of the Swift programming language. The grammar described here is intended to help you understand the language in more detail, rather than to allow you to directly implement a parser or compiler.

The Swift language is relatively small, because many common types, functions, and operators that appear virtually everywhere in Swift code are actually defined in the Swift standard library. Although these types, functions, and operators are not part of the Swift language itself, they are used extensively in the discussions and code examples in this part of the book.

How to Read the Grammar

The notation used to describe the formal grammar of the Swift programming language follows a few conventions:

- An arrow (→) is used to mark grammar productions and can be read as "can consist of."
- Syntactic categories are indicated by *italic* text and appear on both sides of a grammar production rule.
- Literal words and punctuation are indicated by boldface constant width text and appear only on the right-hand side of a grammar production rule.
- Alternative grammar productions are separated by vertical bars (|). When alternative productions are too long to read easily, they are broken into multiple grammar production rules on new lines.
- In a few cases, regular font text is used to describe the right-hand side of a grammar production rule.

• Optional syntactic categories and literals are marked by a trailing subscript, *opt*.

As an example, the grammar of a getter-setter block is defined as follows:

```
GRAMMAR OF A GETTER-SETTER BLOCK

getter-setter-block → { getter-clause setter-clause opt } | { setter-clause getter-
clause }
```

This definition indicates that a getter-setter block can consist of a getter clause followed by an optional setter clause, enclosed in braces, *or* a setter clause followed by a getter clause, enclosed in braces. The grammar production above is equivalent to the following two productions, where the alternatives are spelled out explicitly:

```
GRAMMAR OF A GETTER-SETTER BLOCK

getter-setter-block \rightarrow { getter-clause setter-clause opt }

getter-setter-block \rightarrow { setter-clause getter-clause }
```

Lexical Structure

The *lexical structure* of Swift describes what sequence of characters form valid tokens of the language. These valid tokens form the lowest-level building blocks of the language and are used to describe the rest of the language in subsequent chapters. A token consists of an identifier, keyword, punctuation, literal, or operator.

In most cases, tokens are generated from the characters of a Swift source file by considering the longest possible substring from the input text, within the constraints of the grammar that are specified below. This behavior is referred to as *longest match* or *maximal munch*.

Whitespace and Comments

Whitespace has two uses: to separate tokens in the source file and to help determine whether an operator is a prefix or postfix (see Operators), but is otherwise ignored. The following characters are considered whitespace: space (U+0020), line feed (U+000A), carriage return (U+000D), horizontal tab (U+0009), vertical tab (U+000B), form feed (U+000C) and null (U+0000).

Comments are treated as whitespace by the compiler. Single line comments begin with // and continue until a line feed (U+000A) or carriage return (U+000D). Multiline comments begin with /* and end with */. Nesting multiline comments is allowed, but the comment markers must be balanced.

Comments can contain additional formatting and markup, as described in Markup Formatting Reference.

```
GRAMMAR OF WHITESPACE

whitespace → whitespace-item whitespace opt

whitespace-item → line-break

whitespace-item → comment

whitespace-item → multiline-comment

whitespace-item → U+0000, U+0009, U+000B, U+000C, or U+0020
```

```
line-break → U+000D
line-break → U+000D followed by U+000A
comment → // comment-text line-break
multiline-comment → /* multiline-comment-text */
comment-text → comment-text-item comment-text opt
comment-text-item → Any Unicode scalar value except U+000A or U+000D
multiline-comment-text → multiline-comment-text-item multiline-comment-text opt
multiline-comment-text-item → multiline-comment
multiline-comment-text-item → comment-text-item
multiline-comment-text-item → Any Unicode scalar value except /* or */
```

Identifiers

Identifiers begin with an uppercase or lowercase letter A through Z, an underscore (_), a noncombining alphanumeric Unicode character in the Basic Multilingual Plane, or a character outside the Basic Multilingual Plane that isn't in a Private Use Area. After the first character, digits and combining Unicode characters are also allowed.

To use a reserved word as an identifier, put a backtick (`) before and after it. For example, class is not a valid identifier, but `class` is valid. The backticks aren't considered part of the identifier; `x` and x have the same meaning.

Inside a closure with no explicit parameter names, the parameters are implicitly named \$0, \$1, \$2, and so on. These names are valid identifiers within the scope of the closure.

The compiler synthesizes identifiers that begin with a dollar sign (\$) for properties that have a property wrapper projection. Your code can interact with these identifiers, but you can't declare identifiers with that prefix. For more information, see the <u>propertyWrapper</u> section of the <u>Attributes</u> chapter.

```
GRAMMAR OF AN IDENTIFIER

identifier → identifier-head identifier-characters opt

identifier → identifier-head identifier-characters opt

identifier → implicit-parameter-name

identifier → property-wrapper-projection
```

```
identifier-list → identifier | identifier , identifier-list
identifier-head → Upper- or lowercase letter A through Z
identifier-head →
identifier-head → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2-U+00B5, or U+00B7-
   U+00BA
identifier-head → U+00BC-U+00BE, U+00C0-U+00D6, U+00D8-U+00F6, or U+00F8-
   U+00FF
identifier-head → U+0100-U+02FF, U+0370-U+167F, U+1681-U+180D, or U+180F-
   U+1DBF
identifier-head → U+1E00-U+1FFF
identifier-head → U+200B-U+200D, U+202A-U+202E, U+203F-U+2040, U+2054, or
   U+2060-U+206F
identifier-head → U+2070-U+20CF, U+2100-U+218F, U+2460-U+24FF, or U+2776-
   U+2793
identifier-head → U+2C00-U+2DFF or U+2E80-U+2FFF
identifier-head → U+3004-U+3007, U+3021-U+302F, U+3031-U+303F, or U+3040-
   U+D7FF
identifier-head → U+F900-U+FD3D, U+FD40-U+FDCF, U+FDF0-U+FE1F, or U+FE30-
   U+FE44
identifier-head → U+FE47-U+FFFD
identifier-head → U+10000-U+1FFFD, U+20000-U+2FFFD, U+30000-U+3FFFD, or
   U+40000-U+4FFFD
identifier-head → U+50000-U+5FFFD, U+60000-U+6FFFD, U+70000-U+7FFFD, or
   U+80000-U+8FFFD
identifier-head → U+90000-U+9FFFD, U+A0000-U+AFFFD, U+B0000-U+BFFFD, or
   U+C0000-U+CFFD
identifier-head → U+D0000-U+DFFFD or U+E0000-U+EFFFD
identifier-character → Digit 0 through 9
identifier-character → U+0300-U+036F, U+1DC0-U+1DFF, U+20D0-U+20FF, or
   U+FE20-U+FE2F
identifier-character -- identifier-head
identifier-characters -- identifier-character identifier-characters opt
implicit-parameter-name → $ decimal-digits
property-wrapper-projection → $ identifier-characters
```

Keywords and Punctuation

The following keywords are reserved and can't be used as identifiers, unless they're escaped with backticks, as described above in <u>Identifiers</u>. Keywords other than inout, var, and let can be used as parameter names in a function declaration or function call without being escaped with backticks. When a member has the same name as a keyword, references to that member don't need to be escaped with backticks, except when there's ambiguity between referring

to the member and using the keyword—for example, self, Type, and Protocol have special meaning in an explicit member expression, so they must be escaped with backticks in that context.

- Keywords used in declarations: associated type, class, deinit, enum, extension, fileprivate, func, import, init, inout, internal, let, open, operator, private, protocol, public, rethrows, static, struct, subscript, typealias, and var.
- Keywords used in statements: break, case, continue, default, defer, do, else, fallthrough, for, guard, if, in, repeat, return, switch, where, and while.
- Keywords used in expressions and types: as, Any, catch, false, is, nil, super, self, Self, throw, throws, true, and try.
- Keywords used in patterns: _.
- Keywords that begin with a number sign (#): #available, #colorLiteral, #column, #else, #elseif, #endif, #error, #file, #fileLiteral, #function, #if, #imageLiteral, #line, #selector, #sourceLocation, and #warning.
- Keywords reserved in particular contexts: associativity, convenience, dynamic, didSet, final, get, infix, indirect, lazy, left, mutating, none, nonmutating, optional, override, postfix, precedence, prefix, Protocol, required, right, set, Type, unowned, weak, and willSet. Outside the context in which they appear in the grammar, they can be used as identifiers.

Literals

A *literal* is the source code representation of a value of a type, such as a number or string.

The following are examples of literals:

```
1 42  // Integer literal
2 3.14159  // Floating-point literal
3 "Hello, world!" // String literal
4 true  // Boolean literal
```

A literal doesn't have a type on its own. Instead, a literal is parsed as having infinite precision and Swift's type inference attempts to infer a type for the literal. For example, in the declaration <code>let x: Int8 = 42</code>, Swift uses the explicit type annotation (: Int8) to infer that the type of the integer literal 42 is <code>Int8</code>. If there isn't suitable type information available, Swift infers that the literal's type is one of the default literal types defined in the Swift standard library. The default types are <code>Int</code> for integer literals, <code>Double</code> for floating-point literals, <code>String</code> for string literals, and <code>Bool</code> for <code>Boolean</code> literals. For example, in the declaration <code>let str = "Hello, world"</code>, the default inferred type of the string literal <code>"Hello, world"</code> is <code>String</code>.

When specifying the type annotation for a literal value, the annotation's type must be a type that can be instantiated from that literal value. That is, the type must conform to one of the following Swift standard library protocols: ExpressibleByIntegerLiteral for integer literals, ExpressibleByFloatLiteral for floating-point literals, ExpressibleByStringLiteral for string literals,

ExpressibleByBooleanLiteral for Boolean literals,

ExpressibleByUnicodeScalarLiteral for string literals that contain only a single Unicode scalar, and ExpressibleByExtendedGraphemeClusterLiteral for string literals that contain only a single extended grapheme cluster. For example, Int8 conforms to the ExpressibleByIntegerLiteral protocol, and therefore it can be used in the type annotation for the integer literal 42 in the declaration let x:

Int8 = 42.

```
GRAMMAR OF A LITERAL

literal → numeric-literal | string-literal | boolean-literal | nil-literal

numeric-literal → -opt integer-literal | -opt floating-point-literal

boolean-literal → true | false

nil-literal → nil
```

Integer Literals

Integer literals represent integer values of unspecified precision. By default, integer literals are expressed in decimal; you can specify an alternate base using a prefix. Binary literals begin with 0b, octal literals begin with 0o, and hexadecimal literals begin with 0x.

Decimal literals contain the digits 0 through 9. Binary literals contain 0 and 1, octal literals contain 0 through 7, and hexadecimal literals contain 0 through 9 as well as A through F in upper- or lowercase.

Negative integers literals are expressed by prepending a minus sign (-) to an integer literal, as in -42.

Underscores (_) are allowed between digits for readability, but they're ignored and therefore don't affect the value of the literal. Integer literals can begin with leading zeros (0), but they're likewise ignored and don't affect the base or value of the literal.

Unless otherwise specified, the default inferred type of an integer literal is the Swift standard library type Int. The Swift standard library also defines types for various sizes of signed and unsigned integers, as described in <u>Integers</u>.

```
GRAMMAR OF AN INTEGER LITERAL
integer-literal → binary-literal
integer-literal → <u>octal-literal</u>
integer-literal → <u>decimal-literal</u>
integer-literal → hexadecimal-literal
binary-literal -- 0b binary-digit binary-literal-characters ont
binary-digit → Digit 0 or 1
binary-literal-character → binary-digit | _
binary-literal-characters -> binary-literal-character binary-literal-characters ont
octal-literal - 00 octal-digit octal-literal-characters opt
octal-digit → Digit 0 through 7
octal-literal-character → octal-digit | _
octal-literal-characters -- octal-literal-character octal-literal-characters opt
decimal-literal -- decimal-digit decimal-literal-characters opt
decimal-digit → Digit 0 through 9
decimal-digits → decimal-digit decimal-digits opt
decimal-literal-character → decimal-digit | _
decimal-literal-characters \rightarrow decimal-literal-character decimal-literal-characters ont
hexadecimal-literal -> 0x hexadecimal-digit hexadecimal-literal-characters ont
```

```
hexadecimal-digit → Digit 0 through 9, a through f, or A through F
hexadecimal-literal-character → hexadecimal-digit | _
hexadecimal-literal-characters → hexadecimal-literal-character hexadecimal-literal-characters opt
```

Floating-Point Literals

Floating-point literals represent floating-point values of unspecified precision.

By default, floating-point literals are expressed in decimal (with no prefix), but they can also be expressed in hexadecimal (with a 0x prefix).

Decimal floating-point literals consist of a sequence of decimal digits followed by either a decimal fraction, a decimal exponent, or both. The decimal fraction consists of a decimal point (.) followed by a sequence of decimal digits. The exponent consists of an upper- or lowercase e prefix followed by a sequence of decimal digits that indicates what power of 10 the value preceding the e is multiplied by. For example, 1.25e2 represents 1.25 x 10^2 , which evaluates to 1.25 e Similarly, 1.25e-2 represents 1.25 x 10^2 , which evaluates to 0.0125.

Hexadecimal floating-point literals consist of a $0\times$ prefix, followed by an optional hexadecimal fraction, followed by a hexadecimal exponent. The hexadecimal fraction consists of a decimal point followed by a sequence of hexadecimal digits. The exponent consists of an upper- or lowercase p prefix followed by a sequence of decimal digits that indicates what power of 2 the value preceding the p is multiplied by. For example, $0\times$ Fp2 represents 15 x 2^2 , which evaluates to 60. Similarly, $0\times$ Fp-2 represents 15 x 2^{-2} , which evaluates to 3.75.

Negative floating-point literals are expressed by prepending a minus sign (-) to a floating-point literal, as in -42.5.

Underscores (_) are allowed between digits for readability, but they're ignored and therefore don't affect the value of the literal. Floating-point literals can begin with leading zeros (0), but they're likewise ignored and don't affect the base or value of the literal.

Unless otherwise specified, the default inferred type of a floating-point literal is the Swift standard library type <code>Double</code>, which represents a 64-bit floating-point number. The Swift standard library also defines a <code>Float</code> type, which represents a 32-bit floating-point number.

```
GRAMMAR OF A FLOATING-POINT LITERAL

floating-point-literal → decimal-literal decimal-fraction opt decimal-exponent opt
floating-point-literal → hexadecimal-literal hexadecimal-fraction opt hexadecimal-
exponent
decimal-fraction → . decimal-literal
decimal-exponent → floating-point-e sign opt decimal-literal
hexadecimal-fraction → . hexadecimal-digit hexadecimal-literal-characters opt
hexadecimal-exponent → floating-point-p sign opt decimal-literal
floating-point-e → e | E
floating-point-p → p | P
sign → + | -
```

String Literals

A string literal is a sequence of characters surrounded by quotation marks. A single-line string literal is surrounded by double quotation marks and has the following form:

```
" characters "
```

String literals can't contain an unescaped double quotation mark ("), an unescaped backslash (\), a carriage return, or a line feed.

A multiline string literal is surrounded by three double quotation marks and has the following form:

```
characters
```

Unlike a single-line string literal, a multiline string literal can contain unescaped double quotation marks ("), carriage returns, and line feeds. It can't contain three

unescaped double quotation marks next to each other.

The line break after the """ that begins the multiline string literal is not part of the string. The line break before the """ that ends the literal is also not part of the string. To make a multiline string literal that begins or ends with a line feed, write a blank line as its first or last line.

A multiline string literal can be indented using any combination of spaces and tabs; this indentation is not included in the string. The """ that ends the literal determines the indentation: Every nonblank line in the literal must begin with exactly the same indentation that appears before the closing """; there's no conversion between tabs and spaces. You can include additional spaces and tabs after that indentation; those spaces and tabs appear in the string.

Line breaks in a multiline string literal are normalized to use the line feed character. Even if your source file has a mix of carriage returns and line feeds, all of the line breaks in the string will be the same.

In a multiline string literal, writing a backslash (\) at the end of a line omits that line break from the string. Any whitespace between the backslash and the line break is also omitted. You can use this syntax to hard wrap a multiline string literal in your source code, without changing the value of the resulting string.

Special characters can be included in string literals of both the single-line and multiline forms using the following escape sequences:

- Null character (\0)
- Backslash (\\)
- Horizontal tab (\t)
- Line feed (\n)
- Carriage return (\r)
- Double quotation mark (\")
- Single quotation mark (\')

• Unicode scalar ($\setminus u\{n\}$), where n is a hexadecimal number that has one to eight digits

The value of an expression can be inserted into a string literal by placing the expression in parentheses after a backslash (\). The interpolated expression can contain a string literal, but can't contain an unescaped backslash, a carriage return, or a line feed.

For example, all of the following string literals have the same value:

```
1 "1 2 3"
2 "1 2 \("3")"
3 "1 2 \(3)"
4 "1 2 \(1 + 2)"
5 let x = 3; "1 2 \(x)"
```

A string delimited by extended delimiters is a sequence of characters surrounded by quotation marks and a balanced set of one or more number signs (#). A string delimited by extended delimiters has the following forms:

```
#" characters "#

#"""

characters
"""#
```

Special characters in a string delimited by extended delimiters appear in the resulting string as normal characters rather than as special characters. You can use extended delimiters to create strings with characters that would ordinarily have a special effect such as generating a string interpolation, starting an escape sequence, or terminating the string.

The following example shows a string literal and a string delimited by extended delimiters that create equivalent string values:

```
1 let string = #"\(x) \ " \u{2603}"#
2 let escaped = "\\(x) \\ \" \\u{2603}"
3 print(string)
4 // Prints "\(x) \ " \u{2603}"
5 print(string == escaped)
6 // Prints "true"
```

If you use more than one number sign to form a string delimited by extended delimiters, don't place whitespace in between the number signs:

```
print(###"Line 1\###nLine 2"###) // OK
print(# # #"Line 1\# # #nLine 2"# # #) // Error
```

Multiline string literals that you create using extended delimiters have the same indentation requirements as regular multiline string literals.

The default inferred type of a string literal is String. For more information about the String type, see <u>Strings and Characters</u> and <u>String</u>.

String literals that are concatenated by the + operator are concatenated at compile time. For example, the values of textA and textB in the example below are identical—no runtime concatenation is performed.

```
1 let textA = "Hello " + "world"
2 let textB = "Hello world"
```

```
String-literal → <u>static-string-literal</u> | <u>interpolated-string-literal</u> | <u>string-literal</u> | <u>string-literal-opening-delimiter</u> → <u>extended-string-literal-delimiter</u> <sub>opt</sub>  "

string-literal-closing-delimiter → " <u>extended-string-literal-delimiter</u> <sub>opt</sub> 
static-string-literal → <u>string-literal-opening-delimiter</u> <u>quoted-text</u> <sub>opt</sub> <u>string-literal-closing-delimiter</u> 
static-string-literal → <u>multiline-string-literal-opening-delimiter</u> <u>multiline-quoted-text</u> <sub>opt</sub> 
<u>multiline-string-literal-closing-delimiter</u> 
multiline-string-literal-opening-delimiter → <u>extended-string-literal-delimiter</u> 
multiline-string-literal-closing-delimiter → <u>extended-string-literal-delimiter</u> 
multiline-string-literal-closing-delimiter → <u>extended-string-literal-delimiter</u>
```

```
extended-string-literal-delimiter → # extended-string-literal-delimiter ont
quoted-text → quoted-text-item quoted-text opt
quoted-text-item → escaped-character
quoted-text-item → Any Unicode scalar value except ", \, U+000A, or U+000D
multiline-quoted-text --> multiline-quoted-text-item multiline-quoted-text ont
multiline-quoted-text-item → <u>escaped-character</u>
multiline-quoted-text-item → Any Unicode scalar value except \
multiline-quoted-text-item → <u>escaped-newline</u>
interpolated-string-literal → <u>string-literal-opening-delimiter</u> <u>interpolated-text</u> ont <u>string-</u>
   literal-closing-delimiter
interpolated-string-literal --> multiline-string-literal-opening-delimiter interpolated-text opt
    multiline-string-literal-closing-delimiter
interpolated-text -- interpolated-text-item interpolated-text ont
interpolated-text-item → \( expression ) | quoted-text-item
multiline-interpolated-text \rightarrow multiline-interpolated-text-item multiline-interpolated-text ont
multiline-interpolated-text-item → \( expression \) | multiline-quoted-text-item
escape-sequence - \ extended-string-literal-delimiter
escaped-character → <u>escape-sequence</u> 0 | <u>escape-sequence</u> \ | <u>escape-sequence</u> t |
    escape-sequence n escape-sequence r escape-sequence " escape-sequence
escaped-character → escape-sequence u { unicode-scalar-digits }
unicode-scalar-digits → Between one and eight hexadecimal digits
escaped-newline → escape-sequence whitespace ont line-break
```

Operators

The Swift standard library defines a number of operators for your use, many of which are discussed in <u>Basic Operators</u> and <u>Advanced Operators</u>. The present section describes which characters can be used to define custom operators.

Custom operators can begin with one of the ASCII characters /, =, -, +, !, *, %, <, >, &, |, ^, ?, or ~, or one of the Unicode characters defined in the grammar below (which include characters from the *Mathematical Operators*, *Miscellaneous Symbols*, and *Dingbats* Unicode blocks, among others). After the first character, combining Unicode characters are also allowed.

You can also define custom operators that begin with a dot (.). These operators can contain additional dots. For example, .+. is treated as a single operator. If an operator doesn't begin with a dot, it can't contain a dot elsewhere. For example,

+.+ is treated as the + operator followed by the .+ operator.

Although you can define custom operators that contain a question mark (?), they can't consist of a single question mark character only. Additionally, although operators can contain an exclamation mark (!), postfix operators can't begin with either a question mark or an exclamation mark.

```
NOTE
```

The tokens =, ->, //, /*, */, ., the prefix operators <, &, and ?, the infix operator ?, and the postfix operators >, !, and ? are reserved. These tokens can't be overloaded, nor can they be used as custom operators.

The whitespace around an operator is used to determine whether an operator is used as a prefix operator, a postfix operator, or a binary operator. This behavior is summarized in the following rules:

- If an operator has whitespace around both sides or around neither side, it's treated as a binary operator. As an example, the +++ operator in a+++b and a +++ b is treated as a binary operator.
- If an operator has whitespace on the left side only, it's treated as a prefix unary operator. As an example, the +++ operator in a ++++b is treated as a prefix unary operator.
- If an operator has whitespace on the right side only, it's treated as a postfix unary operator. As an example, the +++ operator in a+++ b is treated as a postfix unary operator.
- If an operator has no whitespace on the left but is followed immediately by a dot (.), it's treated as a postfix unary operator. As an example, the +++ operator in a+++.b is treated as a postfix unary operator (a+++.b rather than a +++.b).

For the purposes of these rules, the characters (, [, and { before an operator, the characters),], and } after an operator, and the characters ,, ;, and : are also considered whitespace.

There's one caveat to the rules above. If the ! or ? predefined operator has no

whitespace on the left, it's treated as a postfix operator, regardless of whether it has whitespace on the right. To use the ? as the optional-chaining operator, it must not have whitespace on the left. To use it in the ternary conditional (? :) operator, it must have whitespace around both sides.

In certain constructs, operators with a leading < or > may be split into two or more tokens. The remainder is treated the same way and may be split again. As a result, there's no need to use whitespace to disambiguate between the closing > characters in constructs like Dictionary<String, Array<Int>>. In this example, the closing > characters are not treated as a single token that may then be misinterpreted as a bit shift >> operator.

To learn how to define new, custom operators, see <u>Custom Operators</u> and <u>Operator Declaration</u>. To learn how to overload existing operators, see <u>Operator Methods</u>.

```
GRAMMAR OF OPERATORS
operator -- operator-head operator-characters opt
operator → <u>dot-operator-head</u> <u>dot-operator-characters</u>
operator-head \rightarrow / | = | - | + | ! | * | % | < | > | & | | | ^ | ~ | ?
operator-head → U+00A1-U+00A7
operator-head → U+00A9 or U+00AB
operator-head → U+00AC or U+00AE
operator-head → U+00B0-U+00B1
operator-head → U+00B6, U+00BB, U+00BF, U+00D7, or U+00F7
operator-head → U+2016–U+2017
operator-head → U+2020-U+2027
operator-head → U+2030-U+203E
operator-head → U+2041–U+2053
operator-head → U+2055-U+205E
operator-head → U+2190-U+23FF
operator-head → U+2500-U+2775
operator-head → U+2794–U+2BFF
operator-head → U+2E00-U+2E7F
operator-head → U+3001-U+3003
operator-head → U+3008–U+3020
operator-head → U+3030
operator-character → operator-head
operator-character → U+0300-U+036F
operator-character → U+1DC0-U+1DFF
operator-character → U+20D0-U+20FF
operator-character → U+FE00-U+FE0F
operator-character → U+FE20-U+FE2F
operator-character → U+E0100-U+E01EF
```

```
operator-characters → operator-character operator-characters opt

dot-operator-head → .

dot-operator-character → . | operator-character

dot-operator-characters → dot-operator-character dot-operator-characters opt

binary-operator → operator

prefix-operator → operator

postfix-operator → operator

postfix-operator → operator
```

Types

In Swift, there are two kinds of types: named types and compound types. A named type is a type that can be given a particular name when it's defined. Named types include classes, structures, enumerations, and protocols. For example, instances of a user-defined class named MyClass have the type MyClass. In addition to user-defined named types, the Swift standard library defines many commonly used named types, including those that represent arrays, dictionaries, and optional values.

Data types that are normally considered basic or primitive in other languages—such as types that represent numbers, characters, and strings—are actually named types, defined and implemented in the Swift standard library using structures. Because they're named types, you can extend their behavior to suit the needs of your program, using an extension declaration, discussed in Extensions and Extension Declaration.

A *compound type* is a type without a name, defined in the Swift language itself. There are two compound types: function types and tuple types. A compound type may contain named types and other compound types. For example, the tuple type (Int, (Int, Int)) contains two elements: The first is the named type Int, and the second is another compound type (Int, Int).

You can put parentheses around a named type or a compound type. However, adding parentheses around a type doesn't have any effect. For example, (Int) is equivalent to Int.

This chapter discusses the types defined in the Swift language itself and describes the type inference behavior of Swift.

```
GRAMMAR OF A TYPE

type \rightarrow function-type

type \rightarrow array-type

type \rightarrow dictionary-type

type \rightarrow type-identifier

type \rightarrow tuple-type

type \rightarrow optional-type

type \rightarrow implicitly-unwrapped-optional-type
```

```
type \rightarrow protocol-composition-type
type \rightarrow opaque-type
type \rightarrow metatype-type
type \rightarrow self-type
type \rightarrow Any
type \rightarrow (type)
```

Type Annotation

A *type annotation* explicitly specifies the type of a variable or expression. Type annotations begin with a colon (:) and end with a type, as the following examples show:

```
1 let someTuple: (Double, Double) = (3.14159, 2.71828)
2 func someFunction(a: Int) { /* ... */ }
```

In the first example, the expression someTuple is specified to have the tuple type (Double, Double). In the second example, the parameter a to the function someFunction is specified to have the type Int.

Type annotations can contain an optional list of type attributes before the type.

```
GRAMMAR OF A TYPE ANNOTATION

type-annotation → : attributes opt inoutopt type
```

Type Identifier

A *type identifier* refers to either a named type or a type alias of a named or compound type.

Most of the time, a type identifier directly refers to a named type with the same name as the identifier. For example, Int is a type identifier that directly refers to the named type Int, and the type identifier Dictionary<String, Int> directly

refers to the named type Dictionary<String, Int>.

There are two cases in which a type identifier doesn't refer to a type with the same name. In the first case, a type identifier refers to a type alias of a named or compound type. For instance, in the example below, the use of Point in the type annotation refers to the tuple type (Int, Int).

```
typealias Point = (Int, Int)
let origin: Point = (0, 0)
```

In the second case, a type identifier uses dot (.) syntax to refer to named types declared in other modules or nested within other types. For example, the type identifier in the following code references the named type MyType that is declared in the ExampleModule module.

```
var someValue: ExampleModule.MyType
```

```
GRAMMAR OF A TYPE IDENTIFIER

type-identifier → type-name generic-argument-clause opt | type-name generic-argument-
clause opt • type-identifier

type-name → identifier
```

Tuple Type

A *tuple type* is a comma-separated list of types, enclosed in parentheses.

You can use a tuple type as the return type of a function to enable the function to return a single tuple containing multiple values. You can also name the elements of a tuple type and use those names to refer to the values of the individual elements. An element name consists of an identifier followed immediately by a colon (:). For an example that demonstrates both of these features, see Functions with Multiple Return Values.

When an element of a tuple type has a name, that name is part of the type.

All tuple types contain two or more types, except for void which is a type alias for the empty tuple type, ().

```
GRAMMAR OF A TUPLE TYPE

tuple-type \rightarrow ( ) | ( tuple-type-element , tuple-type-element-list )

tuple-type-element-list \rightarrow tuple-type-element | tuple-type-element , tuple-type-element-list

tuple-type-element \rightarrow element-name type-annotation | type type-element-name type-element-name type-element-name type-element-name type-element-name type-element-name type-element-name type-element-name type-element-name type-element-name
```

Function Type

A *function type* represents the type of a function, method, or closure and consists of a parameter and return type separated by an arrow (->):

```
( parameter type ) -> return type
```

The *parameter type* is comma-separated list of types. Because the *return type* can be a tuple type, function types support functions and methods that return multiple values.

A parameter of the function type () -> \top (where \top is any type) can apply the autoclosure attribute to implicitly create a closure at its call sites. This provides a syntactically convenient way to defer the evaluation of an expression without needing to write an explicit closure when you call the function. For an example of an autoclosure function type parameter, see <u>Autoclosures</u>.

A function type can have a variadic parameter in its *parameter type*. Syntactically, a variadic parameter consists of a base type name followed

immediately by three dots (...), as in Int.... A variadic parameter is treated as an array that contains elements of the base type name. For instance, the variadic parameter Int... is treated as [Int]. For an example that uses a variadic parameter, see <u>Variadic Parameters</u>.

To specify an in-out parameter, prefix the parameter type with the inout keyword. You can't mark a variadic parameter or a return type with the inout keyword. In-out parameters are discussed in In-Out Parameters.

If a function type has only one parameter and that parameter's type is a tuple type, then the tuple type must be parenthesized when writing the function's type. For example, ((Int, Int)) -> Void is the type of a function that takes a single parameter of the tuple type (Int, Int) and doesn't return any value. In contrast, without parentheses, (Int, Int) -> Void is the type of a function that takes two Int parameters and doesn't return any value. Likewise, because Void is a type alias for (), the function type (Void) -> Void is the same as (()) -> ()—a function that takes a single argument that is an empty tuple. These types are not the same as () -> ()—a function that takes no arguments.

Argument names in functions and methods are not part of the corresponding function type. For example:

```
func someFunction(left: Int, right: Int) {}
1
    func anotherFunction(left: Int, right: Int) {}
    func functionWithDifferentLabels(top: Int, bottom: Int) {}
4
    var f = someFunction // The type of f is (Int, Int) -> Void,
5
          not (left: Int, right: Int) -> Void.
    f = anotherFunction
7
    f = functionWithDifferentLabels // OK
8
    func functionWithDifferentArgumentTypes(left: Int, right:
9
          String) {}
    f = functionWithDifferentArgumentTypes // Error
10
```

Because argument labels are not part of a function's type, you omit them when writing a function type.

```
var operation: (lhs: Int, rhs: Int) -> Int // Error
var operation: (_ lhs: Int, _ rhs: Int) -> Int // OK
var operation: (Int, Int) -> Int // OK
```

If a function type includes more than a single arrow (->), the function types are grouped from right to left. For example, the function type (Int) -> (Int) -> Int is understood as (Int) -> ((Int) -> Int)—that is, a function that takes an Int and returns another function that takes and returns an Int.

Function types that can throw or rethrow an error must be marked with the throws keyword. The throws keyword is part of a function's type, and nonthrowing functions are subtypes of throwing functions. As a result, you can use a nonthrowing function in the same places as a throwing one. Throwing and rethrowing functions are described in Throwing Functions and Methods.

Restrictions for Nonescaping Closures

A parameter that's a nonescaping function can't be stored in a property, variable, or constant of type Any, because that might allow the value to escape.

A parameter that's a nonescaping function can't be passed as an argument to another nonescaping function parameter. This restriction helps Swift perform more of its checks for conflicting access to memory at compile time instead of at runtime. For example:

```
let external: (() -> Void) -> Void = { _ in () }
1
2
    func takesTwoFunctions(first: (() -> Void) -> Void, second: (()
          -> Void) -> Void) {
3
        first { first {} }
                                  // Error
         second { second {} }
                                  // Error
4
5
        first { second {} }
                                  // Error
6
        second { first {} }
                                  // Error
7
8
9
        first { external {} }
                                  // 0K
        external { first {} }
10
                                  // OK
11
    }
```

In the code above, both of the parameters to takesTwoFunctions(first:second:) are functions. Neither parameter is marked @escaping, so they're both nonescaping as a result.

The four function calls marked "Error" in the example above cause compiler errors. Because the first and second parameters are nonescaping functions, they can't be passed as arguments to another nonescaping function parameter. In contrast, the two function calls marked "OK" don't cause a compiler error. These function calls don't violate the restriction because external isn't one of the parameters of takesTwoFunctions(first:second:).

If you need to avoid this restriction, mark one of the parameters as escaping, or temporarily convert one of the nonescaping function parameters to an escaping function by using the withoutActuallyEscaping(_:do:) function. For information about avoiding conflicting access to memory, see Memory Safety.

```
GRAMMAR OF A FUNCTION TYPE function-type \rightarrow attributes opt function-type-argument-clause throws opt -> type function-type-argument-clause \rightarrow ( ) function-type-argument-clause \rightarrow ( function-type-argument-list \rightarrow function-type-argument | function-type-argument , function-type-argument-list
```

```
function-type-argument → <u>attributes</u> <sub>Opt</sub> inout <sub>Opt</sub> <u>type</u> | <u>argument-label</u> <u>type-annotation</u> argument-label → <u>identifier</u>
```

Array Type

The Swift language provides the following syntactic sugar for the Swift standard library Array<Element> type:

```
[type]
```

In other words, the following two declarations are equivalent:

```
let someArray: Array<String> = ["Alex", "Brian", "Dave"]
let someArray: [String] = ["Alex", "Brian", "Dave"]
```

In both cases, the constant someArray is declared as an array of strings. The elements of an array can be accessed through subscripting by specifying a valid index value in square brackets: someArray[0] refers to the element at index 0, "Alex".

You can create multidimensional arrays by nesting pairs of square brackets, where the name of the base type of the elements is contained in the innermost pair of square brackets. For example, you can create a three-dimensional array of integers using three sets of square brackets:

```
var array3D: [[[Int]]] = [[[1, 2], [3, 4]], [[5, 6], [7, 8]]]
```

When accessing the elements in a multidimensional array, the left-most subscript index refers to the element at that index in the outermost array. The next subscript index to the right refers to the element at that index in the array that's nested one level in. And so on. This means that in the example above, array3D[0] refers to [[1, 2], [3, 4]], array3D[0][1] refers to [3, 4], and array3D[0][1][1] refers to the value 4.

For a detailed discussion of the Swift standard library Array type, see Arrays.

```
GRAMMAR OF AN ARRAY TYPE

array-type → [ type ]
```

Dictionary Type

The Swift language provides the following syntactic sugar for the Swift standard library Dictionary<Key, Value> type:

```
[ key type : value type ]
```

In other words, the following two declarations are equivalent:

In both cases, the constant someDictionary is declared as a dictionary with strings as keys and integers as values.

The values of a dictionary can be accessed through subscripting by specifying the corresponding key in square brackets: <code>someDictionary["Alex"]</code> refers to the value associated with the key "Alex". The subscript returns an optional value of the dictionary's value type. If the specified key isn't contained in the dictionary, the subscript returns <code>nil</code>.

The key type of a dictionary must conform to the Swift standard library Hashable protocol.

For a detailed discussion of the Swift standard library Dictionary type, see Dictionaries.

```
GRAMMAR OF A DICTIONARY TYPE
```

```
dictionary-type \rightarrow [ type : type ]
```

Optional Type

The Swift language defines the postfix ? as syntactic sugar for the named type <code>Optional<Wrapped></code>, which is defined in the Swift standard library. In other words, the following two declarations are equivalent:

```
var optionalInteger: Int?
var optionalInteger: Optional<Int>
```

In both cases, the variable optionalInteger is declared to have the type of an optional integer. Note that no whitespace may appear between the type and the ?.

The type <code>Optional<Wrapped></code> is an enumeration with two cases, none and <code>some(Wrapped)</code>, which are used to represent values that may or may not be present. Any type can be explicitly declared to be (or implicitly converted to) an optional type. If you don't provide an initial value when you declare an optional variable or property, its value automatically defaults to <code>nil</code>.

If an instance of an optional type contains a value, you can access that value using the postfix operator !, as shown below:

```
optionalInteger = 42
optionalInteger! // 42
```

Using the ! operator to unwrap an optional that has a value of nil results in a runtime error.

You can also use optional chaining and optional binding to conditionally perform an operation on an optional expression. If the value is nil, no operation is performed and therefore no runtime error is produced.

For more information and to see examples that show how to use optional types,

see Optionals.

```
GRAMMAR OF AN OPTIONAL TYPE

optional-type → type ?
```

Implicitly Unwrapped Optional Type

The Swift language defines the postfix! as syntactic sugar for the named type <code>optional<wrapped></code>, which is defined in the Swift standard library, with the additional behavior that it's automatically unwrapped when it's accessed. If you try to use an implicitly unwrapped optional that has a value of <code>nil</code>, you'll get a runtime error. With the exception of the implicit unwrapping behavior, the following two declarations are equivalent:

```
var implicitlyUnwrappedString: String!
var explicitlyUnwrappedString: Optional<String>
```

Note that no whitespace may appear between the type and the !.

Because implicit unwrapping changes the meaning of the declaration that contains that type, optional types that are nested inside a tuple type or a generic type—such as the element types of a dictionary or array—can't be marked as implicitly unwrapped. For example:

```
1 let tupleOfImplicitlyUnwrappedElements: (Int!, Int!) // Error
2 let implicitlyUnwrappedTuple: (Int, Int)! // OK
3
4 let arrayOfImplicitlyUnwrappedElements: [Int!] // Error
5 let implicitlyUnwrappedArray: [Int]! // OK
```

Because implicitly unwrapped optionals have the same <code>Optional<Wrapped></code> type as optional values, you can use implicitly unwrapped optionals in all the same places in your code that you can use optionals. For example, you can assign

values of implicitly unwrapped optionals to variables, constants, and properties of optionals, and vice versa.

As with optionals, if you don't provide an initial value when you declare an implicitly unwrapped optional variable or property, its value automatically defaults to nil.

Use optional chaining to conditionally perform an operation on an implicitly unwrapped optional expression. If the value is nil, no operation is performed and therefore no runtime error is produced.

For more information about implicitly unwrapped optional types, see <u>Implicitly Unwrapped Optionals</u>.

```
GRAMMAR OF AN IMPLICITLY UNWRAPPED OPTIONAL TYPE

implicitly-unwrapped-optional-type → type!
```

Protocol Composition Type

A *protocol composition type* defines a type that conforms to each protocol in a list of specified protocols, or a type that is a subclass of a given class and conforms to each protocol in a list of specified protocols. Protocol composition types may be used only when specifying a type in type annotations, in generic parameter clauses, and in generic where clauses.

Protocol composition types have the following form:

```
Protocol 1 & Protocol 2
```

A protocol composition type allows you to specify a value whose type conforms to the requirements of multiple protocols without explicitly defining a new, named protocol that inherits from each protocol you want the type to conform to. For example, you can use the protocol composition type ProtocolA & ProtocolB & ProtocolC instead of declaring a new protocol that inherits from ProtocolA, ProtocolB, and ProtocolC. Likewise, you can use SuperClass & ProtocolA

instead of declaring a new protocol that is a subclass of SuperClass and conforms to ProtocolA.

Each item in a protocol composition list is one of the following; the list can contain at most one class:

- The name of a class
- The name of a protocol
- A type alias whose underlying type is a protocol composition type, a protocol, or a class.

When a protocol composition type contains type aliases, it's possible for the same protocol to appear more than once in the definitions—duplicates are ignored. For example, the definition of PQR in the code below is equivalent to P & Q & R.

```
typealias PQ = P & Q
typealias PQR = PQ & Q & R
```

```
GRAMMAR OF A PROTOCOL COMPOSITION TYPE

protocol-composition-type → type-identifier & protocol-composition-continuation

protocol-composition-continuation → type-identifier | protocol-composition-type
```

Opaque Type

An *opaque type* defines a type that conforms to a protocol or protocol composition, without specifying the underlying concrete type.

Opaque types appear as the return type of a function or subscript, or the type of a property. Opaque types can't appear as part of a tuple type or a generic type, such as the element type of an array or the wrapped type of an optional.

Opaque types have the following form:

some (constraint

The *constraint* is a class type, protocol type, protocol composition type, or Any. A value can be used as an instance of the opaque type only if it's an instance of a type that conforms to the listed protocol or protocol composition, or inherits from the listed class. Code that interacts with an opaque value can use the value only in ways that are part of the interface defined by the *constraint*.

Protocol declarations can't include opaque types. Classes can't use an opaque type as the return type of a nonfinal method.

A function that uses an opaque type as its return type must return values that share a single underlying type. The return type can include types that are part of the function's generic type parameters. For example, a function <code>someFunction<T></code> () could return a value of type <code>T</code> or <code>Dictionary<String</code>, <code>T></code>.

```
GRAMMAR OF AN OPAQUE TYPE

opaque-type → some type
```

Metatype Type

A *metatype type* refers to the type of any type, including class types, structure types, enumeration types, and protocol types.

The metatype of a class, structure, or enumeration type is the name of that type followed by .Type. The metatype of a protocol type—not the concrete type that conforms to the protocol at runtime—is the name of that protocol followed by .Protocol. For example, the metatype of the class type SomeClass is SomeClass.Type and the metatype of the protocol SomeProtocol is SomeProtocol.Protocol.

You can use the postfix self expression to access a type as a value. For example, SomeClass.self returns SomeClass itself, not an instance of SomeClass. And SomeProtocol.self returns SomeProtocol itself, not an instance of a type that conforms to SomeProtocol at runtime. You can call the type(of:) function with

an instance of a type to access that instance's dynamic, runtime type as a value, as the following example shows:

```
class SomeBaseClass {
1
2
        class func printClassName() {
             print("SomeBaseClass")
3
4
        }
5
    }
    class SomeSubClass: SomeBaseClass {
6
7
        override class func printClassName() {
             print("SomeSubClass")
9
        }
10
    }
    let someInstance: SomeBaseClass = SomeSubClass()
11
12
    // The compile-time type of someInstance is SomeBaseClass,
    // and the runtime type of someInstance is SomeSubClass
13
    type(of: someInstance).printClassName()
14
    // Prints "SomeSubClass"
15
```

For more information, see <u>type(of:)</u> in the Swift standard library.

Use an initializer expression to construct an instance of a type from that type's metatype value. For class instances, the initializer that's called must be marked with the required keyword or the entire class marked with the final keyword.

```
class AnotherSubClass: SomeBaseClass {
  let string: String
  required init(string: String) {
    self.string = string
  }
  override class func printClassName() {
    print("AnotherSubClass")
```

```
8  }
9  }
10  let metatype: AnotherSubClass.Type = AnotherSubClass.self
11  let anotherInstance = metatype.init(string: "some string")
```

```
GRAMMAR OF A METATYPE TYPE

metatype-type → type . Type | type . Protocol
```

Self Type

The Self type isn't a specific type, but rather lets you conveniently refer to the current type without repeating or knowing that type's name.

In a protocol declaration or a protocol member declaration, the Self type refers to the eventual type that conforms to the protocol.

In a structure, class, or enumeration declaration, the Self type refers to the type introduced by the declaration. Inside the declaration for a member of a type, the Self type refers to that type. In the members of a class declaration, Self can appear as the return type of a method and in the body of a method, but not in any other context. For example, the code below shows an instance method f whose return type is Self.

```
1
   class Superclass {
2
        func f() -> Self { return self }
   }
3
   let x = Superclass()
4
5
   print(type(of: x.f()))
   // Prints "Superclass"
6
7
   class Subclass: Superclass { }
8
   let y = Subclass()
9
```

```
print(type(of: y.f()))

// Prints "Subclass"

let z: Superclass = Subclass()

print(type(of: z.f()))

// Prints "Subclass"
```

The last part of the example above shows that Self refers to the runtime type Subclass of the value of z, not the compile-time type Superclass of the variable itself.

Inside a nested type declaration, the Self type refers to the type introduced by the innermost type declaration.

The Self type refers to the same type as the type(of:) function in the Swift standard library. Writing Self.someStaticMember to access a member of the current type is the same as writing type(of: self).someStaticMember.

```
GRAMMAR OF A SELF TYPE

self-type → Self
```

Type Inheritance Clause

A *type inheritance clause* is used to specify which class a named type inherits from and which protocols a named type conforms to. A type inheritance clause begins with a colon (:), followed by a list of type identifiers.

Class types can inherit from a single superclass and conform to any number of protocols. When defining a class, the name of the superclass must appear first in the list of type identifiers, followed by any number of protocols the class must conform to. If the class doesn't inherit from another class, the list can begin with a protocol instead. For an extended discussion and several examples of class inheritance, see <u>Inheritance</u>.

Other named types can only inherit from or conform to a list of protocols. Protocol types can inherit from any number of other protocols. When a protocol type inherits from other protocols, the set of requirements from those other protocols are aggregated together, and any type that inherits from the current protocol must conform to all of those requirements.

A type inheritance clause in an enumeration definition can be either a list of protocols, or in the case of an enumeration that assigns raw values to its cases, a single, named type that specifies the type of those raw values. For an example of an enumeration definition that uses a type inheritance clause to specify the type of its raw values, see <u>Raw Values</u>.

```
GRAMMAR OF A TYPE INHERITANCE CLAUSE

type-inheritance-clause \rightarrow type-inheritance-list

type-inheritance-list \rightarrow type-identifier \mid type-identifier, type-inheritance-list
```

Type Inference

Swift uses *type inference* extensively, allowing you to omit the type or part of the type of many variables and expressions in your code. For example, instead of writing var x: Int = 0, you can write var x = 0, omitting the type completely —the compiler correctly infers that x names a value of type Int. Similarly, you can omit part of a type when the full type can be inferred from context. For example, if you write let dict: Dictionary = ["A": 1], the compiler infers that dict has the type Dictionary<String, Int>.

In both of the examples above, the type information is passed up from the leaves of the expression tree to its root. That is, the type of \times in var \times : Int = 0 is inferred by first checking the type of 0 and then passing this type information up to the root (the variable \times).

In Swift, type information can also flow in the opposite direction—from the root down to the leaves. In the following example, for instance, the explicit type annotation (: Float) on the constant eFloat causes the numeric literal 2.71828 to have an inferred type of Float instead of Double.

```
let e = 2.71828 // The type of e is inferred to be Double.

let eFloat: Float = 2.71828 // The type of eFloat is Float.
```

Type inference in Swift operates at the level of a single expression or statement. This means that all of the information needed to infer an omitted type or part of a type in an expression must be accessible from type-checking the expression or one of its subexpressions.

Expressions

In Swift, there are four kinds of expressions: prefix expressions, binary expressions, primary expressions, and postfix expressions. Evaluating an expression returns a value, causes a side effect, or both.

Prefix and binary expressions let you apply operators to smaller expressions. Primary expressions are conceptually the simplest kind of expression, and they provide a way to access values. Postfix expressions, like prefix and binary expressions, let you build up more complex expressions using postfixes such as function calls and member access. Each kind of expression is described in detail in the sections below.

```
GRAMMAR OF AN EXPRESSION

expression \rightarrow try-operator_{opt} prefix-expression binary-expressions_{opt}

expression-list \rightarrow expression | expression | expression | expression-list
```

Prefix Expressions

Prefix expressions combine an optional prefix operator with an expression. Prefix operators take one argument, the expression that follows them.

For information about the behavior of these operators, see <u>Basic Operators</u> and <u>Advanced Operators</u>.

For information about the operators provided by the Swift standard library, see Operator Declarations.

In addition to the standard library operators, you use & immediately before the name of a variable that's being passed as an in-out argument to a function call expression. For more information and to see an example, see <u>In-Out Parameters</u>.

```
GRAMMAR OF A PREFIX EXPRESSION

prefix-expression → prefix-operator opt postfix-expression
```

```
prefix-expression → <u>in-out-expression</u>
in-out-expression → & <u>identifier</u>
```

Try Operator

A *try expression* consists of the try operator followed by an expression that can throw an error. It has the following form:

```
try (expression)
```

An *optional-try expression* consists of the try? operator followed by an expression that can throw an error. It has the following form:

```
try? (expression)
```

If the *expression* does not throw an error, the value of the optional-try expression is an optional containing the value of the *expression*. Otherwise, the value of the optional-try expression is nil.

A *forced-try expression* consists of the try! operator followed by an expression that can throw an error. It has the following form:

```
try! expression
```

If the *expression* throws an error, a runtime error is produced.

When the expression on the left-hand side of a binary operator is marked with try, try?, or try!, that operator applies to the whole binary expression. That said, you can use parentheses to be explicit about the scope of the operator's application.

A try expression can't appear on the right-hand side of a binary operator, unless the binary operator is the assignment operator or the try expression is enclosed in parentheses.

For more information and to see examples of how to use try, try?, and try!, see Error Handling.

```
GRAMMAR OF A TRY EXPRESSION

try-operator → try | try ? | try !
```

Binary Expressions

Binary expressions combine an infix binary operator with the expression that it takes as its left-hand and right-hand arguments. It has the following form:

```
left-hand argument operator right-hand argument
```

For information about the behavior of these operators, see <u>Basic Operators</u> and <u>Advanced Operators</u>.

For information about the operators provided by the Swift standard library, see Operator Declarations.

```
{\tt NOTE}
```

At parse time, an expression made up of binary operators is represented as a flat list. This list is transformed into a tree by applying operator precedence. For example, the expression 2 + 3 * 5 is initially understood as a flat list of five items, 2, +, 3, *, and 5. This process transforms it into the tree (2 + (3 * 5)).

```
GRAMMAR OF A BINARY EXPRESSION

binary-expression → binary-operator prefix-expression

binary-expression → assignment-operator try-operator opt prefix-expression
```

```
binary-expression → <u>conditional-operator</u> <u>try-operator</u> <u>opt</u> <u>prefix-expression</u>
binary-expression → <u>type-casting-operator</u>
binary-expressions → <u>binary-expression</u> <u>binary-expressions</u> opt
```

Assignment Operator

The *assignment operator* sets a new value for a given expression. It has the following form:

```
expression = value
```

The value of the *expression* is set to the value obtained by evaluating the *value*. If the *expression* is a tuple, the *value* must be a tuple with the same number of elements. (Nested tuples are allowed.) Assignment is performed from each part of the *value* to the corresponding part of the *expression*. For example:

```
1 (a, _, (b, c)) = ("test", 9.45, (12, 3))
2 // a is "test", b is 12, c is 3, and 9.45 is ignored
```

The assignment operator does not return any value.

```
GRAMMAR OF AN ASSIGNMENT OPERATOR

assignment-operator → =
```

Ternary Conditional Operator

The *ternary conditional operator* evaluates to one of two given values based on the value of a condition. It has the following form:

```
condition ? expression used if true :

expression used if false
```

If the *condition* evaluates to true, the conditional operator evaluates the first expression and returns its value. Otherwise, it evaluates the second expression

and returns its value. The unused expression is not evaluated.

For an example that uses the ternary conditional operator, see <u>Ternary</u> <u>Conditional Operator</u>.

```
GRAMMAR OF A CONDITIONAL OPERATOR

conditional-operator → ? expression :
```

Type-Casting Operators

There are four type-casting operators: the is operator, the as operator, the as? operator, and the as! operator.

They have the following form:

```
expression is type
expression as type
expression as? type
expression as! type
```

The is operator checks at runtime whether the *expression* can be cast to the specified *type*. It returns true if the *expression* can be cast to the specified *type*; otherwise, it returns false.

The as operator performs a cast when it is known at compile time that the cast always succeeds, such as upcasting or bridging. Upcasting lets you use an expression as an instance of its type's supertype, without using an intermediate variable. The following approaches are equivalent:

```
func f(_ any: Any) { print("Function for Any") }
func f(_ int: Int) { print("Function for Int") }
let x = 10
f(x)
// Prints "Function for Int"
```

```
6
7 let y: Any = x
8 f(y)
9 // Prints "Function for Any"
10
11 f(x as Any)
12 // Prints "Function for Any"
```

Bridging lets you use an expression of a Swift standard library type such as String as its corresponding Foundation type such as NSString without needing to create a new instance. For more information on bridging, see Working with Foundation Types.

The as? operator performs a conditional cast of the *expression* to the specified *type*. The as? operator returns an optional of the specified *type*. At runtime, if the cast succeeds, the value of *expression* is wrapped in an optional and returned; otherwise, the value returned is nil. If casting to the specified *type* is guaranteed to fail or is guaranteed to succeed, a compile-time error is raised.

The as! operator performs a forced cast of the *expression* to the specified *type*. The as! operator returns a value of the specified *type*, not an optional type. If the cast fails, a runtime error is raised. The behavior of \times as! \top is the same as the behavior of $(\times$ as? \top)!.

For more information about type casting and to see examples that use the type-casting operators, see <u>Type Casting</u>.

```
GRAMMAR OF A TYPE-CASTING OPERATOR

type-casting-operator → is type

type-casting-operator → as type

type-casting-operator → as ? type

type-casting-operator → as ! type
```

Primary Expressions

Primary expressions are the most basic kind of expression. They can be used as expressions on their own, and they can be combined with other tokens to make prefix expressions, binary expressions, and postfix expressions.

```
primary-expression → identifier generic-argument-clause opt

primary-expression → literal-expression

primary-expression → self-expression

primary-expression → superclass-expression

primary-expression → closure-expression

primary-expression → parenthesized-expression

primary-expression → tuple-expression

primary-expression → implicit-member-expression

primary-expression → wildcard-expression

primary-expression → key-path-expression

primary-expression → selector-expression

primary-expression → key-path-string-expression
```

Literal Expression

A *literal expression* consists of either an ordinary literal (such as a string or a number), an array or dictionary literal, a playground literal, or one of the following special literals:

Literal	Type	Value
#file	String	The name of the file in which it appears.
#line	Int	The line number on which it appears.
#column	Int	The column number in which it begins.
#function	String	The name of the declaration in which it appears.

#dsohandle UnsafeRawPointer The DSO (dynamic shared object) handle in use where it appears.

Inside a function, the value of #function is the name of that function, inside a method it is the name of that method, inside a property getter or setter it is the name of that property, inside special members like init or subscript it is the name of that keyword, and at the top level of a file it is the name of the current module.

When used as the default value of a function or method parameter, the special literal's value is determined when the default value expression is evaluated at the call site.

```
func logFunctionName(string: String = #function) {
   print(string)
}

func myFunction() {
   logFunctionName() // Prints "myFunction()".
}
```

An *array literal* is an ordered collection of values. It has the following form:

```
[ value 1 , value 2 , ...]
```

The last expression in the array can be followed by an optional comma. The value of an array literal has type $[\top]$, where \top is the type of the expressions inside it. If there are expressions of multiple types, \top is their closest common supertype. Empty array literals are written using an empty pair of square brackets and can be used to create an empty array of a specified type.

```
var emptyArray: [Double] = []
```

A *dictionary literal* is an unordered collection of key-value pairs. It has the following form:

```
[ key 1: value 1, key 2: value 2, ...]
```

The last expression in the dictionary can be followed by an optional comma. The value of a dictionary literal has type <code>[Key: Value]</code>, where <code>Key</code> is the type of its key expressions and <code>Value</code> is the type of its value expressions. If there are expressions of multiple types, <code>Key</code> and <code>Value</code> are the closest common supertype for their respective values. An empty dictionary literal is written as a colon inside a pair of brackets (<code>[:]</code>) to distinguish it from an empty array literal. You can use an empty dictionary literal to create an empty dictionary literal of specified key and value types.

```
var emptyDictionary: [String: Double] = [:]
```

A *playground literal* is used by Xcode to create an interactive representation of a color, file, or image within the program editor. Playground literals in plain text outside of Xcode are represented using a special literal syntax.

For information on using playground literals in Xcode, see <u>Add a color, file, or image literal</u> in Xcode Help.

Self Expression

The self expression is an explicit reference to the current type or instance of the type in which it occurs. It has the following forms:

```
self
self. member name
self[subscript index]
self(initializer arguments)
self.init(initializer arguments)
```

In an initializer, subscript, or instance method, self refers to the current instance of the type in which it occurs. In a type method, self refers to the current type in which it occurs.

The self expression is used to specify scope when accessing members, providing disambiguation when there is another variable of the same name in scope, such as a function parameter. For example:

```
class SomeClass {
  var greeting: String
  init(greeting: String) {
    self.greeting = greeting
}
}
```

In a mutating method of a value type, you can assign a new instance of that value type to self. For example:

```
struct Point {
   var x = 0.0, y = 0.0
   mutating func moveBy(x deltaX: Double, y deltaY: Double) {
     self = Point(x: x + deltaX, y: y + deltaY)
   }
}
```

```
Self-expression → self | self-method-expression | self-subscript-expression | self-initializer-expression | self-method-expression | self-method-expression → self . identifier | self-subscript-expression → self [ function-call-argument-list ] | self-initializer-expression → self . init
```

Superclass Expression

A *superclass expression* lets a class interact with its superclass. It has one of the following forms:

```
super. member name
super[subscript index]
super.init(initializer arguments)
```

The first form is used to access a member of the superclass. The second form is used to access the superclass's subscript implementation. The third form is used to access an initializer of the superclass.

Subclasses can use a superclass expression in their implementation of members, subscripting, and initializers to make use of the implementation in their superclass.

```
GRAMMAR OF A SUPERCLASS EXPRESSION

superclass-expression → superclass-method-expression | superclass-subscript-expression | superclass-initializer-expression

superclass-method-expression → super . identifier

superclass-subscript-expression → super [ function-call-argument-list ]

superclass-initializer-expression → super . init
```

Closure Expression

A *closure expression* creates a closure, also known as a *lambda* or an *anonymous function* in other programming languages. Like a function declaration, a closure contains statements, and it captures constants and variables from its enclosing

scope. It has the following form:

```
{ ( parameters ) -> return type in
    statements
}
```

The *parameters* have the same form as the parameters in a function declaration, as described in <u>Function Declaration</u>.

There are several special forms that allow closures to be written more concisely:

- A closure can omit the types of its parameters, its return type, or both. If you omit the parameter names and both types, omit the in keyword before the statements. If the omitted types can't be inferred, a compile-time error is raised.
- A closure may omit names for its parameters. Its parameters are then implicitly named \$ followed by their position: \$0, \$1, \$2, and so on.
- A closure that consists of only a single expression is understood to return the value of that expression. The contents of this expression are also considered when performing type inference on the surrounding expression.

The following closure expressions are equivalent:

```
myFunction { (x: Int, y: Int) -> Int in
1
2
         return x + y
3
    }
4
    myFunction { x, y in
5
6
         return x + y
7
    }
8
    myFunction { return $0 + $1 }
9
10
```

```
11 myFunction { $0 + $1 }
```

For information about passing a closure as an argument to a function, see <u>Function Call Expression</u>.

Closure expressions can be used without being stored in a variable or constant, such as when you immediately use a closure as part of a function call. The closure expressions passed to myFunction in code above are examples of this kind of immediate use. As a result, whether a closure expression is escaping or nonescaping depends on the surrounding context of the expression. A closure expression is nonescaping if it is called immediately or passed as a nonescaping function argument. Otherwise, the closure expression is escaping.

For more information about escaping closures, see **Escaping Closures**.

Capture Lists

By default, a closure expression captures constants and variables from its surrounding scope with strong references to those values. You can use a *capture list* to explicitly control how values are captured in a closure.

A capture list is written as a comma-separated list of expressions surrounded by square brackets, before the list of parameters. If you use a capture list, you must also use the in keyword, even if you omit the parameter names, parameter types, and return type.

The entries in the capture list are initialized when the closure is created. For each entry in the capture list, a constant is initialized to the value of the constant or variable that has the same name in the surrounding scope. For example in the code below, a is included in the capture list but b is not, which gives them different behavior.

```
var a = 0
var b = 0
let closure = { [a] in
print(a, b)
```

```
5 }
6
7 a = 10
8 b = 10
9 closure()
10 // Prints "0 10"
```

There are two different things named a, the variable in the surrounding scope and the constant in the closure's scope, but only one variable named b. The a in the inner scope is initialized with the value of the a in the outer scope when the closure is created, but their values are not connected in any special way. This means that a change to the value of a in the outer scope does not affect the value of a in the inner scope, nor does a change to a inside the closure affect the value of a outside the closure. In contrast, there is only one variable named b—the b in the outer scope—so changes from inside or outside the closure are visible in both places.

This distinction is not visible when the captured variable's type has reference semantics. For example, there are two things named \times in the code below, a variable in the outer scope and a constant in the inner scope, but they both refer to the same object because of reference semantics.

```
1
    class SimpleClass {
 2
         var value: Int = 0
    }
 3
    var x = SimpleClass()
 4
    var y = SimpleClass()
 5
 6
    let closure = { [x] in
 7
         print(x.value, y.value)
    }
 8
 9
10
     x.value = 10
     y.value = 10
11
```

```
12 closure()
13 // Prints "10 10"
```

If the type of the expression's value is a class, you can mark the expression in a capture list with weak or unowned to capture a weak or unowned reference to the expression's value.

```
myFunction { print(self.title) }  // implicit
    strong capture

myFunction { [self] in print(self.title) }  // explicit
    strong capture

myFunction { [weak self] in print(self!.title) }  // weak
    capture

myFunction { [unowned self] in print(self.title) }  // unowned
    capture
```

You can also bind an arbitrary expression to a named value in a capture list. The expression is evaluated when the closure is created, and the value is captured with the specified strength. For example:

```
// Weak capture of "self.parent" as "parent"
myFunction { [weak parent = self.parent] in
print(parent!.title) }
```

For more information and examples of closure expressions, see <u>Closure</u> <u>Expressions</u>. For more information and examples of capture lists, see <u>Resolving</u> <u>Strong Reference Cycles for Closures</u>.

```
GRAMMAR OF A CLOSURE EXPRESSION

closure-expression \rightarrow \{ closure-signature \ opt \ statements \ opt \ \}

closure-signature \rightarrow capture-list \ opt \ closure-parameter-clause \ throws_{opt} \ function-result \ opt \ in

closure-signature \rightarrow capture-list \ in

closure-parameter-clause \rightarrow ( ) | ( closure-parameter-list \ ) | identifier-list \ closure-parameter-list \rightarrow closure-parameter | closure-parameter \ , closure-parameter-list \ )
```

```
      list
      closure-parameter → closure-parameter-name type-annotation opt

      closure-parameter → closure-parameter-name type-annotation ...

      closure-parameter-name → identifier

      capture-list → [ capture-list-items ]

      capture-list-items → capture-list-item | capture-list-item , capture-list-items

      capture-list-item → capture-specifier opt expression

      capture-specifier → weak | unowned | unowned(safe) | unowned(unsafe)
```

Implicit Member Expression

An *implicit member expression* is an abbreviated way to access a member of a type, such as an enumeration case or a type method, in a context where type inference can determine the implied type. It has the following form:

```
. member name
```

For example:

```
var x = MyEnumeration.someValue
x = .anotherValue
```

```
GRAMMAR OF A IMPLICIT MEMBER EXPRESSION implicit-member-expression \rightarrow . identifier
```

Parenthesized Expression

A *parenthesized expression* consists of an expression surrounded by parentheses. You can use parentheses to specify the precedence of operations by explicitly grouping expressions. Grouping parentheses don't change an expression's type —for example, the type of (1) is simply Int.

```
GRAMMAR OF A PARENTHESIZED EXPRESSION

parenthesized-expression → ( expression )
```

Tuple Expression

A *tuple expression* consists of a comma-separated list of expressions surrounded by parentheses. Each expression can have an optional identifier before it, separated by a colon (:). It has the following form:

```
(identifier 1: expression 1, identifier 2: expression 2, ...)
```

Each identifier in a tuple expression must be unique within the scope of the tuple expression. In a nested tuple expression, identifiers at the same level of nesting must be unique. For example, (a: 10, a: 20) is invalid because the label a appears twice at the same level. However, (a: 10, b: (a: 1, \times : 2)) is valid—although a appears twice, it appears once in the outer tuple and once in the inner tuple.

A tuple expression can contain zero expressions, or it can contain two or more expressions. A single expression inside parentheses is a parenthesized expression.

```
NOTE
```

Both an empty tuple expression and an empty tuple type are written () in Swift. Because Void is a type alias for (), you can use it to write an empty tuple type. However, like all type aliases, Void is always a type—you can't use it to write an empty tuple expression.

```
GRAMMAR OF A TUPLE EXPRESSION

tuple-expression \rightarrow ( ) | ( tuple-element , tuple-element-list )

tuple-element-list \rightarrow tuple-element | tuple-element , tuple-element-list

tuple-element \rightarrow expression | identifier : expression
```

Wildcard Expression

A *wildcard expression* is used to explicitly ignore a value during an assignment. For example, in the following assignment 10 is assigned to \times and 20 is ignored:

```
1 (x, _) = (10, 20)
```

```
GRAMMAR OF A WILDCARD EXPRESSION

wildcard-expression → __
```

Key-Path Expression

A *key-path expression* refers to a property or subscript of a type. You use key-path expressions in dynamic programming tasks, such as key-value observing. They have the following form:

```
\type name . path
```

The *type name* is the name of a concrete type, including any generic parameters, such as String, [Int], or Set<Int>.

The *path* consists of property names, subscripts, optional-chaining expressions, and forced unwrapping expressions. Each of these key-path components can be repeated as many times as needed, in any order.

At compile time, a key-path expression is replaced by an instance of the KeyPath class.

To access a value using a key path, pass the key path to the subscript(keyPath:) subscript, which is available on all types. For example:

```
struct SomeStructure {
   var someValue: Int
}

let s = SomeStructure(someValue: 12)
let pathToProperty = \SomeStructure.someValue

let value = s[keyPath: pathToProperty]
```

```
9 // value is 12
```

The *type name* can be omitted in contexts where type inference can determine the implied type. The following code uses \.someProperty instead of

\SomeClass.someProperty:

```
class SomeClass: NSObject {
1
2
        @objc var someProperty: Int
3
        init(someProperty: Int) {
             self.someProperty = someProperty
4
5
        }
6
    }
7
    let c = SomeClass(someProperty: 10)
    c.observe(\.someProperty) { object, change in
10
        // ...
11
    }
```

The *path* can refer to self to create the identity key path (\.self). The identity key path refers to a whole instance, so you can use it to access and change all of the data stored in a variable in a single step. For example:

```
var compoundValue = (a: 1, b: 2)

// Equivalent to compoundValue = (a: 10, b: 20)

compoundValue[keyPath: \.self] = (a: 10, b: 20)
```

The *path* can contain multiple property names, separated by periods, to refer to a property of a property's value. This code uses the key path expression \OuterStructure.outer.someValue to access the someValue property of the OuterStructure type's outer property:

```
struct OuterStructure {
   var outer: SomeStructure
```

```
3
        init(someValue: Int) {
             self.outer = SomeStructure(someValue: someValue)
4
5
        }
    }
6
7
    let nested = OuterStructure(someValue: 24)
8
    let nestedKeyPath = \OuterStructure.outer.someValue
10
    let nestedValue = nested[keyPath: nestedKeyPath]
11
    // nestedValue is 24
12
```

The *path* can include subscripts using brackets, as long as the subscript's parameter type conforms to the Hashable protocol. This example uses a subscript in a key path to access the second element of an array:

```
let greetings = ["hello", "hola", "bonjour", "" ]
let myGreeting = greetings[keyPath: \[String].[1]]
// myGreeting is 'hola'
```

The value used in a subscript can be a named value or a literal. Values are captured in key paths using value semantics. The following code uses the variable index in both a key-path expression and in a closure to access the third element of the greetings array. When index is modified, the key-path expression still references the third element, while the closure uses the new index.

```
var index = 2
let path = \[String].[index]
let fn: ([String]) -> String = { strings in strings[index] }

print(greetings[keyPath: path])
// Prints "bonjour"
print(fn(greetings))
```

```
8
    // Prints "bonjour"
9
    // Setting 'index' to a new value doesn't affect 'path'
10
11
    index += 1
12
    print(greetings[keyPath: path])
    // Prints "bonjour"
13
14
15
    // Because 'fn' closes over 'index', it uses the new value
    print(fn(greetings))
16
    // Prints ""
17
```

The *path* can use optional chaining and forced unwrapping. This code uses optional chaining in a key path to access a property of an optional string:

```
1 let firstGreeting: String? = greetings.first
2 print(firstGreeting?.count as Any)
3 // Prints "Optional(5)"
4
5 // Do the same thing using a key path.
6 let count = greetings[keyPath: \[String].first?.count]
7 print(count as Any)
8 // Prints "Optional(5)"
```

You can mix and match components of key paths to access values that are deeply nested within a type. The following code accesses different values and properties of a dictionary of arrays by using key-path expressions that combine these components.

```
91]]
    print(interestingNumbers[keyPath: \[String: [Int]].["prime"]]
4
          as Any)
    // Prints "Optional([2, 3, 5, 7, 11, 13, 17])"
5
    print(interestingNumbers[keyPath: \[String: [Int]].["prime"]!
6
          [0]
7
    // Prints "2"
    print(interestingNumbers[keyPath: \[String: [Int]].
8
          ["hexagonal"]!.count])
    // Prints "7"
9
    print(interestingNumbers[keyPath: \[String: [Int]].
10
          ["hexagonal"]!.count.bitWidth])
    // Prints "64"
11
```

For more information about using key paths in code that interacts with Objective-C APIs, see <u>Using Objective-C Runtime Features in Swift</u>. For information about key-value coding and key-value observing, see <u>Key-Value Coding Programming Guide</u> and <u>Key-Value Observing Programming Guide</u>.

```
GRAMMAR OF A KEY-PATH EXPRESSION

key-path-expression → \ type opt · key-path-components

key-path-components → key-path-component | key-path-component · key-path-components

key-path-component → identifier key-path-postfixes opt | key-path-postfixes

key-path-postfixes → key-path-postfix key-path-postfixes opt

key-path-postfix → ? | ! | self | [ function-call-argument-list ]
```

Selector Expression

A selector expression lets you access the selector used to refer to a method or to a property's getter or setter in Objective-C. It has the following form:

```
#selector( method name )
```

```
#selector(getter: property name)
#selector(setter: property name)
```

The *method name* and *property name* must be a reference to a method or a property that is available in the Objective-C runtime. The value of a selector expression is an instance of the Selector type. For example:

```
class SomeClass: NSObject {
1
2
        @objc let property: String
        @objc(doSomethingWithInt:)
3
        func doSomething(_ x: Int) {}
4
5
        init(property: String) {
6
7
             self.property = property
        }
8
9
    }
    let selectorForMethod = #selector(SomeClass.doSomething(_:))
10
    let selectorForPropertyGetter = #selector(getter:
11
          SomeClass.property)
```

When creating a selector for a property's getter, the *property name* can be a reference to a variable or constant property. In contrast, when creating a selector for a property's setter, the *property name* must be a reference to a variable property only.

The *method name* can contain parentheses for grouping, as well the as operator to disambiguate between methods that share a name but have different type signatures. For example:

```
1  extension SomeClass {
2    @objc(doSomethingWithString:)
3    func doSomething(_ x: String) { }
4  }
```

Because a selector is created at compile time, not at runtime, the compiler can check that a method or property exists and that they're exposed to the Objective-C runtime.

```
NOTE

Although the method name and the property name are expressions, they're never evaluated.
```

For more information about using selectors in Swift code that interacts with Objective-C APIs, see <u>Using Objective-C Runtime Features in Swift</u>.

```
GRAMMAR OF A SELECTOR EXPRESSION

selector-expression → #selector ( expression )

selector-expression → #selector ( getter: expression )

selector-expression → #selector ( setter: expression )
```

Key-Path String Expression

A key-path string expression lets you access the string used to refer to a property in Objective-C, for use in key-value coding and key-value observing APIs. It has the following form:

```
#keyPath( property name )
```

The *property name* must be a reference to a property that is available in the Objective-C runtime. At compile time, the key-path string expression is replaced by a string literal. For example:

```
class SomeClass: NSObject {
    @objc var someProperty: Int
    init(someProperty: Int) {
        self.someProperty = someProperty
```

```
5
         }
6
    }
7
    let c = SomeClass(someProperty: 12)
8
    let keyPath = #keyPath(SomeClass.someProperty)
9
10
    if let value = c.value(forKey: keyPath) {
11
12
         print(value)
13
    }
    // Prints "12"
14
```

When you use a key-path string expression within a class, you can refer to a property of that class by writing just the property name, without the class name.

```
1  extension SomeClass {
2    func getSomeKeyPath() -> String {
3       return #keyPath(someProperty)
4    }
5  }
6  print(keyPath == c.getSomeKeyPath())
7  // Prints "true"
```

Because the key path string is created at compile time, not at runtime, the compiler can check that the property exists and that the property is exposed to the Objective-C runtime.

For more information about using key paths in Swift code that interacts with Objective-C APIs, see <u>Using Objective-C Runtime Features in Swift</u>. For information about key-value coding and key-value observing, see <u>Key-Value Coding Programming Guide</u> and <u>Key-Value Observing Programming Guide</u>.

NOTE

Although the *property name* is an expression, it is never evaluated.

```
GRAMMAR OF A KEY-PATH STRING EXPRESSION

key-path-string-expression → #keyPath ( expression )
```

Postfix Expressions

Postfix expressions are formed by applying a postfix operator or other postfix syntax to an expression. Syntactically, every primary expression is also a postfix expression.

For information about the behavior of these operators, see <u>Basic Operators</u> and <u>Advanced Operators</u>.

For information about the operators provided by the Swift standard library, see Operator Declarations.

```
GRAMMAR OF A POSTFIX EXPRESSION

postfix-expression → primary-expression

postfix-expression → postfix-expression

postfix-expression → function-call-expression

postfix-expression → initializer-expression

postfix-expression → explicit-member-expression

postfix-expression → postfix-expression

postfix-expression → subscript-expression

postfix-expression → forced-value-expression

postfix-expression → optional-chaining-expression
```

Function Call Expression

A *function call expression* consists of a function name followed by a commaseparated list of the function's arguments in parentheses. Function call expressions have the following form:

```
function name (argument value 1), (argument value 2)
```

The *function name* can be any expression whose value is of a function type.

If the function definition includes names for its parameters, the function call must include names before its argument values separated by a colon (:). This kind of function call expression has the following form:

A function call expression can include a trailing closure in the form of a closure expression immediately after the closing parenthesis. The trailing closure is understood as an argument to the function, added after the last parenthesized argument. The following function calls are equivalent:

```
// someFunction takes an integer and a closure as its arguments
someFunction(x: x, f: {$0 == 13})
someFunction(x: x) {$0 == 13}
```

If the trailing closure is the function's only argument, the parentheses can be omitted.

```
// someMethod takes a closure as its only argument
myData.someMethod() {$0 == 13}
myData.someMethod {$0 == 13}
```

```
GRAMMAR OF A FUNCTION CALL EXPRESSION

function-call-expression → postfix-expression function-call-argument-clause
function-call-expression → postfix-expression function-call-argument-clause opt trailing-
closure
function-call-argument-clause → ( ) | ( function-call-argument-list )
function-call-argument-list → function-call-argument | function-call-argument , function-
call-argument-list
function-call-argument → expression | identifier : expression
function-call-argument → operator | identifier : operator
trailing-closure → closure-expression
```

Initializer Expression

An *initializer expression* provides access to a type's initializer. It has the following form:

```
expression .init( initializer arguments )
```

You use the initializer expression in a function call expression to initialize a new instance of a type. You also use an initializer expression to delegate to the initializer of a superclass.

```
class SomeSubClass: SomeSuperClass {
   override init() {
        // subclass initialization goes here
        super.init()
}
```

Like a function, an initializer can be used as a value. For example:

If you specify a type by name, you can access the type's initializer without using an initializer expression. In all other cases, you must use an initializer expression.

```
1 let s1 = SomeType.init(data: 3) // Valid
2 let s2 = SomeType(data: 1) // Also valid
3
4 let s3 = type(of: someValue).init(data: 7) // Valid
```

```
let s4 = type(of: someValue)(data: 5) // Error

GRAMMAR OF AN INITIALIZER EXPRESSION

initializer-expression → postfix-expression . init
```

initializer-expression → postfix-expression . init (argument-names)

Explicit Member Expression

An *explicit member expression* allows access to the members of a named type, a tuple, or a module. It consists of a period () between the item and the identifier of its member.

```
expression . member name
```

The members of a named type are named as part of the type's declaration or extension. For example:

```
class SomeClass {
  var someProperty = 42
}

let c = SomeClass()

let y = c.someProperty // Member access
```

The members of a tuple are implicitly named using integers in the order they appear, starting from zero. For example:

```
var t = (10, 20, 30)
t.0 = t.1
// Now t is (20, 20, 30)
```

The members of a module access the top-level declarations of that module.

Types declared with the dynamicMemberLookup attribute include members that are looked up at runtime, as described in <u>Attributes</u>.

To distinguish between methods or initializers whose names differ only by the names of their arguments, include the argument names in parentheses, with each argument name followed by a colon (:). Write an underscore (_) for an argument with no name. To distinguish between overloaded methods, use a type annotation. For example:

```
class SomeClass {
1
        func someMethod(x: Int, y: Int) {}
2
        func someMethod(x: Int, z: Int) {}
3
        func overloadedMethod(x: Int, y: Int) {}
        func overloadedMethod(x: Int, y: Bool) {}
5
    }
6
7
    let instance = SomeClass()
8
    let a = instance.someMethod
                                              // Ambiguous
    let b = instance.someMethod(x:y:)
                                              // Unambiguous
10
11
    let d = instance.overloadedMethod
12
                                              // Ambiguous
    let d = instance.overloadedMethod(x:y:) // Still ambiguous
13
14
    let d: (Int, Bool) -> Void = instance.overloadedMethod(x:y:)
          // Unambiguous
```

If a period appears at the beginning of a line, it is understood as part of an explicit member expression, not as an implicit member expression. For example, the following listing shows chained method calls split over several lines:

```
explicit-member-expression → postfix-expression · decimal-digits
explicit-member-expression → postfix-expression · identifier generic-argument-clause
opt
explicit-member-expression → postfix-expression · identifier ( argument-names )
argument-names → argument-name argument-names opt
argument-name → identifier :
```

Postfix Self Expression

A postfix self expression consists of an expression or the name of a type, immediately followed by .self. It has the following forms:

```
expression self
```

The first form evaluates to the value of the *expression*. For example, \times . self evaluates to \times .

The second form evaluates to the value of the *type*. Use this form to access a type as a value. For example, because <code>SomeClass.self</code> evaluates to the <code>SomeClass</code> type itself, you can pass it to a function or method that accepts a type-level argument.

```
GRAMMAR OF A POSTFIX SELF EXPRESSION

postfix-self-expression → postfix-expression . self
```

Subscript Expression

A *subscript expression* provides subscript access using the getter and setter of the corresponding subscript declaration. It has the following form:

```
expression [ index expressions ]
```

To evaluate the value of a subscript expression, the subscript getter for the *expression*'s type is called with the *index expressions* passed as the subscript parameters. To set its value, the subscript setter is called in the same way.

For information about subscript declarations, see **Protocol Subscript Declaration**.

```
GRAMMAR OF A SUBSCRIPT EXPRESSION

subscript-expression → postfix-expression [ function-call-argument-list ]
```

Forced-Value Expression

A *forced-value expression* unwraps an optional value that you are certain is not nil. It has the following form:

```
expression!
```

If the value of the *expression* is not nil, the optional value is unwrapped and returned with the corresponding non-optional type. Otherwise, a runtime error is raised.

The unwrapped value of a forced-value expression can be modified, either by mutating the value itself, or by assigning to one of the value's members. For example:

```
var x: Int? = 0

x! += 1

// x is now 1

var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]

someDictionary["a"]![0] = 100

// someDictionary is now ["a": [100, 2, 3], "b": [10, 20]]
```

```
GRAMMAR OF A FORCED-VALUE EXPRESSION
forced-value-expression → postfix-expression !
```

Optional-Chaining Expression

An *optional-chaining expression* provides a simplified syntax for using optional values in postfix expressions. It has the following form:

```
expression?
```

The postfix ? operator makes an optional-chaining expression from an expression without changing the expression's value.

Optional-chaining expressions must appear within a postfix expression, and they cause the postfix expression to be evaluated in a special way. If the value of the optional-chaining expression is nil, all of the other operations in the postfix expression are ignored and the entire postfix expression evaluates to nil. If the value of the optional-chaining expression is not nil, the value of the optional-chaining expression is unwrapped and used to evaluate the rest of the postfix expression. In either case, the value of the postfix expression is still of an optional type.

If a postfix expression that contains an optional-chaining expression is nested inside other postfix expressions, only the outermost expression returns an optional type. In the example below, when c is not nil, its value is unwrapped and used to evaluate .property, the value of which is used to evaluate .performAction(). The entire expression c?.property.performAction() has a value of an optional type.

```
var c: SomeClass?
var result: Bool? = c?.property.performAction()
```

The following example shows the behavior of the example above without using optional chaining.

```
var result: Bool?

if let unwrappedC = c {
    result = unwrappedC.property.performAction()
}
```

The unwrapped value of an optional-chaining expression can be modified, either

by mutating the value itself, or by assigning to one of the value's members. If the value of the optional-chaining expression is nil, the expression on the right-hand side of the assignment operator is not evaluated. For example:

```
1
    func someFunctionWithSideEffects() -> Int {
        return 42 // No actual side effects.
2
3
    }
    var someDictionary = ["a": [1, 2, 3], "b": [10, 20]]
4
5
    someDictionary["not here"]?[0] = someFunctionWithSideEffects()
6
    // someFunctionWithSideEffects is not evaluated
7
8
    // someDictionary is still ["a": [1, 2, 3], "b": [10, 20]]
9
    someDictionary["a"]?[0] = someFunctionWithSideEffects()
10
11
    // someFunctionWithSideEffects is evaluated and returns 42
    // someDictionary is now ["a": [42, 2, 3], "b": [10, 20]]
12
```

```
GRAMMAR OF AN OPTIONAL-CHAINING EXPRESSION

optional-chaining-expression → postfix-expression ?
```

Statements

In Swift, there are three kinds of statements: simple statements, compiler control statements, and control flow statements. Simple statements are the most common and consist of either an expression or a declaration. Compiler control statements allow the program to change aspects of the compiler's behavior and include a conditional compilation block and a line control statement.

Control flow statements are used to control the flow of execution in a program. There are several types of control flow statements in Swift, including loop statements, branch statements, and control transfer statements. Loop statements allow a block of code to be executed repeatedly, branch statements allow a certain block of code to be executed only when certain conditions are met, and control transfer statements provide a way to alter the order in which code is executed. In addition, Swift provides a do statement to introduce scope, and catch and handle errors, and a defer statement for running cleanup actions just before the current scope exits.

A semicolon (;) can optionally appear after any statement and is used to separate multiple statements if they appear on the same line.

```
Statement → expression ; opt

statement → declaration ; opt

statement → loop-statement ; opt

statement → branch-statement ; opt

statement → labeled-statement ; opt

statement → control-transfer-statement ; opt

statement → defer-statement ; opt

statement → do-statement ; opt

statement → compiler-control-statement

statements → statement statement statement statement
```

Loop Statements

Loop statements allow a block of code to be executed repeatedly, depending on the conditions specified in the loop. Swift has three loop statements: a for-in statement, a while statement, and a repeat-while statement.

Control flow in a loop statement can be changed by a break statement and a continue statement and is discussed in <u>Break Statement</u> and <u>Continue Statement</u> below.

```
GRAMMAR OF A LOOP STATEMENT

loop-statement → for-in-statement
loop-statement → while-statement
loop-statement → repeat-while-statement
```

For-In Statement

A for-in statement allows a block of code to be executed once for each item in a collection (or any type) that conforms to the <u>Sequence</u> protocol.

A for-in statement has the following form:

```
for item in collection {
    statements
}
```

The makeIterator() method is called on the *collection* expression to obtain a value of an iterator type—that is, a type that conforms to the IteratorProtocol protocol. The program begins executing a loop by calling the next() method on the iterator. If the value returned is not nil, it is assigned to the item pattern, the program executes the statements, and then continues execution at the beginning of the loop. Otherwise, the program does not perform assignment or execute the statements, and it is finished executing the for-in statement.

```
GRAMMAR OF A FOR-IN STATEMENT

for-in-statement → for case<sub>opt</sub> pattern in expression where-clause opt code-block
```

While Statement

A while statement allows a block of code to be executed repeatedly, as long as a condition remains true.

A while statement has the following form:

```
while condition {
    statements
}
```

A while statement is executed as follows:

1. The *condition* is evaluated.

If true, execution continues to step 2. If false, the program is finished executing the while statement.

2. The program executes the *statements*, and execution returns to step 1.

Because the value of the *condition* is evaluated before the *statements* are executed, the *statements* in a while statement can be executed zero or more times.

The value of the *condition* must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in Optional Binding.

```
While-statement → while condition-list code-block condition-list → condition | condition | condition | condition | case-condition | optional-binding-condition optional-binding-condition → case pattern initializer optional-binding-condition → let pattern initializer | var pattern initializer
```

Repeat-While Statement

A repeat-while statement allows a block of code to be executed one or more times, as long as a condition remains true.

A repeat-while statement has the following form:

```
repeat {
     statements
} while condition
```

A repeat-while statement is executed as follows:

- 1. The program executes the *statements*, and execution continues to step 2.
- 2. The *condition* is evaluated.

If true, execution returns to step 1. If false, the program is finished executing the repeat-while statement.

Because the value of the *condition* is evaluated after the *statements* are executed, the *statements* in a repeat-while statement are executed at least once.

The value of the *condition* must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in Optional Binding.

```
GRAMMAR OF A REPEAT-WHILE STATEMENT

repeat-while-statement → repeat code-block while expression
```

Branch Statements

Branch statements allow the program to execute certain parts of code depending on the value of one or more conditions. The values of the conditions specified in a branch statement control how the program branches and, therefore, what block of code is executed. Swift has three branch statements: an if statement, a guard statement, and a switch statement.

Control flow in an if statement or a switch statement can be changed by a break statement and is discussed in **Break Statement** below.

```
GRAMMAR OF A BRANCH STATEMENT

branch-statement → <u>if-statement</u>

branch-statement → <u>guard-statement</u>

branch-statement → <u>switch-statement</u>
```

If Statement

An if statement is used for executing code based on the evaluation of one or more conditions.

There are two basic forms of an if statement. In each form, the opening and closing braces are required.

The first form allows code to be executed only when a condition is true and has the following form:

```
if condition {
    statements
}
```

The second form of an if statement provides an additional *else clause* (introduced by the else keyword) and is used for executing one part of code when the condition is true and another part of code when the same condition is false. When a single else clause is present, an if statement has the following form:

```
if condition {
    statements to execute if condition is true
} else {
    statements to execute if condition is false
}
```

The else clause of an if statement can contain another if statement to test more

than one condition. An if statement chained together in this way has the following form:

```
if condition 1 {
    statements to execute if condition 1 is true
} else if condition 2 {
    statements to execute if condition 2 is true
} else {
    statements to execute if both conditions are false
}
```

The value of any condition in an if statement must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in Optional Binding.

```
GRAMMAR OF AN IF STATEMENT

if-statement → if condition-list code-block else-clause opt

else-clause → else code-block | else if-statement
```

Guard Statement

A guard statement is used to transfer program control out of a scope if one or more conditions aren't met.

A guard statement has the following form:

```
guard condition else {
    statements
}
```

The value of any condition in a guard statement must be of type Bool or a type bridged to Bool. The condition can also be an optional binding declaration, as discussed in Optional Binding.

Any constants or variables assigned a value from an optional binding declaration

in a guard statement condition can be used for the rest of the guard statement's enclosing scope.

The else clause of a guard statement is required, and must either call a function with the Never return type or transfer program control outside the guard statement's enclosing scope using one of the following statements:

- return
- break
- continue
- throw

Control transfer statements are discussed in <u>Control Transfer Statements</u> below. For more information on functions with the <u>Never return type</u>, see <u>Functions that Never Return</u>.

```
GRAMMAR OF A GUARD STATEMENT

guard-statement → guard condition-list else code-block
```

Switch Statement

A switch statement allows certain blocks of code to be executed depending on the value of a control expression.

A switch statement has the following form:

```
switch control expression {
case pattern 1:
    statements
case pattern 2 where condition:
    statements
case pattern 3 where condition,
    pattern 4 where condition:
```

```
statements

default:

statements
}
```

The *control expression* of the switch statement is evaluated and then compared with the patterns specified in each case. If a match is found, the program executes the *statements* listed within the scope of that case. The scope of each case can't be empty. As a result, you must include at least one statement following the colon (:) of each case label. Use a single break statement if you don't intend to execute any code in the body of a matched case.

The values of expressions your code can branch on are very flexible. For example, in addition to the values of scalar types, such as integers and characters, your code can branch on the values of any type, including floating-point numbers, strings, tuples, instances of custom classes, and optionals. The value of the *control expression* can even be matched to the value of a case in an enumeration and checked for inclusion in a specified range of values. For examples of how to use these various types of values in switch statements, see <a href="Switch tiple-width-switch-

A switch case can optionally contain a where clause after each pattern. A where clause is introduced by the where keyword followed by an expression, and is used to provide an additional condition before a pattern in a case is considered matched to the *control expression*. If a where clause is present, the *statements* within the relevant case are executed only if the value of the *control expression* matches one of the patterns of the case and the expression of the where clause evaluates to true. For example, a *control expression* matches the case in the example below only if it is a tuple that contains two elements of the same value, such as (1, 1).

```
case let (x, y) where x == y:
```

As the above example shows, patterns in a case can also bind constants using the let keyword (they can also bind variables using the var keyword). These constants (or variables) can then be referenced in a corresponding where clause

and throughout the rest of the code within the scope of the case. If the case contains multiple patterns that match the control expression, all of the patterns must contain the same constant or variable bindings, and each bound variable or constant must have the same type in all of the case's patterns.

A switch statement can also include a default case, introduced by the default keyword. The code within a default case is executed only if no other cases match the control expression. A switch statement can include only one default case, which must appear at the end of the switch statement.

Although the actual execution order of pattern-matching operations, and in particular the evaluation order of patterns in cases, is unspecified, pattern matching in a switch statement behaves as if the evaluation is performed in source order—that is, the order in which they appear in source code. As a result, if multiple cases contain patterns that evaluate to the same value, and thus can match the value of the control expression, the program executes only the code within the first matching case in source order.

Switch Statements Must Be Exhaustive

In Swift, every possible value of the control expression's type must match the value of at least one pattern of a case. When this simply isn't feasible (for example, when the control expression's type is Int), you can include a default case to satisfy the requirement.

Switching Over Future Enumeration Cases

A *nonfrozen enumeration* is a special kind of enumeration that may gain new enumeration cases in the future—even after you compile and ship an app. Switching over a nonfrozen enumeration requires extra consideration. When a library's authors mark an enumeration as nonfrozen, they reserve the right to add new enumeration cases, and any code that interacts with that enumeration *must* be able to handle those future cases without being recompiled. Code that's compiled in library evolution mode, code in the standard library, Swift overlays for Apple frameworks, and C and Objective-C code can declare nonfrozen enumerations. For information about frozen and nonfrozen enumerations, see frozen.

When switching over a nonfrozen enumeration value, you always need to include a default case, even if every case of the enumeration already has a corresponding switch case. You can apply the <code>@unknown</code> attribute to the default case, which indicates that the default case should match only enumeration cases that are added in the future. Swift produces a warning if the default case matches any enumeration case that is known at compiler time. This future warning informs you that the library author added a new case to the enumeration that doesn't have a corresponding switch case.

The following example switches over all three existing cases of the standard library's Mirror. AncestorRepresentation enumeration. If you add additional cases in the future, the compiler generates a warning to indicate that you need to update the switch statement to take the new cases into account.

```
1
    let representation: Mirror.AncestorRepresentation = .generated
    switch representation {
    case customized:
        print("Use the nearest ancestor's implementation.")
4
    case .generated:
5
        print("Generate a default mirror for all ancestor
6
          classes.")
7
    case suppressed:
8
        print("Suppress the representation of all ancestor
          classes.")
    @unknown default:
9
10
        print("Use a representation that was unknown when this
          code was compiled.")
    }
11
    // Prints "Generate a default mirror for all ancestor
12
          classes."
```

Execution Does Not Fall Through Cases Implicitly

After the code within a matched case has finished executing, the program exits from the switch statement. Program execution does not continue or "fall through" to the next case or default case. That said, if you want execution to continue from one case to the next, explicitly include a fallthrough statement, which simply consists of the fallthrough keyword, in the case from which you want execution to continue. For more information about the fallthrough statement, see <u>Fallthrough Statement</u> below.

```
GRAMMAR OF A SWITCH STATEMENT
switch-statement → switch expression { switch-cases opt }
switch-cases - switch-case switch-cases opt
switch-case → <u>case-label</u> <u>statements</u>
switch-case → default-label statements
switch-case → conditional-switch-case
case-label → <u>attributes</u> opt case <u>case-item-list</u>:
case-item-list → pattern where-clause opt | pattern where-clause opt , case-item-list
default-label → <u>attributes</u> <sub>Opt</sub> default:
where-clause → where where-expression
where-expression → expression
conditional-switch-case -> switch-if-directive-clause switch-elseif-directive-clauses opt
    switch-else-directive-clause opt endif-directive
switch-if-directive-clause - if-directive compilation-condition switch-cases opt
switch-elseif-directive-clauses - elseif-directive-clause switch-elseif-directive-clauses ont
switch-elseif-directive-clause -> elseif-directive compilation-condition switch-cases ont
switch-else-directive-clause -> else-directive switch-cases opt
```

Labeled Statement

You can prefix a loop statement, an if statement, a switch statement, or a do statement with a *statement label*, which consists of the name of the label followed immediately by a colon (:). Use statement labels with break and continue statements to be explicit about how you want to change control flow in a loop statement or a switch statement, as discussed in Break Statement and Continue Statement below.

The scope of a labeled statement is the entire statement following the statement label. You can nest labeled statements, but the name of each statement label must

be unique.

For more information and to see examples of how to use statement labels, see <u>Labeled Statements</u> in <u>Control Flow</u>.

```
GRAMMAR OF A LABELED STATEMENT

| labeled-statement → statement-label | loop-statement |
| labeled-statement → statement-label | switch-statement |
| labeled-statement → statement-label | do-statement |
| labeled-statement → statement-label | do-statement |
| statement-label → label-name :
| label-name → identifier
```

Control Transfer Statements

Control transfer statements can change the order in which code in your program is executed by unconditionally transferring program control from one piece of code to another. Swift has five control transfer statements: a break statement, a continue statement, a fallthrough statement, a return statement, and a throw statement.

```
GRAMMAR OF A CONTROL TRANSFER STATEMENT

control-transfer-statement → break-statement
control-transfer-statement → continue-statement
control-transfer-statement → fallthrough-statement
control-transfer-statement → return-statement
control-transfer-statement → throw-statement
```

Break Statement

A break statement ends program execution of a loop, an if statement, or a switch statement. A break statement can consist of only the break keyword, or it can consist of the break keyword followed by the name of a statement label, as shown below.

break

```
break label name
```

When a break statement is followed by the name of a statement label, it ends program execution of the loop, if statement, or switch statement named by that label.

When a break statement is not followed by the name of a statement label, it ends program execution of the switch statement or the innermost enclosing loop statement in which it occurs. You can't use an unlabeled break statement to break out of an if statement.

In both cases, program control is then transferred to the first line of code following the enclosing loop or switch statement, if any.

For examples of how to use a break statement, see <u>Break</u> and <u>Labeled</u> Statements in Control Flow.

```
GRAMMAR OF A BREAK STATEMENT

break-statement → break label-name opt
```

Continue Statement

A continue statement ends program execution of the current iteration of a loop statement but does not stop execution of the loop statement. A continue statement can consist of only the continue keyword, or it can consist of the continue keyword followed by the name of a statement label, as shown below.

```
continue
continue label name
```

When a continue statement is followed by the name of a statement label, it ends program execution of the current iteration of the loop statement named by that label.

When a continue statement is not followed by the name of a statement label, it ends program execution of the current iteration of the innermost enclosing loop

statement in which it occurs.

In both cases, program control is then transferred to the condition of the enclosing loop statement.

In a for statement, the increment expression is still evaluated after the continue statement is executed, because the increment expression is evaluated after the execution of the loop's body.

For examples of how to use a continue statement, see <u>Continue</u> and <u>Labeled</u> <u>Statements in Control Flow</u>.

```
GRAMMAR OF A CONTINUE STATEMENT

continue-statement → continue <u>label-name</u> opt
```

Fallthrough Statement

A fallthrough statement consists of the fallthrough keyword and occurs only in a case block of a switch statement. A fallthrough statement causes program execution to continue from one case in a switch statement to the next case. Program execution continues to the next case even if the patterns of the case label do not match the value of the switch statement's control expression.

A fallthrough statement can appear anywhere inside a switch statement, not just as the last statement of a case block, but it can't be used in the final case block. It also cannot transfer control into a case block whose pattern contains value binding patterns.

For an example of how to use a fallthrough statement in a switch statement, see Control Transfer Statements in Control Flow.

```
GRAMMAR OF A FALLTHROUGH STATEMENT

fallthrough-statement → fallthrough
```

Return Statement

A return statement occurs in the body of a function or method definition and causes program execution to return to the calling function or method. Program execution continues at the point immediately following the function or method call.

A return statement can consist of only the return keyword, or it can consist of the return keyword followed by an expression, as shown below.

```
return expression
```

When a return statement is followed by an expression, the value of the expression is returned to the calling function or method. If the value of the expression does not match the value of the return type declared in the function or method declaration, the expression's value is converted to the return type before it is returned to the calling function or method.

```
NOTE
```

As described in <u>Failable Initializers</u>, a special form of the return statement (return nil) can be used in a failable initializer to indicate initialization failure.

When a return statement is not followed by an expression, it can be used only to return from a function or method that does not return a value (that is, when the return type of the function or method is Void or ()).

```
GRAMMAR OF A RETURN STATEMENT

return-statement → return expression opt
```

Throw Statement

A throw statement occurs in the body of a throwing function or method, or in the body of a closure expression whose type is marked with the throws keyword.

A throw statement causes a program to end execution of the current scope and begin error propagation to its enclosing scope. The error that's thrown continues

to propagate until it's handled by a catch clause of a do statement.

A throw statement consists of the throw keyword followed by an expression, as shown below.

```
throw expression
```

The value of the *expression* must have a type that conforms to the Error protocol.

For an example of how to use a throw statement, see <u>Propagating Errors Using Throwing Functions</u> in <u>Error Handling</u>.

```
GRAMMAR OF A THROW STATEMENT

throw-statement → throw expression
```

Defer Statement

A defer statement is used for executing code just before transferring program control outside of the scope that the defer statement appears in.

A defer statement has the following form:

```
defer {
     statements
}
```

The statements within the defer statement are executed no matter how program control is transferred. This means that a defer statement can be used, for example, to perform manual resource management such as closing file descriptors, and to perform actions that need to happen even if an error is thrown.

If multiple defer statements appear in the same scope, the order they appear is the reverse of the order they are executed. Executing the last defer statement in a given scope first means that statements inside that last defer statement can refer to resources that will be cleaned up by other defer statements.

```
func f() {
    defer { print("First defer") }
    defer { print("Second defer") }
    print("End of function")
}

f()

// Prints "End of function"
// Prints "Second defer"
// Prints "First defer"
```

The statements in the defer statement can't transfer program control outside of the defer statement.

```
GRAMMAR OF A DEFER STATEMENT

defer-statement → defer code-block
```

Do Statement

The do statement is used to introduce a new scope and can optionally contain one or more catch clauses, which contain patterns that match against defined error conditions. Variables and constants declared in the scope of a do statement can be accessed only within that scope.

A do statement in Swift is similar to curly braces ({}) in C used to delimit a code block, and does not incur a performance cost at runtime.

A do statement has the following form:

```
do {
```

```
try expression
    statements
} catch pattern 1 {
    statements
} catch pattern 2 where condition {
    statements
}
```

Like a switch statement, the compiler attempts to infer whether catch clauses are exhaustive. If such a determination can be made, the error is considered handled. Otherwise, the error can propagate out of the containing scope, which means the error must be handled by an enclosing catch clause or the containing function must be declared with throws.

To ensure that an error is handled, use a catch clause with a pattern that matches all errors, such as a wildcard pattern (_). If a catch clause does not specify a pattern, the catch clause matches and binds any error to a local constant named error. For more information about the patterns you can use in a catch clause, see Patterns.

To see an example of how to use a do statement with several catch clauses, see Handling Errors.

```
GRAMMAR OF A DO STATEMENT

do-statement \rightarrow do <u>code-block</u> <u>catch-clauses</u> <sub>opt</sub>

catch-clauses \rightarrow <u>catch-clause</u> <u>catch-clauses</u> <sub>opt</sub>

catch-clause \rightarrow catch <u>pattern</u> <sub>opt</sub> <u>where-clause</u> <sub>opt</sub> <u>code-block</u>
```

Compiler Control Statements

Compiler control statements allow the program to change aspects of the compiler's behavior. Swift has three compiler control statements: a conditional compilation block a line control statement, and a compile-time diagnostic statement.

GRAMMAR OF A COMPILER CONTROL STATEMENT

compiler-control-statement → <u>conditional-compilation-block</u>

compiler-control-statement → <u>line-control-statement</u> compiler-control-statement → <u>diagnostic-statement</u>

Conditional Compilation Block

A conditional compilation block allows code to be conditionally compiled depending on the value of one or more compilation conditions.

Every conditional compilation block begins with the #if compilation directive and ends with the #endif compilation directive. A simple conditional compilation block has the following form:

```
#if compilation condition
statements
#endif
```

Unlike the condition of an if statement, the *compilation condition* is evaluated at compile time. As a result, the *statements* are compiled and executed only if the *compilation condition* evaluates to true at compile time.

The *compilation condition* can include the true and false Boolean literals, an identifier used with the -D command line flag, or any of the platform conditions listed in the table below.

Platform condition

Valid arguments

```
os() macOS, iOS, watchOS, tvOS, Linux

arch() i386, x86_64, arm, arm64

swift() >= or < followed by a version number
```

```
compiler() >= or < followed by a version number

canImport() A module name

targetEnvironment() simulator, macCatalyst</pre>
```

The version number for the <code>swift()</code> and <code>compiler()</code> platform conditions consists of a major number, optional minor number, optional patch number, and so on, with a dot (.) separating each part of the version number. There must not be whitespace between the comparison operator and the version number. The version for <code>compiler()</code> is the compiler version, regardless of the Swift version setting passed to the compiler. The version for <code>swift()</code> is the language version currently being compiled. For example, if you compile your code using the Swift 5 compiler in Swift 4.2 mode, the compiler version is 5 and the language version is 4.2. With those settings, the following code prints all three messages:

```
#if compiler(>=5)
1
    print("Compiled with the Swift 5 compiler or later")
3
    #endif
    #if swift(>=4.2)
4
    print("Compiled in Swift 4.2 mode or later")
5
    #endif
6
    #if compiler(>=5) && swift(<5)</pre>
7
    print("Compiled with the Swift 5 compiler or later in a Swift
          mode earlier than 5")
    #endif
9
10
    // Prints "Compiled with the Swift 5 compiler or later"
11
    // Prints "Compiled in Swift 4.2 mode or later"
12
    // Prints "Compiled with the Swift 5 compiler or later in a
          Swift mode earlier than 5"
```

The argument for the <code>canImport()</code> platform condition is the name of a module that may not be present on all platforms. This condition tests whether it's possible to import the module, but doesn't actually import it. If the module is present, the platform condition returns <code>true</code>; otherwise, it returns <code>false</code>.

The targetEnvironment() platform condition returns true when code is compiled for a simulator; otherwise, it returns false.

NOTE

The arch(arm) platform condition does not return true for ARM 64 devices. The arch(i386) platform condition returns true when code is compiled for the 32–bit iOS simulator.

You can combine compilation conditions using the logical operators &&, ||, and | and use parentheses for grouping. These operators have the same associativity and precedence as the logical operators that are used to combine ordinary Boolean expressions.

Similar to an if statement, you can add multiple conditional branches to test for different compilation conditions. You can add any number of additional branches using #elseif clauses. You can also add a final additional branch using an #else clause. Conditional compilation blocks that contain multiple branches have the following form:

```
#if compilation condition 1
statements to compile if compilation condition 1 is true
#elseif compilation condition 2
statements to compile if compilation condition 2 is true
#else
statements to compile if both compilation conditions are false
#endif
```

NOTE

Each statement in the body of a conditional compilation block is parsed even if it's not compiled. However, there is an exception if the compilation condition includes a Swift() platform condition: The statements are parsed only if the compiler's version of Swift matches what is specified in the

platform condition. This exception ensures that an older compiler doesn't attempt to parse syntax introduced in a newer version of Swift.

```
GRAMMAR OF A CONDITIONAL COMPILATION BLOCK
conditional-compilation-block → <u>if-directive-clause</u> <u>elseif-directive-clauses</u> <sub>Opt</sub> <u>else-</u>
   directive-clause opt endif-directive
if-directive-clause -- if-directive compilation-condition statements opt
elseif-directive-clauses -> elseif-directive-clause elseif-directive-clauses opt
elseif-directive-clause -> elseif-directive compilation-condition statements opt
else-directive-clause -> else-directive statements opt
if-directive → #if
elseif-directive → #elseif
else-directive → #else
endif-directive → #endif
compilation-condition → platform-condition
compilation-condition → <u>identifier</u>
compilation-condition → boolean-literal
compilation-condition → ( compilation-condition )
compilation-condition → ! compilation-condition
compilation-condition → compilation-condition && compilation-condition
compilation-condition → compilation-condition | compilation-condition
platform-condition → os ( <u>operating-system</u> )
platform-condition → arch ( <u>architecture</u> )
platform-condition \rightarrow swift ( >= swift-version ) | swift ( < swift-version )
platform-condition → compiler ( >= <u>swift-version</u> ) | compiler ( < <u>swift-version</u>
    )
platform-condition \rightarrow canImport ( \underline{module}-name )
platform-condition → targetEnvironment ( environment )
operating-system → macOS | iOS | watchOS | tvOS
architecture → i386 | x86_64 | arm | arm64
swift-version -> decimal-digits swift-version-continuation ont
swift-version-continuation → • decimal-digits swift-version-continuation ont
module-name → identifier
environment → simulator
```

Line Control Statement

A line control statement is used to specify a line number and filename that can be different from the line number and filename of the source code being compiled. Use a line control statement to change the source code location used by Swift for diagnostic and debugging purposes.

A line control statement has the following forms:

```
#sourceLocation(file: filename, line: line number)
#sourceLocation()
```

The first form of a line control statement changes the values of the #line and #file literal expressions, beginning with the line of code following the line control statement. The *line number* changes the value of #line and is any integer literal greater than zero. The *filename* changes the value of #file and is a string literal.

The second form of a line control statement, #sourceLocation(), resets the source code location back to the default line numbering and filename.

```
GRAMMAR OF A LINE CONTROL STATEMENT

line-control-statement → #sourceLocation ( file: file-name , line: line-number )

line-control-statement → #sourceLocation ( )

line-number → A decimal integer greater than zero

file-name → static-string-literal
```

Compile-Time Diagnostic Statement

A compile-time diagnostic statement causes the compiler to emit an error or a warning during compilation. A compile-time diagnostic statement has the following forms:

```
#error(" error message")
#warning(" warning message")
```

The first form emits the *error message* as a fatal error and terminates the compilation process. The second form emits the *warning message* as a nonfatal warning and allows compilation to proceed. You write the diagnostic message as a static string literal. Static string literals can't use features like string interpolation or concatenation, but they can use the multiline string literal syntax.

```
GRAMMAR OF A COMPILE-TIME DIAGNOSTIC STATEMENT

diagnostic-statement → #error ( diagnostic-message )

diagnostic-statement → #warning ( diagnostic-message )

diagnostic-message → static-string-literal
```

Availability Condition

An *availability condition* is used as a condition of an if, while, and guard statement to query the availability of APIs at runtime, based on specified platforms arguments.

An availability condition has the following form:

```
if #available( platform name version , ..., *) {
    statements to execute if the APIs are available
} else {
    fallback statements to execute if the APIs are unavailable
}
```

You use an availability condition to execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

The availability condition takes a comma-separated list of platform names and versions. Use ios, macos, watchos, and tvos for the platform names, and include the corresponding version numbers. The * argument is required and specifies that on any other platform, the body of the code block guarded by the availability condition executes on the minimum deployment target specified by your target.

Unlike Boolean conditions, you can't combine availability conditions using

logical operators such as && and ||.

```
GRAMMAR OF AN AVAILABILITY CONDITION

availability-condition → #available ( availability-arguments )

availability-arguments → availability-argument | availability-argument , availability-argument savailability-argument → platform-name platform-version

availability-argument → *

platform-name → iOS | iOSApplicationExtension

platform-name → macOS | macOSApplicationExtension

platform-name → watchOS

platform-version → decimal-digits

platform-version → decimal-digits . decimal-digits

platform-version → decimal-digits . decimal-digits . decimal-digits . decimal-digits . decimal-digits
```

Declarations

A *declaration* introduces a new name or construct into your program. For example, you use declarations to introduce functions and methods, to introduce variables and constants, and to define enumeration, structure, class, and protocol types. You can also use a declaration to extend the behavior of an existing named type and to import symbols into your program that are declared elsewhere.

In Swift, most declarations are also definitions in the sense that they are implemented or initialized at the same time they are declared. That said, because protocols don't implement their members, most protocol members are declarations only. For convenience and because the distinction isn't that important in Swift, the term *declaration* covers both declarations and definitions.

```
GRAMMAR OF A DECLARATION
declaration → import-declaration
declaration → constant-declaration
declaration → variable-declaration
declaration → typealias-declaration
declaration → <u>function-declaration</u>
declaration → enum-declaration
declaration → <u>struct-declaration</u>
declaration → class-declaration
declaration → protocol-declaration
declaration → initializer-declaration
declaration → deinitializer-declaration
declaration → extension-declaration
declaration → subscript-declaration
declaration → operator-declaration
declaration → precedence-group-declaration
declarations - declaration declarations opt
```

Top-Level Code

The top-level code in a Swift source file consists of zero or more statements, declarations, and expressions. By default, variables, constants, and other named declarations that are declared at the top-level of a source file are accessible to

code in every source file that is part of the same module. You can override this default behavior by marking the declaration with an access-level modifier, as described in Access Control Levels.

```
GRAMMAR OF A TOP-LEVEL DECLARATION

top-level-declaration → statements opt
```

Code Blocks

A *code block* is used by a variety of declarations and control structures to group statements together. It has the following form:

```
{
    statements
}
```

The *statements* inside a code block include declarations, expressions, and other kinds of statements and are executed in order of their appearance in source code.

```
GRAMMAR OF A CODE BLOCK

code-block → { statements opt }
```

Import Declaration

An *import declaration* lets you access symbols that are declared outside the current file. The basic form imports the entire module; it consists of the import keyword followed by a module name:

```
import module
```

Providing more detail limits which symbols are imported—you can specify a

specific submodule or a specific declaration within a module or submodule. When this detailed form is used, only the imported symbol (and not the module that declares it) is made available in the current scope.

```
import import kind module. symbol name
import module. submodule
```

```
GRAMMAR OF AN IMPORT DECLARATION

import-declaration → attributes opt import import-kind opt import-path

import-kind → typealias | struct | class | enum | protocol | let | var |

func

import-path → import-path-identifier | import-path-identifier . import-path

import-path-identifier → identifier | operator
```

Constant Declaration

A *constant declaration* introduces a constant named value into your program. Constant declarations are declared using the let keyword and have the following form:

```
let constant name : (type = expression
```

A constant declaration defines an immutable binding between the *constant name* and the value of the initializer *expression*; after the value of a constant is set, it cannot be changed. That said, if a constant is initialized with a class object, the object itself can change, but the binding between the constant name and the object it refers to can't.

When a constant is declared at global scope, it must be initialized with a value. When a constant declaration occurs in the context of a function or method, it can be initialized later, as long as it is guaranteed to have a value set before the first time its value is read. When a constant declaration occurs in the context of a class or structure declaration, it is considered a *constant property*. Constant declarations are not computed properties and therefore do not have getters or setters.

If the *constant name* of a constant declaration is a tuple pattern, the name of each item in the tuple is bound to the corresponding value in the initializer *expression*.

```
let (firstNumber, secondNumber) = (10, 42)
```

In this example, firstNumber is a named constant for the value 10, and secondNumber is a named constant for the value 42. Both constants can now be used independently:

```
print("The first number is \((firstNumber).")

// Prints "The first number is 10."

print("The second number is \((secondNumber)."))

// Prints "The second number is 42."
```

The type annotation (: *type*) is optional in a constant declaration when the type of the *constant name* can be inferred, as described in <u>Type Inference</u>.

To declare a constant type property, mark the declaration with the static declaration modifier. A constant type property of a class is always implicitly final; you can't mark it with the class or final declaration modifier to allow or disallow overriding by subclasses. Type properties are discussed in Type Properties.

For more information about constants and for guidance about when to use them, see <u>Constants and Variables</u> and <u>Stored Properties</u>.

```
GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → <u>attributes</u> opt <u>declaration-modifiers</u> opt <u>let pattern-initializer-list</u>

pattern-initializer-list → <u>pattern-initializer | pattern-initializer | pattern-initializer | pattern-initializer | pattern-initializer | pattern-initializer | opt initializer → <u>expression</u></u>
```

Variable Declaration

A *variable declaration* introduces a variable named value into your program and is declared using the var keyword.

Variable declarations have several forms that declare different kinds of named, mutable values, including stored and computed variables and properties, stored variable and property observers, and static variable properties. The appropriate form to use depends on the scope at which the variable is declared and the kind of variable you intend to declare.

NOTE

You can also declare properties in the context of a protocol declaration, as described in <u>Protocol Property Declaration</u>.

You can override a property in a subclass by marking the subclass's property declaration with the override declaration modifier, as described in <u>Overriding</u>.

Stored Variables and Stored Variable Properties

The following form declares a stored variable or stored variable property:

```
var variable name : type = expression
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class or structure declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, it is referred to as a *stored variable*. When it is declared in the context of a class or structure declaration, it is referred to as a *stored variable property*.

The initializer *expression* can't be present in a protocol declaration, but in all other contexts, the initializer *expression* is optional. That said, if no initializer *expression* is present, the variable declaration must include an explicit type annotation (: *type*).

As with constant declarations, if the *variable name* is a tuple pattern, the name of each item in the tuple is bound to the corresponding value in the initializer *expression*.

As their names suggest, the value of a stored variable or a stored variable property is stored in memory.

Computed Variables and Computed Properties

The following form declares a computed variable or computed property:

```
var variable name : type {
    get {
        statements
    }
    set(setter name) {
        statements
    }
}
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class, structure, enumeration, or extension declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, it is referred to as a *computed variable*. When it is declared in the context of a class, structure, or extension declaration, it is referred to as a *computed property*.

The getter is used to read the value, and the setter is used to write the value. The setter clause is optional, and when only a getter is needed, you can omit both clauses and simply return the requested value directly, as described in Read-Only Computed Properties. But if you provide a setter clause, you must also provide a getter clause.

The *setter name* and enclosing parentheses is optional. If you provide a setter name, it is used as the name of the parameter to the setter. If you do not provide a setter name, the default parameter name to the setter is newValue, as described in Shorthand Setter Declaration.

Unlike stored named values and stored variable properties, the value of a

computed named value or a computed property is not stored in memory.

For more information and to see examples of computed properties, see Computed Properties.

Stored Variable Observers and Property Observers

You can also declare a stored variable or property with willSet and didSet observers. A stored variable or property declared with observers has the following form:

```
var variable name: type = expression {
   willSet(setter name) {
       statements
   }
   didSet(setter name) {
       statements
   }
}
```

You define this form of a variable declaration at global scope, the local scope of a function, or in the context of a class or structure declaration. When a variable declaration of this form is declared at global scope or the local scope of a function, the observers are referred to as *stored variable observers*. When it is declared in the context of a class or structure declaration, the observers are referred to as *property observers*.

You can add property observers to any stored property. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass, as described in Overriding Property
Observers.

The initializer *expression* is optional in the context of a class or structure declaration, but required elsewhere. The *type* annotation is optional when the type can be inferred from the initializer *expression*.

The willset and didset observers provide a way to observe (and to respond appropriately) when the value of a variable or property is being set. The observers are not called when the variable or property is first initialized. Instead, they are called only when the value is set outside of an initialization context.

A willset observer is called just before the value of the variable or property is set. The new value is passed to the willset observer as a constant, and therefore it can't be changed in the implementation of the willset clause. The didset observer is called immediately after the new value is set. In contrast to the willset observer, the old value of the variable or property is passed to the didset observer in case you still need access to it. That said, if you assign a value to a variable or property within its own didset observer clause, that new value that you assign will replace the one that was just set and passed to the willset observer.

The *setter name* and enclosing parentheses in the willset and didset clauses are optional. If you provide setter names, they are used as the parameter names to the willset and didset observers. If you do not provide setter names, the default parameter name to the willset observer is newValue and the default parameter name to the didset observer is oldValue.

The didSet clause is optional when you provide a willSet clause. Likewise, the willSet clause is optional when you provide a didSet clause.

For more information and to see an example of how to use property observers, see <u>Property Observers</u>.

Type Variable Properties

To declare a type variable property, mark the declaration with the static declaration modifier. Classes can mark type computed properties with the class declaration modifier instead to allow subclasses to override the superclass's implementation. Type properties are discussed in Type Properties.

```
GRAMMAR OF A VARIABLE DECLARATION

variable-declaration → variable-declaration-head pattern-initializer-list

variable-declaration → variable-declaration-head variable-name type-annotation code-
```

```
block
variable-declaration → variable-declaration-head variable-name type-annotation getter-
   setter-block
variable-declaration → variable-declaration-head variable-name type-annotation getter-
   setter-keyword-block
variable-declaration → variable-declaration-head variable-name initializer willSet-didSet-
   block
variable-declaration → variable-declaration-head variable-name type-annotation
    initializer opt willSet-didSet-block
variable-declaration-head → attributes opt declaration-modifiers opt var
variable-name → identifier
getter-setter-block → code-block
getter-setter-block → { getter-clause setter-clause opt }
getter-setter-block → { setter-clause getter-clause }
getter-clause → <u>attributes</u> opt <u>mutation-modifier</u> opt get <u>code-block</u>
setter-clause → <u>attributes ont mutation-modifier ont</u> set <u>setter-name ont code-block</u>
setter-name \rightarrow ( identifier )
getter-setter-keyword-block → { getter-keyword-clause ont }
getter-setter-keyword-block → { setter-keyword-clause getter-keyword-clause }
getter-keyword-clause → <u>attributes</u> opt <u>mutation-modifier</u> opt get
setter-keyword-clause -> attributes opt mutation-modifier opt set
willSet-didSet-block → { willSet-clause didSet-clause ont }
willSet-didSet-block → { didSet-clause willSet-clause ont }
willSet-clause → <u>attributes</u> ont willSet <u>setter-name</u> ont <u>code-block</u>
didSet-clause → <u>attributes</u> <sub>ODT</sub> didSet <u>setter-name</u> <sub>ODT</sub> <u>code-block</u>
```

Type Alias Declaration

A *type alias declaration* introduces a named alias of an existing type into your program. Type alias declarations are declared using the typealias keyword and have the following form:

```
typealias name = existing type
```

After a type alias is declared, the aliased *name* can be used instead of the *existing type* everywhere in your program. The *existing type* can be a named type or a compound type. Type aliases do not create new types; they simply allow a name to refer to an existing type.

A type alias declaration can use generic parameters to give a name to an existing generic type. The type alias can provide concrete types for some or all of the generic parameters of the existing type. For example:

```
typealias StringDictionary<Value> = Dictionary<String, Value>

// The following dictionaries have the same type.

var dictionary1: StringDictionary<Int> = [:]

var dictionary2: Dictionary<String, Int> = [:]
```

When a type alias is declared with generic parameters, the constraints on those parameters must match exactly the constraints on the existing type's generic parameters. For example:

Because the type alias and the existing type can be used interchangeably, the type alias can't introduce additional generic constraints.

A type alias can forward an existing type's generic parameters by omitting all generic parameters from the declaration. For example, the Diccionario type alias declared here has the same generic parameters and constraints as Dictionary.

```
typealias Diccionario = Dictionary
```

Inside a protocol declaration, a type alias can give a shorter and more convenient name to a type that is used frequently. For example:

```
protocol Sequence {
    associatedtype Iterator: IteratorProtocol
    typealias Element = Iterator.Element
}
```

Without this type alias, the sum function would have to refer to the associated type as T.Iterator.Element instead of T.Element.

See also Protocol Associated Type Declaration.

```
GRAMMAR OF A TYPE ALIAS DECLARATION

typealias-declaration → attributes opt access-level-modifier opt typealias typealias-
name generic-parameter-clause opt typealias-assignment

typealias-name → identifier

typealias-assignment → = type
```

Function Declaration

A *function declaration* introduces a function or method into your program. A function declared in the context of class, structure, enumeration, or protocol is referred to as a *method*. Function declarations are declared using the func keyword and have the following form:

```
func function name ( parameters ) -> return type {
    statements
}
```

If the function has a return type of Void, the return type can be omitted as follows:

```
func function name (parameters) {
    statements
}
```

The type of each parameter must be included—it can't be inferred. If you write inout in front of a parameter's type, the parameter can be modified inside the scope of the function. In-out parameters are discussed in detail in In-Out Parameters, below.

A function declaration whose *statements* include only a single expression is understood to return the value of that expression.

Functions can return multiple values using a tuple type as the return type of the function.

A function definition can appear inside another function declaration. This kind of function is known as a *nested function*.

A nested function is nonescaping if it captures a value that is guaranteed to never escape—such as an in-out parameter—or passed as a nonescaping function argument. Otherwise, the nested function is an escaping function.

For a discussion of nested functions, see <u>Nested Functions</u>.

Parameter Names

Function parameters are a comma-separated list where each parameter has one of several forms. The order of arguments in a function call must match the order of parameters in the function's declaration. The simplest entry in a parameter list has the following form:

```
parameter name: parameter type
```

A parameter has a name, which is used within the function body, as well as an argument label, which is used when calling the function or method. By default, parameter names are also used as argument labels. For example:

```
func f(x: Int, y: Int) -> Int { return x + y }
f(x: 1, y: 2) // both x and y are labeled
```

You can override the default behavior for argument labels with one of the following forms:

```
argument label parameter name: parameter type
_ parameter name: parameter type
```

A name before the parameter name gives the parameter an explicit argument label, which can be different from the parameter name. The corresponding argument must use the given argument label in function or method calls.

An underscore (_) before a parameter name suppresses the argument label. The corresponding argument must have no label in function or method calls.

```
func repeatGreeting(_ greeting: String, count n: Int) { /*
    Greet n times */ }
repeatGreeting("Hello, world!", count: 2) // count is labeled,
    greeting is not
```

In-Out Parameters

In-out parameters are passed as follows:

- 1. When the function is called, the value of the argument is copied.
- 2. In the body of the function, the copy is modified.
- 3. When the function returns, the copy's value is assigned to the original argument.

This behavior is known as *copy-in copy-out* or *call by value result*. For example, when a computed property or a property with observers is passed as an in-out parameter, its getter is called as part of the function call and its setter is called as part of the function return.

As an optimization, when the argument is a value stored at a physical address in memory, the same memory location is used both inside and outside the function

body. The optimized behavior is known as *call by reference*; it satisfies all of the requirements of the copy-in copy-out model while removing the overhead of copying. Write your code using the model given by copy-in copy-out, without depending on the call-by-reference optimization, so that it behaves correctly with or without the optimization.

Within a function, don't access a value that was passed as an in-out argument, even if the original value is available in the current scope. Accessing the original is a simultaneous access of the value, which violates Swift's memory exclusivity guarantee. For the same reason, you can't pass the same value to multiple in-out parameters.

For more information about memory safety and memory exclusivity, see Memory Safety.

A closure or nested function that captures an in-out parameter must be nonescaping. If you need to capture an in-out parameter without mutating it or to observe changes made by other code, use a capture list to explicitly capture the parameter immutably.

```
func someFunction(a: inout Int) -> () -> Int {
   return { [a] in return a + 1 }
}
```

If you need to capture and mutate an in-out parameter, use an explicit local copy, such as in multithreaded code that ensures all mutation has finished before the function returns.

```
func multithreadedFunction(queue: DispatchQueue, x: inout Int)
{
    // Make a local copy and manually copy it back.
    var localX = x
    defer { x = localX }

    // Operate on localX asynchronously, then wait before
```

```
returning.

queue.async { someMutatingOperation(&localX) }

queue.sync {}

}
```

For more discussion and examples of in-out parameters, see **In-Out Parameters**.

Special Kinds of Parameters

Parameters can be ignored, take a variable number of values, and provide default values using the following forms:

```
_ : parameter type

parameter name : parameter type ...

parameter name : parameter type = default argument value
```

An underscore (_) parameter is explicitly ignored and can't be accessed within the body of the function.

A parameter with a base type name followed immediately by three dots (...) is understood as a variadic parameter. A function can have at most one variadic parameter. A variadic parameter is treated as an array that contains elements of the base type name. For example, the variadic parameter Int... is treated as [Int]. For an example that uses a variadic parameter, see <u>Variadic Parameters</u>.

A parameter with an equals sign (=) and an expression after its type is understood to have a default value of the given expression. The given expression is evaluated when the function is called. If the parameter is omitted when calling the function, the default value is used instead.

Special Kinds of Methods

Methods on an enumeration or a structure that modify self must be marked with the mutating declaration modifier.

Methods that override a superclass method must be marked with the override declaration modifier. It's a compile-time error to override a method without the override modifier or to use the override modifier on a method that doesn't override a superclass method.

Methods associated with a type rather than an instance of a type must be marked with the static declaration modifier for enumerations and structures, or with either the static or class declaration modifier for classes. A class type method marked with the class declaration modifier can be overridden by a subclass implementation; a class type method marked with class final or static can't be overridden.

Throwing Functions and Methods

Functions and methods that can throw an error must be marked with the throws keyword. These functions and methods are known as *throwing functions* and *throwing methods*. They have the following form:

```
func function name (parameters) throws -> return type {
    statements
}
```

Calls to a throwing function or method must be wrapped in a try or try! expression (that is, in the scope of a try or try! operator).

The throws keyword is part of a function's type, and nonthrowing functions are subtypes of throwing functions. As a result, you can use a nonthrowing function in the same places as a throwing one.

You can't overload a function based only on whether the function can throw an error. That said, you can overload a function based on whether a function *parameter* can throw an error.

A throwing method can't override a nonthrowing method, and a throwing method can't satisfy a protocol requirement for a nonthrowing method. That said, a nonthrowing method can override a throwing method, and a nonthrowing method can satisfy a protocol requirement for a throwing method.

Rethrowing Functions and Methods

A function or method can be declared with the rethrows keyword to indicate that it throws an error only if one of its function parameters throws an error. These functions and methods are known as *rethrowing functions* and *rethrowing methods*. Rethrowing functions and methods must have at least one throwing function parameter.

```
func someFunction(callback: () throws -> Void) rethrows {
   try callback()
}
```

A rethrowing function or method can contain a throw statement only inside a catch clause. This lets you call the throwing function inside a do-catch block and handle errors in the catch clause by throwing a different error. In addition, the catch clause must handle only errors thrown by one of the rethrowing function's throwing parameters. For example, the following is invalid because the catch clause would handle the error thrown by alwaysThrows().

```
1
   func alwaysThrows() throws {
2
        throw SomeError.error
   }
3
   func someFunction(callback: () throws -> Void) rethrows {
4
5
        do {
            try callback()
6
            try alwaysThrows() // Invalid, alwaysThrows() isn't a
          throwing parameter
8
        } catch {
```

```
9 throw AnotherError.error
10 }
11 }
```

A throwing method can't override a rethrowing method, and a throwing method can't satisfy a protocol requirement for a rethrowing method. That said, a rethrowing method can override a throwing method, and a rethrowing method can satisfy a protocol requirement for a throwing method.

Functions that Never Return

Swift defines a Never type, which indicates that a function or method doesn't return to its caller. Functions and methods with the Never return type are called *nonreturning*. Nonreturning functions and methods either cause an irrecoverable error or begin a sequence of work that continues indefinitely. This means that code that would otherwise run immediately after the call is never executed. Throwing and rethrowing functions can transfer program control to an appropriate catch block, even when they are nonreturning.

A nonreturning function or method can be called to conclude the else clause of a guard statement, as discussed in <u>Guard Statement</u>.

You can override a nonreturning method, but the new method must preserve its return type and nonreturning behavior.

```
function-declaration → function-head function-name generic-parameter-clause opt

function-signature generic-where-clause opt function-body opt

function-head → attributes opt declaration-modifiers opt function-name

function-name → identifier | operator

function-signature → parameter-clause throwsopt function-result opt

function-signature → parameter-clause rethrows function-result opt

function-result → -> attributes opt type

function-body → code-block

parameter-clause → () | ( parameter-list )

parameter-list → parameter | parameter , parameter-name type-annotation
```

Enumeration Declaration

An *enumeration declaration* introduces a named enumeration type into your program.

Enumeration declarations have two basic forms and are declared using the enum keyword. The body of an enumeration declared using either form contains zero or more values—called *enumeration cases*—and any number of declarations, including computed properties, instance methods, type methods, initializers, type aliases, and even other enumeration, structure, and class declarations. Enumeration declarations can't contain deinitializer or protocol declarations.

Enumeration types can adopt any number of protocols, but can't inherit from classes, structures, or other enumerations.

Unlike classes and structures, enumeration types do not have an implicitly provided default initializer; all initializers must be declared explicitly. Initializers can delegate to other initializers in the enumeration, but the initialization process is complete only after an initializer assigns one of the enumeration cases to self.

Like structures but unlike classes, enumerations are value types; instances of an enumeration are copied when assigned to variables or constants, or when passed as arguments to a function call. For information about value types, see Structures and Enumerations Are Value Types.

You can extend the behavior of an enumeration type with an extension declaration, as discussed in Extension Declaration.

Enumerations with Cases of Any Type

The following form declares an enumeration type that contains enumeration cases of any type:

```
enum enumeration name: adopted protocols {
   case enumeration case 1
   case enumeration case 2 (associated value types)
}
```

Enumerations declared in this form are sometimes called *discriminated unions* in other programming languages.

In this form, each case block consists of the case keyword followed by one or more enumeration cases, separated by commas. The name of each case must be unique. Each case can also specify that it stores values of a given type. These types are specified in the *associated value types* tuple, immediately following the name of the case.

Enumeration cases that store associated values can be used as functions that create instances of the enumeration with the specified associated values. And just like functions, you can get a reference to an enumeration case and apply it later in your code.

```
1
   enum Number {
2
        case integer(Int)
        case real(Double)
3
4
   }
   let f = Number.integer
5
   // f is a function of type (Int) -> Number
6
7
8
   // Apply f to create an array of Number instances with integer
          values
   let evenInts: [Number] = [0, 2, 4, 6].map(f)
```

For more information and to see examples of cases with associated value types, see Associated Values.

Enumerations with Indirection

Enumerations can have a recursive structure, that is, they can have cases with associated values that are instances of the enumeration type itself. However, instances of enumeration types have value semantics, which means they have a fixed layout in memory. To support recursion, the compiler must insert a layer of indirection.

To enable indirection for a particular enumeration case, mark it with the indirect declaration modifier. An indirect case must have an associated value.

```
1 enum Tree<T> {
2    case empty
3    indirect case node(value: T, left: Tree, right: Tree)
4 }
```

To enable indirection for all the cases of an enumeration that have an associated value, mark the entire enumeration with the indirect modifier—this is convenient when the enumeration contains many cases that would each need to be marked with the indirect modifier.

An enumeration that is marked with the indirect modifier can contain a mixture of cases that have associated values and cases those that don't. That said, it can't contain any cases that are also marked with the indirect modifier.

Enumerations with Cases of a Raw-Value Type

The following form declares an enumeration type that contains enumeration cases of the same basic type:

```
enum enumeration name : raw-value type , adopted protocols {
   case enumeration case 1 = raw value 1
```

```
case enumeration case 2 = raw value 2
}
```

In this form, each case block consists of the case keyword, followed by one or more enumeration cases, separated by commas. Unlike the cases in the first form, each case has an underlying value, called a *raw value*, of the same basic type. The type of these values is specified in the *raw-value type* and must represent an integer, floating-point number, string, or single character. In particular, the *raw-value type* must conform to the Equatable protocol and one of the following protocols: ExpressibleByIntegerLiteral for integer literals, ExpressibleByFloatLiteral for floating-point literals, ExpressibleByStringLiteral for string literals that contain any number of characters, and ExpressibleByUnicodeScalarLiteral or ExpressibleByExtendedGraphemeClusterLiteral for string literals that contain only a single character. Each case must have a unique name and be assigned a unique raw value.

If the raw-value type is specified as Int and you don't assign a value to the cases explicitly, they are implicitly assigned the values 0, 1, 2, and so on. Each unassigned case of type Int is implicitly assigned a raw value that is automatically incremented from the raw value of the previous case.

```
1 enum ExampleEnum: Int {
2    case a, b, c = 5, d
3 }
```

In the above example, the raw value of <code>ExampleEnum.a</code> is 0 and the value of <code>ExampleEnum.b</code> is 1. And because the value of <code>ExampleEnum.c</code> is explicitly set to 5, the value of <code>ExampleEnum.d</code> is automatically incremented from 5 and is therefore 6.

If the raw-value type is specified as String and you don't assign values to the cases explicitly, each unassigned case is implicitly assigned a string with the same text as the name of that case.

```
1 enum GamePlayMode: String {
```

```
case cooperative, individual, competitive
}
```

In the above example, the raw value of GamePlayMode.cooperative is "cooperative", the raw value of GamePlayMode.individual is "individual", and the raw value of GamePlayMode.competitive is "competitive".

Enumerations that have cases of a raw-value type implicitly conform to the RawRepresentable protocol, defined in the Swift standard library. As a result, they have a rawValue property and a failable initializer with the signature init? (rawValue: RawValue). You can use the rawValue property to access the raw value of an enumeration case, as in ExampleEnum.b.rawValue. You can also use a raw value to find a corresponding case, if there is one, by calling the enumeration's failable initializer, as in ExampleEnum(rawValue: 5), which returns an optional case. For more information and to see examples of cases with raw-value types, see Raw Values.

Accessing Enumeration Cases

To reference the case of an enumeration type, use dot (.) syntax, as in <code>EnumerationType.enumerationCase</code>. When the enumeration type can be inferred from context, you can omit it (the dot is still required), as described in <code>Enumeration Syntax</code> and <code>Implicit Member Expression</code>.

To check the values of enumeration cases, use a switch statement, as shown in <u>Matching Enumeration Values with a Switch Statement</u>. The enumeration type is pattern-matched against the enumeration case patterns in the case blocks of the switch statement, as described in <u>Enumeration Case Pattern</u>.

```
GRAMMAR OF AN ENUMERATION DECLARATION

enum-declaration \rightarrow attributes opt access-level-modifier opt union-style-enum

enum-declaration \rightarrow attributes opt access-level-modifier opt raw-value-style-enum

union-style-enum \rightarrow indirectopt enum enum-name generic-parameter-clause opt

type-inheritance-clause opt generic-where-clause opt { union-style-enum-members opt }

union-style-enum-members \rightarrow union-style-enum-member union-style-enum-members opt union-style-enum-member \rightarrow declaration | union-style-enum-case-clause | compiler-
```

```
control-statement
union-style-enum-case-clause \rightarrow <u>attributes</u> opt indirect case <u>union-style-enum-</u>
          case-list
union-style-enum-case | union-style-enum-case | union-style-enum-case , union-
          style-enum-case-list
union-style-enum-case → enum-case-name tuple-type opt
enum-name → <u>identifier</u>
enum-case-name → identifier
raw-value-style-enum → enum enum-name generic-parameter-clause opt type-
          inheritance-clause generic-where-clause opt { raw-value-style-enum-members }
raw-value-style-enum-members → raw-value-style-enum-member raw-value-style-enum-
          members opt
raw-value-style-enum-member → <u>declaration</u> | <u>raw-value-style-enum-case-clause</u> |
           compiler-control-statement
raw-value-style-enum-case-clause → attributes opt case raw-value-style-enum-case-
raw-value-style-enum-case | r
          case , raw-value-style-enum-case-list
raw-value-style-enum-case → enum-case-name raw-value-assignment ont
raw-value-assignment → = <u>raw-value-literal</u>
raw-value-literal → <u>numeric-literal</u> | <u>static-string-literal</u> | <u>boolean-literal</u>
```

Structure Declaration

A *structure declaration* introduces a named structure type into your program. Structure declarations are declared using the struct keyword and have the following form:

```
struct structure name: adopted protocols {
    declarations
}
```

The body of a structure contains zero or more *declarations*. These *declarations* can include both stored and computed properties, type properties, instance methods, type methods, initializers, subscripts, type aliases, and even other structure, class, and enumeration declarations. Structure declarations can't contain deinitializer or protocol declarations. For a discussion and several examples of structures that include various kinds of declarations, see <u>Structures</u>

and Classes.

Structure types can adopt any number of protocols, but can't inherit from classes, enumerations, or other structures.

There are three ways to create an instance of a previously declared structure:

- Call one of the initializers declared within the structure, as described in <u>Initializers</u>.
- If no initializers are declared, call the structure's memberwise initializer, as described in <u>Memberwise Initializers for Structure Types</u>.
- If no initializers are declared, and all properties of the structure declaration were given initial values, call the structure's default initializer, as described in Default Initializers.

The process of initializing a structure's declared properties is described in **Initialization**.

Properties of a structure instance can be accessed using dot (.) syntax, as described in <u>Accessing Properties</u>.

Structures are value types; instances of a structure are copied when assigned to variables or constants, or when passed as arguments to a function call. For information about value types, see Structures and Enumerations Are Value Types.

You can extend the behavior of a structure type with an extension declaration, as discussed in Extension Declaration.

```
\begin{array}{lll} \textit{Struct-declaration} & \rightarrow & \underline{\textit{attributes}} \ \textit{opt} & \underline{\textit{access-level-modifier}} \ \textit{opt} & \underline{\textit{struct-name}} \ \\ & & \underline{\textit{generic-parameter-clause}} \ \textit{opt} & \underline{\textit{type-inheritance-clause}} \ \textit{opt} & \underline{\textit{generic-where-clause}} \ \textit{opt} \ \\ & \underline{\textit{struct-body}} \ \\ & \underline{\textit{struct-name}} \ \rightarrow & \underline{\textit{identifier}} \ \\ & \underline{\textit{struct-members}} \ \textit{opt} \ \\ & \underline{\textit{struct-members}} \ \textit{opt} \ \\ & \underline{\textit{struct-members}} \ \rightarrow & \underline{\textit{struct-members}} \ \textit{opt} \ \\ & \underline{\textit{struct-members}} \ \rightarrow & \underline{\textit{struct-members}} \ \textit{opt} \ \\ & \underline{\textit{struct-members}} \ \rightarrow & \underline{\textit{declaration}} \ \mid & \underline{\textit{compiler-control-statement}} \ \end{array}
```

Class Declaration

A *class declaration* introduces a named class type into your program. Class declarations are declared using the class keyword and have the following form:

```
class class name : superclass , adopted protocols {
    declarations
}
```

The body of a class contains zero or more *declarations*. These *declarations* can include both stored and computed properties, instance methods, type methods, initializers, a single deinitializer, subscripts, type aliases, and even other class, structure, and enumeration declarations. Class declarations can't contain protocol declarations. For a discussion and several examples of classes that include various kinds of declarations, see <u>Structures and Classes</u>.

A class type can inherit from only one parent class, its *superclass*, but can adopt any number of protocols. The *superclass* appears first after the *class name* and colon, followed by any *adopted protocols*. Generic classes can inherit from other generic and nongeneric classes, but a nongeneric class can inherit only from other nongeneric classes. When you write the name of a generic superclass class after the colon, you must include the full name of that generic class, including its generic parameter clause.

As discussed in <u>Initializer Declaration</u>, classes can have designated and convenience initializers. The designated initializer of a class must initialize all of the class's declared properties and it must do so before calling any of its superclass's designated initializers.

A class can override properties, methods, subscripts, and initializers of its superclass. Overridden properties, methods, subscripts, and designated initializers must be marked with the override declaration modifier.

To require that subclasses implement a superclass's initializer, mark the superclass's initializer with the required declaration modifier. The subclass's implementation of that initializer must also be marked with the required declaration modifier.

Although properties and methods declared in the *superclass* are inherited by the current class, designated initializers declared in the *superclass* are only inherited when the subclass meets the conditions described in <u>Automatic Initializer</u> <u>Inheritance</u>. Swift classes do not inherit from a universal base class.

There are two ways to create an instance of a previously declared class:

- Call one of the initializers declared within the class, as described in Initializers.
- If no initializers are declared, and all properties of the class declaration were given initial values, call the class's default initializer, as described in Default Initializers.

Access properties of a class instance with dot (.) syntax, as described in Accessing Properties.

Classes are reference types; instances of a class are referred to, rather than copied, when assigned to variables or constants, or when passed as arguments to a function call. For information about reference types, see Structures and Enumerations Are Value Types.

You can extend the behavior of a class type with an extension declaration, as discussed in Extension Declaration.

```
class-declaration → attributes opt access-level-modifier opt finalopt class class-
name generic-parameter-clause opt type-inheritance-clause opt generic-where-clause
opt class-body
class-declaration → attributes opt final access-level-modifier opt class class-
name generic-parameter-clause opt type-inheritance-clause opt generic-where-clause
opt class-body
class-name → identifier
class-body → { class-members opt }
class-members → class-member class-members opt
class-member → declaration | compiler-control-statement
```

Protocol Declaration

A *protocol declaration* introduces a named protocol type into your program. Protocol declarations are declared at global scope using the protocol keyword and have the following form:

```
protocol protocol name: inherited protocols {
    protocol member declarations
}
```

The body of a protocol contains zero or more *protocol member declarations*, which describe the conformance requirements that any type adopting the protocol must fulfill. In particular, a protocol can declare that conforming types must implement certain properties, methods, initializers, and subscripts. Protocols can also declare special kinds of type aliases, called *associated types*, that can specify relationships among the various declarations of the protocol. Protocol declarations can't contain class, structure, enumeration, or other protocol declarations. The *protocol member declarations* are discussed in detail below.

Protocol types can inherit from any number of other protocols. When a protocol type inherits from other protocols, the set of requirements from those other protocols are aggregated, and any type that inherits from the current protocol must conform to all those requirements. For an example of how to use protocol inheritance, see Protocol Inheritance.

```
NOTE
```

You can also aggregate the conformance requirements of multiple protocols using protocol composition types, as described in <u>Protocol Composition Type</u> and <u>Protocol Composition</u>.

You can add protocol conformance to a previously declared type by adopting the protocol in an extension declaration of that type. In the extension, you must implement all of the adopted protocol's requirements. If the type already implements all of the requirements, you can leave the body of the extension declaration empty.

By default, types that conform to a protocol must implement all properties,

methods, and subscripts declared in the protocol. That said, you can mark these protocol member declarations with the <code>optional</code> declaration modifier to specify that their implementation by a conforming type is optional. The <code>optional</code> modifier can be applied only to members that are marked with the <code>objc</code> attribute, and only to members of protocols that are marked with the <code>objc</code> attribute. As a result, only class types can adopt and conform to a protocol that contains optional member requirements. For more information about how to use the <code>optional</code> declaration modifier and for guidance about how to access optional protocol members—for example, when you're not sure whether a conforming type implements them—see <code>Optional Protocol Requirements</code>.

To restrict the adoption of a protocol to class types only, include the AnyObject protocol in the *inherited protocols* list after the colon. For example, the following protocol can be adopted only by class types:

```
protocol SomeProtocol: AnyObject {
    /* Protocol members go here */
}
```

Any protocol that inherits from a protocol that's marked with the AnyObject requirement can likewise be adopted only by class types.

```
NOTE
```

If a protocol is marked with the <code>objc</code> attribute, the <code>AnyObject</code> requirement is implicitly applied to that protocol; there's no need to mark the protocol with the <code>AnyObject</code> requirement explicitly.

Protocols are named types, and thus they can appear in all the same places in your code as other named types, as discussed in <u>Protocols as Types</u>. However, you can't construct an instance of a protocol, because protocols do not actually provide the implementations for the requirements they specify.

You can use protocols to declare which methods a delegate of a class or structure should implement, as described in <u>Delegation</u>.

```
GRAMMAR OF A PROTOCOL DECLARATION

protocol-declaration → <u>attributes</u> opt <u>access-level-modifier</u> opt protocol <u>protocol-</u>
```

```
name type-inheritance-clause opt generic-where-clause opt protocol-body protocol-name → identifier protocol-body → { protocol-members opt } protocol-members → protocol-member protocol-members opt protocol-member → protocol-member-declaration | compiler-control-statement protocol-member-declaration → protocol-property-declaration protocol-member-declaration → protocol-method-declaration protocol-member-declaration → protocol-initializer-declaration protocol-member-declaration → protocol-subscript-declaration protocol-member-declaration → protocol-associated-type-declaration protocol-member-declaration → typealias-declaration
```

Protocol Property Declaration

Protocols declare that conforming types must implement a property by including a *protocol property declaration* in the body of the protocol declaration. Protocol property declarations have a special form of a variable declaration:

```
var property name : type { get set }
```

As with other protocol member declarations, these property declarations declare only the getter and setter requirements for types that conform to the protocol. As a result, you don't implement the getter or setter directly in the protocol in which it is declared.

The getter and setter requirements can be satisfied by a conforming type in a variety of ways. If a property declaration includes both the get and set keywords, a conforming type can implement it with a stored variable property or a computed property that is both readable and writeable (that is, one that implements both a getter and a setter). However, that property declaration can't be implemented as a constant property or a read-only computed property. If a property declaration includes only the get keyword, it can be implemented as any kind of property. For examples of conforming types that implement the property requirements of a protocol, see Property Requirements.

To declare a type property requirement in a protocol declaration, mark the property declaration with the static keyword. Structures and enumerations that conform to the protocol declare the property with the static keyword, and

classes that conform to the protocol declare the property with either the static or class keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the same keyword as the type they extend uses. Extensions that provide a default implementation for a type property requirement use the static keyword.

See also Variable Declaration.

GRAMMAR OF A PROTOCOL PROPERTY DECLARATION

protocol-property-declaration → variable-declaration-head variable-name type-annotation getter-setter-keyword-block

Protocol Method Declaration

Protocols declare that conforming types must implement a method by including a protocol method declaration in the body of the protocol declaration. Protocol method declarations have the same form as function declarations, with two exceptions: They don't include a function body, and you can't provide any default parameter values as part of the function declaration. For examples of conforming types that implement the method requirements of a protocol, see Method Requirements.

To declare a class or static method requirement in a protocol declaration, mark the method declaration with the static declaration modifier. Structures and enumerations that conform to the protocol declare the method with the static keyword, and classes that conform to the protocol declare the method with either the static or class keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the same keyword as the type they extend uses. Extensions that provide a default implementation for a type method requirement use the static keyword.

See also Function Declaration.

GRAMMAR OF A PROTOCOL METHOD DECLARATION

 $\begin{array}{rcl} \textit{protocol-method-declaration} & \rightarrow & \underline{\textit{function-head}} & \underline{\textit{function-name}} & \underline{\textit{generic-parameter-clause}} \\ & \textit{opt} & \underline{\textit{function-signature}} & \underline{\textit{generic-where-clause}} & \textit{opt} \\ \end{array}$

Protocol Initializer Declaration

Protocols declare that conforming types must implement an initializer by including a protocol initializer declaration in the body of the protocol declaration. Protocol initializer declarations have the same form as initializer declarations, except they don't include the initializer's body.

A conforming type can satisfy a nonfailable protocol initializer requirement by implementing a nonfailable initializer or an init! failable initializer. A conforming type can satisfy a failable protocol initializer requirement by implementing any kind of initializer.

When a class implements an initializer to satisfy a protocol's initializer requirement, the initializer must be marked with the required declaration modifier if the class is not already marked with the final declaration modifier.

See also Initializer Declaration.

```
GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION

protocol-initializer-declaration → initializer-head generic-parameter-clause opt

parameter-clause throws<sub>opt</sub> generic-where-clause opt

protocol-initializer-declaration → initializer-head generic-parameter-clause opt

parameter-clause rethrows generic-where-clause opt
```

Protocol Subscript Declaration

Protocols declare that conforming types must implement a subscript by including a protocol subscript declaration in the body of the protocol declaration. Protocol subscript declarations have a special form of a subscript declaration:

```
subscript ( parameters ) -> return type { get set }
```

Subscript declarations only declare the minimum getter and setter implementation requirements for types that conform to the protocol. If the subscript declaration includes both the get and set keywords, a conforming type must implement both a getter and a setter clause. If the subscript declaration includes only the get keyword, a conforming type must implement *at least* a

getter clause and optionally can implement a setter clause.

To declare a static subscript requirement in a protocol declaration, mark the subscript declaration with the static declaration modifier. Structures and enumerations that conform to the protocol declare the subscript with the static keyword, and classes that conform to the protocol declare the subscript with either the static or class keyword. Extensions that add protocol conformance to a structure, enumeration, or class use the same keyword as the type they extend uses. Extensions that provide a default implementation for a static subscript requirement use the static keyword.

See also **Subscript Declaration**.

```
GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION

protocol-subscript-declaration → subscript-head subscript-result generic-where-clause opt getter-setter-keyword-block
```

Protocol Associated Type Declaration

Protocols declare associated types using the associated type keyword. An associated type provides an alias for a type that is used as part of a protocol's declaration. Associated types are similar to type parameters in generic parameter clauses, but they're associated with Self in the protocol in which they're declared. In that context, Self refers to the eventual type that conforms to the protocol. For more information and examples, see <u>Associated Types</u>.

You use a generic where clause in a protocol declaration to add constraints to an associated types inherited from another protocol, without redeclaring the associated types. For example, the declarations of SubProtocol below are equivalent:

```
protocol SomeProtocol {
    associatedtype SomeType
}
```

```
5  protocol SubProtocolA: SomeProtocol {
6     // This syntax produces a warning.
7     associatedtype SomeType: Equatable
8  }
9
10  // This syntax is preferred.
11  protocol SubProtocolB: SomeProtocol where SomeType: Equatable
     { }
```

See also **Type Alias Declaration**.

```
GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION

protocol-associated-type-declaration → <u>attributes</u> opt <u>access-level-modifier</u> opt

associatedtype <u>typealias-name</u> <u>type-inheritance-clause</u> opt <u>typealias-assignment</u>

opt <u>generic-where-clause</u> opt
```

Initializer Declaration

An *initializer declaration* introduces an initializer for a class, structure, or enumeration into your program. Initializer declarations are declared using the init keyword and have two basic forms.

Structure, enumeration, and class types can have any number of initializers, but the rules and associated behavior for class initializers are different. Unlike structures and enumerations, classes have two kinds of initializers: designated initializers and convenience initializers, as described in <u>Initialization</u>.

The following form declares initializers for structures, enumerations, and designated initializers of classes:

```
init( parameters ) {
    statements
}
```

A designated initializer of a class initializes all of the class's properties directly. It can't call any other initializers of the same class, and if the class has a superclass, it must call one of the superclass's designated initializers. If the class inherits any properties from its superclass, one of the superclass's designated initializers must be called before any of these properties can be set or modified in the current class.

Designated initializers can be declared in the context of a class declaration only and therefore can't be added to a class using an extension declaration.

Initializers in structures and enumerations can call other declared initializers to delegate part or all of the initialization process.

To declare convenience initializers for a class, mark the initializer declaration with the convenience declaration modifier.

```
convenience init( parameters ) {
    statements
}
```

Convenience initializers can delegate the initialization process to another convenience initializer or to one of the class's designated initializers. That said, the initialization processes must end with a call to a designated initializer that ultimately initializes the class's properties. Convenience initializers can't call a superclass's initializers.

You can mark designated and convenience initializers with the required declaration modifier to require that every subclass implement the initializer. A subclass's implementation of that initializer must also be marked with the required declaration modifier.

By default, initializers declared in a superclass are not inherited by subclasses. That said, if a subclass initializes all of its stored properties with default values and doesn't define any initializers of its own, it inherits all of the superclass's initializers. If the subclass overrides all of the superclass's designated initializers, it inherits the superclass's convenience initializers.

As with methods, properties, and subscripts, you need to mark overridden

designated initializers with the override declaration modifier.

```
NOTE
```

If you mark an initializer with the required declaration modifier, you don't also mark the initializer with the override modifier when you override the required initializer in a subclass.

Just like functions and methods, initializers can throw or rethrow errors. And just like functions and methods, you use the throws or rethrows keyword after an initializer's parameters to indicate the appropriate behavior.

To see examples of initializers in various type declarations, see <u>Initialization</u>.

Failable Initializers

A *failable initializer* is a type of initializer that produces an optional instance or an implicitly unwrapped optional instance of the type the initializer is declared on. As a result, a failable initializer can return nil to indicate that initialization failed.

To declare a failable initializer that produces an optional instance, append a question mark to the init keyword in the initializer declaration (init?). To declare a failable initializer that produces an implicitly unwrapped optional instance, append an exclamation mark instead (init!). The example below shows an init? failable initializer that produces an optional instance of a structure.

```
struct SomeStruct {
1
2
        let property: String
        // produces an optional instance of 'SomeStruct'
3
        init?(input: String) {
4
            if input isEmpty {
5
                // discard 'self' and return 'nil'
6
                return nil
7
            }
8
```

You call an init? failable initializer in the same way that you call a nonfailable initializer, except that you must deal with the optionality of the result.

```
if let actualInstance = SomeStruct(input: "Hello") {
    // do something with the instance of 'SomeStruct'
} else {
    // initialization of 'SomeStruct' failed and the initializer returned 'nil'
}
```

A failable initializer can return nil at any point in the implementation of the initializer's body.

A failable initializer can delegate to any kind of initializer. A nonfailable initializer can delegate to another nonfailable initializer or to an init! failable initializer. A nonfailable initializer can delegate to an init? failable initializer by force-unwrapping the result of the superclass's initializer—for example, by writing super.init()!.

Initialization failure propagates through initializer delegation. Specifically, if a failable initializer delegates to an initializer that fails and returns nil, then the initializer that delegated also fails and implicitly returns nil. If a nonfailable initializer delegates to an init! failable initializer that fails and returns nil, then a runtime error is raised (as if you used the! operator to unwrap an optional that has a nil value).

A failable designated initializer can be overridden in a subclass by any kind of designated initializer. A nonfailable designated initializer can be overridden in a subclass by a nonfailable designated initializer only.

For more information and to see examples of failable initializers, see Failable

<u>Initializers</u>.

```
initializer-declaration → initializer-head generic-parameter-clause opt parameter-clause

throws<sub>opt</sub> generic-where-clause opt initializer-body

initializer-declaration → initializer-head generic-parameter-clause opt parameter-clause

rethrows generic-where-clause opt initializer-body

initializer-head → attributes opt declaration-modifiers opt init

initializer-head → attributes opt declaration-modifiers opt init ?

initializer-head → attributes opt declaration-modifiers opt init !

initializer-body → code-block
```

Deinitializer Declaration

A *deinitializer declaration* declares a deinitializer for a class type. Deinitializers take no parameters and have the following form:

```
deinit {
     statements
}
```

A deinitializer is called automatically when there are no longer any references to a class object, just before the class object is deallocated. A deinitializer can be declared only in the body of a class declaration—but not in an extension of a class—and each class can have at most one.

A subclass inherits its superclass's deinitializer, which is implicitly called just before the subclass object is deallocated. The subclass object is not deallocated until all deinitializers in its inheritance chain have finished executing.

Deinitializers are not called directly.

For an example of how to use a deinitializer in a class declaration, see Deinitialization.

Extension Declaration

An *extension declaration* allows you to extend the behavior of existing types. Extension declarations are declared using the extension keyword and have the following form:

```
extension type name where requirements {
    declarations
}
```

The body of an extension declaration contains zero or more *declarations*. These *declarations* can include computed properties, computed type properties, instance methods, type methods, initializers, subscript declarations, and even class, structure, and enumeration declarations. Extension declarations can't contain deinitializer or protocol declarations, stored properties, property observers, or other extension declarations. Declarations in a protocol extension can't be marked final. For a discussion and several examples of extensions that include various kinds of declarations, see Extensions.

If the *type name* is a class, structure, or enumeration type, the extension extends that type. If the *type name* is a protocol type, the extension extends all types that conform to that protocol.

Extension declarations that extend a generic type or a protocol with associated types can include *requirements*. If an instance of the extended type or of a type that conforms to the extended protocol satisfies the *requirements*, the instance gains the behavior specified in the declaration.

Extension declarations can contain initializer declarations. That said, if the type you're extending is defined in another module, an initializer declaration must delegate to an initializer already defined in that module to ensure members of that type are properly initialized.

Properties, methods, and initializers of an existing type can't be overridden in an extension of that type.

Extension declarations can add protocol conformance to an existing class, structure, or enumeration type by specifying *adopted protocols*:

```
extension type name: adopted protocols where requirements
{
    declarations
}
```

Extension declarations can't add class inheritance to an existing class, and therefore you can specify only a list of protocols after the *type name* and colon.

Conditional Conformance

You can extend a generic type to conditionally conform to a protocol, so that instances of the type conform to the protocol only when certain requirements are met. You add conditional conformance to a protocol by including *requirements* in an extension declaration.

Overridden Requirements Aren't Used in Some Generic Contexts

In some generic contexts, types that get behavior from conditional conformance to a protocol don't always use the specialized implementations of that protocol's requirements. To illustrate this behavior, the following example defines two protocols and a generic type that conditionally conforms to both protocols.

```
protocol Loggable {
  func log()
}

extension Loggable {
  func log() {
    print(self)
```

```
7
        }
8
    }
9
    protocol TitledLoggable: Loggable {
10
11
         static var logTitle: String { get }
12
    }
    extension TitledLoggable {
13
         func log() {
14
             print("\(Self.logTitle): \(self)")
15
        }
16
17
    }
18
19
    struct Pair<T>: CustomStringConvertible {
        let first: T
20
21
        let second: T
22
        var description: String {
23
             return "(\(first), \(second))"
        }
24
25
    }
26
27
    extension Pair: Loggable where T: Loggable { }
     extension Pair: TitledLoggable where T: TitledLoggable {
28
        static var logTitle: String {
29
             return "Pair of '\(T.logTitle)'"
30
31
        }
32
    }
33
    extension String: TitledLoggable {
34
35
         static var logTitle: String {
36
             return "String"
```

```
37 }
38 }
```

The Pair structure conforms to Loggable and TitledLoggable whenever its generic type conforms to Loggable or TitledLoggable, respectively. In the example below, oneAndTwo is an instance of Pair<String>, which conforms to TitledLoggable because String conforms to TitledLoggable. When the log() method is called on oneAndTwo directly, the specialized version containing the title string is used.

```
1 let oneAndTwo = Pair(first: "one", second: "two")
2 oneAndTwo.log()
3 // Prints "Pair of 'String': (one, two)"
```

However, when <code>oneAndTwo</code> is used in a generic context or as an instance of the <code>Loggable</code> protocol, the specialized version isn't used. Swift picks which implementation of <code>log()</code> to call by consulting only the minimum requirements that <code>Pair</code> needs to conform to <code>Loggable</code>. For this reason, the default implementation provided by the <code>Loggable</code> protocol is used instead.

```
func doSomething<T: Loggable>(with x: T) {
    x.log()
}
doSomething(with: oneAndTwo)
// Prints "(one, two)"
```

When log() is called on the instance that's passed to doSomething(_:), the customized title is omitted from the logged string.

Protocol Conformance Must Not Be Redundant

A concrete type can conform to a particular protocol only once. Swift marks redundant protocol conformances as an error. You're likely to encounter this kind of error in two kinds of situations. The first situation is when you explicitly

conform to the same protocol multiple times, but with different requirements. The second situation is when you implicitly inherit from the same protocol multiple times. These situations are discussed in the sections below.

Resolving Explicit Redundancy

Multiple extensions on a concrete type can't add conformance to the same protocol, even if the extensions' requirements are mutually exclusive. This restriction is demonstrated in the example below. Two extension declarations attempt to add conditional conformance to the Serializable protocol, one for for arrays with Int elements, and one for arrays with String elements.

```
protocol Serializable {
1
2
         func serialize() -> Any
    }
3
4
    extension Array: Serializable where Element == Int {
5
6
         func serialize() -> Any {
7
             // implementation
         }
8
    }
9
     extension Array: Serializable where Element == String {
10
         func serialize() -> Any {
11
             // implementation
12
         }
13
14
     }
    // Error: redundant conformance of 'Array<Element>' to
15
           protocol 'Serializable'
```

If you need to add conditional conformance based on multiple concrete types, create a new protocol that each type can conform to and use that protocol as the requirement when declaring conditional conformance.

```
1
   protocol SerializableInArray { }
2
   extension Int: SerializableInArray { }
3
   extension String: SerializableInArray { }
4
   extension Array: Serializable where Element:
5
          SerializableInArray {
        func serialize() -> Any {
6
7
            // implementation
        }
8
9
   }
```

Resolving Implicit Redundancy

When a concrete type conditionally conforms to a protocol, that type implicitly conforms to any parent protocols with the same requirements.

If you need a type to conditionally conform to two protocols that inherit from a single parent, explicitly declare conformance to the parent protocol. This avoids implicitly conforming to the parent protocol twice with different requirements.

The following example explicitly declares the conditional conformance of Array to Loggable to avoid a conflict when declaring its conditional conformance to both TitledLoggable and the new MarkedLoggable protocol.

```
protocol MarkedLoggable: Loggable {
  func markAndLog()
}

extension MarkedLoggable {
  func markAndLog() {
    print("-----")
  log()
```

```
9
       }
10
    }
11
12
    extension Array: Loggable where Element: Loggable { }
13
    extension Array: TitledLoggable where Element: TitledLoggable
          {
        static var logTitle: String {
14
             return "Array of '\(Element.logTitle)'"
15
16
        }
17
    }
    extension Array: MarkedLoggable where Element: MarkedLoggable
18
          { }
```

Without the extension to explicitly declare conditional conformance to Loggable, the other Array extensions would implicitly create these declarations, resulting in an error:

```
extension Array: Loggable where Element: TitledLoggable { }
extension Array: Loggable where Element: MarkedLoggable { }

// Error: redundant conformance of 'Array<Element>' to protocol
'Loggable'
```

```
GRAMMAR OF AN EXTENSION DECLARATION

extension-declaration → <u>attributes</u> opt <u>access-level-modifier</u> opt <u>extension type-identifier</u> type-inheritance-clause opt <u>generic-where-clause</u> opt <u>extension-body</u>

extension-body → { <u>extension-members</u> opt }

extension-members → <u>extension-member</u> extension-members opt

extension-member → <u>declaration</u> | <u>compiler-control-statement</u>
```

Subscript Declaration

A *subscript* declaration allows you to add subscripting support for objects of a particular type and are typically used to provide a convenient syntax for accessing the elements in a collection, list, or sequence. Subscript declarations are declared using the subscript keyword and have the following form:

```
subscript ( parameters ) -> return type {
    get {
        statements
    }
    set(setter name) {
        statements
    }
}
```

Subscript declarations can appear only in the context of a class, structure, enumeration, extension, or protocol declaration.

The *parameters* specify one or more indexes used to access elements of the corresponding type in a subscript expression (for example, the i in the expression object[i]). Although the indexes used to access the elements can be of any type, each parameter must include a type annotation to specify the type of each index. The *return type* specifies the type of the element being accessed.

As with computed properties, subscript declarations support reading and writing the value of the accessed elements. The getter is used to read the value, and the setter is used to write the value. The setter clause is optional, and when only a getter is needed, you can omit both clauses and simply return the requested value directly. That said, if you provide a setter clause, you must also provide a getter clause.

The *setter name* and enclosing parentheses are optional. If you provide a setter name, it is used as the name of the parameter to the setter. If you do not provide a setter name, the default parameter name to the setter is value. The type of the parameter to the setter is the same as the *return type*.

You can overload a subscript declaration in the type in which it is declared, as

long as the *parameters* or the *return type* differ from the one you're overloading. You can also override a subscript declaration inherited from a superclass. When you do so, you must mark the overriden subscript declaration with the override declaration modifier.

Subscript parameters follow the same rules as function parameters, with two exceptions. By default, the parameters used in subscripting don't have argument labels, unlike functions, methods, and initializers. However, you can provide explicit argument labels using the same syntax that functions, methods, and initializers use. In addition, subscripts can't have in-out parameters.

You can also declare subscripts in the context of a protocol declaration, as described in <u>Protocol Subscript Declaration</u>.

For more information about subscripting and to see examples of subscript declarations, see <u>Subscripts</u>.

Type Subscript Declarations

To declare a subscript that's exposed by the type, rather than by instances of the type, mark the subscript declaration with the static declaration modifier. Classes can mark type computed properties with the class declaration modifier instead to allow subclasses to override the superclass's implementation. In a class declaration, the static keyword has the same effect as marking the declaration with both the class and final declaration modifiers.

```
Subscript-declaration → Subscript-head Subscript-result generic-where-clause opt code-block

Subscript-declaration → Subscript-head Subscript-result generic-where-clause opt getter-setter-block

Subscript-declaration → Subscript-head Subscript-result generic-where-clause opt getter-setter-block

Subscript-declaration → Subscript-head Subscript-result generic-where-clause opt getter-setter-keyword-block

Subscript-head → attributes opt declaration-modifiers opt subscript generic-parameter-clause opt parameter-clause

Subscript-result → -> attributes opt type
```

Operator Declaration

An *operator declaration* introduces a new infix, prefix, or postfix operator into your program and is declared using the operator keyword.

You can declare operators of three different fixities: infix, prefix, and postfix. The *fixity* of an operator specifies the relative position of an operator to its operands.

There are three basic forms of an operator declaration, one for each fixity. The fixity of the operator is specified by marking the operator declaration with the infix, prefix, or postfix declaration modifier before the operator keyword. In each form, the name of the operator can contain only the operator characters defined in Operators.

The following form declares a new infix operator:

```
infix operator operator name: precedence group
```

An *infix operator* is a binary operator that is written between its two operands, such as the familiar addition operator (+) in the expression 1 + 2.

Infix operators can optionally specify a precedence group. If you omit the precedence group for an operator, Swift uses the default precedence group, DefaultPrecedence, which specifies a precedence just higher than TernaryPrecedence. For more information, see <u>Precedence Group Declaration</u>.

The following form declares a new prefix operator:

```
prefix operator operator name
```

A *prefix operator* is a unary operator that is written immediately before its operand, such as the prefix logical NOT operator (!) in the expression !a.

Prefix operators declarations don't specify a precedence level. Prefix operators are nonassociative.

The following form declares a new postfix operator:

```
postfix operator operator name
```

A *postfix operator* is a unary operator that is written immediately after its operand, such as the postfix forced-unwrap operator (!) in the expression a!.

As with prefix operators, postfix operator declarations don't specify a precedence level. Postfix operators are nonassociative.

After declaring a new operator, you implement it by declaring a static method that has the same name as the operator. The static method is a member of one of the types whose values the operator takes as an argument—for example, an operator that multiplies a <code>Double</code> by an <code>Int</code> is implemented as a static method on either the <code>Double</code> or <code>Int</code> structure. If you're implementing a prefix or postfix operator, you must also mark that method declaration with the corresponding <code>prefix</code> or <code>postfix</code> declaration modifier. To see an example of how to create and implement a new operator, see Custom Operators.

```
GRAMMAR OF AN OPERATOR DECLARATION

operator-declaration → prefix-operator-declaration | postfix-operator-declaration | infix-operator-declaration

prefix-operator-declaration → prefix operator operator

postfix-operator-declaration → postfix operator operator

infix-operator-declaration → infix operator operator infix-operator-group opt

infix-operator-group → : precedence-group-name
```

Precedence Group Declaration

A *precedence group declaration* introduces a new grouping for infix operator precedence into your program. The precedence of an operator specifies how tightly the operator binds to its operands, in the absence of grouping parentheses.

A precedence group declaration has the following form:

```
precedencegroup precedence group name {
   higherThan: lower group names
```

```
lowerThan: higher group names
associativity: associativity
assignment: assignment
}
```

The *lower group names* and *higher group names* lists specify the new precedence group's relation to existing precedence groups. The <code>lowerThan</code> precedence group attribute may only be used to refer to precedence groups declared outside of the current module. When two operators compete with each other for their operands, such as in the expression <code>2 + 3 * 5</code>, the operator with the higher relative precedence binds more tightly to its operands.

NOTE

Precedence groups related to each other using *lower group names* and *higher group names* must fit into a single relational hierarchy, but they *don't* have to form a linear hierarchy. This means it is possible to have precedence groups with undefined relative precedence. Operators from those precedence groups can't be used next to each other without grouping parentheses.

Swift defines numerous precedence groups to go along with the operators provided by the standard library. For example, the addition (+) and subtraction (-) operators belong to the AdditionPrecedence group, and the multiplication (*) and division (/) operators belong to the MultiplicationPrecedence group. For a complete list of precedence groups provided by the Swift standard library, see Operator Declarations.

The *associativity* of an operator specifies how a sequence of operators with the same precedence level are grouped together in the absence of grouping parentheses. You specify the associativity of an operator by writing one of the context-sensitive keywords left, right, or none—if your omit the associativity, the default is none. Operators that are left-associative group left-to-right. For example, the subtraction operator (-) is left-associative, so the expression 4 - 5 - 6 is grouped as (4 - 5) - 6 and evaluates to -7. Operators that are right-associative group right-to-left, and operators that are specified with an associativity of none don't associate at all. Nonassociative operators of the same precedence level can't appear adjacent to each to other. For example, the < operator has an associativity of none, which means 1 < 2 < 3 is not a valid expression.

The *assignment* of a precedence group specifies the precedence of an operator when used in an operation that includes optional chaining. When set to true, an operator in the corresponding precedence group uses the same grouping rules during optional chaining as the assignment operators from the standard library. Otherwise, when set to false or omitted, operators in the precedence group follows the same optional chaining rules as operators that don't perform assignment.

```
GRAMMAR OF A PRECEDENCE GROUP DECLARATION
precedence-group-declaration → precedencegroup precedence-group-name {
   precedence-group-attributes ont }
precedence-group-attributes → <u>precedence-group-attribute</u> <u>precedence-group-attributes</u>
precedence-group-attribute → precedence-group-relation
precedence-group-attribute → precedence-group-assignment
precedence-group-attribute → precedence-group-associativity
precedence-group-relation → higherThan : precedence-group-names
precedence-group-relation → lowerThan: precedence-group-names
precedence-group-assignment → assignment : boolean-literal
precedence-group-associativity → associativity : left
precedence-group-associativity → associativity : right
precedence-group-associativity → associativity : none
precedence-group-names → <u>precedence-group-name</u> | <u>precedence-group-name</u> ,
   precedence-group-names
precedence-group-name → <u>identifier</u>
```

Declaration Modifiers

Declaration modifiers are keywords or context-sensitive keywords that modify the behavior or meaning of a declaration. You specify a declaration modifier by writing the appropriate keyword or context-sensitive keyword between a declaration's attributes (if any) and the keyword that introduces the declaration.

class

Apply this modifier to a member of a class to indicate that the member is a member of the class itself, rather than a member of instances of the class. Members of a superclass that have this modifier and don't have the final modifier can be overridden by subclasses.

dynamic

Apply this modifier to any member of a class that can be represented by Objective-C. When you mark a member declaration with the dynamic modifier, access to that member is always dynamically dispatched using the Objective-C runtime. Access to that member is never inlined or devirtualized by the compiler.

Because declarations marked with the dynamic modifier are dispatched using the Objective-C runtime, they must be marked with the objc attribute.

final

Apply this modifier to a class or to a property, method, or subscript member of a class. It's applied to a class to indicate that the class can't be subclassed. It's applied to a property, method, or subscript of a class to indicate that a class member can't be overridden in any subclass. For an example of how to use the final attribute, see <u>Preventing Overrides</u>.

lazy

Apply this modifier to a stored variable property of a class or structure to indicate that the property's initial value is calculated and stored at most once, when the property is first accessed. For an example of how to use the lazy modifier, see <u>Lazy Stored Properties</u>.

optional

Apply this modifier to a protocol's property, method, or subscript members to indicate that a conforming type isn't required to implement those members.

You can apply the optional modifier only to protocols that are marked with the objc attribute. As a result, only class types can adopt and conform to a protocol that contains optional member requirements. For more information about how to use the optional modifier and for guidance about how to access optional protocol members—for example, when you're not sure whether a conforming type implements them—see Optional Protocol Requirements.

required

Apply this modifier to a designated or convenience initializer of a class to indicate that every subclass must implement that initializer. The subclass's implementation of that initializer must also be marked with the required modifier.

static

Apply this modifier to a member of a structure, class, enumeration, or protocol to indicate that the member is a member of the type, rather than a member of instances of that type. In the scope of a class declaration, writing the static modifier on a member declaration has the same effect as writing the class and final modifiers on that member declaration. However, constant type properties of a class are an exception: static has its normal, nonclass meaning there because you can't write class or final on those declarations.

unowned

Apply this modifier to a stored variable, constant, or stored property to indicate that the variable or property has an unowned reference to the object stored as its value. If you try to access the variable or property after the object has been deallocated, a runtime error is raised. Like a weak reference, the type of the property or value must be a class type; unlike a weak reference, the type is non-optional. For an example and more information about the unowned modifier, see <u>Unowned References</u>.

unowned(safe)

An explicit spelling of unowned.

unowned(unsafe)

Apply this modifier to a stored variable, constant, or stored property to indicate that the variable or property has an unowned reference to the object stored as its value. If you try to access the variable or property after the object has been deallocated, you'll access the memory at the location where the object used to be, which is a memory-unsafe operation. Like a weak reference, the type of the property or value must be a class type; unlike a

weak reference, the type is non-optional. For an example and more information about the unowned modifier, see <u>Unowned References</u>.

weak

Apply this modifier to a stored variable or stored variable property to indicate that the variable or property has a weak reference to the object stored as its value. The type of the variable or property must be an optional class type. If you access the variable or property after the object has been deallocated, its value is nil. For an example and more information about the weak modifier, see Weak References.

Access Control Levels

Swift provides five levels of access control: open, public, internal, file private, and private. You can mark a declaration with one of the access-level modifiers below to specify the declaration's access level. Access control is discussed in detail in Access Control.

open

Apply this modifier to a declaration to indicate the declaration can be accessed and subclassed by code in the same module as the declaration. Declarations marked with the open access-level modifier can also be accessed and subclassed by code in a module that imports the module that contains that declaration.

public

Apply this modifier to a declaration to indicate the declaration can be accessed and subclassed by code in the same module as the declaration. Declarations marked with the public access-level modifier can also be accessed (but not subclassed) by code in a module that imports the module that contains that declaration.

internal

Apply this modifier to a declaration to indicate the declaration can be

accessed only by code in the same module as the declaration. By default, most declarations are implicitly marked with the internal access-level modifier.

fileprivate

Apply this modifier to a declaration to indicate the declaration can be accessed only by code in the same source file as the declaration.

private

Apply this modifier to a declaration to indicate the declaration can be accessed only by code within the declaration's immediate enclosing scope.

For the purpose of access control, extensions to the same type that are in the same file share an access-control scope. If the type they extend is also in the same file, they share the type's access-control scope. Private members declared in the type's declaration can be accessed from extensions, and private members declared in one extension can be accessed from other extensions and from the type's declaration.

Each access-level modifier above optionally accepts a single argument, which consists of the set keyword enclosed in parentheses (for example, private(set)). Use this form of an access-level modifier when you want to specify an access level for the setter of a variable or subscript that's less than or equal to the access level of the variable or subscript itself, as discussed in <u>Getters and Setters</u>.

```
declaration-modifier → class | convenience | dynamic | final | infix | lazy |
optional | override | postfix | prefix | required | static | unowned |
unowned ( safe ) | unowned ( unsafe ) | weak
declaration-modifier → access-level-modifier
declaration-modifier → mutation-modifier
declaration-modifiers → declaration-modifier opt
access-level-modifier → private | private ( set )
access-level-modifier → fileprivate | fileprivate ( set )
access-level-modifier → public | public ( set )
access-level-modifier → open | open ( set )
mutation-modifier → mutating | nonmutating
```

Attributes

There are two kinds of attributes in Swift—those that apply to declarations and those that apply to types. An attribute provides additional information about the declaration or type. For example, the discardableResult attribute on a function declaration indicates that, although the function returns a value, the compiler shouldn't generate a warning if the return value is unused.

You specify an attribute by writing the @ symbol followed by the attribute's name and any arguments that the attribute accepts:

```
@ attribute name
@ attribute name ( attribute arguments )
```

Some declaration attributes accept arguments that specify more information about the attribute and how it applies to a particular declaration. These *attribute arguments* are enclosed in parentheses, and their format is defined by the attribute they belong to.

Declaration Attributes

You can apply a declaration attribute to declarations only.

available

Apply this attribute to indicate a declaration's life cycle relative to certain Swift language versions or certain platforms and operating system versions.

The available attribute always appears with a list of two or more commaseparated attribute arguments. These arguments begin with one of the following platform or language names:

- i0S
- iOSApplicationExtension
- macOS
- macOSApplicationExtension
- watch0S
- watchOSApplicationExtension
- tvos
- tvOSApplicationExtension
- swift

You can also use an asterisk (*) to indicate the availability of the declaration on all of the platform names listed above. An available attribute that specifies availability using a Swift version number can't use the asterisk.

The remaining arguments can appear in any order and specify additional information about the declaration's life cycle, including important milestones.

- The unavailable argument indicates that the declaration isn't available on the specified platform. This argument can't be used when specifying Swift version availability.
- The introduced argument indicates the first version of the specified platform or language in which the declaration was introduced. It has the following form:

introduced: version number

The *version number* consists of one to three positive integers, separated by periods.

• The deprecated argument indicates the first version of the specified

platform or language in which the declaration was deprecated. It has the following form:

```
deprecated: (version number)
```

The optional *version number* consists of one to three positive integers, separated by periods. Omitting the version number indicates that the declaration is currently deprecated, without giving any information about when the deprecation occurred. If you omit the version number, omit the colon (:) as well.

• The obsoleted argument indicates the first version of the specified platform or language in which the declaration was obsoleted. When a declaration is obsoleted, it's removed from the specified platform or language and can no longer be used. It has the following form:

```
obsoleted: version number
```

The *version number* consists of one to three positive integers, separated by periods.

 The message argument provides a textual message that the compiler displays when emitting a warning or error about the use of a deprecated or obsoleted declaration. It has the following form:

```
message: message
```

The *message* consists of a string literal.

 The renamed argument provides a textual message that indicates the new name for a declaration that's been renamed. The compiler displays the new name when emitting an error about the use of a renamed declaration. It has the following form:

```
renamed: new name
```

The *new name* consists of a string literal.

You can apply the available attribute with the renamed and unavailable arguments to a type alias declaration, as shown below, to indicate that the name of a declaration changed between releases of a framework or library. This combination results in a compile-time error that the declaration has been renamed.

```
// First release
1
2
   protocol MyProtocol {
3
       // protocol definition
4
   }
1
   // Subsequent release renames MyProtocol
   protocol MyRenamedProtocol {
       // protocol definition
4
   }
5
   @available(*, unavailable, renamed: "MyRenamedProtocol")
6
   typealias MyProtocol = MyRenamedProtocol
7
```

You can apply multiple available attributes on a single declaration to specify the declaration's availability on different platforms and different versions of Swift. The declaration that the available attribute applies to is ignored if the attribute specifies a platform or language version that doesn't match the current target. If you use multiple available attributes, the effective availability is the combination of the platform and Swift availabilities.

If an available attribute only specifies an introduced argument in addition to a platform or language name argument, you can use the following shorthand syntax instead:

```
@available( platform name version number , *)
@available(swift version number )
```

The shorthand syntax for available attributes concisely expresses availability

for multiple platforms. Although the two forms are functionally equivalent, the shorthand form is preferred whenever possible.

```
1  @available(iOS 10.0, macOS 10.12, *)
2  class MyClass {
3    // class definition
4 }
```

An available attribute that specifies availability using a Swift version number can't additionally specify a declaration's platform availability. Instead, use separate available attributes to specify a Swift version availability and one or more platform availabilities.

```
1  @available(swift 3.0.2)
2  @available(macOS 10.12, *)
3  struct MyStruct {
    // struct definition
5 }
```

discardableResult

Apply this attribute to a function or method declaration to suppress the compiler warning when the function or method that returns a value is called without using its result.

dynamicCallable

Apply this attribute to a class, structure, enumeration, or protocol to treat instances of the type as callable functions. The type must implement either a dynamicallyCall(withArguments:) method, a dynamicallyCall(withKeywordArguments:) method, or both.

You can call an instance of a dynamically callable type as if it's a function that

takes any number of arguments.

```
1
    @dynamicCallable
2
    struct TelephoneExchange {
        func dynamicallyCall(withArguments phoneNumber: [Int]) {
3
             if phoneNumber == [4, 1, 1] {
4
                 print("Get Swift help on forums.swift.org")
5
6
             } else {
                 print("Unrecognized number")
7
8
            }
9
        }
10
    }
11
12
    let dial = TelephoneExchange()
13
14
    // Use a dynamic method call.
15
    dial(4, 1, 1)
16
    // Prints "Get Swift help on forums.swift.org"
17
18
    dial(8, 6, 7, 5, 3, 0, 9)
    // Prints "Unrecognized number"
19
20
21
    // Call the underlying method directly.
22
    dial.dynamicallyCall(withArguments: [4, 1, 1])
```

The declaration of the dynamicallyCall(withArguments:) method must have a single parameter that conforms to the ExpressibleByArrayLiteral protocol—like [Int] in the example above. The return type can be any type.

You can include labels in a dynamic method call if you implement the dynamicallyCall(withKeywordArguments:) method.

```
@dynamicCallable
1
2
    struct Repeater {
3
         func dynamicallyCall(withKeywordArguments pairs:
           KeyValuePairs<String, Int>) -> String {
             return pairs
4
                 .map { label, count in
5
                     repeatElement(label, count:
6
           count).joined(separator: " ")
7
                 }
                 .joined(separator: "\n")
8
         }
9
10
    }
11
12
    let repeatLabels = Repeater()
13
    print(repeatLabels(a: 1, b: 2, c: 3, b: 2, a: 1))
14
    // a
15
    // b b
16
    // c c c
    // b b
17
18
    // a
```

The declaration of the dynamicallyCall(withKeywordArguments:) method must have a single parameter that conforms to the ExpressibleByDictionaryLiteral protocol, and the return type can be any type. The parameter's Key must be ExpressibleByStringLiteral. The previous example uses KeyValuePairs as the parameter type so that callers can include duplicate parameter labels—a and b appear multiple times in the call to repeat.

If you implement both dynamicallyCall methods, dynamicallyCall(withKeywordArguments:) is called when the method call includes keyword arguments. In all other cases, dynamicallyCall(withArguments:) is called.

You can only call a dynamically callable instance with arguments and a return value that match the types you specify in one of your dynamicallyCall method implementations. The call in the following example doesn't compile because there isn't an implementation of dynamicallyCall(withArguments:) that takes KeyValuePairs<String, String>.

```
repeatLabels(a: "four") // Error
```

dynamicMemberLookup

Apply this attribute to a class, structure, enumeration, or protocol to enable members to be looked up by name at runtime. The type must implement a subscript(dynamicMemberLookup:) subscript.

In an explicit member expression, if there isn't a corresponding declaration for the named member, the expression is understood as a call to the type's subscript(dynamicMemberLookup:) subscript, passing information about the member as the argument. The subscript can accept a parameter that's either a key path or a member name; if you implement both subscripts, the subscript that takes key path argument is used.

An implementation of subscript(dynamicMemberLookup:) can accept key paths using an argument of type KeyPath, WritableKeyPath, or ReferenceWritableKeyPath. It can accept member names using an argument of a type that conforms to the ExpressibleByStringLiteral protocol—in most cases, String. The subscript's return type can be any type.

Dynamic member lookup by member name can be used to create a wrapper type around data that can't be type checked at compile time, such as when bridging data from other languages into Swift. For example:

```
5
        subscript(dynamicMember member: String) -> Int {
             return dictionary[member] ?? 1054
6
        }
7
    }
8
    let s = DynamicStruct()
9
10
11
    // Use dynamic member lookup.
12
    let dynamic = s.someDynamicMember
    print(dynamic)
13
    // Prints "325"
14
15
16
    // Call the underlying subscript directly.
    let equivalent = s[dynamicMember: "someDynamicMember"]
17
18
    print(dynamic == equivalent)
    // Prints "true"
19
```

Dynamic member lookup by key path can be used to implement a wrapper type in a way that supports compile-time type checking. For example:

```
1
    struct Point { var x, y: Int }
2
3
    @dynamicMemberLookup
    struct PassthroughWrapper<Value> {
4
         var value: Value
5
         subscript<T>(dynamicMember member: KeyPath<Value, T>) -> T
6
           {
             get { return value[keyPath: member] }
7
8
         }
9
    }
10
```

```
11  let point = Point(x: 381, y: 431)
12  let wrapper = PassthroughWrapper(value: point)
13  print(wrapper.x)
```

frozen

Apply this attribute to a structure or enumeration declaration to restrict the kinds of changes you can make to the type. This attribute is allowed only when compiling in library evolution mode. Future versions of the library can't change the declaration by adding, removing, or reordering an enumeration's cases or a structure's stored instance properties. These changes are allowed on nonfrozen types, but they break ABI compatibility for frozen types.

NOTE

When the compiler isn't in library evolution mode, all structures and enumerations are implicitly frozen, and you can't use this attribute.

In library evolution mode, code that interacts with members of nonfrozen structures and enumerations is compiled in a way that allows it to continue working without recompiling even if a future version of the library adds, removes, or reorders some of that type's members. The compiler makes this possible using techniques like looking up information at runtime and adding a layer of indirection. Marking a structure or enumeration as frozen gives up this flexibility to gain performance: Future versions of the library can make only limited changes to the type, but the compiler can make additional optimizations in code that interacts with the type's members.

Frozen types, the types of the stored properties of frozen structures, and the associated values of frozen enumeration cases must be public or marked with the usableFromInline attribute. The properties of a frozen structure can't have property observers, and expressions that provide the initial value for stored instance properties must follow the same restrictions as inlinable functions, as discussed in inlinable.

To enable library evolution mode on the command line, pass the -enable-

library-evolution option to the Swift compiler. To enable it in Xcode, set the "Build Libraries for Distribution" build setting (BUILD_LIBRARY_FOR_DISTRIBUTION) to Yes, as described in Xcode Help.

A switch statement over a frozen enumeration doesn't require a default case, as discussed in <u>Switching Over Future Enumeration Cases</u>. Including a default or @unknown default case when switching over a frozen enumeration produces a warning because that code is never executed.

GKInspectable

Apply this attribute to expose a custom GameplayKit component property to the SpriteKit editor UI. Applying this attribute also implies the objc attribute.

inlinable

Apply this attribute to a function, method, computed property, subscript, convenience initializer, or deinitializer declaration to expose that declaration's implementation as part of the module's public interface. The compiler is allowed to replace calls to an inlinable symbol with a copy of the symbol's implementation at the call site.

Inlinable code can interact with public symbols declared in any module, and it can interact with internal symbols declared in the same module that are marked with the usableFromInline attribute. Inlinable code can't interact with private or fileprivate symbols.

This attribute can't be applied to declarations that are nested inside functions or to fileprivate or private declarations. Functions and closures that are defined inside an inlinable function are implicitly inlinable, even though they can't be marked with this attribute.

nonobjc

Apply this attribute to a method, property, subscript, or initializer declaration to

suppress an implicit objc attribute. The nonobjc attribute tells the compiler to make the declaration unavailable in Objective-C code, even though it's possible to represent it in Objective-C.

Applying this attribute to an extension has the same effect as applying it to every member of that extension that isn't explicitly marked with the objc attribute.

You use the nonobjc attribute to resolve circularity for bridging methods in a class marked with the objc attribute, and to allow overloading of methods and initializers in a class marked with the objc attribute.

A method marked with the nonobjc attribute can't override a method marked with the objc attribute. However, a method marked with the objc attribute can override a method marked with the nonobjc attribute. Similarly, a method marked with the nonobjc attribute can't satisfy a protocol requirement for a method marked with the objc attribute.

NSApplicationMain

Apply this attribute to a class to indicate that it's the application delegate. Using this attribute is equivalent to calling the $NSApplicationMain(_:_:)$ function.

If you don't use this attribute, supply a main.swift file with code at the top level that calls the NSApplicationMain(_:_:) function as follows:

```
import AppKit
NSApplicationMain(CommandLine.argc, CommandLine.unsafeArgv)
```

NSCopying

Apply this attribute to a stored variable property of a class. This attribute causes the property's setter to be synthesized with a *copy* of the property's value—returned by the <code>copyWithZone(_:)</code> method—instead of the value of the property itself. The type of the property must conform to the <code>NSCopying</code> protocol.

The NSCopying attribute behaves in a way similar to the Objective-C copy property attribute.

NSManaged

Apply this attribute to an instance method or stored variable property of a class that inherits from NSManagedObject to indicate that Core Data dynamically provides its implementation at runtime, based on the associated entity description. For a property marked with the NSManaged attribute, Core Data also provides the storage at runtime. Applying this attribute also implies the objc attribute.

objc

Apply this attribute to any declaration that can be represented in Objective-C—for example, nonnested classes, protocols, nongeneric enumerations (constrained to integer raw-value types), properties and methods (including getters and setters) of classes, protocols and optional members of a protocol, initializers, and subscripts. The <code>objc</code> attribute tells the compiler that a declaration is available to use in Objective-C code.

Applying this attribute to an extension has the same effect as applying it to every member of that extension that isn't explicitly marked with the nonobjc attribute.

The compiler implicitly adds the <code>objc</code> attribute to subclasses of any class defined in Objective-C. However, the subclass must not be generic, and must not inherit from any generic classes. You can explicitly add the <code>objc</code> attribute to a subclass that meets these criteria, to specify its Objective-C name as discussed below. Protocols that are marked with the <code>objc</code> attribute can't inherit from protocols that aren't marked with this attribute.

The objc attribute is also implicitly added in the following cases:

• The declaration is an override in a subclass, and the superclass's declaration has the objc attribute.

- The declaration satisfies a requirement from a protocol that has the object attribute.
- The declaration has the IBAction, IBSegueAction, IBOutlet, IBDesignable, IBInspectable, NSManaged, or GKInspectable attribute.

If you apply the <code>objc</code> attribute to an enumeration, each enumeration case is exposed to Objective-C code as the concatenation of the enumeration name and the case name. The first letter of the case name is capitalized. For example, a case named <code>venus</code> in a Swift <code>Planet</code> enumeration is exposed to Objective-C code as a case named <code>PlanetVenus</code>.

The objc attribute optionally accepts a single attribute argument, which consists of an identifier. The identifier specifies the name to be exposed to Objective-C for the entity that the objc attribute applies to. You can use this argument to name classes, enumerations, enumeration cases, protocols, methods, getters, setters, and initializers. If you specify the Objective-C name for a class, protocol, or enumeration, include a three-letter prefix on the name, as described in Conventions in Programming with Objective-C. The example below exposes the getter for the enabled property of the ExampleClass to Objective-C code as isEnabled rather than just as the name of the property itself.

objcMembers

Apply this attribute to a class declaration, to implicitly apply the <code>objc</code> attribute to all Objective-C compatible members of the class, its extensions, its subclasses, and all of the extensions of its subclasses.

Most code should use the <code>objc</code> attribute instead, to expose only the declarations that are needed. If you need to expose many declarations, you can group them in an extension that has the <code>objc</code> attribute. The <code>objcMembers</code> attribute is a convenience for libraries that make heavy use of the introspection facilities of the Objective-C runtime. Applying the <code>objc</code> attribute when it isn't needed can increase your binary size and adversely affect performance.

propertyWrapper

Apply this attribute to a class, structure, or enumeration declaration to use that type as a property wrapper. When you apply this attribute to a type, you create a custom attribute with the same name as the type. Apply that new attribute to a property of a class, structure, or enumeration to wrap access to the property through an instance of the wrapper type. Local and global variables can't use property wrappers.

The wrapper must define a wrappedvalue instance property. The wrapped value of the property is the value that the getter and setter for this property expose. In most cases, wrappedvalue is a computed value, but it can be a stored value instead. The wrapper is responsible for defining and managing any underlying storage needed by its wrapped value. The compiler synthesizes storage for the instance of the wrapper type by prefixing the name of the wrapped property with an underscore (_)—for example, the wrapper for someProperty is stored as __someProperty. The synthesized storage for the wrapper has an access control level of private.

A property that has a property wrapper can include willset and didSet blocks, but it can't override the compiler-synthesized get or set blocks.

Swift provides two forms of syntactic sugar for initialization of a property wrapper. You can use assignment syntax in the definition of a wrapped value to pass the expression on the right-hand side of the assignment as the argument to the wrappedValue parameter of the property wrapper's initializer. You can also provide arguments to the attribute when you apply it to a property, and those arguments are passed to the property wrapper's initializer. For example, in the code below, SomeStruct calls each of the initializers that SomeWrapper defines.

```
@propertyWrapper
1
    struct SomeWrapper {
2
3
        var wrappedValue: Int
4
        var someValue: Double
        init() {
5
             self.wrappedValue = 100
6
7
             self.someValue = 12.3
8
        }
9
        init(wrappedValue: Int) {
             self.wrappedValue = wrappedValue
10
             self.someValue = 45.6
11
12
         }
13
         init(wrappedValue value: Int, custom: Double) {
14
             self.wrappedValue = value
15
             self.someValue = custom
16
        }
17
    }
18
19
    struct SomeStruct {
20
        // Uses init()
21
         @SomeWrapper var a: Int
22
23
         // Uses init(wrappedValue:)
         @SomeWrapper var b = 10
24
25
26
         // Both use init(wrappedValue:custom:)
27
         @SomeWrapper(custom: 98.7) var c = 30
         @SomeWrapper(wrappedValue: 30, custom: 98.7) var d
28
29
    }
```

The *projected value* for a wrapped property is a second value that a property wrapper can use to expose additional functionality. The author of a property wrapper type is responsible for determining the meaning of its projected value and defining the interface that the projected value exposes. To project a value from a property wrapper, define a projected value instance property on the wrapper type. The compiler synthesizes an identifier for the projected value by prefixing the name of the wrapped property with a dollar sign (\$)—for example, the projected value for someProperty is \$someProperty. The projected value has the same access control level as the original wrapped property.

```
1
    @propertyWrapper
    struct WrapperWithProjection {
2
3
        var wrappedValue: Int
        var projectedValue: SomeProjection {
4
5
             return SomeProjection(wrapper: self)
6
        }
7
    }
    struct SomeProjection {
8
        var wrapper: WrapperWithProjection
10
    }
11
    struct SomeStruct {
12
13
         @WrapperWithProjection var x = 123
    }
14
    let s = SomeStruct()
15
                  // Int value
16
    S.X
17
    s.$x
                  // SomeProjection value
     s.$x.wrapper // WrapperWithProjection value
18
```

requires_stored_property_inits

Apply this attribute to a class declaration to require all stored properties within

the class to provide default values as part of their definitions. This attribute is inferred for any class that inherits from NSManagedObject.

testable

Apply this attribute to an import declaration to import that module with changes to its access control that simplify testing the module's code. Entities in the imported module that are marked with the internal access-level modifier are imported as if they were declared with the public access-level modifier. Classes and class members that are marked with the internal or public access-level modifier are imported as if they were declared with the open access-level modifier. The imported module must be compiled with testing enabled.

UIApplicationMain

Apply this attribute to a class to indicate that it's the application delegate. Using this attribute is equivalent to calling the UIApplicationMain function and passing this class's name as the name of the delegate class.

If you don't use this attribute, supply a main.swift file with code at the top level that calls the <u>UIApplicationMain(:::::)</u> function. For example, if your app uses a custom subclass of UIApplication as its principal class, call the UIApplicationMain(:::::::) function instead of using this attribute.

usableFromInline

Apply this attribute to a function, method, computed property, subscript, initializer, or deinitializer declaration to allow that symbol to be used in inlinable code that's defined in the same module as the declaration. The declaration must have the internal access level modifier. A structure or class marked usableFromInline can use only types that are public or usableFromInline for its properties. An enumeration marked usableFromInline can use only types that are public or usableFromInline for the raw values and associated values of its cases.

Like the public access level modifier, this attribute exposes the declaration as

part of the module's public interface. Unlike public, the compiler doesn't allow declarations marked with usableFromInline to be referenced by name in code outside the module, even though the declaration's symbol is exported. However, code outside the module might still be able to interact with the declaration's symbol by using runtime behavior.

Declarations marked with the inlinable attribute are implicitly usable from inlinable code. Although either inlinable or usableFromInline can be applied to internal declarations, applying both attributes is an error.

warn_unqualified_access

Apply this attribute to a top-level function, instance method, or class or static method to trigger warnings when that function or method is used without a preceding qualifier, such as a module name, type name, or instance variable or constant. Use this attribute to help discourage ambiguity between functions with the same name that are accessible from the same scope.

For example, the Swift standard library includes both a top-level $\min(::)$ function and a $\min()$ method for sequences with comparable elements. The sequence method is declared with the warn_unqualified_access attribute to help reduce confusion when attempting to use one or the other from within a Sequence extension.

Declaration Attributes Used by Interface Builder

Interface Builder attributes are declaration attributes used by Interface Builder to synchronize with Xcode. Swift provides the following Interface Builder attributes: IBAction, IBSegueAction, IBOutlet, IBDesignable, and IBInspectable. These attributes are conceptually the same as their Objective-C counterparts.

You apply the IBOutlet and IBInspectable attributes to property declarations of a class. You apply the IBAction and IBSegueAction attribute to method declarations of a class and the IBDesignable attribute to class declarations.

Applying the IBAction, IBSegueAction, IBOutlet, IBDesignable, or

IBInspectable attribute also implies the objc attribute.

Type Attributes

You can apply type attributes to types only.

autoclosure

Apply this attribute to delay the evaluation of an expression by automatically wrapping that expression in a closure with no arguments. You apply it to a parameter's type in a method or function declaration, for a parameter whose type is a function type that takes no arguments and that returns a value of the type of the expression. For an example of how to use the autoclosure attribute, see Autoclosures and Function Type.

convention

Apply this attribute to the type of a function to indicate its calling conventions.

The convention attribute always appears with one of the following arguments:

- The swift argument indicates a Swift function reference. This is the standard calling convention for function values in Swift.
- The block argument indicates an Objective-C compatible block reference. The function value is represented as a reference to the block object, which is an id-compatible Objective-C object that embeds its invocation function within the object. The invocation function uses the C calling convention.
- The c argument indicates a C function reference. The function value carries no context and uses the C calling convention.

With a few exceptions, a function of any calling convention can be used when a function any other calling convention is needed. A nongeneric global function, a

local function that doesn't capture any local variables or a closure that doesn't capture any local variables can be converted to the C calling convention. Other Swift functions can't be converted to the C calling convention. A function with the Objective-C block calling convention can't be converted to the C calling convention.

escaping

Apply this attribute to a parameter's type in a method or function declaration to indicate that the parameter's value can be stored for later execution. This means that the value is allowed to outlive the lifetime of the call. Function type parameters with the escaping type attribute require explicit use of self. for properties or methods. For an example of how to use the escaping attribute, see <u>Escaping Closures</u>.

Switch Case Attributes

You can apply switch case attributes to switch cases only.

unknown

Apply this attribute to a switch case to indicate that it isn't expected to be matched by any case of the enumeration that's known at the time the code is compiled. For an example of how to use the unknown attribute, see Switching Over Future Enumeration Cases.

```
GRAMMAR OF AN ATTRIBUTE

attribute \rightarrow @ attribute-name attribute-argument-clause opt

attribute-name \rightarrow identifier

attribute-argument-clause \rightarrow ( balanced-tokens opt )

attributes \rightarrow attribute attributes opt

balanced-tokens \rightarrow balanced-token balanced-tokens opt

balanced-token \rightarrow ( balanced-tokens opt )

balanced-token \rightarrow [ balanced-tokens opt ]
```

```
balanced-token → { balanced-tokens opt }
balanced-token → Any identifier, keyword, literal, or operator
balanced-token → Any punctuation except (, ), [, ], {, or }
```

Patterns

A *pattern* represents the structure of a single value or a composite value. For example, the structure of a tuple (1, 2) is a comma-separated list of two elements. Because patterns represent the structure of a value rather than any one particular value, you can match them with a variety of values. For instance, the pattern (x, y) matches the tuple (1, 2) and any other two-element tuple. In addition to matching a pattern with a value, you can extract part or all of a composite value and bind each part to a constant or variable name.

In Swift, there are two basic kinds of patterns: those that successfully match any kind of value, and those that may fail to match a specified value at runtime.

The first kind of pattern is used for destructuring values in simple variable, constant, and optional bindings. These include wildcard patterns, identifier patterns, and any value binding or tuple patterns containing them. You can specify a type annotation for these patterns to constrain them to match only values of a certain type.

The second kind of pattern is used for full pattern matching, where the values you're trying to match against may not be there at runtime. These include enumeration case patterns, optional patterns, expression patterns, and typecasting patterns. You use these patterns in a case label of a switch statement, a catch clause of a do statement, or in the case condition of an if, while, guard, or for-in statement.

```
grammar of a pattern
→ wildcard-pattern type-annotation opt

pattern → identifier-pattern type-annotation opt

pattern → value-binding-pattern

pattern → tuple-pattern type-annotation opt

pattern → enum-case-pattern

pattern → optional-pattern

pattern → type-casting-pattern

pattern → expression-pattern
```

Wildcard Pattern

A *wildcard pattern* matches and ignores any value and consists of an underscore (_). Use a wildcard pattern when you don't care about the values being matched against. For example, the following code iterates through the closed range 1...3, ignoring the current value of the range on each iteration of the loop:

```
for _ in 1...3 {
    // Do something three times.
}
```

```
GRAMMAR OF A WILDCARD PATTERN

wildcard-pattern → _
```

Identifier Pattern

An *identifier pattern* matches any value and binds the matched value to a variable or constant name. For example, in the following constant declaration, someValue is an identifier pattern that matches the value 42 of type Int:

```
let someValue = 42
```

When the match succeeds, the value 42 is bound (assigned) to the constant name someValue.

When the pattern on the left-hand side of a variable or constant declaration is an identifier pattern, the identifier pattern is implicitly a subpattern of a value-binding pattern.

```
GRAMMAR OF AN IDENTIFIER PATTERN

identifier-pattern → identifier
```

Value-Binding Pattern

A *value-binding pattern* binds matched values to variable or constant names. Value-binding patterns that bind a matched value to the name of a constant begin with the let keyword; those that bind to the name of variable begin with the var keyword.

Identifiers patterns within a value-binding pattern bind new named variables or constants to their matching values. For example, you can decompose the elements of a tuple and bind the value of each element to a corresponding identifier pattern.

```
1 let point = (3, 2)
2 switch point {
3     // Bind x and y to the elements of point.
4 case let (x, y):
5     print("The point is at (\(x), \(y)).")
6 }
7     // Prints "The point is at (3, 2)."
```

In the example above, let distributes to each identifier pattern in the tuple pattern (x, y). Because of this behavior, the switch cases case let (x, y): and case (let x, let y): match the same values.

```
GRAMMAR OF A VALUE-BINDING PATTERN

value-binding-pattern → var pattern | let pattern
```

Tuple Pattern

A *tuple pattern* is a comma-separated list of zero or more patterns, enclosed in parentheses. Tuple patterns match values of corresponding tuple types.

You can constrain a tuple pattern to match certain kinds of tuple types by using

type annotations. For example, the tuple pattern (x, y): (Int, Int) in the constant declaration let (x, y): (Int, Int) = (1, 2) matches only tuple types in which both elements are of type Int.

When a tuple pattern is used as the pattern in a for-in statement or in a variable or constant declaration, it can contain only wildcard patterns, identifier patterns, optional patterns, or other tuple patterns that contain those. For example, the following code isn't valid because the element 0 in the tuple pattern (x, 0) is an expression pattern:

The parentheses around a tuple pattern that contains a single element have no effect. The pattern matches values of that single element's type. For example, the following are equivalent:

```
1 let a = 2 // a: Int = 2
2 let (a) = 2 // a: Int = 2
3 let (a): Int = 2 // a: Int = 2
```

```
GRAMMAR OF A TUPLE PATTERN

tuple-pattern \rightarrow ( tuple-pattern-element-list opt )

tuple-pattern-element-list \rightarrow tuple-pattern-element | tuple-pattern-element opt, opt opt

opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt opt
```

Enumeration Case Pattern

An enumeration case pattern matches a case of an existing enumeration type.

Enumeration case patterns appear in switch statement case labels and in the case conditions of if, while, guard, and for-in statements.

If the enumeration case you're trying to match has any associated values, the corresponding enumeration case pattern must specify a tuple pattern that contains one element for each associated value. For an example that uses a switch statement to match enumeration cases containing associated values, see Associated Values.

An enumeration case pattern also matches values of that case wrapped in an optional. This simplified syntax lets you omit an optional pattern. Note that, because <code>Optional</code> is implemented as an enumeration, <code>.none</code> and <code>.some</code> can appear in the same switch as the cases of the enumeration type.

```
1
    enum SomeEnum { case left, right }
    let x: SomeEnum? = .left
2
    switch x {
3
    case left:
        print("Turn left")
5
    case right:
6
        print("Turn right")
7
8
    case nil:
        print("Keep going straight")
10
    }
    // Prints "Turn left"
11
```

```
GRAMMAR OF AN ENUMERATION CASE PATTERN

enum-case-pattern → type-identifier opt • enum-case-name tuple-pattern opt
```

Optional Pattern

An $optional\ pattern\ matches\ values\ wrapped\ in\ a\ some(Wrapped)\ case\ of\ an$

Optional

Wrapped> enumeration. Optional patterns consist of an identifier
pattern followed immediately by a question mark and appear in the same places
as enumeration case patterns.

Because optional patterns are syntactic sugar for Optional enumeration case patterns, the following are equivalent:

```
let someOptional: Int? = 42
1
    // Match using an enumeration case pattern.
    if case .some(let x) = someOptional {
3
        print(x)
4
    }
5
6
    // Match using an optional pattern.
7
8
    if case let x? = someOptional {
9
        print(x)
10
    }
```

The optional pattern provides a convenient way to iterate over an array of optional values in a for-in statement, executing the body of the loop only for non-nil elements.

```
1 let arrayOfOptionalInts: [Int?] = [nil, 2, 3, nil, 5]
2 // Match only non-nil values.
3 for case let number? in arrayOfOptionalInts {
    print("Found a \((number)\)")
5 }
6 // Found a 2
7 // Found a 3
8 // Found a 5
```

```
GRAMMAR OF AN OPTIONAL PATTERN

optional-pattern → <u>identifier-pattern</u> ?
```

Type-Casting Patterns

There are two type-casting patterns, the is pattern and the as pattern. The is pattern appears only in switch statement case labels. The is and as patterns have the following form:

```
is type pattern as type
```

The is pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the is pattern—or a subclass of that type. The is pattern behaves like the is operator in that they both perform a type cast but discard the returned type.

The as pattern matches a value if the type of that value at runtime is the same as the type specified in the right-hand side of the as pattern—or a subclass of that type. If the match succeeds, the type of the matched value is cast to the *pattern* specified in the right-hand side of the as pattern.

For an example that uses a switch statement to match values with is and as patterns, see Type Casting for Any and AnyObject.

```
GRAMMAR OF A TYPE CASTING PATTERN

type\text{-}casting\text{-}pattern \rightarrow \underline{is\text{-}pattern} \mid \underline{as\text{-}pattern}

is\text{-}pattern \rightarrow \underline{is} \underline{type}

as\text{-}pattern \rightarrow \underline{pattern} \quad as \quad \underline{type}
```

Expression Pattern

An *expression pattern* represents the value of an expression. Expression patterns appear only in switch statement case labels.

The expression represented by the expression pattern is compared with the value of an input expression using the Swift standard library ~= operator. The matches succeeds if the ~= operator returns true. By default, the ~= operator compares two values of the same type using the == operator. It can also match a value with a range of values, by checking whether the value is contained within the range, as the following example shows.

```
1
    let point = (1, 2)
2
    switch point {
3
    case (0, 0):
         print("(0, 0) is at the origin.")
4
    case (-2...2, -2...2):
5
         print("(\(point.0), \(point.1)) is near the origin.")
6
    default:
         print("The point is at (\((point.0), \((point.1))."))
8
9
    }
    // Prints "(1, 2) is near the origin."
10
```

You can overload the ~= operator to provide custom expression matching behavior. For example, you can rewrite the above example to compare the point expression with a string representations of points.

```
// Overload the ~= operator to match a string with an integer.
1
    func ~= (pattern: String, value: Int) -> Bool {
2
         return pattern == "\(value)"
3
    }
4
    switch point {
5
    case ("0", "0"):
6
7
         print("(0, 0) is at the origin.")
    default:
8
         print("The point is at (\((point.0), \((point.1))."))
9
     }
10
```

11 // Prints "The point is at (1, 2)."

GRAMMAR OF AN EXPRESSION PATTERN

expression-pattern → expression

Generic Parameters and Arguments

This chapter describes parameters and arguments for generic types, functions, and initializers. When you declare a generic type, function, subscript, or initializer, you specify the type parameters that the generic type, function, or initializer can work with. These type parameters act as placeholders that are replaced by actual concrete type arguments when an instance of a generic type is created or a generic function or initializer is called.

For an overview of generics in Swift, see Generics.

Generic Parameter Clause

A *generic parameter clause* specifies the type parameters of a generic type or function, along with any associated constraints and requirements on those parameters. A generic parameter clause is enclosed in angle brackets (<>) and has the following form:

```
< generic parameter list >
```

The *generic parameter list* is a comma-separated list of generic parameters, each of which has the following form:

```
type parameter: constraint
```

A generic parameter consists of a *type parameter* followed by an optional *constraint*. A *type parameter* is simply the name of a placeholder type (for example, T, U, V, Key, Value, and so on). You have access to the type parameters (and any of their associated types) in the rest of the type, function, or initializer declaration, including in the signature of the function or initializer.

The *constraint* specifies that a type parameter inherits from a specific class or conforms to a protocol or protocol composition. For example, in the generic

function below, the generic parameter \top : Comparable indicates that any type argument substituted for the type parameter \top must conform to the Comparable protocol.

```
func simpleMax<T: Comparable>(_ x: T, _ y: T) -> T {
   if x < y {
      return y
   }
   return x
}</pre>
```

Because Int and Double, for example, both conform to the Comparable protocol, this function accepts arguments of either type. In contrast with generic types, you don't specify a generic argument clause when you use a generic function or initializer. The type arguments are instead inferred from the type of the arguments passed to the function or initializer.

```
simpleMax(17, 42) // T is inferred to be Int
simpleMax(3.14159, 2.71828) // T is inferred to be Double
```

Generic Where Clauses

You can specify additional requirements on type parameters and their associated types by including a generic where clause right before the opening curly brace of a type or function's body. A generic where clause consists of the where keyword, followed by a comma-separated list of one or more *requirements*.

```
where requirements
```

The *requirements* in a generic where clause specify that a type parameter inherits from a class or conforms to a protocol or protocol composition. Although the generic where clause provides syntactic sugar for expressing simple constraints on type parameters (for example, <T: Comparable> is equivalent to <T> where T: Comparable and so on), you can use it to provide more complex constraints on

type parameters and their associated types. For example, you can constrain the associated types of type parameters to conform to protocols. For example, <S: Sequence> where S.Iterator.Element: Equatable specifies that S conforms to the Sequence protocol and that the associated type S.Iterator.Element conforms to the Equatable protocol. This constraint ensures that each element of the sequence is equatable.

You can also specify the requirement that two types be identical, using the == operator. For example, <S1: Sequence, S2: Sequence> where S1.Iterator.Element == S2.Iterator.Element expresses the constraints that S1 and S2 conform to the Sequence protocol and that the elements of both sequences must be of the same type.

Any type argument substituted for a type parameter must meet all the constraints and requirements placed on the type parameter.

You can overload a generic function or initializer by providing different constraints, requirements, or both on the type parameters. When you call an overloaded generic function or initializer, the compiler uses these constraints to resolve which overloaded function or initializer to invoke.

For more information about generic where clauses and to see an example of one in a generic function declaration, see <u>Generic Where Clauses</u>.

```
generic-parameter-clause → < generic-parameter-list >
generic-parameter-list → generic-parameter | generic-parameter , generic-parameter-

list
generic-parameter → type-name
generic-parameter → type-name : type-identifier
generic-parameter → type-name : protocol-composition-type
generic-where-clause → where requirement-list
requirement-list → requirement | requirement | same-type-requirement
conformance-requirement → type-identifier : type-identifier
conformance-requirement → type-identifier : protocol-composition-type
same-type-requirement → type-identifier : protocol-composition-type
same-type-requirement → type-identifier == type
```

Generic Argument Clause

A *generic argument clause* specifies the type arguments of a generic type. A generic argument clause is enclosed in angle brackets (<>) and has the following form:

```
< generic argument list >
```

The *generic argument list* is a comma-separated list of type arguments. A *type argument* is the name of an actual concrete type that replaces a corresponding type parameter in the generic parameter clause of a generic type. The result is a specialized version of that generic type. The example below shows a simplified version of the Swift standard library's generic dictionary type.

The specialized version of the generic Dictionary type, Dictionary<String, Int> is formed by replacing the generic parameters Key: Hashable and Value with the concrete type arguments String and Int. Each type argument must satisfy all the constraints of the generic parameter it replaces, including any additional requirements specified in a generic where clause. In the example above, the Key type parameter is constrained to conform to the Hashable protocol and therefore String must also conform to the Hashable protocol.

You can also replace a type parameter with a type argument that is itself a specialized version of a generic type (provided it satisfies the appropriate constraints and requirements). For example, you can replace the type parameter <code>Element</code> in <code>Array<Element></code> with a specialized version of an array, <code>Array<Int></code>, to form an array whose elements are themselves arrays of integers.

As mentioned in <u>Generic Parameter Clause</u>, you don't use a generic argument clause to specify the type arguments of a generic function or initializer.

```
GRAMMAR OF A GENERIC ARGUMENT CLAUSE generic-argument-clause \rightarrow < generic-argument-list > generic-argument-list \rightarrow generic-argument | generic-argument , generic-argument-list generic-argument \rightarrow type
```

Summary of the Grammar

Lexical Structure

```
GRAMMAR OF WHITESPACE
whitespace -- whitespace opt
whitespace-item → line-break
whitespace-item → comment
whitespace-item → multiline-comment
whitespace-item → U+0000, U+0009, U+000B, U+000C, or U+0020
line-break → U+000A
line-break → U+000D
line-break → U+000D followed by U+000A
comment → // comment-text line-break
multiline-comment → /* multiline-comment-text */
comment-text → comment-text-item comment-text opt
comment-text-item → Any Unicode scalar value except U+000A or U+000D
multiline-comment-text -> multiline-comment-text item multiline-comment-text ont
multiline-comment-text-item → multiline-comment
multiline-comment-text-item → comment-text-item
multiline-comment-text-item → Any Unicode scalar value except /* or */
```

```
GRAMMAR OF AN IDENTIFIER
identifier -- identifier-head identifier-characters opt
identifier -- identifier-head identifier-characters ont
identifier → implicit-parameter-name
identifier → <u>property-wrapper-projection</u>
identifier-list → identifier | identifier , identifier-list
identifier-head → Upper- or lowercase letter A through Z
identifier-head →
identifier-head → U+00A8, U+00AA, U+00AD, U+00AF, U+00B2-U+00B5, or U+00B7-
   U+00BA
identifier-head → U+00BC-U+00BE, U+00C0-U+00D6, U+00D8-U+00F6, or U+00F8-
   U+00FF
identifier-head → U+0100-U+02FF, U+0370-U+167F, U+1681-U+180D, or U+180F-
   U+1DBF
identifier-head → U+1E00-U+1FFF
identifier-head → U+200B-U+200D, U+202A-U+202E, U+203F-U+2040, U+2054, or
   U+2060-U+206F
identifier-head → U+2070-U+20CF, U+2100-U+218F, U+2460-U+24FF, or U+2776-
```

```
U+2793
identifier-head → U+2C00-U+2DFF or U+2E80-U+2FFF
identifier-head → U+3004-U+3007, U+3021-U+302F, U+3031-U+303F, or U+3040-
   U+D7FF
identifier-head → U+F900-U+FD3D, U+FD40-U+FDCF, U+FDF0-U+FE1F, or U+FE30-
   U+FE44
identifier-head → U+FE47-U+FFFD
identifier-head → U+10000-U+1FFFD, U+20000-U+2FFFD, U+30000-U+3FFFD, or
   U+40000-U+4FFFD
identifier-head → U+50000-U+5FFFD, U+60000-U+6FFFD, U+70000-U+7FFFD, or
   U+80000-U+8FFFD
identifier-head → U+90000-U+9FFFD, U+A0000-U+AFFFD, U+B0000-U+BFFFD, or
   U+C0000-U+CFFD
identifier-head → U+D0000-U+DFFFD or U+E0000-U+EFFFD
identifier-character → Digit 0 through 9
identifier-character → U+0300-U+036F, U+1DC0-U+1DFF, U+20D0-U+20FF, or
   U+FE20-U+FE2F
identifier-character → identifier-head
identifier-characters - identifier-character identifier-characters opt
implicit-parameter-name → $ decimal-digits
property-wrapper-projection → $ identifier-characters
```

GRAMMAR OF A LITERAL literal \rightarrow numeric-literal | string-literal | boolean-literal | nil-literal numeric-literal \rightarrow -opt integer-literal | -opt floating-point-literal boolean-literal \rightarrow true | false nil-literal \rightarrow nil

```
GRAMMAR OF AN INTEGER LITERAL
integer-literal → binary-literal
integer-literal → octal-literal
integer-literal → <u>decimal-literal</u>
integer-literal → <u>hexadecimal-literal</u>
binary-literal -> 9b binary-digit binary-literal-characters opt
binary-digit → Digit 0 or 1
binary-literal-character → binary-digit | _
binary-literal-characters -> binary-literal-character binary-literal-characters ont
octal-literal -> 00 octal-digit octal-literal-characters ont
octal-digit → Digit 0 through 7
octal-literal-character → octal-digit | _
octal-literal-characters - octal-literal-character octal-literal-characters ont
decimal-literal -- decimal-digit decimal-literal-characters ont
decimal-digit → Digit 0 through 9
decimal-digits - decimal-digit decimal-digits ont
decimal-literal-character → decimal-digit | _
```

```
decimal-literal-characters → decimal-literal-character decimal-literal-characters opt
hexadecimal-literal → 0x hexadecimal-digit hexadecimal-literal-characters opt
hexadecimal-digit → Digit 0 through 9, a through f, or A through F
hexadecimal-literal-character → hexadecimal-digit | _
hexadecimal-literal-characters → hexadecimal-literal-character hexadecimal-literal-characters opt
```

```
\begin{array}{lll} \text{GRAMMAR OF A FLOATING-POINT LITERAL} \\ \text{floating-point-literal} & \rightarrow & \underline{\text{decimal-literal}} & \underline{\text{decimal-fraction}} & \underline{\text{opt}} & \underline{\text{decimal-exponent}} & \underline{\text{opt}} \\ \text{floating-point-literal} & \rightarrow & \underline{\text{hexadecimal-literal}} & \underline{\text{hexadecimal-fraction}} & \underline{\text{opt}} & \underline{\text{hexadecimal-literal}} \\ \underline{\text{decimal-fraction}} & \rightarrow & \underline{\text{decimal-literal}} & \underline{\text{decimal-literal}} \\ \underline{\text{decimal-exponent}} & \rightarrow & \underline{\text{floating-point-e}} & \underline{\text{sign}} & \underline{\text{opt}} & \underline{\text{decimal-literal-characters}} & \underline{\text{opt}} \\ \underline{\text{hexadecimal-exponent}} & \rightarrow & \underline{\text{floating-point-p}} & \underline{\text{sign}} & \underline{\text{opt}} & \underline{\text{decimal-literal}} \\ \underline{\text{floating-point-e}} & \rightarrow & \mathbf{e} & \| \mathbf{E} \\ \underline{\text{floating-point-p}} & \rightarrow & \mathbf{p} & \| \mathbf{P} \\ \underline{\text{sign}} & \rightarrow & \mathbf{+} \| - \\ \end{array}
```

```
GRAMMAR OF A STRING LITERAL
string-literal → <u>static-string-literal</u> | <u>interpolated-string-literal</u>
string-literal-opening-delimiter --> extended-string-literal-delimiter opt "
string-literal-closing-delimiter \rightarrow " extended-string-literal-delimiter opt
static-string-literal → <u>string-literal-opening-delimiter</u> <u>quoted-text</u> opt <u>string-literal-closing-</u>
    delimiter
static-string-literal --> multiline-string-literal-opening-delimiter multiline-quoted-text opt
    multiline-string-literal-closing-delimiter
multiline-string-literal-opening-delimiter → <u>extended-string-literal-delimiter</u> """
multiline-string-literal-closing-delimiter → """ extended-string-literal-delimiter
extended-string-literal-delimiter → # extended-string-literal-delimiter opt
quoted-text → quoted-text-item quoted-text opt
quoted-text-item → <u>escaped-character</u>
quoted-text-item → Any Unicode scalar value except ", \, U+000A, or U+000D
multiline-quoted-text -- multiline-quoted-text opt
multiline-quoted-text-item → escaped-character
multiline-quoted-text-item → Any Unicode scalar value except \
multiline-quoted-text-item → <u>escaped-newline</u>
interpolated-string-literal → <u>string-literal-opening-delimiter</u> <u>interpolated-text</u> opt <u>string-</u>
    literal-closing-delimiter
interpolated-string-literal → multiline-string-literal-opening-delimiter interpolated-text opt
    multiline-string-literal-closing-delimiter
interpolated-text → interpolated-text-item interpolated-text ont
interpolated-text-item → \( expression ) | quoted-text-item
```

```
multiline-interpolated-text → multiline-interpolated-text-item multiline-interpolated-text opt
multiline-interpolated-text-item → \( expression ) | multiline-quoted-text-item
escape-sequence → \( extended-string-literal-delimiter \)
escape-sequence o | escape-sequence \( | escape-sequence t | escape-sequence r | escape-sequence " | escape-sequence contact r | escape-sequence r | esc
```

```
GRAMMAR OF OPERATORS
operator → operator-head operator-characters opt
operator → <u>dot-operator-head</u> <u>dot-operator-characters</u>
operator-head \rightarrow / | = | - | + | ! | * | % | < | > | & | | | ^ | ~ | ?
operator-head → U+00A1–U+00A7
operator-head → U+00A9 or U+00AB
operator-head → U+00AC or U+00AE
operator-head → U+00B0–U+00B1
operator-head → U+00B6, U+00BB, U+00BF, U+00D7, or U+00F7
operator-head → U+2016–U+2017
operator-head → U+2020–U+2027
operator-head → U+2030–U+203E
operator-head → U+2041–U+2053
operator-head → U+2055–U+205E
operator-head → U+2190–U+23FF
operator-head → U+2500–U+2775
operator-head → U+2794–U+2BFF
operator-head → U+2E00–U+2E7F
operator-head → U+3001–U+3003
operator-head → U+3008–U+3020
operator-head → U+3030
operator-character → <u>operator-head</u>
operator-character → U+0300–U+036F
operator-character → U+1DC0-U+1DFF
operator-character → U+20D0–U+20FF
operator-character → U+FE00–U+FE0F
operator-character → U+FE20-U+FE2F
operator-character → U+E0100–U+E01EF
operator-characters - operator-character operator-characters opt
dot-operator-head \rightarrow .
dot-operator-character → . | operator-character
dot-operator-characters → dot-operator-character dot-operator-characters opt
binary-operator → operator
prefix-operator → operator
postfix-operator → <u>operator</u>
```

Types

```
GRAMMAR OF A TYPE
type → <u>function-type</u>
type → array-type
type → dictionary-type
type → type-identifier
type → tuple-type
type → optional-type
type → implicitly-unwrapped-optional-type
type → protocol-composition-type
type → opaque-type
type → metatype-type
type → <u>self-type</u>
type → Any
type \rightarrow (type)
GRAMMAR OF A TYPE ANNOTATION
type-annotation \rightarrow : <u>attributes</u> opt inout opt <u>type</u>
GRAMMAR OF A TYPE IDENTIFIER
type-identifier → type-name generic-argument-clause opt | type-name generic-argument-
    <u>clause</u> opt <u>type-identifier</u>
type-name \rightarrow identifier
GRAMMAR OF A TUPLE TYPE
tuple-type \rightarrow ( ) | ( \underline{tuple-type-element} , \underline{tuple-type-element-list} )
tuple-type-element-list → tuple-type-element | tuple-type-element , tuple-type-element-
    <u>list</u>
tuple-type-element → <u>element-name</u> <u>type-annotation</u> | <u>type</u>
element-name → <u>identifier</u>
GRAMMAR OF A FUNCTION TYPE
function-type → <u>attributes</u> <sub>Opt</sub> <u>function-type-argument-clause</u> throws <sub>Opt</sub> -> <u>type</u>
function-type-argument-clause → ( )
function-type-argument-clause \rightarrow [ \underline{function-type-argument-list} ...opt ]
function-type-argument-list → <u>function-type-argument</u> | <u>function-type-argument</u> ,
    function-type-argument-list
function-type-argument \rightarrow <u>attributes</u> opt inout opt type | <u>argument-label</u> type-
    annotation
argument-label → <u>identifier</u>
```

```
GRAMMAR OF AN ARRAY TYPE
array-type \rightarrow [ type ]
GRAMMAR OF A DICTIONARY TYPE
dictionary-type \rightarrow [ \underline{type} : \underline{type} ]
GRAMMAR OF AN OPTIONAL TYPE
optional-type → type ?
GRAMMAR OF AN IMPLICITLY UNWRAPPED OPTIONAL TYPE
implicitly-unwrapped-optional-type \rightarrow type!
GRAMMAR OF A PROTOCOL COMPOSITION TYPE
protocol-composition-type -> type-identifier & protocol-composition-continuation
protocol-composition-continuation → <u>type-identifier</u> | <u>protocol-composition-type</u>
GRAMMAR OF AN OPAQUE TYPE
opaque-type → some type
GRAMMAR OF A METATYPE TYPE
metatype-type \rightarrow type . Type | type . Protocol
GRAMMAR OF A SELF TYPE
self-type → Self
GRAMMAR OF A TYPE INHERITANCE CLAUSE
type-inheritance-clause → : type-inheritance-list
type-inheritance-list \rightarrow type-identifier | type-identifier , type-inheritance-list
```

Expressions

```
GRAMMAR OF AN EXPRESSION

expression \rightarrow try-operator opt prefix-expression binary-expressions opt expression-list \rightarrow expression | expression , expression-list
```

```
GRAMMAR OF A PREFIX EXPRESSION
```

prefix-expression → prefix-operator opt postfix-expression

prefix-expression → <u>in-out-expression</u> in-out-expression → & <u>identifier</u>

GRAMMAR OF A TRY EXPRESSION

try-operator \rightarrow try | try ? | try !

GRAMMAR OF A BINARY EXPRESSION

binary-expression → binary-operator prefix-expression

binary-expression → assignment-operator try-operator opt prefix-expression

binary-expression → <u>conditional-operator</u> <u>try-operator</u> <u>opt</u> <u>prefix-expression</u>

binary-expression → type-casting-operator

binary-expressions → binary-expression binary-expressions opt

GRAMMAR OF AN ASSIGNMENT OPERATOR

assignment-operator → =

GRAMMAR OF A CONDITIONAL OPERATOR

conditional-operator → ? <u>expression</u> :

GRAMMAR OF A TYPE-CASTING OPERATOR

type-casting-operator → is type

type-casting-operator → as type

type-casting-operator → as ? type

type-casting-operator → as ! type

GRAMMAR OF A PRIMARY EXPRESSION

primary-expression \rightarrow identifier generic-argument-clause ont

primary-expression → literal-expression

primary-expression → self-expression

primary-expression → superclass-expression

primary-expression → <u>closure-expression</u>

primary-expression → <u>parenthesized-expression</u>

primary-expression → *tuple-expression*

primary-expression → <u>implicit-member-expression</u>

primary-expression → <u>wildcard-expression</u>

primary-expression → key-path-expression

primary-expression → <u>selector-expression</u>

primary-expression → <u>key-path-string-expression</u>

```
GRAMMAR OF A LITERAL EXPRESSION
literal-expression → <u>literal</u>
literal-expression → <u>array-literal</u> | <u>dictionary-literal</u> | <u>playground-literal</u>
literal-expression → #file | #line | #column | #function | #dsohandle
array-literal → [ array-literal-items ont ]
array-literal-items \rightarrow array-literal-item , array-literal-item , array-literal-items
array-literal-item → <u>expression</u>
dictionary-literal → [ dictionary-literal-items ] | [ : ]
dictionary-literal-item → <u>dictionary-literal-item</u> , <u>opt</u> | <u>dictionary-literal-item</u> , <u>dictionary-</u>
       literal-items
dictionary-literal-item → <u>expression</u> : <u>expression</u>
playground-literal → #colorLiteral ( red : <u>expression</u> , green : <u>expression</u>
       , blue : <u>expression</u> , alpha : <u>expression</u> )
playground-literal → #fileLiteral ( resourceName : <u>expression</u> )
playground-literal → #imageLiteral ( resourceName : <u>expression</u> )
GRAMMAR OF A SELF EXPRESSION
self-expression → self | self-method-expression | self-subscript-expression | self-
      initializer-expression
self-method-expression → self . <u>identifier</u>
self-subscript-expression → self [ <u>function-call-argument-list</u> ]
self-initializer-expression → self . init
GRAMMAR OF A SUPERCLASS EXPRESSION
superclass-expression → superclass-method-expression | superclass-subscript-
       expression | superclass-initializer-expression
superclass-method-expression → super . identifier
superclass-subscript-expression → super [ <u>function-call-argument-list</u> ]
superclass-initializer-expression → super . init
GRAMMAR OF A CLOSURE EXPRESSION
closure-expression → { closure-signature opt statements opt }
closure-signature → <u>capture-list</u> ont <u>closure-parameter-clause</u> throwsont <u>function-</u>
       <u>result</u> opt in
closure-signature → <u>capture-list</u> in
closure-parameter-clause → ( ) | ( closure-parameter-list ) | identifier-list
closure-parameter | closu
       list
closure-parameter → <u>closure-parameter-name</u> <u>type-annotation</u> opt
closure-parameter → closure-parameter-name type-annotation ...
closure-parameter-name → identifier
capture-list \rightarrow [ capture-list-items ]
capture-list-item → <u>capture-list-item</u> | <u>capture-list-item</u> , <u>capture-list-item</u>
```

```
capture-list-item → <u>capture-specifier</u> opt <u>expression</u>
capture-specifier → weak | unowned | unowned(safe) | unowned(unsafe)
GRAMMAR OF A IMPLICIT MEMBER EXPRESSION
implicit-member-expression → • identifier
GRAMMAR OF A PARENTHESIZED EXPRESSION
parenthesized-expression \rightarrow ( expression )
GRAMMAR OF A TUPLE EXPRESSION
tuple-expression \rightarrow ( ) | ( \underline{tuple-element} , \underline{tuple-element-list} )
tuple-element-list → tuple-element | tuple-element , tuple-element-list
tuple-element → <u>expression</u> | <u>identifier</u> : <u>expression</u>
GRAMMAR OF A WILDCARD EXPRESSION
wildcard-expression →
GRAMMAR OF A KEY-PATH EXPRESSION
key-path-expression \rightarrow \bigvee \underline{type}_{opt} . \underline{key}-path-components
key-path-component | key-path
        components
key-path-component → <u>identifier</u> <u>key-path-postfixes</u> <sub>opt</sub> | <u>key-path-postfixes</u>
key-path-postfixes → key-path-postfix key-path-postfixes opt
key-path-postfix \rightarrow ? | ! | self | [ function-call-argument-list ]
GRAMMAR OF A SELECTOR EXPRESSION
selector-expression → #selector ( <u>expression</u> )
selector-expression → #selector ( getter: expression )
selector-expression → #selector ( setter: <u>expression</u> )
GRAMMAR OF A KEY-PATH STRING EXPRESSION
key-path-string-expression → #keyPath ( expression )
GRAMMAR OF A POSTFIX EXPRESSION
postfix-expression → primary-expression
postfix-expression → <u>postfix-expression</u> <u>postfix-operator</u>
postfix-expression → <u>function-call-expression</u>
postfix-expression → <u>initializer-expression</u>
postfix-expression → explicit-member-expression
```

```
postfix-expression → postfix-self-expression
postfix-expression → <u>subscript-expression</u>
postfix-expression → <u>forced-value-expression</u>
postfix-expression → optional-chaining-expression
GRAMMAR OF A FUNCTION CALL EXPRESSION
function-call-expression → postfix-expression function-call-argument-clause
function-call-expression → postfix-expression function-call-argument-clause ont trailing-
       closure
function-call-argument-clause \rightarrow ( ) | ( \underline{function-call-argument-list} )
function-call-argument | function-call-argume
        call-argument-list
function-call-argument → <u>expression</u> | <u>identifier</u> : <u>expression</u>
function-call-argument → <u>operator</u> | <u>identifier</u> : <u>operator</u>
trailing-closure → <u>closure-expression</u>
GRAMMAR OF AN INITIALIZER EXPRESSION
initializer-expression → <u>postfix-expression</u> . init
initializer-expression → postfix-expression . init ( argument-names )
GRAMMAR OF AN EXPLICIT MEMBER EXPRESSION
explicit-member-expression → postfix-expression · decimal-digits
explicit-member-expression → postfix-expression identifier generic-argument-clause
      opt
explicit-member-expression → postfix-expression · identifier ( argument-names )
argument-names → argument-name argument-names ont
argument-name → <u>identifier</u>:
GRAMMAR OF A POSTFIX SELF EXPRESSION
postfix-self-expression → postfix-expression . self
GRAMMAR OF A SUBSCRIPT EXPRESSION
subscript-expression → postfix-expression [ function-call-argument-list ]
GRAMMAR OF A FORCED-VALUE EXPRESSION
forced-value-expression → postfix-expression !
GRAMMAR OF AN OPTIONAL-CHAINING EXPRESSION
optional-chaining-expression → postfix-expression ?
```

Statements

```
GRAMMAR OF A STATEMENT
statement → <u>expression</u> ; opt
statement → declaration ; opt
statement → loop-statement ; opt
statement → branch-statement ; opt
statement - labeled-statement ; opt
statement - control-transfer-statement ; opt
statement → <u>defer-statement</u>; opt
statement → do-statement ; opt
statement → compiler-control-statement
statements -> statement statements opt
GRAMMAR OF A LOOP STATEMENT
loop-statement → for-in-statement
loop-statement → while-statement
loop-statement → repeat-while-statement
GRAMMAR OF A FOR-IN STATEMENT
for-in-statement → for case<sub>opt</sub> pattern in expression where-clause opt code-block
GRAMMAR OF A WHILE STATEMENT
while-statement → while condition-list code-block
condition-list \rightarrow condition | condition , condition-list
condition → expression | availability-condition | case-condition | optional-binding-
   condition
case-condition → case <u>pattern</u> <u>initializer</u>
optional-binding-condition → let <u>pattern</u> <u>initializer</u> | var <u>pattern</u> <u>initializer</u>
GRAMMAR OF A REPEAT-WHILE STATEMENT
repeat-while-statement → repeat <u>code-block</u> while <u>expression</u>
GRAMMAR OF A BRANCH STATEMENT
branch-statement → <u>if-statement</u>
branch-statement → <u>guard-statement</u>
branch-statement → switch-statement
```

GRAMMAR OF AN IF STATEMENT

```
if-statement → if condition-list code-block else-clause opt
else-clause → else code-block | else if-statement

GRAMMAR OF A GUARD STATEMENT
guard-statement → guard condition-list else code-block

GRAMMAR OF A SWITCH STATEMENT
switch-statement → switch expression { switch-cases opt }
```

```
SWITCH STATEMENT

SWITCH-STATEMENT

SWITCH-STATEMENT

SWITCH-STATEMENT

SWITCH-STATEMENT

SWITCH-STATEMENT

SWITCH-STATEMENT

SWITCH-CASES → SWITCH EXPRESSION { SWITCH-CASES OPT }

SWITCH-CASES → CASE-LABEL STATEMENTS

SWITCH-CASE → SWITCH-CASE OPT | PATTERN WHERE-CLAUSE OP
```

```
GRAMMAR OF A LABELED STATEMENT

labeled-statement → <u>statement-label</u> <u>loop-statement</u>

labeled-statement → <u>statement-label</u> <u>if-statement</u>

labeled-statement → <u>statement-label</u> <u>switch-statement</u>

labeled-statement → <u>statement-label</u> <u>do-statement</u>

statement-label → <u>label-name</u> :

label-name → <u>identifier</u>
```

```
GRAMMAR OF A CONTROL TRANSFER STATEMENT

control-transfer-statement → break-statement
control-transfer-statement → continue-statement
control-transfer-statement → fallthrough-statement
control-transfer-statement → return-statement
control-transfer-statement → throw-statement
```

```
GRAMMAR OF A BREAK STATEMENT

break-statement → break label-name opt
```

```
GRAMMAR OF A CONTINUE STATEMENT

continue-statement → continue <u>label-name</u> opt
```

```
GRAMMAR OF A FALLTHROUGH STATEMENT
```

 $fall through-statement \rightarrow fall through$

```
GRAMMAR OF A RETURN STATEMENT

return-statement → return expression opt
```

```
GRAMMAR OF A THROW STATEMENT

throw-statement → throw expression
```

```
GRAMMAR OF A DEFER STATEMENT

defer-statement → defer code-block
```

```
GRAMMAR OF A DO STATEMENT
```

```
do-statement → do <u>code-block</u> <u>catch-clauses</u> <sub>opt</sub> 
catch-clause → <u>catch-clause</u> <u>catch-clauses</u> <sub>opt</sub> 
catch-clause → catch <u>pattern</u> <sub>opt</sub> <u>where-clause</u> <sub>opt</sub> <u>code-block</u>
```

GRAMMAR OF A COMPILER CONTROL STATEMENT

```
compiler-control-statement → <u>conditional-compilation-block</u>
compiler-control-statement → <u>line-control-statement</u>
compiler-control-statement → <u>diagnostic-statement</u>
```

```
GRAMMAR OF A CONDITIONAL COMPILATION BLOCK
```

```
conditional-compilation-block → if-directive-clause elseif-directive-clauses opt else-
directive-clause opt endif-directive
if-directive-clause → if-directive compilation-condition statements opt
elseif-directive-clauses → elseif-directive-clause elseif-directive-clauses opt
elseif-directive-clause → elseif-directive compilation-condition statements opt
else-directive-clause → else-directive statements opt
if-directive → #if
elseif-directive → #elseif
else-directive → #else
endif-directive → #endif
compilation-condition → platform-condition
```

```
compilation-condition → <u>identifier</u>
compilation-condition → boolean-literal
compilation-condition → ( compilation-condition )
compilation-condition → ! compilation-condition
compilation-condition → compilation-condition && compilation-condition
compilation-condition \rightarrow compilation-condition \mid \mid compilation-condition
platform-condition → os ( <u>operating-system</u> )
platform-condition → arch ( <u>architecture</u> )
platform-condition \rightarrow swift ( >= swift-version ) | swift ( < swift-version )
platform-condition → compiler ( >= <u>swift-version</u> ) | compiler ( < <u>swift-version</u>
platform-condition \rightarrow canImport ( \underline{module}-name )
platform-condition → targetEnvironment ( <u>environment</u> )
operating-system → macOS | iOS | watchOS | tvOS
architecture → i386 | x86_64 | arm | arm64
swift-version -> decimal-digits swift-version-continuation opt
swift-version-continuation - . decimal-digits swift-version-continuation opt
module-name → <u>identifier</u>
environment → simulator
GRAMMAR OF A LINE CONTROL STATEMENT
line-control-statement → #sourceLocation ( file: file-name , line: line-
   <u>number</u>
line-control-statement → #sourceLocation ( )
line-number → A decimal integer greater than zero
file-name → <u>static-string-literal</u>
GRAMMAR OF A COMPILE-TIME DIAGNOSTIC STATEMENT
diagnostic-statement → #error ( diagnostic-message )
diagnostic-statement → #warning ( diagnostic-message )
diagnostic-message → <u>static-string-literal</u>
GRAMMAR OF AN AVAILABILITY CONDITION
availability-condition → #available ( <u>availability-arguments</u> )
availability-arguments - availability-argument | availability-argument , availability-
   arguments
availability-argument → <u>platform-name</u> <u>platform-version</u>
availability-argument → *
platform-name → iOS | iOSApplicationExtension
platform-name → macOS | macOSApplicationExtension
platform-name → watch0S
platform-name → tv0S
platform-version → <u>decimal-digits</u>
platform-version → <u>decimal-digits</u> · <u>decimal-digits</u>
```

Declarations

```
GRAMMAR OF A DECLARATION
declaration → import-declaration
declaration → constant-declaration
declaration → variable-declaration
declaration → typealias-declaration
declaration → function-declaration
declaration → <u>enum-declaration</u>
declaration → struct-declaration
declaration → <u>class-declaration</u>
declaration → <u>protocol-declaration</u>
declaration → <u>initializer-declaration</u>
declaration → <u>deinitializer-declaration</u>
declaration → <u>extension-declaration</u>
declaration → <u>subscript-declaration</u>
declaration → <u>operator-declaration</u>
declaration → precedence-group-declaration
declarations → declaration declarations opt
GRAMMAR OF A TOP-LEVEL DECLARATION
```

```
GRAMMAR OF A TOP-LEVEL DECLARATION

top-level-declaration → statements opt
```

```
GRAMMAR OF A CODE BLOCK

code-block → { statements opt }
```

```
GRAMMAR OF AN IMPORT DECLARATION

import-declaration → <u>attributes</u> opt <u>import import-kind</u> opt <u>import-path</u>

import-kind → typealias | struct | class | enum | protocol | let | var |

func

import-path → <u>import-path-identifier</u> | <u>import-path-identifier</u> . <u>import-path</u>

import-path-identifier → <u>identifier</u> | <u>operator</u>
```

```
GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → <u>attributes</u> <sub>Opt</sub> <u>declaration-modifiers</u> <sub>Opt</sub> let <u>pattern-initializer-list</u>

pattern-initializer | <u>pattern-initializer</u> , <u>pattern-initializer-list</u>
```

```
pattern-initializer \rightarrow pattern initializer opt
pattern-initializer \rightarrow pattern initializer opt
```

```
GRAMMAR OF A VARIABLE DECLARATION
variable-declaration → <u>variable-declaration-head</u> <u>pattern-initializer-list</u>
variable-declaration → variable-declaration-head variable-name type-annotation code-
    block
variable-declaration → <u>variable-declaration-head</u> <u>variable-name</u> <u>type-annotation</u> <u>getter-</u>
    setter-block
variable-declaration → variable-declaration-head variable-name type-annotation getter-
    setter-keyword-block
variable-declaration → variable-declaration-head variable-name initializer willSet-didSet-
    block
variable-declaration → variable-declaration-head variable-name type-annotation
    initializer opt willSet-didSet-block
variable-declaration-head → <u>attributes</u> opt <u>declaration-modifiers</u> opt var
variable-name → <u>identifier</u>
getter-setter-block → code-block
getter-setter-block → { getter-clause setter-clause ont }
getter-setter-block → { setter-clause getter-clause }
getter-clause → <u>attributes</u> opt <u>mutation-modifier</u> opt get <u>code-block</u>
setter-clause → <u>attributes</u> opt <u>mutation-modifier</u> opt set <u>setter-name</u> opt <u>code-block</u>
setter-name \rightarrow ( identifier )
getter-setter-keyword-block → { getter-keyword-clause ont }
getter-setter-keyword-block → { setter-keyword-clause getter-keyword-clause }
getter-keyword-clause → <u>attributes</u> opt <u>mutation-modifier</u> opt get
setter-keyword-clause → <u>attributes</u> opt <u>mutation-modifier</u> opt set
willSet-didSet-block → { willSet-clause didSet-clause opt }
willSet-didSet-block → { didSet-clause willSet-clause opt }
willSet-clause → <u>attributes</u> opt willSet <u>setter-name</u> opt <u>code-block</u>
didSet-clause → <u>attributes</u> <sub>Opt</sub> didSet <u>setter-name</u> <sub>Opt</sub> <u>code-block</u>
```

```
GRAMMAR OF A TYPE ALIAS DECLARATION

typealias-declaration \rightarrow attributes opt access-level-modifier opt typealias typealias-

name generic-parameter-clause opt typealias-assignment

typealias-name \rightarrow identifier

typealias-assignment \rightarrow = type
```

```
GRAMMAR OF A FUNCTION DECLARATION

function-declaration → function-head function-name generic-parameter-clause opt

function-signature generic-where-clause opt function-body opt

function-head → attributes opt declaration-modifiers opt func
```

```
function-name → identifier | operator function-signature → parameter-clause throws<sub>Opt</sub> function-result<sub>Opt</sub> function-signature → parameter-clause rethrows function-result<sub>Opt</sub> function-result → -> attributes<sub>Opt</sub> type
function-body → code-block
parameter-clause → ( ) | ( parameter-list<sub>Opt</sub>)
parameter-list → parameter | parameter<sub>Opt</sub> | local-parameter-name<sub>Opt</sub> | local-parameter-name<sub>Opt</sub>
```

```
GRAMMAR OF AN ENUMERATION DECLARATION
enum-declaration → <u>attributes</u> opt <u>access-level-modifier</u> opt <u>union-style-enum</u>
enum-declaration → <u>attributes</u> opt <u>access-level-modifier</u> opt <u>raw-value-style-enum</u>
union-style-enum \rightarrow indirect<sub>opt</sub> enum <u>enum-name</u> <u>generic-parameter-clause</u> opt
         type-inheritance-clause opt generic-where-clause opt { union-style-enum-members
       opt }
union-style-enum-members - union-style-enum-member union-style-enum-members ont
union-style-enum-member → <u>declaration</u> | <u>union-style-enum-case-clause</u> | <u>compiler-</u>
        control-statement
union-style-enum-case-clause \rightarrow <u>attributes</u> opt indirect case <u>union-style-enum-</u>
union-style-enum-case | union-style-enum-c
        <u>style-enum-case-list</u>
union-style-enum-case → <u>enum-case-name</u> <u>tuple-type</u> opt
enum-name → <u>identifier</u>
enum-case-name → <u>identifier</u>
raw-value-style-enum → enum <u>enum-name</u> <u>generic-parameter-clause</u> <sub>Opt</sub> <u>type-</u>
        <u>inheritance-clause</u> <u>generic-where-clause</u> <sub>Opt</sub> { <u>raw-value-style-enum-members</u> }
raw-value-style-enum-members → <u>raw-value-style-enum-member</u> <u>raw-value-style-enum-</u>
        members opt
raw-value-style-enum-member → <u>declaration</u> | <u>raw-value-style-enum-case-clause</u> |
         compiler-control-statement
raw-value-style-enum-case-clause → <u>attributes</u> <sub>Opt</sub> case <u>raw-value-style-enum-case-</u>
raw-value-style-enum-case-list → <u>raw-value-style-enum-case</u> | <u>raw-value-style-enum-</u>
        <u>case</u> , <u>raw-value-style-enum-case-list</u>
raw-value-style-enum-case → <u>enum-case-name</u> <u>raw-value-assignment</u> <sub>Opt</sub>
raw-value-assignment \rightarrow = raw-value-literal
raw-value-literal → <u>numeric-literal</u> | <u>static-string-literal</u> | <u>boolean-literal</u>
```

```
Class-declaration → attributes opt access-level-modifier opt final_opt class class-
name generic-parameter-clause opt type-inheritance-clause opt generic-where-clause
opt class-body

class-declaration → attributes opt final access-level-modifier opt class class-
name generic-parameter-clause opt type-inheritance-clause opt generic-where-clause
opt class-body

class-body

class-body → { class-members opt }

class-members → class-member class-members opt
class-member → declaration | compiler-control-statement
```

```
protocol-declaration → attributes opt access-level-modifier opt protocol-
name type-inheritance-clause opt generic-where-clause opt protocol-body

protocol-name → identifier
protocol-body → { protocol-members opt }

protocol-members → protocol-member protocol-members opt protocol-member → protocol-member-declaration | compiler-control-statement protocol-member-declaration → protocol-property-declaration protocol-member-declaration → protocol-method-declaration protocol-member-declaration → protocol-initializer-declaration protocol-member-declaration → protocol-subscript-declaration protocol-member-declaration → protocol-subscript-declaration protocol-member-declaration → protocol-associated-type-declaration protocol-member-declaration → typealias-declaration
```

GRAMMAR OF A PROTOCOL PROPERTY DECLARATION

protocol-property-declaration → <u>variable-declaration-head</u> <u>variable-name</u> <u>type-annotation</u> getter-setter-keyword-block

 $protocol-method-declaration \rightarrow function-head function-name generic-parameter-clause opt function-signature generic-where-clause opt$

GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION

protocol-initializer-declaration → <u>initializer-head</u> <u>generic-parameter-clause</u> opt <u>parameter-clause</u> **throws**_{Opt} <u>generic-where-clause</u> opt protocol-initializer-declaration → <u>initializer-head</u> <u>generic-parameter-clause</u> opt <u>parameter-clause</u> **rethrows** <u>generic-where-clause</u> opt

GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION

 $protocol\text{-}subscript\text{-}declaration \rightarrow \underline{subscript\text{-}head} \underline{subscript\text{-}result} \underline{generic\text{-}where\text{-}clause} \\ \underline{opt} \underline{getter\text{-}setter\text{-}keyword\text{-}block}$

GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION

protocol-associated-type-declaration \rightarrow <u>attributes</u> opt <u>access-level-modifier</u> opt **associatedtype** <u>type-alias-name</u> <u>type-inheritance-clause</u> opt <u>type-alias-assignment</u> opt <u>generic-where-clause</u> opt

GRAMMAR OF AN INITIALIZER DECLARATION

initializer-declaration → initializer-head generic-parameter-clause opt parameter-clause throws_{opt} generic-where-clause opt initializer-body initializer-declaration → initializer-head generic-parameter-clause opt parameter-clause

initializer-declaration → <u>initializer-head generic-parameter-clause</u> opt <u>parameter-clause</u>

rethrows generic-where-clause opt initializer-body initializer-head \rightarrow attributes opt declaration-modifiers opt init

initializer-head → <u>attributes</u> opt <u>declaration-modifiers</u> opt init ?

initializer-head → <u>attributes</u> _{Opt} <u>declaration-modifiers</u> _{Opt} **init**!

initializer-body → <u>code-block</u>

GRAMMAR OF A DEINITIALIZER DECLARATION

deinitializer- $declaration \rightarrow \underline{attributes}_{Opt}$ deinit \underline{code} - \underline{block}

GRAMMAR OF AN EXTENSION DECLARATION

extension-declaration \rightarrow attributes opt access-level-modifier opt extension type-identifier type-inheritance-clause opt generic-where-clause opt extension-body extension-body \rightarrow { extension-members opt } extension-members \rightarrow extension-member extension-members opt extension-member \rightarrow declaration | compiler-control-statement

```
Subscript-declaration → subscript-head subscript-result generic-where-clause opt code-block

subscript-declaration → subscript-head subscript-result generic-where-clause opt getter-setter-block

subscript-declaration → subscript-head subscript-result generic-where-clause opt getter-setter-block

subscript-declaration → subscript-head subscript-result generic-where-clause opt getter-setter-keyword-block

subscript-head → attributes opt declaration-modifiers opt subscript generic-parameter-clause opt parameter-clause

subscript-result → -> attributes opt type
```

```
GRAMMAR OF AN OPERATOR DECLARATION

operator-declaration → prefix-operator-declaration | postfix-operator-declaration | infix-operator-declaration

prefix-operator-declaration → prefix operator operator

postfix-operator-declaration → postfix operator operator

infix-operator-declaration → infix operator operator infix-operator-group opt

infix-operator-group → : precedence-group-name
```

```
GRAMMAR OF A PRECEDENCE GROUP DECLARATION
precedence-group-declaration → precedencegroup <u>precedence-group-name</u> {
    precedence-group-attributes opt }
precedence-group-attributes → <u>precedence-group-attribute</u> <u>precedence-group-attributes</u>
precedence-group-attribute → <u>precedence-group-relation</u>
precedence-group-attribute → <u>precedence-group-assignment</u>
precedence-group-attribute → precedence-group-associativity
precedence-group-relation \rightarrow higherThan : precedence-group-names
precedence-group-relation → lowerThan : <u>precedence-group-names</u>
precedence-group-assignment \rightarrow assignment : <u>boolean-literal</u>
precedence-group-associativity → associativity : left
precedence-group-associativity 	o associativity : right
precedence-group-associativity → associativity : none
precedence-group-name → <u>precedence-group-name</u> | <u>precedence-group-name</u> ,
    precedence-group-names
precedence-group-name → <u>identifier</u>
```

```
declaration-modifier → class | convenience | dynamic | final | infix | lazy |
optional | override | postfix | prefix | required | static | unowned |
unowned ( safe ) | unowned ( unsafe ) | weak
declaration-modifier → access-level-modifier
declaration-modifier → mutation-modifier
```

```
declaration-modifiers → declaration-modifier declaration-modifiers opt

access-level-modifier → private | private ( set )

access-level-modifier → fileprivate | fileprivate ( set )

access-level-modifier → internal | internal ( set )

access-level-modifier → public | public ( set )

access-level-modifier → open | open ( set )

mutation-modifier → mutating | nonmutating
```

Attributes

```
GRAMMAR OF AN ATTRIBUTE

attribute \rightarrow @ attribute-name attribute-argument-clause opt

attribute-name \rightarrow identifier

attribute-argument-clause \rightarrow ( balanced-tokens opt )

attributes \rightarrow attribute attributes opt

balanced-tokens \rightarrow balanced-token balanced-tokens opt

balanced-token \rightarrow ( balanced-tokens opt )

balanced-token \rightarrow [ balanced-tokens opt ]

balanced-token \rightarrow { balanced-tokens opt }

balanced-token \rightarrow Any identifier, keyword, literal, or operator balanced-token \rightarrow Any punctuation except (, ), [, ], {, or }
```

Patterns

```
grammar of a pattern
wildcard-pattern type-annotation opt

pattern → identifier-pattern type-annotation opt

pattern → value-binding-pattern

pattern → tuple-pattern type-annotation opt

pattern → enum-case-pattern

pattern → optional-pattern

pattern → type-casting-pattern

pattern → expression-pattern
```

```
GRAMMAR OF A WILDCARD PATTERN

wildcard-pattern → _
```

```
GRAMMAR OF AN IDENTIFIER PATTERN
identifier-pattern → identifier
GRAMMAR OF A VALUE-BINDING PATTERN
value-binding-pattern → var <u>pattern</u> | let <u>pattern</u>
GRAMMAR OF A TUPLE PATTERN
tuple-pattern → ( tuple-pattern-element-list opt )
tuple-pattern-element-list → tuple-pattern-element | tuple-pattern-element , tuple-
   pattern-element-list
tuple-pattern-element → <u>pattern</u> | <u>identifier</u> : <u>pattern</u>
GRAMMAR OF AN ENUMERATION CASE PATTERN
enum-case-pattern → <u>type-identifier</u> <sub>Opt</sub> · <u>enum-case-name</u> <u>tuple-pattern</u> <sub>Opt</sub>
GRAMMAR OF AN OPTIONAL PATTERN
optional-pattern → <u>identifier-pattern</u> ?
GRAMMAR OF A TYPE CASTING PATTERN
type-casting-pattern → <u>is-pattern</u> | <u>as-pattern</u>
is-pattern → is <u>type</u>
as-pattern → <u>pattern</u> as <u>type</u>
GRAMMAR OF AN EXPRESSION PATTERN
expression-pattern → <u>expression</u>
```

Generic Parameters and Arguments

```
GRAMMAR OF A GENERIC PARAMETER CLAUSE

generic-parameter-clause → < generic-parameter-list >
generic-parameter-list → generic-parameter | generic-parameter , generic-parameter-

list
generic-parameter → type-name
generic-parameter → type-name : type-identifier
generic-parameter → type-name : protocol-composition-type
```

```
\begin{array}{lll} \textit{generic-where-clause} & \rightarrow & \textbf{where} & \textit{requirement-list} \\ \textit{requirement-list} & \rightarrow & \textit{requirement} & | & \textit{requirement-list} \\ \textit{requirement} & \rightarrow & & \textit{conformance-requirement} & | & \textit{same-type-requirement} \\ \textit{conformance-requirement} & \rightarrow & \textit{type-identifier} & : & \textit{type-identifier} \\ \textit{conformance-requirement} & \rightarrow & \textit{type-identifier} & : & \textit{protocol-composition-type} \\ \textit{same-type-requirement} & \rightarrow & & \textit{type-identifier} & = & & \textit{type-identifier} \\ \end{array}
```

```
GRAMMAR OF A GENERIC ARGUMENT CLAUSE
```

 $\begin{array}{ll} \textit{generic-argument-clause} & \rightarrow & < & \underline{\textit{generic-argument-list}} & > \\ \textit{generic-argument-list} & \rightarrow & \underline{\textit{generic-argument}} & | & \underline{\textit{generic-argument}} & | & \underline{\textit{generic-argument-list}} \\ \textit{generic-argument} & \rightarrow & \underline{\textit{type}} \\ \end{array}$

Revision History

Document Revision History

- Updated for Swift 5.1.
- Added information about functions that specify a protocol that their return value conforms to, instead of providing a specific named return type, to the Opaque Types chapter.
- Added information about property wrappers to the <u>Property Wrappers</u> section.
- Added information enumerations and structures that are frozen for library evolution to the <u>frozen</u> section.
- Added the <u>Functions With an Implicit Return</u> and <u>Shorthand Getter</u>

 <u>Declaration sections with information about functions that omit return.</u>
- Added information about using subscripts on types to the <u>Type Subscripts</u> section.
- Updated the <u>Enumeration Case Pattern</u> section, now that an enumeration case pattern can match an optional value.
- Updated the <u>Memberwise Initializers for Structure Types</u> section, now that memberwise initializers support omitting parameters for properties that have a default value.
- Added information about dynamic members that are looked up by key path at run time to the dynamicMemberLookup section.
- Added macCatalyst to the list of target environments in <u>Conditional</u> <u>Compilation Block</u>.
- Updated the <u>Self Type</u> section, now that Self can be used to refer to the type introduced by the current class, structure, or enumeration declaration.

- Updated for Swift 5.0.
- Added the <u>Extended String Delimiters</u> section and updated the <u>String Literals</u> section with information about extended string delimiters.
- Added the <u>dynamicCallable</u> section with information about dynamically calling instances as functions using the dynamicCallable attribute.
- Added the <u>unknown</u> and <u>Switching Over Future Enumeration Cases</u> sections with information about handling future enumeration cases in switch statements using the <u>unknown</u> switch case attribute.
- Added information about the identity key path (\.self) to the <u>Key-Path</u> <u>Expression</u> section.
- Added information about using the less than (<) operator in platform conditions to the <u>Conditional Compilation Block</u> section.

- Updated for Swift 4.2.
- Added information about accessing all of an enumeration's cases to the Iterating over Enumeration Cases section.
- Added information about #error and #warning to the Compile-Time Diagnostic Statement section.
- Added information about inlining to the <u>Declaration Attributes</u> section under the inlinable and usableFromInline attributes.
- Added information about members that are looked up by name at runtime to the <u>Declaration Attributes</u> section under the dynamicMemberLookup attribute.
- Added information about the requires_stored_property_inits and warn_unqualified_access attributes to the <u>Declaration Attributes</u> section.

- Added information about how to conditionally compile code depending on the Swift compiler version being used to the <u>Conditional Compilation</u> <u>Block</u> section.
- Added information about #dsohandle to the <u>Literal Expression</u> section.

- Updated for Swift 4.1.
- Added information about synthesized implementations of equivalence operators to the <u>Equivalence Operators</u> section.
- Added information about conditional protocol conformance to the
 <u>Extension Declaration</u> section of the <u>Declarations</u> chapter, and to the
 <u>Conditionally Conforming to a Protocol</u> section of the <u>Protocols</u> chapter.
- Added information about recursive protocol constraints to the <u>Using a Protocol in Its Associated Type's Constraints</u> section.
- Added information about the canImport() and targetEnvironment() platform conditions to <u>Conditional Compilation Block</u>.

2017-12-04

- Updated for Swift 4.0.3.
- Updated the <u>Key-Path Expression</u> section, now that key paths support subscript components.

- Updated for Swift 4.0.
- Added information about exclusive access to memory to the <u>Memory Safety</u> chapter.
- Added the <u>Associated Types with a Generic Where Clause</u> section, now that you can use generic where clauses to constrain associated types.

- Added information about multiline string literals to the <u>String Literals</u> section of the <u>Strings and Characters</u> chapter, and to the <u>String Literals</u> section of the <u>Lexical Structure</u> chapter.
- Updated the discussion of the objc attribute in <u>Declaration Attributes</u>, now that this attribute is inferred in fewer places.
- Added the Generic Subscripts section, now that subscripts can be generic.
- Updated the discussion in the <u>Protocol Composition</u> section of the <u>Protocols</u> chapter, and in the <u>Protocol Composition Type</u> section of the <u>Types</u> chapter, now that protocol composition types can contain a superclass requirement.
- Updated the discussion of protocol extensions in <u>Extension Declaration</u> now that final isn't allowed in them.
- Added information about preconditions and fatal errors to the <u>Assertions</u> and <u>Preconditions</u> section.

- Updated for Swift 3.1.
- Added the <u>Extensions with a Generic Where Clause</u> section with information about extensions that include requirements.
- Added examples of iterating over a range to the <u>For-In Loops</u> section.
- Added an example of failable numeric conversions to the <u>Failable Initializers</u> section.
- Added information to the <u>Declaration Attributes</u> section about using the available attribute with a Swift language version.
- Updated the discussion in the <u>Function Type</u> section to note that argument labels are not allowed when writing a function type.
- Updated the discussion of Swift language version numbers in the <u>Conditional Compilation Block</u> section, now that an optional patch number

is allowed.

- Updated the discussion in the <u>Function Type</u> section, now that Swift distinguishes between functions that take multiple parameters and functions that take a single parameter of a tuple type.
- Removed the Dynamic Type Expression section from the <u>Expressions</u> chapter, now that type(of:) is a Swift standard library function.

2016-10-27

- Updated for Swift 3.0.1.
- Updated the discussion of weak and unowned references in the <u>Automatic</u> <u>Reference Counting</u> chapter.
- Added information about the unowned, unowned(safe), and unowned(unsafe) declaration modifiers in the Declaration Modifiers section.
- Added a note to the <u>Type Casting for Any and AnyObject</u> section about using an optional value when a value of type Any is expected.
- Updated the <u>Expressions</u> chapter to separate the discussion of parenthesized expressions and tuple expressions.

- Updated for Swift 3.0.
- Updated the discussion of functions in the <u>Functions</u> chapter and the <u>Function Declaration</u> section to note that all parameters get an argument label by default.
- Updated the discussion of operators in the <u>Advanced Operators</u> chapter, now that you implement them as type methods instead of as global functions.
- Added information about the open and fileprivate access-level modifiers to the <u>Access Control</u> chapter.

- Updated the discussion of inout in the <u>Function Declaration</u> section to note that it appears in front of a parameter's type instead of in front of a parameter's name.
- Updated the discussion of the @noescape and @autoclosure attributes in the <u>Escaping Closures</u> and <u>Autoclosures</u> sections and the <u>Attributes</u> chapter now that they are type attributes, rather than declaration attributes.
- Added information about operator precedence groups to the <u>Precedence for Custom Infix Operators</u> section of the <u>Advanced Operators</u> chapter, and to the <u>Precedence Group Declaration</u> section of the <u>Declarations</u> chapter.
- Updated discussion throughout to use macOS instead of OS X, Error instead of ErrorProtocol, and protocol names such as ExpressibleByStringLiteral instead of StringLiteralConvertible.
- Updated the discussion in the <u>Generic Where Clauses</u> section of the <u>Generics</u> chapter and in the <u>Generic Parameters and Arguments</u> chapter, now that generic where clauses are written at the end of a declaration.
- Updated the discussion in the <u>Escaping Closures</u> section, now that closures are nonescaping by default.
- Updated the discussion in the <u>Optional Binding</u> section of the <u>The Basics</u> chapter and the <u>While Statement</u> section of the <u>Statements</u> chapter, now that if, while, and guard statements use a comma-separated list of conditions without where clauses.
- Added information about switch cases that have multiple patterns to the <u>Switch</u> section of the <u>Control Flow</u> chapter and the <u>Switch Statement</u> section of the <u>Statements</u> chapter.
- Updated the discussion of function types in the <u>Function Type</u> section now that function argument labels are no longer part of a function's type.
- Updated the discussion of protocol composition types in the <u>Protocol</u>
 <u>Composition</u> section of the <u>Protocols</u> chapter and in the <u>Protocol</u>
 <u>Composition Type</u> section of the <u>Types</u> chapter to use the new <u>Protocol1</u> & <u>Protocol2</u> syntax.

- Updated the discussion in the Dynamic Type Expression section to use the new type(of:) syntax for dynamic type expressions.
- Updated the discussion of line control statements to use the #sourceLocation(file:line:) syntax in the <u>Line Control Statement</u> section.
- Updated the discussion in <u>Functions that Never Return</u> to use the new Never type.
- Added information about playground literals to the <u>Literal Expression</u> section.
- Updated the discussion in the <u>In-Out Parameters</u> section to note that only nonescaping closures can capture in-out parameters.
- Updated the discussion about default parameters in the <u>Default Parameter</u> <u>Values</u> section, now that they can't be reordered in function calls.
- Updated attribute arguments to use a colon in the <u>Attributes</u> chapter.
- Added information about throwing an error inside the catch block of a rethrowing function to the <u>Rethrowing Functions and Methods</u> section.
- Added information about accessing the selector of an Objective-C property's getter or setter to the <u>Selector Expression</u> section.
- Added information to the <u>Type Alias Declaration</u> section about generic type aliases and using type aliases inside of protocols.
- Updated the discussion of function types in the <u>Function Type</u> section to note that parentheses around the parameter types are required.
- Updated the <u>Attributes</u> chapter to note that the @IBAction, @IBOutlet, and @NSManaged attributes imply the @objc attribute.
- Added information about the @GKInspectable attribute to the <u>Declaration</u>
 <u>Attributes</u> section.

- Updated the discussion of optional protocol requirements in the <u>Optional Protocol Requirements</u> section to clarify that they are used only in code that interoperates with Objective-C.
- Removed the discussion of explicitly using let in function parameters from the Function Declaration section.
- Removed the discussion of the Boolean protocol from the <u>Statements</u> chapter, now that the protocol has been removed from the Swift standard library.
- Corrected the discussion of the @NSApplicationMain attribute in the Declaration Attributes section.

- Updated for Swift 2.2.
- Added information about how to conditionally compile code depending on the version of Swift being used to the <u>Conditional Compilation Block</u> section.
- Added information about how to distinguish between methods or initializers whose names differ only by the names of their arguments to the <u>Explicit Member Expression</u> section.
- Added information about the #selector syntax for Objective-C selectors to the <u>Selector Expression</u> section.
- Updated the discussion of associated types to use the associated type keyword in the <u>Associated Types</u> and <u>Protocol Associated Type Declaration</u> sections.
- Updated information about initializers that return nil before the instance is fully initialized in the <u>Failable Initializers</u> section.
- Added information about comparing tuples to the <u>Comparison Operators</u> section.

- Added information about using keywords as external parameter names to the <u>Keywords and Punctuation</u> section.
- Updated the discussion of the <code>@objc</code> attribute in the <u>Declaration Attributes</u> section to note that enumerations and enumeration cases can use this attribute.
- Updated the <u>Operators</u> section with discussion of custom operators that contain a dot.
- Added a note to the <u>Rethrowing Functions and Methods</u> section that rethrowing functions can't directly throw errors.
- Added a note to the <u>Property Observers</u> section about property observers being called when you pass a property as an in-out parameter.
- Added a section about error handling to the <u>A Swift Tour</u> chapter.
- Updated figures in the <u>Weak References</u> section to show the deallocation process more clearly.
- Removed discussion of C-style for loops, the ++ prefix and postfix operators, and the -- prefix and postfix operators.
- Removed discussion of variable function arguments and the special syntax for curried functions.

2015-10-20

- Updated for Swift 2.1.
- Updated the <u>String Interpolation</u> and <u>String Literals</u> sections now that string interpolations can contain string literals.
- Added the <u>Escaping Closures</u> section with information about the @noescape attribute.
- Updated the <u>Declaration Attributes</u> and <u>Conditional Compilation Block</u> sections with information about tvOS.

- Added information about the behavior of in-out parameters to the <u>In-Out</u> <u>Parameters</u> section.
- Added information to the <u>Capture Lists</u> section about how values specified in closure capture lists are captured.
- Updated the <u>Accessing Properties Through Optional Chaining</u> section to clarify how assignment through optional chaining behaves.
- Improved the discussion of autoclosures in the <u>Autoclosures</u> section.
- Added an example that uses the ?? operator to the <u>A Swift Tour</u> chapter.

- Updated for Swift 2.0.
- Added information about error handling to the <u>Error Handling</u> chapter, the <u>Do Statement</u> section, the <u>Throw Statement</u> section, the <u>Defer Statement</u> section, and the <u>Try Operator</u> section.
- Updated the <u>Representing and Throwing Errors</u> section, now that all types can conform to the ErrorType protocol.
- Added information about the new try? keyword to the <u>Converting Errors to</u> <u>Optional Values</u> section.
- Added information about recursive enumerations to the <u>Recursive</u>
 <u>Enumerations</u> section of the <u>Enumerations</u> chapter and the <u>Enumerations</u> with <u>Cases of Any Type</u> section of the <u>Declarations</u> chapter.
- Added information about API availability checking to the <u>Checking API</u>
 <u>Availability</u> section of the <u>Control Flow</u> chapter and the <u>Availability</u>
 <u>Condition</u> section of the <u>Statements</u> chapter.
- Added information about the new guard statement to the <u>Early Exit</u> section of the <u>Control Flow</u> chapter and the <u>Guard Statement</u> section of the <u>Statements</u> chapter.

- Added information about protocol extensions to the <u>Protocol Extensions</u> section of the <u>Protocols</u> chapter.
- Added information about access control for unit testing to the <u>Access</u> <u>Levels for Unit Test Targets</u> section of the <u>Access Control</u> chapter.
- Added information about the new optional pattern to the <u>Optional Pattern</u> section of the <u>Patterns</u> chapter.
- Updated the <u>Repeat-While</u> section with information about the repeat-while loop.
- Updated the <u>Strings and Characters</u> chapter, now that String no longer conforms to the CollectionType protocol from the Swift standard library.
- Added information about the new Swift standard library print(_:separator:terminator) function to the <u>Printing Constants and</u> Variables section.
- Added information about the behavior of enumeration cases with String raw values to the <u>Implicitly Assigned Raw Values</u> section of the <u>Enumerations</u> chapter and the <u>Enumerations with Cases of a Raw-Value</u> <u>Type</u> section of the <u>Declarations</u> chapter.
- Added information about the @autoclosure attribute—including its @autoclosure(escaping) form—to the <u>Autoclosures</u> section.
- Updated the <u>Declaration Attributes</u> section with information about the @available and @warn_unused_result attributes.
- Updated the <u>Type Attributes</u> section with information about the @convention attribute.
- Added an example of using multiple optional bindings with a where clause to the Optional Binding section.
- Added information to the <u>String Literals</u> section about how concatenating string literals using the + operator happens at compile time.

- Added information to the <u>Metatype Type</u> section about comparing metatype values and using them to construct instances with initializer expressions.
- Added a note to the <u>Debugging with Assertions</u> section about when user-defined assertions are disabled.
- Updated the discussion of the @NSManaged attribute in the <u>Declaration</u>

 <u>Attributes</u> section, now that the attribute can be applied to certain instance methods.
- Updated the <u>Variadic Parameters</u> section, now that variadic parameters can be declared in any position in a function's parameter list.
- Added information to the <u>Overriding a Failable Initializer</u> section about how a nonfailable initializer can delegate up to a failable initializer by force-unwrapping the result of the superclass's initializer.
- Added information about using enumeration cases as functions to the Enumerations with Cases of Any Type section.
- Added information about explicitly referencing an initializer to the <u>Initializer Expression</u> section.
- Added information about build configuration and line control statements to the <u>Compiler Control Statements</u> section.
- Added a note to the <u>Metatype Type</u> section about constructing class instances from metatype values.
- Added a note to the <u>Weak References</u> section about weak references being unsuitable for caching.
- Updated a note in the <u>Type Properties</u> section to mention that stored type properties are lazily initialized.
- Updated the <u>Capturing Values</u> section to clarify how variables and constants are captured in closures.
- Updated the **Declaration Attributes** section to describe when you can apply

the @objc attribute to classes.

- Added a note to the <u>Handling Errors</u> section about the performance of executing a throw statement. Added similar information about the do statement in the <u>Do Statement</u> section.
- Updated the <u>Type Properties</u> section with information about stored and computed type properties for classes, structures, and enumerations.
- Updated the <u>Break Statement</u> section with information about labeled break statements.
- Updated a note in the <u>Property Observers</u> section to clarify the behavior of willset and didset observers.
- Added a note to the <u>Access Levels</u> section with information about the scope of private access.
- Added a note to the <u>Weak References</u> section about the differences in weak references between garbage collected systems and ARC.
- Updated the <u>Special Characters in String Literals</u> section with a more precise definition of Unicode scalars.

2015-04-08

- Updated for Swift 1.2.
- Swift now has a native Set collection type. For more information, see <u>Sets</u>.
- @autoclosure is now an attribute of the parameter declaration, not its type. There is also a new @noescape parameter declaration attribute. For more information, see <u>Declaration Attributes</u>.
- Type methods and properties now use the static keyword as a declaration modifier. For more information see Type Variable Properties.
- Swift now includes the as? and as! failable downcast operators. For more information, see Checking for Protocol Conformance.

- Added a new guide section about <u>String Indices</u>.
- Removed the overflow division (&/) and overflow remainder (&%) operators from Overflow Operators.
- Updated the rules for constant and constant property declaration and initialization. For more information, see <u>Constant Declaration</u>.
- Updated the definition of Unicode scalars in string literals. See <u>Special</u> <u>Characters in String Literals</u>.
- Updated <u>Range Operators</u> to note that a half-open range with the same start and end index will be empty.
- Updated <u>Closures Are Reference Types</u> to clarify the capturing rules for variables.
- Updated <u>Value Overflow</u> to clarify the overflow behavior of signed and unsigned integers
- Updated <u>Protocol Declaration</u> to clarify protocol declaration scope and members.
- Updated <u>Defining a Capture List</u> to clarify the syntax for weak and unowned references in closure capture lists.
- Updated <u>Operators</u> to explicitly mention examples of supported characters for custom operators, such as those in the Mathematical Operators, Miscellaneous Symbols, and Dingbats Unicode blocks.
- Constants can now be declared without being initialized in local function scope. They must have a set value before first use. For more information, see Constant Declaration.
- In an initializer, constant properties can now only assign a value once. For more information, see <u>Assigning Constant Properties During Initialization</u>.
- Multiple optional bindings can now appear in a single if statement as a comma-separated list of assignment expressions. For more information, see

Optional Binding.

- An Optional-Chaining Expression must appear within a postfix expression.
- Protocol casts are no longer limited to @objc protocols.
- Type casts that can fail at runtime now use the as? or as! operator, and type casts that are guaranteed not to fail use the as operator. For more information, see Type-Casting Operators.

2014-10-16

- Updated for Swift 1.1.
- Added a full guide to <u>Failable Initializers</u>.
- Added a description of <u>Failable Initializer Requirements</u> for protocols.
- Constants and variables of type Any can now contain function instances. Updated the example in Type Casting for Any and AnyObject to show how to check for and cast to a function type within a switch statement.
- Enumerations with raw values now have a rawValue property rather than a toRaw() method and a failable initializer with a rawValue parameter rather than a fromRaw() method. For more information, see Raw Values and Enumerations with Cases of a Raw-Value Type.
- Added a new reference section about <u>Failable Initializers</u>, which can trigger initialization failure.
- Custom operators can now contain the ? character. Updated the <u>Operators</u> reference to describe the revised rules. Removed a duplicate description of the valid set of operator characters from <u>Custom Operators</u>.

2014-08-18

• New document that describes Swift 1.0, Apple's new programming language for building iOS and OS X apps.

- Added a new section about <u>Initializer Requirements</u> in protocols.
- Added a new section about <u>Class-Only Protocols</u>.
- <u>Assertions and Preconditions</u> can now use string interpolation. Removed a note to the contrary.
- Updated the <u>Concatenating Strings and Characters</u> section to reflect the fact that String and Character values can no longer be combined with the addition operator (+) or addition assignment operator (+=). These operators are now used only with String values. Use the String type's append(_:) method to append a single Character value onto the end of a string.
- Added information about the availability attribute to the <u>Declaration</u> Attributes section.
- Optionals no longer implicitly evaluate to true when they have a value and false when they do not, to avoid confusion when working with optional Bool values. Instead, make an explicit check against nil with the == or != operators to find out if an optional contains a value.
- Swift now has a <u>Nil-Coalescing Operator</u> (a ?? b), which unwraps an optional's value if it exists, or returns a default value if the optional is nil.
- Updated and expanded the <u>Comparing Strings</u> section to reflect and demonstrate that string and character comparison and prefix / suffix comparison are now based on Unicode canonical equivalence of extended grapheme clusters.
- You can now try to set a property's value, assign to a subscript, or call a
 mutating method or operator through <u>Optional Chaining</u>. The information
 about <u>Accessing Properties Through Optional Chaining</u> has been updated
 accordingly, and the examples of checking for method call success in
 <u>Calling Methods Through Optional Chaining</u> have been expanded to show
 how to check for property setting success.
- Added a new section about <u>Accessing Subscripts of Optional Type</u> through optional chaining.

- Updated the <u>Accessing and Modifying an Array</u> section to note that you can no longer append a single item to an array with the += operator. Instead, use the append(_:) method, or append a single-item array with the += operator.
- Added a note that the start value a for the Range Operators a...b and a..
b must not be greater than the end value b.
- Rewrote the <u>Inheritance</u> chapter to remove its introductory coverage of initializer overrides. This chapter now focuses more on the addition of new functionality in a subclass, and the modification of existing functionality with overrides. The chapter's example of <u>Overriding Property Getters and Setters</u> has been rewritten to show how to override a description property. (The examples of modifying an inherited property's default value in a subclass initializer have been moved to the <u>Initialization</u> chapter.)
- Updated the <u>Initializer Inheritance and Overriding</u> section to note that overrides of a designated initializer must now be marked with the override modifier.
- Updated the <u>Required Initializers</u> section to note that the required modifier is now written before every subclass implementation of a required initializer, and that the requirements for required initializers can now be satisfied by automatically inherited initializers.
- Infix Operator Methods no longer require the @infix attribute.
- The <code>@prefix</code> and <code>@postfix</code> attributes for <u>Prefix</u> and <u>Postfix</u> Operators have been replaced by <code>prefix</code> and <code>postfix</code> declaration modifiers.
- Added a note about the order in which <u>Prefix and Postfix Operators</u> are applied when both a prefix and a postfix operator are applied to the same operand.
- Operator functions for <u>Compound Assignment Operators</u> no longer use the <code>@assignment</code> attribute when defining the function.
- The order in which modifiers are specified when defining <u>Custom</u> <u>Operators</u> has changed. You now write prefix operator rather than operator prefix, for example.

- Added information about the dynamic declaration modifier in <u>Declaration</u> Modifiers.
- Added information about how type inference works with <u>Literals</u>.
- Added more information about curried functions.
- Added a new chapter about <u>Access Control</u>.
- Updated the <u>Strings and Characters</u> chapter to reflect the fact that Swift's character type now represents a single Unicode extended grapheme cluster. Includes a new section on <u>Extended Grapheme Clusters</u> and more information about <u>Unicode Scalar Values</u> and <u>Comparing Strings</u>.
- Updated the <u>String Literals</u> section to note that Unicode scalars inside string literals are now written as \u{n}, where n is a hexadecimal number between 0 and 10FFFF, the range of Unicode's codespace.
- The NSString length property is now mapped onto Swift's native String type as utf16Count, not utf16count.
- Swift's native String type no longer has an uppercaseString or lowercaseString property. The corresponding section in <u>Strings and Characters</u> has been removed, and various code examples have been updated.
- Added a new section about <u>Initializer Parameters Without Argument</u> Labels.
- Added a new section about <u>Required Initializers</u>.
- Added a new section about <u>Optional Tuple Return Types</u>.
- Updated the <u>Type Annotations</u> section to note that multiple related variables can be defined on a single line with one type annotation.
- The @optional, @lazy, @final, and @required attributes are now the optional, lazy, final, and required <u>Declaration Modifiers</u>.

- Updated the entire book to refer to ... < as the <u>Half-Open Range Operator</u> (rather than the "half-closed range operator").
- Updated the <u>Accessing and Modifying a Dictionary</u> section to note that Dictionary now has a Boolean is Empty property.
- Clarified the full list of characters that can be used when defining <u>Custom Operators</u>.
- nil and the Booleans true and false are now Literals.
- Swift's Array type now has full value semantics. Updated the information about <u>Mutability of Collections</u> and <u>Arrays</u> to reflect the new approach. Also clarified the assignment and copy behavior for strings arrays and dictionaries.
- <u>Array Type Shorthand Syntax</u> is now written as [SomeType] rather than SomeType[].
- Added a new section about <u>Dictionary Type Shorthand Syntax</u>, which is written as [KeyType: ValueType].
- Added a new section about <u>Hash Values for Set Types</u>.
- Examples of <u>Closure Expressions</u> now use the global sorted(_:_:) function rather than the global sort(_:_:) function, to reflect the new array value semantics.
- Updated the information about <u>Memberwise Initializers for Structure Types</u> to clarify that the memberwise structure initializer is made available even if a structure's stored properties do not have default values.
- Updated to ... < rather than ... for the <u>Half-Open Range Operator</u>.
- Added an example of <u>Extending a Generic Type</u>.

Copyright and Notices

Apple Inc. Copyright © 2019 Apple Inc.

This document is made available under a Creative Commons Attribution 4.0 International (CC BY 4.0) License: https://creativecommons.org/licenses/by/4.0/

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document.

Apple Inc.
One Apple Park Way
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Bonjour, Cocoa, Logic, Mac, Numbers, Objective-C, OS X, Retina, Sand, Shake, and Xcode are trademarks of Apple Inc., registered in the U.S. and other countries.

Swift and tvOS are trademarks of Apple Inc.

IOS is a trademark or registered trademark of Cisco in the U.S. and other countries and is used under license.

Times is a registered trademark of Heidelberger Druckmaschinen AG, available from Linotype Library GmbH.