Swift

```
var people = ["Dave", "Brian", "Alex", "A
let name = "Alex"
if let index = find(people, name) {
    println("\(name) is person \(index +
    delegate?.didFindPersonWithName(name,
} else {
    println("Unable to find \(name) in th
}
```

Welcome to Swift

About Swift

Swift is a new programming language for iOS and OS X apps that builds on the best of C and Objective-C, without the constraints of C compatibility. Swift adopts safe programming patterns and adds modern features to make programming easier, more flexible, and more fun. Swift's clean slate, backed by the mature and much-loved Cocoa and Cocoa Touch frameworks, is an opportunity to reimagine how software development works.

Swift has been years in the making. Apple laid the foundation for Swift by advancing our existing compiler, debugger, and framework infrastructure. We simplified memory management with Automatic Reference Counting (ARC). Our framework stack, built on the solid base of Foundation and Cocoa, has been modernized and standardized throughout. Objective-C itself has evolved to support blocks, collection literals, and modules, enabling framework adoption of modern language technologies without disruption. Thanks to this groundwork, we can now introduce a new language for the future of Apple software development.

Swift feels familiar to Objective-C developers. It adopts the readability of Objective-C's named parameters and the power of Objective-C's dynamic object model. It provides seamless access to existing Cocoa frameworks and mix-and-match interoperability with Objective-C code. Building from this common ground, Swift introduces many new features and unifies the procedural and object-oriented portions of the language.

Swift is friendly to new programmers. It is the first industrial-quality systems programming language that is as expressive and enjoyable as a scripting language. It supports playgrounds, an innovative feature that allows programmers to experiment with Swift code and see the results immediately, without the overhead of building and running an app.

Swift combines the best in modern language thinking with wisdom from the wider Apple engineering culture. The compiler is optimized for performance, and the language is optimized for development, without compromising on either. It's designed to scale from "hello, world" to an entire operating system. All this makes Swift a sound future investment for developers and for Apple.

Swift is a fantastic way to write iOS and OS X apps, and will continue to evolve with new features and capabilities. Our goals for Swift are ambitious. We can't wait to see what you create with it.

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:

```
1  println("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
>
> For the best experience, open this chapter as a playground in Xcode. Playgrounds allow you to edit the code listings and see the result immediately.

Simple Values

Use `let` to make a constant and `var` to make a variable. The value of a constant doesn’t need to be known at compile time, but you must assign it a value exactly once. 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
```

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 a 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.

```
let implicitInteger = 70
let implicitDouble = 70.0
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.

```
let label = "The width is "
let width = 94
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:

```
let apples = 3
let oranges = 5
let appleSummary = "I have \(apples) apples."
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.

Create arrays and dictionaries using brackets (`[]`), and access their elements by writing the index or key in brackets.

```
var shoppingList = ["catfish", "water", "tulips", "blue paint"]
shoppingList[1] = "bottle of water"

var occupations = [
    "Malcolm": "Captain",
    "Kaylee": "Mechanic",
]
occupations["Jayne"] = "Public Relations"
```

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

```
let emptyArray = String[]()
let emptyDictionary = Dictionary<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 = []   // Went shopping and bought everything.
```

Control Flow

Use `if` and `switch` to make conditionals, and use `for-in`, `for`, `while`, and `do-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]
var teamScore = 0
for score in individualScores {
    if score > 50 {
        teamScore += 3
    } else {
        teamScore += 1
    }
}
   10  teamScore
```

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 the value is missing. Write a question mark (`?`) after the type of a value to mark the value as optional.

```
var optionalString: String? = "Hello"
```

```
optionalString == nil

var optionalName: String? = "John Appleseed"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

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.

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

```
let vegetable = "red pepper"
switch vegetable {
case "celery":
    let vegetableComment = "Add some raisins and make ants on a log."
case "cucumber", "watercress":
    let vegetableComment = "That would make a good tea sandwich."
case let x where x.hasSuffix("pepper"):
    let vegetableComment = "Is it a spicy \(x)?"
default:
    let vegetableComment = "Everything tastes good in soup."
}
```

EXPERIMENT

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

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.

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

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.

```
var n = 2
while n < 100 {
    n = n * 2
}
n

var m = 2
do {
    m = m * 2
} while m < 100
m
```

You can keep an index in a loop—either by using `..` to make a range of indexes or by writing an explicit initialization, condition, and increment. These two loops do the same thing:

```
var firstForLoop = 0
for i in 0..3 {
    firstForLoop += i
}
firstForLoop

var secondForLoop = 0
for var i = 0; i < 3; ++i {
    secondForLoop += 1
}
secondForLoop
```

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

EXPERIMENT

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

Use a tuple to return multiple values from a function.

```
func getGasPrices() -> (Double, Double, Double) {
    return (3.59, 3.69, 3.79)
}
getGasPrices()
```

Functions can also take a variable number of arguments, collecting them into an array.

```
func sumOf(numbers: Int...) -> Int {
    var sum = 0
    for number in numbers {
        sum += number
    }
    return sum
}
sumOf()
sumOf(42, 597, 12)
```

EXPERIMENT

Write a function that calculates the average of its arguments.

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.

```
func hasAnyMatches(list: Int[], condition: Int -> Bool) -> Bool {
```

```
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}
func lessThanTen(number: Int) -> Bool {
    return number < 10
}
var numbers = [20, 19, 7, 12]
hasAnyMatches(numbers, lessThanTen)
```

Functions are actually a special case of closures. 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.

```
numbers.map({ number in 3 * number })
```

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.

```
sort([1, 5, 3, 12, 2]) { $0 > $1 }
```

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."
    }
}
```

EXPERIMENT

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 {
    var numberOfSides: Int = 0
    var name: String

    init(name: String) {
        self.name = name
    }

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

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.

```
class Square: NamedShape {
    var sideLength: Double

    init(sideLength: Double, name: String) {
```

```
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 4
    }

    func area() ->  Double {
        return sideLength * sideLength
    }

    override func simpleDescription() -> String {
        return "A square with sides of length \(sideLength)."
    }
}
let test = Square(sideLength: 5.2, name: "my test square")
test.area()
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 `describe` method on the `Circle` class.

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

```
class EquilateralTriangle: NamedShape {
    var sideLength: Double = 0.0

    init(sideLength: Double, name: String) {
        self.sideLength = sideLength
        super.init(name: name)
        numberOfSides = 3
    }
```

```
9
10      var perimeter: Double {
11      get {
12          return 3.0 * sideLength
13      }
14      set {
15          sideLength = newValue / 3.0
16      }
17      }
18
19      override func simpleDescription() -> String {
20          return "An equilateral triagle with sides of length \
            (sideLength)."
21      }
22  }
23  var triangle = EquilateralTriangle(sideLength: 3.1, name: "a
            triangle")
24  triangle.perimeter
25  triangle.perimeter = 9.9
26  triangle.sideLength
```

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`. 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
    }
    }
    var square: Square {
    willSet {
        triangle.sideLength = newValue.sideLength
      }
      }
      init(size: Double, name: String) {
          square = Square(sideLength: size, name: name)
          triangle = EquilateralTriangle(sideLength: size, name:
             name)
      }
}
var triangleAndSquare = TriangleAndSquare(size: 10, name:
        "another test shape")
triangleAndSquare.square.sideLength
triangleAndSquare.triangle.sideLength
triangleAndSquare.square = Square(sideLength: 50, name: "larger
        square")
triangleAndSquare.triangle.sideLength
```

Methods on classes have one important difference from functions. Parameter names in functions are used only within the function, but parameters names in methods are also used when you call the method (except for the first parameter). By default, a method has the same name for its parameters when you call it and within the method itself. You can specify a second name, which is used inside the method.

```
class Counter {
    var count: Int = 0
    func incrementBy(amount: Int, numberOfTimes times: Int) {
        count += amount * times
    }
}
```

```
var counter = Counter()
counter.incrementBy(2, numberOfTimes: 7)
```

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.

```
let optionalSquare: Square? = Square(sideLength: 2.5, name: "optional
             square")
let sideLength = optionalSquare?.sideLength
```

Enumerations and Structures

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

```
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
    func simpleDescription() -> String {
        switch self {
        case .Ace:
            return "ace"
        case .Jack:
            return "jack"
        case .Queen:
            return "queen"
        case .King:
            return "king"
        default:
            return String(self.toRaw())
```

```
            }
        }
    }
    let ace = Rank.Ace
    let aceRawValue = ace.toRaw()
```

EXPERIMENT

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

In the example above, the raw value type of the enumeration is `Int`, so you only have to specify the first raw value. 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 `toRaw` and `fromRaw` functions to convert between the raw value and the enumeration value.

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

The member 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.

```
enum Suit {
    case Spades, Hearts, Diamonds, Clubs
    func simpleDescription() -> String {
        switch self {
        case .Spades:
            return "spades"
        case .Hearts:
            return "hearts"
```

```
        case .Diamonds:
            return "diamonds"
        case .Clubs:
            return "clubs"
        }
    }
}
let hearts = Suit.Hearts
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` member of the enumeration is referred to above: When assigning a value to the `hearts` constant, the enumeration member `Suit.Hearts` is referred to by its full name because the constant doesn’t have an explicit type specified. Inside the switch, the enumeration 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.

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.

```
struct Card {
    var rank: Rank
    var suit: Suit
    func simpleDescription() -> String {
        return "The \(rank.simpleDescription()) of \
               (suit.simpleDescription())"
    }
```

```
}
let threeOfSpades = Card(rank: .Three, suit: .Spades)
let threeOfSpadesDescription = threeOfSpades.simpleDescription()
```

EXPERIMENT

Add a method to `Card` that creates a full deck of cards, with one card of each combination of rank and suit.

An instance of an enumeration member can have values associated with the instance. Instances of the same enumeration member can have different values associated with them. You provide the associated values when you create the instance. Associated values and raw values are different: The raw value of an enumeration member is the same for all of its instances, and you provide the raw value when you define the enumeration.

For example, consider the case of requesting the sunrise and sunset time from a server. The server either responds with the information or it responds with some error information.

```
enum ServerResponse {
    case Result(String, String)
    case Error(String)
}

let success = ServerResponse.Result("6:00 am", "8:09 pm")
let failure = ServerResponse.Error("Out of cheese.")

switch success {
    case let .Result(sunrise, sunset):
        let serverResponse = "Sunrise is at \(sunrise) and sunset
            is at \(sunset)."
    case let .Error(error):
        let serverResponse = "Failure...  \(error)"
    }
```

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.

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.

```
class SimpleClass: ExampleProtocol {
    var simpleDescription: String = "A very simple class."
    var anotherProperty: Int = 69105
    func adjust() {
        simpleDescription += "  Now 100% adjusted."
    }
}
var a = SimpleClass()
a.adjust()
let aDescription = a.simpleDescription

struct SimpleStructure: ExampleProtocol {
```

```
    var simpleDescription: String = "A simple structure"
    mutating func adjust() {
        simpleDescription += " (adjusted)"
    }
}
var b = SimpleStructure()
b.adjust()
let bDescription = b.simpleDescription
```

EXPERIMENT

Write an enumeration that conforms to this 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.

```
extension Int: ExampleProtocol {
    var simpleDescription: String {
    return "The number \(self)"
    }
    mutating func adjust() {
        self += 42
    }
}
7.simpleDescription
```

EXPERIMENT

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.

```
let protocolValue: ExampleProtocol = a
protocolValue.simpleDescription
// protocolValue.anotherProperty  // Uncomment to see the error
```

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.

Generics

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

```
func repeat<ItemType>(item: ItemType, times: Int) -> ItemType[] {
    var result = ItemType[]()
    for i in 0..times {
        result += item
    }
    return result
}
repeat("knock", 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<T> {
    case None
    case Some(T)
}
var possibleInteger: OptionalValue<Int> = .None
possibleInteger = .Some(100)
```

Use `where` after the type name 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.

```
func anyCommonElements <T, U where T: Sequence, U: Sequence,
             T.GeneratorType.Element: Equatable,
             T.GeneratorType.Element == U.GeneratorType.Element>
             (lhs: T, rhs: U) -> Bool {
    for lhsItem in lhs {
        for rhsItem in rhs {
            if lhsItem == rhsItem {
                return true
            }
        }
    }
    return false
}
anyCommonElements([1, 2, 3], [3])
```

EXPERIMENT

Modify the `anyCommonElements` function to make a function that returns an array of the elements that any two sequences have in common.

In the simple cases, you can omit `where` and simply write the protocol or class name after a colon. Writing

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

Language Guide

The Basics

Swift is a new programming language for iOS and OS X 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 two primary collection types, `Array` 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 cannot 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 do not need to change.

In addition to familiar types, Swift introduces advanced types not found in Objective-C. These include tuples, which enable you to create and pass around groupings of values. Tuples can 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". Optionals are similar to using `nil` with pointers in Objective-C, but they work for any type, not just classes. Optionals are safer and more expressive than `nil` pointers in Objective-C and are at the heart of many of Swift's most powerful features.

Optionals are an example of the fact that Swift is a *type safe* language. Swift helps you to be clear about the types of values your code can work with. If part of your code expects a `String`, type safety prevents you from passing it an `Int` by mistake. This enables you to 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* cannot be changed once it is 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 are 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:

```
let maximumNumberOfLoginAttempts = 10
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 x = 0.0, y = 0.0, z = 0.0
```

> NOTE
>
> If a stored value in your code is not going to change, always declare it as a constant with the `let` keyword. Use variables only for storing values that need to be able to change.

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"
```

> NOTE
>
> It is 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 is 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 specified with a type annotation rather than being inferred from an initial value.

Naming Constants and Variables

You can use almost any character you like for constant and variable names, including Unicode characters:

```
let π = 3.14159
let 你好 = "你好世界"
let 🐶🐮 = "dogcow"
```

Constant and variable names cannot contain mathematical symbols, arrows, private-use (or invalid) Unicode code points, 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 redeclare 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, you can do so by surrounding the keyword with back ticks (`) when using it as a name. However, you should 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 cannot be changed once it is set. Attempting to do so is reported as an

error when your code is compiled:

```
let languageName = "Swift"
languageName = "Swift++"
// 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 `println` function:

```
println(friendlyWelcome)
// prints "Bonjour!"
```

`println` is a global function that prints a value, followed by a line break, to an appropriate output. If you are working in Xcode, for example, `println` prints its output in Xcode's "console" pane. (A second function, `print`, performs the same task without appending a line break to the end of the value to be printed.)

The `println` function prints any `String` value you pass to it:

```
println("This is a string")
// prints "This is a string"
```

The `println` function can print more complex logging messages, in a similar manner to Cocoa's `NSLog` function. These messages can include the current values of constants and variables.

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:

```
println("The current value of friendlyWelcome is \(friendlyWelcome)")
// prints "The current value of friendlyWelcome is Bonjour!"
```

> NOTE
>
> All options you can use with string interpolation are described in String Interpolation.

Comments

Use comments to include non-executable 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
```

You can also write multiline comments, which 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 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 does not require you to write a semicolon (`;`) after each statement in your code, although you can do so if you wish. Semicolons *are* required, however, if you want to write multiple separate statements on a single line:

```
let cat = "□□"; println(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 is not the case, `Int` is preferred, even when the values to be stored are known to be non-negative. 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. Use it when floating-point values must be very large or particularly precise.

`Float` represents a 32-bit floating-point number. Use it when floating-point values do not require 64-bit precision.

> 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.

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 expects 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:

```
let meaningOfLife = 42
// 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`:

```
let pi = 3.14159
// 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 `Double` will be inferred from the context:

```
let anotherPi = 3 + 0.14159
// 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 `0o` prefix

A *hexadecimal* number, with a `0x` prefix

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

```
let decimalInteger = 17
let binaryInteger = 0b10001       // 17 in binary notation
let octalInteger = 0o21           // 17 in octal notation
let hexadecimalInteger = 0x11     // 17 in hexadecimal notation
```

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

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 1.25×10^{-2}, or `0.0125`.

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

`0xFp2` means 15×2^2, or `60.0`.

`0xFp-2` means 15×2^{-2}, or `3.75`.

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

```
let decimalDouble = 12.1875
let exponentDouble = 1.21875e1
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 zeroes and can contain underscores to help with readability. Neither type of formatting affects the underlying value of the literal:

```
let paddedDouble = 000123.456
let oneMillion = 1_000_000
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 are known to be non-negative. 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 are are 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 will not fit into a constant or variable of a sized integer type is reported as an error when your code is compiled:

```
let cannotBeNegative: UInt8 = -1
// UInt8 cannot store negative numbers, and so this will report an
```

```
                error
let tooBig: Int8 = Int8.max + 1
// Int8 cannot store a number larger than its maximum value,
// 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 cannot be added together directly, because they are 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:

```
let twoThousand: UInt16 = 2_000
let one: UInt8 = 1
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 is 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:

```
let three = 3
```

```
let pointOneFourOneFiveNine = 0.14159
let pi = Double(three) + pointOneFourOneFiveNine
// 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 `Double`, so that both sides of the addition are of the same type. Without this conversion in place, the addition would not be allowed.

The reverse is also true for floating-point to integer conversion, in that an integer type can be initialized with a `Double` or `Float` value:

```
let integerPi = Int(pi)
// 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 do not have an explicit type in and of themselves. Their type is inferred only at the point that they are 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 is 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`:

```
let orangesAreOrange = true
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 {
    println("Mmm, tasty turnips!")
} else {
    println("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 be substituted for `Bool`. The following example reports a compile-time error:

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

However, the alternative example below is valid:

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

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 do not 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:

```
let (statusCode, statusMessage) = http404Error
println("The status code is \(statusCode)")
// prints "The status code is 404"
println("The status message is \(statusMessage)")
// 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:

```
let (justTheStatusCode, _) = http404Error
println("The status code is \(justTheStatusCode)")
// prints "The status code is 404"
```

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

```
println("The status code is \(http404Error.0)")
// prints "The status code is 404"
println("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:

```
println("The status code is \(http200Status.statusCode)")
// prints "The status code is 200"
println("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 Functions with Multiple Return Values.

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

Optionals

You use *optionals* in situations where a value may be absent. An optional says:

There *is* a value, and it equals *x*

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 structs, 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 is 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. Swift's `String` type has a method called `toInt`, 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"` does not have an obvious numeric value to convert to.

The example below uses the `toInt` method to try to convert a `String` into an `Int`:

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

Because the `toInt` method 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.)

If Statements and Forced Unwrapping

You can use an `if` statement to find out whether an optional contains a value. If an optional does have a value, it evaluates to `true`; if it has no value at all, it evaluates to `false`.

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:

```
if convertedNumber {
    println("\(possibleNumber) has an integer value of \
            (convertedNumber!)")
} else {
    println("\(possibleNumber) could not be converted to an integer")
}
// prints "123 has an integer value of 123"
```

For more on the `if` statement, see Control Flow.

> NOTE
>
> Trying to use `!` to access a non-existent 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 optional bindings for the `if` statement as follows:

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

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

```
if let actualNumber = possibleNumber.toInt() {
    println("\(possibleNumber) has an integer value of \
            (actualNumber)")
} else {
    println("\(possibleNumber) could not be converted to an integer")
}
// prints "123 has an integer value of 123"
```

This can be read as:

"If the optional `Int` returned by `possibleNumber.toInt` 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 is 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.

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
>
> `nil` cannot be used with non-optional constants and variables. If a constant or variable in your code needs to be able to cope 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 constant or variable without providing a default value, the constant or variable is automatically set to `nil` for you:

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

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

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 is clear from a program’s structure that an optional will *always* have a value, after that value is first set. In these cases, it is useful to remove the need to check and unwrap the optional’s value every time it is 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 nonoptional value, without the need to unwrap the optional value each time it is accessed. The following example shows the difference in behavior between an optional `String` and an implicitly unwrapped optional `String`:

```
1 let possibleString: String? = "An optional string."
2 println(possibleString!) // requires an exclamation mark to access its
                value
3 // prints "An optional string."
4
5 let assumedString: String! = "An implicitly unwrapped optional
                string."
6 println(assumedString)  // no exclamation mark is needed to access its
                value
7 // prints "An implicitly unwrapped optional string."
```

You can think of an implicitly unwrapped optional as giving permission for the optional to be unwrapped automatically whenever it is 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 you try to access an implicitly unwrapped optional when it does not contain a value, you will trigger a runtime error. The result is exactly the same as if you place an exclamation mark after a normal optional that does not contain a value.

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

```
if assumedString {
    println(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 {
    println(definiteString)
}
// prints "An implicitly unwrapped optional string."
```

> NOTE
>
> Implicitly unwrapped optionals should not be used when there is 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.

Assertions

Optionals enable you to check for values that may or may not exist, and to write code that copes gracefully with the absence of a value. In some cases, however, it is simply not possible for your code to continue execution if a value does not exist, or if a provided value does not satisfy certain conditions. In these situations, you can trigger an *assertion* in your code to end code execution and to provide an opportunity to debug the cause of the absent or invalid value.

Debugging with Assertions

An assertion is a runtime check that a logical condition definitely evaluates to `true`. Literally put, an assertion "asserts" that a condition is true. You use an assertion to make sure that an essential condition is satisfied before executing any further code. If the condition evaluates to `true`, code execution continues as usual; if the condition evaluates to `false`, code execution ends, and your app is terminated.

If your code triggers an assertion while running in a debug environment, such as when you build and run an app in Xcode, you can see exactly where the invalid state occurred and query the state of your app at the time that the assertion was triggered. An assertion also lets you provide a suitable debug message as to the nature of the assert.

You write an assertion by calling the global `assert` function. You pass the `assert` function an expression that evaluates to `true` or `false` and a message that should be displayed if the result of the condition is `false`:

```
let age = -3
assert(age >= 0, "A person's age cannot be less than zero")
// this causes the assertion to trigger, because age is not >= 0
```

In this example, code execution will continue only if `age >= 0` evaluates to `true`, that is, if the value of `age` is non-negative. If the value of `age` *is* negative, as in the code above, then `age >= 0` evaluates to `false`, and the assertion is triggered, terminating the application.

Assertion messages cannot use string interpolation. The assertion message can be omitted if desired, as in the following example:

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

When to Use Assertions

Use an assertion whenever a condition has the potential to be false, but must *definitely* be true in order for your code to continue execution. Suitable scenarios for an assertion check include:

An integer subscript index is passed to a custom subscript implementation, but the subscript index value could be too low or too high.

A value is passed to a function, but an invalid value means that the function cannot fulfill its task.

An optional value is currently `nil`, but a non-`nil` value is essential for subsequent code to execute successfully.

See also Subscripts and Functions.

> NOTE
>
> Assertions cause your app to terminate and are not a substitute for designing your code in such a way that invalid conditions are unlikely to arise. Nonetheless, in situations where invalid conditions are possible, an assertion is an effective way to ensure that such conditions are highlighted and noticed during development, before your app is published.

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 together (as in `let i = 1 + 2`). More complex examples include the logical AND operator `&&` (as in `if enteredDoorCode && passedRetinaScan`) and the increment operator `++i`, which is a shortcut to increase the value of `i` by `1`.

Swift supports most standard C operators and improves several capabilities to eliminate common coding errors. The assignment operator (`=`) does not 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.

Unlike C, Swift lets you perform remainder (`%`) calculations on floating-point numbers. Swift also provides two range operators (`a..b` and `a...b`) not found in C, as a shortcut for expressing a range of values.

This chapter describes the common operators in Swift. Advanced Operators 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 `i++`).
>
> *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`:

```
let b = 10
var a = 5
a = b
// 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:

```
let (x, y) = (1, 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 x = 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 (/)

```
1 + 2       // equals 3
5 - 3       // equals 2
2 * 3       // equals 6
10.0 / 2.5  // equals 4.0
```

Unlike the arithmetic operators in C and Objective-C, the Swift arithmetic operators do not 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 `String` concatenation:

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

Two `Character` values, or one `Character` value and one `String` value, can be added together to make a new `String` value:

```
let dog: Character = "□□"
let cow: Character = "□□"
let dogCow = dog + cow
// dogCow is equal to "□□□□"
```

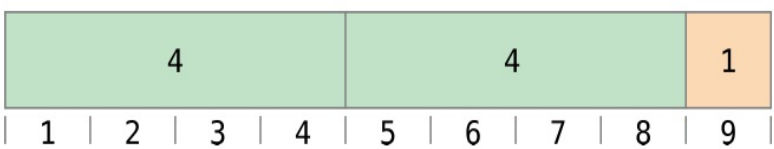
See also Concatenating Strings and Characters.

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 it is, strictly speaking, a remainder rather than a modulo operation.

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



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

In Swift, this would be written as:

```
9 % 4    // equals 1
```

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

`a` = (`b` × `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` × `2`) + `1`

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

```
1  -9 % 4   // equals -1
```

Inserting `-9` and `4` into the equation yields:

`-9` = (`4` × `-2`) + `-1`

giving a remainder value of `-1`.

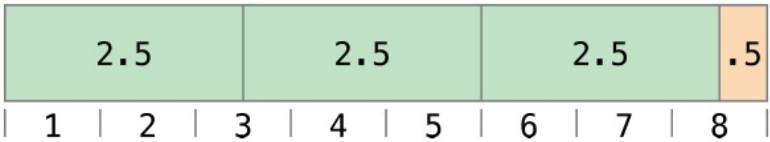
The sign of `b` is ignored for negative values of `b`. This means that `a % b` and `a % -b` always give the same answer.

Floating-Point Remainder Calculations

Unlike the remainder operator in C and Objective-C, Swift's remainder operator can also operate on floating-point numbers:

```
1  8 % 2.5   // equals 0.5
```

In this example, `8` divided by `2.5` equals `3`, with a remainder of `0.5`, so the remainder operator returns a `Double` value of `0.5`.



Increment and Decrement Operators

Like C, Swift provides an *increment operator* (`++`) and a *decrement operator* (`--`) as a shortcut to increase or decrease the value of a numeric variable by `1`. You can use these operators with variables of any integer or floating-point type.

```
var i = 0
++i      // i now equals 1
```

Each time you call `++i`, the value of `i` is increased by `1`. Essentially, `++i` is shorthand for saying `i = i + 1`. Likewise, `--i` can be used as shorthand for `i = i - 1`.

The `++` and `--` symbols can be used as prefix operators or as postfix operators. `++i` and `i++` are both valid ways to increase the value of `i` by `1`. Similarly, `--i` and `i--` are both valid ways to decrease the value of `i` by `1`.

Note that these operators modify `i` and also return a value. If you only want to increment or decrement the value stored in `i`, you can ignore the returned value. However, if you *do* use the returned value, it will be different based on whether you used the prefix or postfix version of the operator, according to the following rules:

> If the operator is written *before* the variable, it increments the variable *before* returning its value.
>
> If the operator is written *after* the variable, it increments the variable *after* returning its value.

For example:

```
var a = 0
```

```
let b = ++a
// a and b are now both equal to 1
let c = a++
// a is now equal to 2, but c has been set to the pre-increment value
                of 1
```

In the example above, `let b = ++a` increments `a` *before* returning its value. This is why both `a` and `b` are equal to to the new value of `1`.

However, `let c = a++` increments `a` *after* returning its value. This means that `c` gets the old value of `1`, and `a` is then updated to equal `2`.

Unless you need the specific behavior of `i++`, it is recommended that you use `++i` and `--i` in all cases, because they have the typical expected behavior of modifying `i` and returning the result.

Unary Minus Operator

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

```
let three = 3
let minusThree = -three       // minusThree equals -3
let plusThree = -minusThree   // plusThree equals 3, or "minus minus
                three"
```

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:

```
let minusSix = -6
```

```
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* (+=):

```
var a = 1
a += 2
// 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 do not return a value. You cannot write `let b = a += 2`, for example. This behavior is different from the increment and decrement operators mentioned above.

A complete list of compound assignment operators can be found in Expressions.

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 Classes and Structures.

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

```
1 == 1   // true, because 1 is equal to 1
2 != 1   // true, because 2 is not equal to 1
2 > 1    // true, because 2 is greater than 1
1 < 2    // true, because 1 is less than 2
1 >= 1   // true, because 1 is greater than or equal to 1
2 <= 1   // false, because 2 is not less than or equal to 1
```

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

```
let name = "world"
if name == "world" {
    println("hello, world")
} else {
    println("I'm sorry \(name), but I don't recognize you")
```

```
}
// prints "hello, world", because name is indeed equal to "world"
```

For more on the `if` statement, see Control Flow.

Ternary Conditional Operator

The *ternary conditional operator* is a special operator with three parts, which takes the form `question ? answer1 : answer2`. It is 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 pixel height for a table row. The row height should be 50 pixels taller than the content height if the row has a header, and 20 pixels taller if the row doesn't have a header:

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

The preceding example is shorthand for the code below:

```
let contentHeight = 40
let hasHeader = true
```

```
var rowHeight = contentHeight
if hasHeader {
    rowHeight = rowHeight + 50
} else {
    rowHeight = rowHeight + 20
}
// 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. This is more concise than the second example, and removes the need for `rowHeight` to be a variable, because its value does not need to be modified within an `if` statement.

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.

Range Operators

Swift includes two *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 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 {
    println("\(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 on `for-in` loops, see Control Flow.

Half-Closed Range Operator

The *half-closed range operator* (`a..b`) defines a range that runs from `a` to `b`, but does not include `b`. It is said to be *half-closed* because it contains its first value, but not its final value.

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

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..count {
    println("Person \(i + 1) is called \(names[i])")
}
// Person 1 is called Anna
// Person 2 is called Alex
// Person 3 is called Brian
// 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 is a half-closed range. For more on arrays, see Arrays.

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:

```
let allowedEntry = false
if !allowedEntry {
    println("ACCESS DENIED")
}
// 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`:

```
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    println("Welcome!")
} else {
    println("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 cannot 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:

```
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
```

```
// prints "Welcome!"
```

Combining Logical Operators

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

```
if enteredDoorCode && passedRetinaScan || hasDoorKey ||
            knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "Welcome!"
```

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. It 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 mini-expressions are `false`. However, the emergency override password is known, so the overall compound expression still evaluates to `true`.

Explicit Parentheses

It is sometimes useful to include parentheses when they are not strictly needed, to make the intention of a complex expression easier to read. In the door access example above, it is useful to add parentheses around the first part of the compound expression to make its intent explicit:

```
if (enteredDoorCode && passedRetinaScan) || hasDoorKey ||
            knowsOverridePassword {
    println("Welcome!")
} else {
    println("ACCESS DENIED")
}
// prints "Welcome!"
```

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 an ordered collection of characters, such as `"hello, world"` or `"albatross"`. Swift strings are represented by the `String` type, which in turn represents a collection of values of `Character` type.

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 similar syntax to C strings. String concatenation is as simple as adding together 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.

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.

Strings can also be used 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.

> NOTE
>
> Swift's `String` type is bridged seamlessly to Foundation's `NSString` class. If you are working with the Foundation framework in Cocoa or Cocoa Touch, the entire `NSString` API is available to call on any `String` value you create, in addition to the `String` features described in this chapter. You can also use a `String` value with any API that requires an `NSString` instance.
>
> For more information about using `String` with Foundation and Cocoa, see *Using Swift with Cocoa and Objective-C*.

String Literals

You can include predefined `String` values within your code as *string literals*. A string literal is a fixed sequence of textual characters surrounded by a pair of double quotes (`""`).

A string literal can be used to provide 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 is initialized with a string literal value.

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 quote) and `\'` (single quote)
>
> Single-byte Unicode scalars, written as `\xnn`, where `nn` is two hexadecimal digits
>
> Two-byte Unicode scalars, written as `\unnnn`, where `nnnn` is four hexadecimal digits
>
> Four-byte Unicode scalars, written as `\Unnnnnnnn`, where `nnnnnnnn` is eight hexadecimal digits

The code below shows an example of each kind of special character. The `wiseWords` constant contains two escaped double quote characters. The `dollarSign`, `blackHeart`, and `sparklingHeart` constants demonstrate the three different Unicode scalar character formats:

```
let wiseWords = "\"Imagination is more important than knowledge\" - Einstein"
// "Imagination is more important than knowledge" - Einstein
let dollarSign = "\x24"        // $,  Unicode scalar U+0024
let blackHeart = "\u2665"      // ♥,  Unicode scalar U+2665
let sparklingHeart = "\U0001F496"  // 💖, Unicode scalar U+1F496
```

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
var anotherEmptyString = String()  // initializer syntax
// these two strings are both empty, and are equivalent to each other
```

You can find out whether a `String` value is empty by checking its Boolean `isEmpty` property:

```
if emptyString.isEmpty {
    println("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 cannot be modified):

```
var 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 is passed to a function or method, or when it is 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 Structures and Enumerations Are Value Types.

NOTE

This behavior differs from that of `NSString` in Cocoa. When you create an `NSString` instance in Cocoa, and pass it to a function or method or assign it to a variable, you are always passing or assigning a *reference* to the same single `NSString`. No copying of the string takes place, unless you specifically request it.

Swift's copy-by-default `String` behavior ensures that when a function or method passes you a `String` value, it is clear that you own that exact `String` value, regardless of where it came from. You can be confident that the string you are passed will not 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

Swift's `String` type represents a collection of `Character` values in a specified order. Each `Character` value represents a single Unicode character. You can access the individual `Character` values in a string by

iterating over that string with a `for-in` loop:

```
for character in "Dog!🐶" {
    println(character)
}
// D
// o
// g
// !
// 🐶
```

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

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

```
let yenSign: Character = "¥"
```

Counting Characters

To retrieve a count of the characters in a string, call the global `countElements` function and pass in a string as the function's sole parameter:

```
let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
println("unusualMenagerie has \(countElements(unusualMenagerie)) characters")
// prints "unusualMenagerie has 40 characters"
```

NOTE

> Different Unicode characters and different representations of the same Unicode character can require different amounts of memory to store. Because of this, characters in Swift do not each take up the same amount of memory within a string's representation. As a result, the length of a string cannot be calculated without iterating through the string to consider each of its characters in turn. If you are working with particularly long string values, be aware that the `countElements` function must iterate over the characters within a string in order to calculate an accurate character count for that string.
>
> Note also that the character count returned by `countElements` is not 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 characters within the string. To reflect this fact, the `length` property from `NSString` is called `utf16count` when it is accessed on a Swift `String` value.

Concatenating Strings and Characters

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

```
let string1 = "hello"
let string2 = " there"
let character1: Character = "!"
let character2: Character = "?"

let stringPlusCharacter = string1 + character1        // equals
                "hello!"
let stringPlusString = string1 + string2              // equals "hello
                there"
let characterPlusString = character1 + string1        // equals
                "!hello"
let characterPlusCharacter = character1 + character2  // equals "!?"
```

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

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

var welcome = "good morning"
welcome += character1
// welcome now equals "good morning!"
```

NOTE

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

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. Each item that you insert into the string literal is wrapped in a pair of parentheses, prefixed by a backslash:

```
let multiplier = 3
let message = "\(multiplier) times 2.5 is \(Double(multiplier) * 2.5)"
// message is "3 times 2.5 is 7.5"
```

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 is included inside the string literal.

> NOTE
>
> The expressions you write inside parentheses within an interpolated string cannot contain an unescaped double quote (") or backslash (\), and cannot contain a carriage return or line feed.

Comparing Strings

Swift provides three ways to compare `String` values: string equality, prefix equality, and suffix equality.

String Equality

Two `String` values are considered equal if they contain exactly the same characters in the same order:

```
let quotation = "We're a lot alike, you and I."
let sameQuotation = "We're a lot alike, you and I."
if quotation == sameQuotation {
    println("These two strings are considered equal")
}
// prints "These two strings are considered equal"
```

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. Both methods perform a character-by-character comparison between the base string and the prefix or suffix string.

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

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

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
    }
}
println("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
for scene in romeoAndJuliet {
    if scene.hasSuffix("Capulet's mansion") {
```

```
        ++mansionCount
    } else if scene.hasSuffix("Friar Lawrence's cell") {
        ++cellCount
    }
}
println("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
// prints "6 mansion scenes; 2 cell scenes"
```

Uppercase and Lowercase Strings

You can access an uppercase or lowercase version of a string with its `uppercaseString` and `lowercaseString` properties:

```
let normal = "Could you help me, please?"
let shouty = normal.uppercaseString
// shouty is equal to "COULD YOU HELP ME, PLEASE?"
let whispered = normal.lowercaseString
// whispered is equal to "could you help me, please?"
```

Unicode

Unicode is an international standard for encoding and representing text. 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. They support a number of different Unicode encodings, as described below.

Unicode Terminology

Every character in Unicode can be represented by one or more *unicode scalars*. A unicode scalar is a unique 21-bit number (and name) for a character or modifier, such as `U+0061` for `LOWERCASE LATIN LETTER A` (`"a"`), or `U+1F425` for `FRONT-FACING BABY CHICK` (`"🐥"`).

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

Unicode Representations of Strings

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 characters. 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 (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`, `!`, and the 🐶 character (`DOG FACE`, or Unicode scalar `U+1F436`):

```
let dogString = "Dog!🐶"
```

UTF-8

You can access a UTF-8 representation of a `String` by iterating over its `utf8` property. This property is of

type `UTF8View`, which is a collection of unsigned 8-bit (`UInt8`) values, one for each byte in the string's UTF-8 representation:

```
for codeUnit in dogString.utf8 {
    print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 240 159 144 182
```

In the example above, the first four decimal `codeUnit` values (`68`, `111`, `103`, `33`) represent the characters `D`, `o`, `g`, and `!`, whose UTF-8 representation is the same as their ASCII representation. The last four `codeUnit` values (`240`, `159`, `144`, `182`) are a four-byte UTF-8 representation of the `DOG FACE` character.

UTF-16

You can access a UTF-16 representation of a `String` by iterating over its `utf16` property. This property is of type `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:

```
for codeUnit in dogString.utf16 {
    print("\(codeUnit) ")
}
print("\n")
// 68 111 103 33 55357 56374
```

Again, the first four `codeUnit` values (`68`, `111`, `103`, `33`) represent the characters `D`, `o`, `g`, and `!`, whose UTF-16 code units have the same values as in the string's UTF-8 representation.

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 lead surrogate value of `U+D83D` (decimal value `55357`) and a trail surrogate value of `U+DC36` (decimal value `56374`).

Unicode Scalars

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`. A Unicode scalar is any 21-bit Unicode code point that is not a lead surrogate or trail surrogate code point.

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

```
for scalar in dogString.unicodeScalars {
    print("\(scalar.value) ")
}
print("\n")
// 68 111 103 33 128054
```

The `value` properties for the first four `UnicodeScalar` values (`68`, `111`, `103`, `33`) once again represent the characters `D`, `o`, `g`, and `!`. The `value` property of the fifth and final `UnicodeScalar`, `128054`, is a decimal equivalent of the hexadecimal value `1F436`, which is equivalent to 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 {
    println("\(scalar) ")
}
// D
// o
// g
// !
// □□
```

Collection Types

Swift provides two *collection types*, known as arrays and dictionaries, for storing collections of values. Arrays store ordered lists of values of the same type. Dictionaries store unordered collections of values of the same type, which can be referenced and looked up through a unique identifier (also known as a key).

Arrays 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 an array or dictionary by mistake. It also means you can be confident about the types of values you will retrieve from an array or dictionary. Swift's use of explicitly typed collections ensures that your code is always clear about the types of values it can work with and enables you to catch any type mismatches early in your code's development.

> NOTE
>
> Swift's `Array` type exhibits different behavior to other types when assigned to a constant or variable, or when passed to a function or method. For more information, see Mutability of Collections and Assignment and Copy Behavior for Collection Types.

Arrays

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

Swift arrays are specific about the kinds of values they can store. They differ from Objective-C's `NSArray` and `NSMutableArray` classes, which can store any kind of object and do not provide any information about the nature of the objects they return. In Swift, the type of values that a particular array can store is always made clear, either through an explicit type annotation, or through type inference, and does not have to be a class type. If you create an array of `Int` values, for example, you can't insert any value other than `Int` values into that array. Swift arrays are type safe, and are always clear about what they may contain.

Array Type Shorthand Syntax

The type of a Swift array is written in full as `Array<SomeType>`, where `SomeType` is the type that the array is allowed to store. You can also write the type of an array in shorthand form as `SomeType[]`. 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.

Array Literals

You can 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 *only* allowed to store `String` values. 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:

```
println("The shopping list contains \(shoppingList.count) items.")
// prints "The shopping list contains 2 items."
```

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

```
if shoppingList.isEmpty {
    println("The shopping list is empty.")
} else {
    println("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")
// shoppingList now contains 3 items, and someone is making pancakes
```

Alternatively, add a new item to the end of an array with the addition assignment operator (`+=`):

```
shoppingList += "Baking Powder"
// shoppingList now contains 4 items
```

You can also append an array of compatible items with the addition assignment operator (`+=`):

```
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 that 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"
```

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
```

> NOTE
>
> You can't use subscript syntax to append a new item to the end of an array. If you try to use subscript syntax to retrieve or set a value for an index that is outside of an array's existing bounds, you will trigger a runtime error. However, you can check that an index is valid before using it, by comparing it to the array's `count` property. Except when `count` is `0` (meaning the array is empty), the largest valid index in an array will always be `count - 1`, because arrays are indexed from zero.

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

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

This call to the `insert` 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 `removeAtIndex` 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):

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

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 `removeAtIndex` method to avoid the need to query the array's `count` property. Like the `removeAtIndex` method, `removeLast` returns the removed item:

```
let apples = shoppingList.removeLast()
// the last item in the array has just been removed
// shoppingList now contains 5 items, and no cheese
// the apples constant is now equal to the removed "Apples" string
```

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 {
    println(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

If you need the integer index of each item as well as its value, use the global `enumerate` function to iterate over the array instead. The `enumerate` function returns a tuple for each item in the array composed of the index and the value for that item. You can decompose the tuple into temporary constants or variables as part of the iteration:

```
for (index, value) in enumerate(shoppingList) {
```

```
    println("Item \(index + 1): \(value)")
}
// Item 1: Six eggs
// Item 2: Milk
// Item 3: Flour
// Item 4: Baking Powder
// Item 5: Bananas
```

For more about the `for-in` loop, see For Loops.

Creating and Initializing an Array

You can create an empty array of a certain type (without setting any initial values) using initializer syntax:

```
var someInts = Int[]()
println("someInts is of type Int[] with \(someInts.count) items.")
// prints "someInts is of type Int[] with 0 items."
```

Note that the type of the `someInts` variable is inferred to be `Int[]`, because it is set to the output of an `Int[]` 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[]
```

Swift's `Array` type also provides an initializer for creating an array of a certain size with all of its values set to a provided default value. You pass this initializer the number of items to be added to the new array (called `count`) and a default value of the appropriate type (called `repeatedValue`):

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

Thanks to type inference, you don't need to specify the type to be stored in the array when using this initializer, because it can be inferred from the default value:

```
var anotherThreeDoubles = Array(count: 3, repeatedValue: 2.5)
// anotherThreeDoubles is inferred as Double[], and equals [2.5, 2.5,
                2.5]
```

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

```
var sixDoubles = threeDoubles + anotherThreeDoubles
// sixDoubles is inferred as Double[], and equals [0.0, 0.0, 0.0, 2.5,
                2.5, 2.5]
```

Dictionaries

A *dictionary* is a container that stores multiple values of the same type. 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.

Swift dictionaries are specific about the types of keys and values they can store. They differ from Objective-C's `NSDictionary` and `NSMutableDictionary` classes, which can use any kind of object as their keys and values and do not provide any information about the nature of these objects. In Swift, the type of keys and values that a particular dictionary can store is always made clear, either through an explicit type annotation or through type inference.

Swift's dictionary type is written as `Dictionary<KeyType, ValueType>`, where `KeyType` is the type

of value that can be used as a dictionary key, and `ValueType` is the type of value that the dictionary stores for those keys.

The only restriction is that `KeyType` must be *hashable*—that is, it must provide a way to make itself uniquely representable. All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default, and all of these types can be used as the keys of a dictionary. Enumeration member values without associated values (as described in Enumerations) are also hashable by default.

Dictionary Literals

You can initialize a dictionary with 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 `Dictionary` 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:

```
var airports: Dictionary<String, String> = ["TYO": "Tokyo", "DUB":
                "Dublin"]
```

The `airports` dictionary is declared as having a type of `Dictionary<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 will be 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 `"TYO"` and a value of `"Tokyo"`. The second pair has a key of `"DUB"` and a value of `"Dublin"`.

This dictionary literal contains two `String: String` pairs. This 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 be written in a shorter form instead:

```
var airports = ["TYO": "Tokyo", "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 `Dictionary<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 can find out the number of items in a `Dictionary` by checking its read-only `count` property:

```
println("The dictionary of airports contains \(airports.count)
             items.")
// prints "The dictionary of airports contains 2 items."
```

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:

```
airports["LHR"] = "London Heathrow"
// 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 International", forKey:
                "DUB") {
    println("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"] {
    println("The name of the airport is \(airportName).")
```

```
} else {
    println("That airport is not in the airports dictionary.")
}
// prints "The name of the airport is Dublin International."
```

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 `removeValueForKey` method. This method removes the key-value pair if it exists and returns the removed value, or returns `nil` if no value existed:

```
if let removedValue = airports.removeValueForKey("DUB") {
    println("The removed airport's name is \(removedValue).")
} else {
    println("The airports dictionary does not contain a value for
             DUB.")
}
// prints "The removed airport's name is Dublin International."
```

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 {
```

```
    println("\(airportCode): \(airportName)")
}
// TYO: Tokyo
// LHR: London Heathrow
```

For more about the `for-in` loop, see For Loops.

You can also retrieve an iteratable collection of a dictionary's keys or values by accessing its `keys` and `values` properties:

```
for airportCode in airports.keys {
    println("Airport code: \(airportCode)")
}
// Airport code: TYO
// Airport code: LHR

for airportName in airports.values {
    println("Airport name: \(airportName)")
}
// Airport name: Tokyo
// Airport name: London Heathrow
```

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:

```
let airportCodes = Array(airports.keys)
// airportCodes is ["TYO", "LHR"]

let airportNames = Array(airports.values)
// airportNames is ["Tokyo", "London Heathrow"]
```

NOTE

Swift's `Dictionary` type is an unordered collection. The order in which keys, values, and key-value pairs are retrieved when iterating over a dictionary is not specified.

Creating an Empty Dictionary

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

```
var namesOfIntegers = Dictionary<Int, String>()
// namesOfIntegers is an empty Dictionary<Int, String>
```

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, 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
```

NOTE

Behind the scenes, Swift's array and dictionary types are implemented as *generic collections*. For more on generic types and collections, see Generics.

Mutability of Collections

Arrays and dictionaries store multiple values together in a single collection. If you create an array 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 size of the collection after it is created by adding more items to the collection, or by removing existing items from the ones it already contains. Conversely, if you assign an array or a dictionary to a constant, that array or dictionary is *immutable*, and its size cannot be changed.

For dictionaries, immutability also means that you cannot replace the value for an existing key in the dictionary. An immutable dictionary's contents cannot be changed once they are set.

Immutability has a slightly different meaning for arrays, however. You are still not allowed to perform any action that has the potential to change the size of an immutable array, but you *are* allowed to set a new value for an existing index in the array. This enables Swift's `Array` type to provide optimal performance for array operations when the size of an array is fixed.

The mutability behavior of Swift's `Array` type also affects how array instances are assigned and modified. For more information, see Assignment and Copy Behavior for Collection Types.

> NOTE
>
> It is good practice to create immutable collections in all cases where the collection's size does not need to change. Doing so enables the Swift compiler to optimize the performance of the collections you create.

Control Flow

Swift provides all the familiar control flow constructs of C-like languages. These include `for` and `while` loops to perform a task multiple times; `if` 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.

In addition to the traditional `for`-`condition`-`increment` loop found in C, Swift adds a `for`-`in` loop that makes it easy to iterate over arrays, dictionaries, ranges, strings, and other sequences.

Swift's `switch` statement is also considerably more powerful than its counterpart in C. The cases of a `switch` statement do not "fall through" to the next case in Swift, avoiding common C errors caused by missing `break` statements. Cases can match many different types of pattern, including range 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 Loops

A `for` loop performs a set of statements a certain number of times. Swift provides two kinds of `for` loop:

> `for`-`in` performs a set of statements for each item in a range, sequence, collection, or progression.
>
> `for`-`condition`-`increment` performs a set of statements until a specific condition is met, typically by incrementing a counter each time the loop ends.

For-In

You use the `for`-`in` loop to iterate over collections of items, such as ranges of numbers, items in an array, or characters in a string.

This example prints the first few entries in the five-times-table:

```
for index in 1...5 {
    println("\(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 collection of items being iterated is a closed 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 `println` 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, it 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.

> NOTE
>
> The `index` constant exists only within the scope of the loop. If you want to check the value of `index` after the loop completes, or if you want to work with its value as a variable rather than a constant, you must declare it yourself before its use in the loop.

If you don't need each value from the range, you can ignore the values by using an underscore in place of a variable name:

```
let base = 3
```

```
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
println("\(base) to the power of \(power) is \(answer)")
// prints "3 to the power of 10 is 59049"
```

This example 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 half-closed loop that starts with `0` and ends with `9`. This calculation doesn't need to know the individual counter values each time through the loop—it simply needs to execute 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.

Use the `for-in` loop with an array to iterate over its items:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    println("Hello, \(name)!")
}
// Hello, Anna!
// Hello, Alex!
// Hello, Brian!
// 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 in the body of the `for-in` loop. Here, the dictionary's keys are decomposed into a constant called `animalName`, and the dictionary's values are decomposed into a constant called `legCount`:

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    println("\(animalName)s have \(legCount) legs")
```

```
}
// spiders have 8 legs
// ants have 6 legs
// cats have 4 legs
```

Items in a `Dictionary` may not necessarily be iterated in the same order as they were inserted. The contents of a `Dictionary` are inherently unordered, and iterating over them does not guarantee the order in which they will be retrieved. For more on arrays and dictionaries, see Collection Types.)

In addition to arrays and dictionaries, you can also use the `for-in` loop to iterate over the `Character` values in a string:

```
for character in "Hello" {
    println(character)
}
// H
// e
// l
// l
// o
```

For-Condition-Increment

In addition to `for-in` loops, Swift supports traditional C-style `for` loops with a condition and an incrementer:

```
for var index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
```

Here's the general form of this loop format:

```
for initialization; condition; increment {
    statements
}
```

Semicolons separate the three parts of the loop's definition, as in C. However, unlike C, Swift doesn't need parentheses around the entire "initialization; condition; increment" block.

The loop is executed as follows:

1. When the loop is first entered, the *initialization expression* is evaluated once, to set up any constants or variables that are needed for the loop.
2. The *condition expression* is evaluated. If it evaluates to `false`, the loop ends, and code execution continues after the `for` loop's closing brace (`}`). If the expression evaluates to `true`, code execution continues by executing the statements inside the braces.
3. After all statements are executed, the *increment expression* is evaluated. It might increase or decrease the value of a counter, or set one of the initialized variables to a new value based on the outcome of the statements. After the increment expression has been evaluated, execution returns to step 2, and the condition expression is evaluated again.

The loop format and execution process described above is shorthand for (and equivalent to) the outline below:

```
initialization
while condition {
    statements
    increment
}
```

Constants and variables declared within the initialization expression (such as `var index = 0`) are only valid within the scope of the `for` loop itself. To retrieve the final value of `index` after the loop ends, you must declare `index` before the loop's scope begins:

```
var index: Int
for index = 0; index < 3; ++index {
    println("index is \(index)")
}
// index is 0
// index is 1
// index is 2
println("The loop statements were executed \(index) times")
// prints "The loop statements were executed 3 times"
```

Note that the final value of `index` after this loop is completed is `3`, not `2`. The last time the increment statement `++index` is called, it sets `index` to `3`, which causes `index < 3` to equate to `false`, ending the loop.

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` loop:

- `while` evaluates its condition at the start of each pass through the loop.
- `do-while` evaluates its condition at the end of each pass through the loop.

While

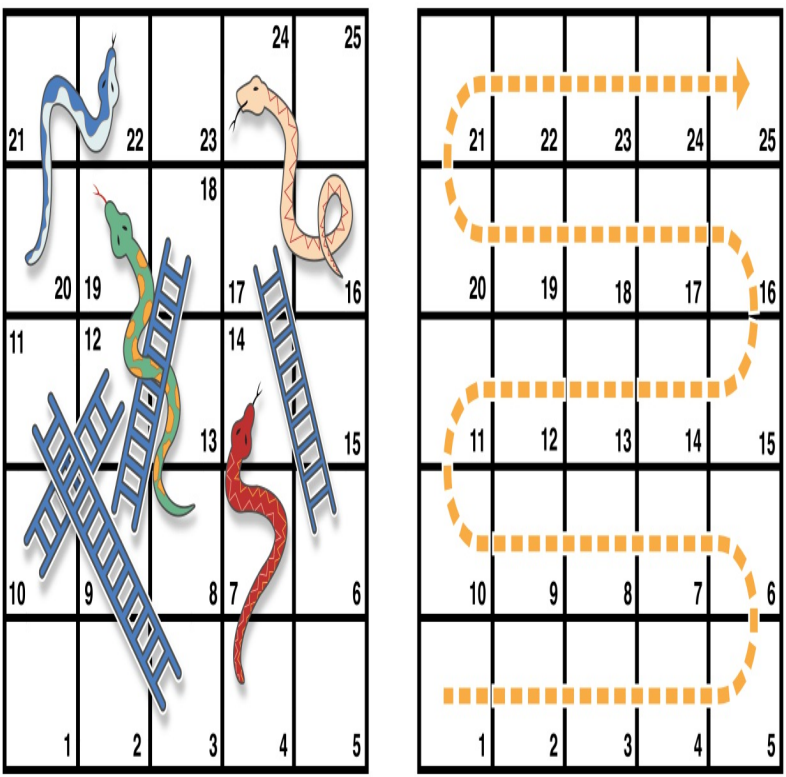
A `while` loop starts by evaluating a single condition. If the condition is `true`, a set of statements is repeated until the condition becomes `false`.

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.

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. The board is initialized with 26 zero `Int` values, not 25 (one each at indices `0` through `25` inclusive):

```
let finalSquare = 25
var board = Int[](count: finalSquare + 1, repeatedValue: 0)
```

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:

```
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
```

Square 3 contains the bottom of a ladder that moves you up to square 11. To represent this, `board[03]` is equal to `+08`, which is equivalent to an integer value of `8` (the difference between `3` and `11`). The unary plus operator (`+i`) balances with the unary minus operator (`-i`), and numbers lower than `10` are padded with zeros so that all board definitions align. (Neither stylistic tweak is strictly necessary, but they lead to neater code.)

The player's starting square is "square zero", which is just off the bottom left corner of the board. The first dice roll always moves the player on to the board:

```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // roll the dice
    if ++diceRoll == 7 { diceRoll = 1 }
    // move by the rolled amount
    square += diceRoll
    if square < board.count {
        // if we're still on the board, move up or down for a snake or
              a ladder
```

```
            square += board[square]
        }
    }
    println("Game over!")
```

This example uses a very simple approach to dice rolling. Instead of a random number generator, it starts with a `diceRoll` value of `0`. Each time through the `while` loop, `diceRoll` is incremented with the prefix increment operator (`++i`), and is then checked to see if it has become too large. The return value of `++diceRoll` is equal to the value of `diceRoll` *after* it is incremented. Whenever this return value equals `7`, the dice roll has become too large, and is reset to a value of `1`. This gives 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 before adding the value stored in `board[square]` onto the current `square` value to move the player up or down any ladders or snakes.

Had this check not been performed, `board[square]` might try to access a value outside the bounds of the `board` array, which would trigger an error. If `square` is now equal to `26`, the code would try to check the value of `board[26]`, which is larger than the size of the array.

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.

Do-While

The other variation of the `while` loop, known as the `do-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`.

Here's the general form of a `do-while` loop:

```
do {
    statements
} while condition
```

Here's the *Snakes and Ladders* example again, written as a `do-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:

```
let finalSquare = 25
var board = Int[](count: finalSquare + 1, repeatedValue: 0)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
var diceRoll = 0
```

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 is not possible to win the game by moving up a ladder. Therefore, it is 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:

```
do {
    // move up or down for a snake or ladder
    square += board[square]
    // roll the dice
    if ++diceRoll == 7 { diceRoll = 1 }
    // move by the rolled amount
    square += diceRoll
} while square < finalSquare
println("Game over!")
```

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 is not evaluated until the *end* of the first run through the loop. The structure of the `do-while` loop is better suited to this game than the `while` loop in the previous example. In the `do-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 earlier version of the game.

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, known as 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 {
    println("It's very cold. Consider wearing a scarf.")
}
// prints "It's very cold. Consider wearing a scarf."
```

The preceding example 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 when the `if` condition is `false`. These statements are indicated by the `else` keyword:

```
temperatureInFahrenheit = 40
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing a scarf.")
} else {
    println("It's not that cold. Wear a t-shirt.")
}
// prints "It's not that cold. Wear a t-shirt."
```

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
if temperatureInFahrenheit <= 32 {
    println("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    println("It's really warm. Don't forget to wear sunscreen.")
} else {
    println("It's not that cold. Wear a t-shirt.")
}
// prints "It's really warm. Don't forget to wear sunscreen."
```

Here, an additional `if` statement is added to respond to particularly warm temperatures. The final `else` clause remains, and 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 {
    println("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    println("It's really warm. Don't forget to wear sunscreen.")
}
```

In this example, the temperature is neither too cold nor too warm to trigger the `if` or `else if` conditions, and so 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 section.

The body of each `switch` case is a separate branch of code execution, in a similar manner to the branches of an `if` statement. The `switch` statement determines which branch should be selected. This 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 is not appropriate to provide a `switch` case for every possible value, you can define a default catch-all case to cover any values that are not addressed explicitly. This catch-all case is indicated by the keyword `default`, and must always appear last.

This example uses a `switch` statement to consider a single lowercase character called `someCharacter`:

```
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    println("\(someCharacter) is a vowel")
case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
"n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
    println("\(someCharacter) is a consonant")
default:
    println("\(someCharacter) is not a vowel or a consonant")
}
// prints "e is a vowel"
```

The `switch` statement's first case matches all five lowercase vowels in the English language. Similarly, its second case matches all lowercase English consonants.

It is not practical to write all other possible characters as part of a `switch` case, and so this `switch` statement provides a `default` case to match all other characters that are not vowels or consonants. 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 in C, and avoids executing more than one `switch` case by mistake.

> NOTE
>
> You can still break out of a matched `switch` case before that case has completed its execution if you need to. See Break in a Switch Statement for details.

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:

```
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a":
case "A":
    println("The letter A")
default:
    println("Not the letter A")
}
// 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.

Multiple matches for a single `switch` case can be separated by commas, and can be written over multiple lines if the list is long:

```
switch some value to consider {
case value 1,
 value 2:
     statements
}
```

NOTE

To opt in to fallthrough behavior for a particular `switch` case, use the `fallthrough` keyword, as described in Fallthrough.

Range Matching

Values in `switch` cases can be checked for their inclusion in a range. This example uses number ranges to provide a natural-language count for numbers of any size:

```
let count = 3_000_000_000_000
let countedThings = "stars in the Milky Way"
var naturalCount: String
switch count {
case 0:
    naturalCount = "no"
case 1...3:
    naturalCount = "a few"
case 4...9:
    naturalCount = "several"
case 10...99:
    naturalCount = "tens of"
```

```
case 100...999:
    naturalCount = "hundreds of"
case 1000...999_999:
    naturalCount = "thousands of"
default:
    naturalCount = "millions and millions of"
}
println("There are \(naturalCount) \(countedThings).")
// prints "There are millions and millions of stars in the
        Milky Way."
```

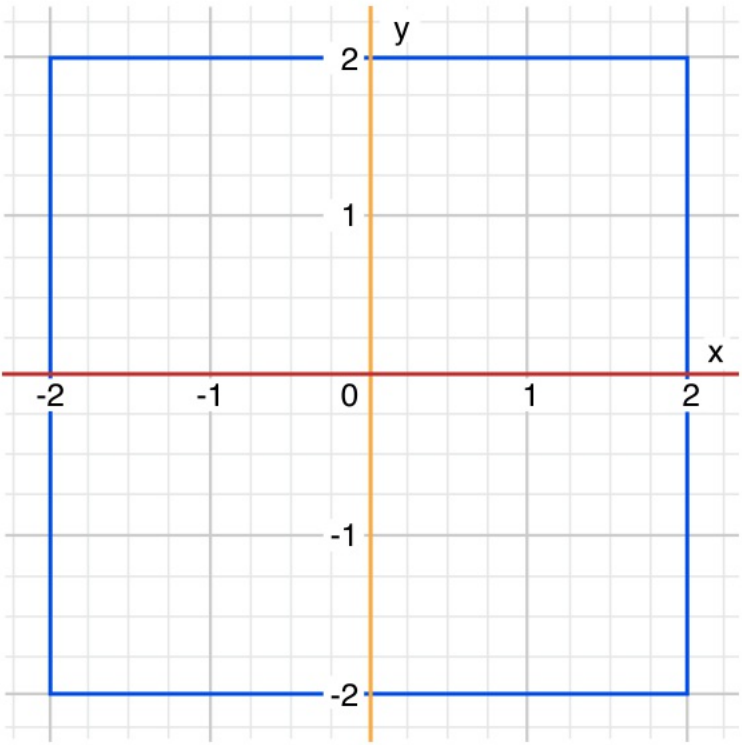
Tuples

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 range of values. Alternatively, use the underscore (_) identifier 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:

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    println("(0, 0) is at the origin")
case (_, 0):
    println("(\(somePoint.0), 0) is on the x-axis")
case (0, _):
    println("(0, \(somePoint.1)) is on the y-axis")
case (-2...2, -2...2):
    println("(\(somePoint.0), \(somePoint.1)) is inside the
        box")
default:
    println("(\(somePoint.0), \(somePoint.1)) is outside of the
        box")
```

```
}
// prints "(1, 1) is inside the box"
```



The `switch` statement determines if 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.

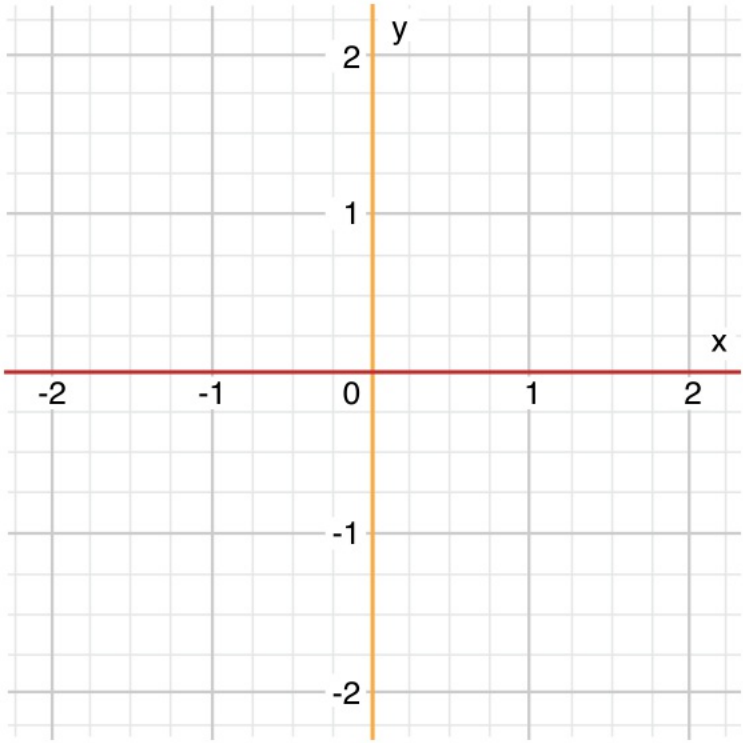
Unlike C, Swift allows multiple `switch` cases to consider the same value or values. In fact, the point (0, 0) could match all *four* of the cases in this example. However, if multiple matches are possible, the first matching case is always used. The point (0, 0) would match `case (0, 0)` first, and so all other matching cases would be ignored.

Value Bindings

A `switch` case can bind the value or values it matches to temporary constants or variables, for use in the body of the case. This 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:

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    println("on the x-axis with an x value of \(x)")
case (0, let y):
    println("on the y-axis with a y value of \(y)")
case let (x, y):
    println("somewhere else at (\(x), \(y))")
}
// prints "on the x-axis with an x value of 2"
```



The `switch` statement determines if 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 `x` and `y`, which temporarily take on one or both tuple values from `anotherPoint`. The first case, `case (let x, 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`.

Once the temporary constants are declared, they can be used within the case's code block. Here, they are used as shorthand for printing the values with the `println` function.

Note that 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. As a result, it matches all possible remaining values, and a `default` case is not needed to make the `switch` statement exhaustive.

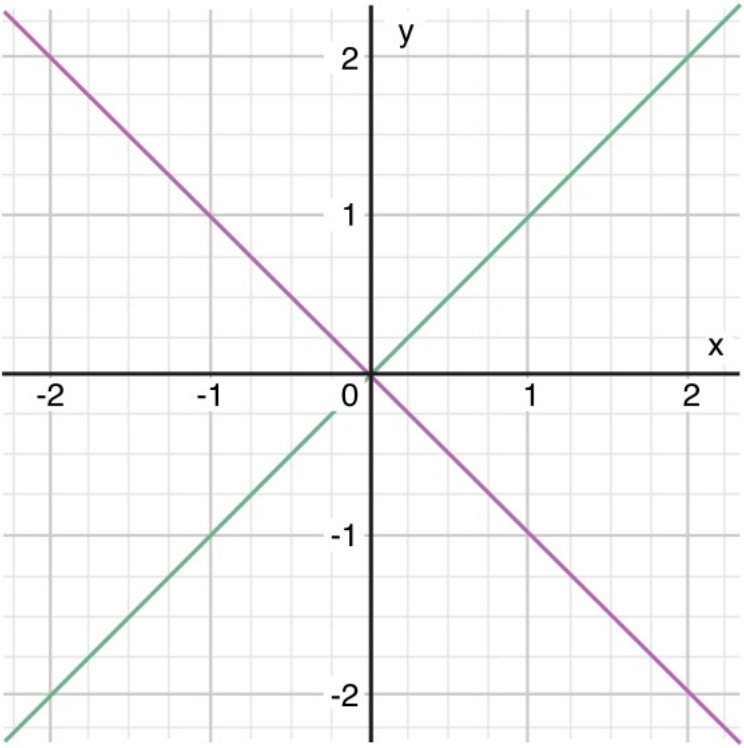
In the example above, `x` and `y` are declared as constants with the `let` keyword, because there is no need to modify their values within the body of the case. However, they could have been declared as variables instead, with the `var` keyword. If this had been done, a temporary variable would have been created and initialized with the appropriate value. Any changes to that variable would only have an effect within the body of the case.

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:

```
let yetAnotherPoint = (1, -1)
switch yetAnotherPoint {
case let (x, y) where x == y:
    println("(\(x), \(y)) is on the line x == y")
case let (x, y) where x == -y:
    println("(\(x), \(y)) is on the line x == -y")
case let (x, y):
    println("(\(x), \(y)) is just some arbitrary point")
}
// prints "(1, -1) is on the line x == -y"
```



The `switch` statement determines if 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 `x` and `y`, which temporarily take on the two tuple values from `point`. 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.

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 four control transfer statements:

```
continue
break
fallthrough
return
```

The `control`, `break` and `fallthrough` statements are described below. The `return` statement is described in Functions.

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.

> NOTE
>
> In a `for`-`condition`-`increment` loop, the incrementer is still evaluated after calling the `continue` statement. The loop itself continues to work as usual; only the code within the loop's body is skipped.

The following example removes all vowels and spaces from a lowercase string to create a cryptic puzzle phrase:

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
for character in puzzleInput {
    switch character {
```

```
    case "a", "e", "i", "o", "u", " ":
        continue
    default:
        puzzleOutput += character
    }
}
println(puzzleOutput)
// prints "grtmndsthnklk"
```

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. This behavior enables the switch block to match (and ignore) only the vowel and space characters, rather than requiring the block to match every character that should get printed.

Break

The `break` statement ends execution of an entire control flow statement immediately. The `break` statement can be used inside a `switch` statement 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 first line of 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 first line of 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 only contains 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. Multiple values are covered in a single `switch` case for brevity:

```
let numberSymbol: Character = "三"  // Simplified Chinese for the
                number 3
var possibleIntegerValue: Int?
switch numberSymbol {
case "1", "١", "一", "๑":
    possibleIntegerValue = 1
case "2", "٢", "二", "๒":
    possibleIntegerValue = 2
case "3", "٣", "三", "๓":
    possibleIntegerValue = 3
case "4", "٤", "四", "๔":
    possibleIntegerValue = 4
default:
    break
}
if let integerValue = possibleIntegerValue {
    println("The integer value of \(numberSymbol) is \
```

```
            (integerValue).")
} else {
    println("An integer value could not be found for \
        (numberSymbol).")
}
// prints "The integer value of 三 is 3."
```

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.

It is not practical to list every possible `Character` value in the example above, so a `default` case provides a catchall for 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` statement is matched, the `break` statement ends the `switch` statement's execution, and code execution continues from the `if let` statement.

Fallthrough

Switch statements in Swift do not fall through the bottom of each case and into the next one. Instead, 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 really 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
var description = "The number \(integerToDescribe) is"
switch integerToDescribe {
case 2, 3, 5, 7, 11, 13, 17, 19:
    description += " a prime number, and also"
    fallthrough
default:
    description += " an integer."
}
println(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.

If the value of `integerToDescribe` is *not* in the list of known prime numbers, it is not matched by the first `switch` case at all. There are no other specific cases, and so `integerToDescribe` is matched by the catchall `default` case.

After the `switch` statement has finished executing, the number's description is printed using the `println` 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

You can nest loops and `switch` statements inside other loops and `switch` statements in Swift to create complex control flow structures. However, loops and `switch` statements can both use the `break` statement to end their execution prematurely. Therefore, it is sometimes useful to be explicit about which loop or `switch` 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 `switch` statement with a *statement label*, and use this label with the `break` statement 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:

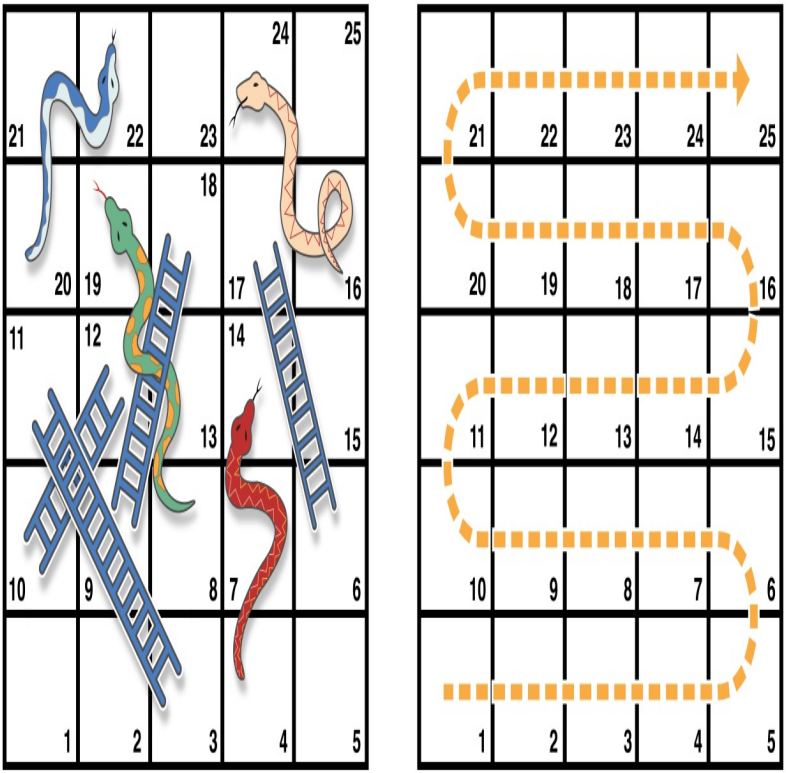
```
label name: while condition {
    statements
}
```

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:

```
let finalSquare = 25
var board = Int[](count: finalSquare + 1, repeatedValue: 0)
board[03] = +08; board[06] = +11; board[09] = +09; board[10] = +02
board[14] = -10; board[19] = -11; board[22] = -02; board[24] = -08
var square = 0
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 {
    if ++diceRoll == 7 { diceRoll = 1 }
    switch square + diceRoll {
    case finalSquare:
        // diceRoll will move us to the final square, so the game is over
        break gameLoop
    case let newSquare where newSquare > finalSquare:
        // diceRoll will move us beyond the final square, so roll again
        continue gameLoop
    default:
        // this is a valid move, so find out its effect
        square += diceRoll
        square += board[square]
    }
}
println("Game over!")
```

The dice is rolled at the start of each loop. Rather than moving the player immediately, a `switch` statement is used to consider the result of the move, and to work out if 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.

Note also that it is not strictly necessary to use the `gameLoop` label when calling `continue gameLoop` to jump to the next iteration of the loop. There is only one loop in the game, and so there is no ambiguity as to which loop the `continue` statement will affect. However, there is no harm in using the `gameLoop` label with the `continue` statement. Doing so is consistent with the label's use alongside the `break` statement, and helps make the game's logic clearer to read and understand.

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 local and external parameter names 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*), and/or 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 `greetingForPerson`, 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 `personName`—and a return type of `String`, which will contain a greeting for that person:

```
1 func sayHello(personName: String) -> String {
2     let greeting = "Hello, " + personName + "!"
```

```
    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 elsewhere in your code in a clear and unambiguous way:

```
println(sayHello("Anna"))
// prints "Hello, Anna!"
println(sayHello("Brian"))
// prints "Hello, Brian!"
```

You call the `sayHello` function by passing it a `String` argument value in parentheses, such as `sayHello("Anna")`. Because the function returns a `String` value, `sayHello` can be wrapped in a call to the `println` function to print that string and see its return value, as shown above.

The body of the `sayHello` function starts by defining a new `String` constant called `greeting` and setting it to a simple greeting message for `personName`. This greeting is then passed back out of the function using the `return` keyword. As soon as `return greeting` is called, the function finishes its execution and returns the current value of `greeting`.

You can call the `sayHello` 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 simplify the body of this function, combine the message creation and the return statement into one line:

```
func sayHelloAgain(personName: String) -> String {
    return "Hello again, " + personName + "!"
}
println(sayHelloAgain("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.

Multiple Input Parameters

Functions can have multiple input parameters, which are written within the function's parentheses, separated by commas.

This function takes a start and an end index for a half-open range, and works out how many elements the range contains:

```
func halfOpenRangeLength(start: Int, end: Int) -> Int {
    return end - start
}
println(halfOpenRangeLength(1, 10))
// prints "9"
```

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"
}
```

```
println(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 Without Return Values

Functions are not required to define a return type. Here's a version of the `sayHello` function, called `waveGoodbye`, which prints its own `String` value rather than returning it:

```
func sayGoodbye(personName: String) {
    println("Goodbye, \(personName)!")
}
sayGoodbye("Dave")
// prints "Goodbye, 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, the `sayGoodbye` 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 `Void`. This is simply an empty tuple, in effect a tuple with zero elements, which can be written as `()`.

The return value of a function can be ignored when it is called:

```
func printAndCount(stringToPrint: String) -> Int {
```

```
    println(stringToPrint)
    return countElements(stringToPrint)
}
func printWithoutCounting(stringToPrint: String) {
    printAndCount(stringToPrint)
}
printAndCount("hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting("hello, world")
// prints "hello, world" but does not return a value
```

The first function, `printAndCount`, prints a string, and then returns its character count as an `Int`. The second function, `printWithoutCounting`, 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 `count`, which counts the number of vowels, consonants, and other characters in a string, based on the standard set of vowels and consonants used in American English:

```
func count(string: String) -> (vowels: Int, consonants: Int, others:
```

```
            Int) {
    var vowels = 0, consonants = 0, others = 0
    for character in string {
        switch String(character).lowercaseString {
        case "a", "e", "i", "o", "u":
            ++vowels
        case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
        "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
            ++consonants
           default:
               ++others
           }
       }
       return (vowels, consonants, others)
}
```

You can use this `count` function to count the characters in an arbitrary string, and to retrieve the counted totals as a tuple of three named `Int` values:

```
let total = count("some arbitrary string!")
println("\(total.vowels) vowels and \(total.consonants) consonants")
// prints "6 vowels and 13 consonants"
```

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.

Function Parameter Names

All of the above functions define *parameter names* for their parameters:

```
func someFunction(parameterName: Int) {
    // function body goes here, and can use parameterName
    // to refer to the argument value for that parameter
```

```
}
```

However, these parameter names are only used within the body of the function itself, and cannot be used when calling the function. These kinds of parameter names are known as *local parameter names*, because they are only available for use within the function's body.

External Parameter Names

Sometimes it's useful to name each parameter when you *call* a function, to indicate the purpose of each argument you pass to the function.

If you want users of your function to provide parameter names when they call your function, define an *external parameter name* for each parameter, in addition to the local parameter name. You write an external parameter name before the local parameter name it supports, separated by a space:

```
func someFunction(externalParameterName localParameterName: Int) {
    // function body goes here, and can use localParameterName
    // to refer to the argument value for that parameter
}
```

> NOTE
>
> If you provide an external parameter name for a parameter, that external name must *always* be used when calling the function.

As an example, consider the following function, which joins two strings by inserting a third "joiner" string between them:

```
func join(s1: String, s2: String, joiner: String) -> String {
    return s1 + joiner + s2
```

```
}
```

When you call this function, the purpose of the three strings that you pass to the function is unclear:

```
join("hello", "world", ", ")
// returns "hello, world"
```

To make the purpose of these `String` values clearer, define external parameter names for each `join` function parameter:

```
func join(string s1: String, toString s2: String, withJoiner joiner:
              String)
    -> String {
        return s1 + joiner + s2
}
```

In this version of the `join` function, the first parameter has an external name of `string` and a local name of `s1`; the second parameter has an external name of `toString` and a local name of `s2`; and the third parameter has an external name of `withJoiner` and a local name of `joiner`.

You can now use these external parameter names to call the function in a clear and unambiguous way:

```
join(string: "hello", toString: "world", withJoiner: ", ")
// returns "hello, world"
```

The use of external parameter names enables this second version of the `join` function to be called in an expressive, sentence-like manner by users of the function, while still providing a function body that is readable and clear in intent.

> NOTE
>
> Consider using external parameter names whenever the purpose of a function's arguments would

be unclear to someone reading your code for the first time. You do not need to specify external parameter names if the purpose of each parameter is clear and unambiguous when the function is called.

Shorthand External Parameter Names

If you want to provide an external parameter name for a function parameter, and the local parameter name is already an appropriate name to use, you do not need to write the same name twice for that parameter. Instead, write the name once, and prefix the name with a hash symbol (#). This tells Swift to use that name as both the local parameter name and the external parameter name.

This example defines a function called `containsCharacter`, which defines external parameter names for both of its parameters by placing a hash symbol before their local parameter names:

```
func containsCharacter(#string: String, #characterToFind: Character) -
                > Bool {
    for character in string {
        if character == characterToFind {
            return true
        }
    }
    return false
}
```

This function's choice of parameter names makes for a clear, readable function body, while also enabling the function to be called without ambiguity:

```
let containsAVee = containsCharacter(string: "aardvark",
                characterToFind: "v")
// containsAVee equals true, because "aardvark" contains a "v"
```

Default Parameter Values

You can define a *default value* for any parameter as part of a function's definition. If a default value is defined, you can omit that parameter when calling the function.

> NOTE
>
> Place parameters with default values at the end of a function's parameter list. This ensures that all calls to the function use the same order for their non-default arguments, and makes it clear that the same function is being called in each case.

Here's a version of the `join` function from earlier, which provides a default value for its `joiner` parameter:

```
func join(string s1: String, toString s2: String,
    withJoiner joiner: String = " ") -> String {
        return s1 + joiner + s2
}
```

If a string value for `joiner` is provided when the `join` function is called, that string value is used to join the two strings together, as before:

```
join(string: "hello", toString: "world", withJoiner: "-")
// returns "hello-world"
```

However, if no value of `joiner` is provided when the function is called, the default value of a single space (`" "`) is used instead:

```
join(string: "hello", toString: "world")
// returns "hello world"
```

External Names for Parameters with Default Values

In most cases, it is useful to provide (and therefore require) an external name for any parameter with a default value. This ensures that the argument for that parameter is clear in purpose if a value is provided when the function is called.

To make this process easier, Swift provides an automatic external name for any defaulted parameter you define, if you do not provide an external name yourself. The automatic external name is the same as the local name, as if you had written a hash symbol before the local name in your code.

Here's a version of the `join` function from earlier, which does not provide external names for any of its parameters, but still provides a default value for its `joiner` parameter:

```
func join(s1: String, s2: String, joiner: String = " ") -> String {
    return s1 + joiner + s2
}
```

In this case, Swift automatically provides an external parameter name of `joiner` for the defaulted parameter. The external name must therefore be provided when calling the function, making the parameter's purpose clear and unambiguous:

```
join("hello", "world", joiner: "-")
// returns "hello-world"
```

> NOTE
>
> You can opt out of this behavior by writing an underscore (_) instead of an explicit external name when you define the parameter. However, external names for defaulted parameters are always preferred where appropriate.

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 `Double...` is made available within the function's body as a constant array called `numbers` of type `Double[]`.

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
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
// returns 3.0, which is the arithmetic mean of these five numbers
arithmeticMean(3, 8, 19)
// returns 10.0, which is the arithmetic mean of these three numbers
```

NOTE

A function may have at most one variadic parameter, and it must always appear last in the parameter list, to avoid ambiguity when calling the function with multiple parameters.

If your function has one or more parameters with a default value, and also has a variadic parameter, place the variadic parameter after all the defaulted parameters at the very end of the list.

Constant and Variable 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.

However, sometimes it is useful for a function to have a *variable* copy of a parameter's value to work with. You can avoid defining a new variable yourself within the function by specifying one or more parameters as *variable parameters* instead. Variable parameters are available as variables rather than as constants, and give a new modifiable copy of the parameter's value for your function to work with.

Define variable parameters by prefixing the parameter name with the keyword `var`:

```
func alignRight(var string: String, count: Int, pad: Character) ->
                String {
    let amountToPad = count - countElements(string)
    for _ in 1...amountToPad {
        string = pad + string
    }
    return string
}
let originalString = "hello"
let paddedString = alignRight(originalString, 10, "-")
   10  // paddedString is equal to "-----hello"
   11  // originalString is still equal to "hello"
```

This example defines a new function called `alignRight`, which aligns an input string to the right edge of a longer output string. Any space on the left is filled with a specified padding character. In this example, the string `"hello"` is converted to the string `"-----hello"`.

The `alignRight` function defines the input parameter `string` to be a variable parameter. This means that `string` is now available as a local variable, initialized with the passed-in string value, and can be manipulated within the body of the function.

The function starts by working out how many characters need to be added to the left of `string` in order to right-align it within the overall string. This value is stored in a local constant called `amountToPad`. The function then adds `amountToPad` copies of the `pad` character to the left of the existing string and returns the result. It uses the `string` variable parameter for all its string manipulation.

> NOTE
>
> The changes you make to a variable parameter do not persist beyond the end of each call to the function, and are not visible outside the function's body. The variable parameter only exists for the lifetime of that function call.

In-Out Parameters

Variable parameters, as described above, can only be changed within the function itself. 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 at the start of its parameter definition. 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.

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 inout 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`. If you mark a parameter as `inout`, it cannot also be marked as `var` or `let`.

Here's an example of a function called `swapTwoInts`, which has two in-out integer parameters called `a` and `b`:

```
func swapTwoInts(inout a: Int, inout b: 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)
println("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 type 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() {
    println("hello, world")
}
```

The type of this function is `() -> ()`, or "a function that has no parameters, and returns `Void`." Functions that don't specify a return value always return `Void`, which is equivalent to an empty tuple in Swift, shown as `()`.

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`:

```
println("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 non-function types:

```
mathFunction = multiplyTwoInts
println("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:

```
let anotherMathFunction = addTwoInts
// anotherMathFunction is inferred to be of type (Int, Int) -> Int
```

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:

```
func printMathResult(mathFunction: (Int, Int) -> Int, a: Int, b: Int)
                {
    println("Result: \(mathFunction(a, b))")
}
printMathResult(addTwoInts, 3, 5)
// prints "Result: 8"
```

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 `stepForward` and `stepBackward`. The `stepForward` function returns a value one more than its input value, and the `stepBackward` function returns a value one less than its input value. Both functions have a type of `(Int) -> Int`:

```
func stepForward(input: Int) -> Int {
    return input + 1
}
func stepBackward(input: Int) -> Int {
    return input - 1
}
```

Here's a function called `chooseStepFunction`, whose return type is "a function of type `(Int) -> Int`". `chooseStepFunction` returns the `stepForward` function or the `stepBackward` function based on a Boolean parameter called `backwards`:

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    return backwards ? stepBackward : stepForward
}
```

You can now use `chooseStepFunction` to obtain a function that will step in one direction or the other:

```
var currentValue = 3
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the stepBackward() function
```

The preceding example works out 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` 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:

```
println("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// 3...
// 2...
// 1...
// 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` example above to use and return nested functions:

```
func chooseStepFunction(backwards: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backwards ? stepBackward : stepForward
}
```

```
var currentValue = -4
let moveNearerToZero = chooseStepFunction(currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    println("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
println("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

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, hence the name “closures”. 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 Capturing Values.

Global and nested functions, as introduced in Functions, 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

Closure Expressions

Nested functions, as introduced in Nested Functions, 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 that take other 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 their simplest form without loss of clarity or intent. The closure expression examples below illustrate these optimizations by refining a single example of the `sort` function over several iterations, each of which expresses the same functionality in a more succinct way.

The Sort Function

Swift's standard library provides a function called `sort`, 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 `sort` function returns a new array of the same type and size as the old one, with its elements in the correct sorted order.

The closure expression examples below use the `sort` function 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 `sort` function takes two arguments:

An array of values of a known type.

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 the `sort` function's second parameter:

```
func backwards(s1: String, s2: String) -> Bool {
    return s1 > s2
}
var reversed = sort(names, backwards)
// reversed is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

If the first string (`s1`) is greater than the second string (`s2`), the `backwards` 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
}
```

Closure expression syntax can use constant parameters, variable parameters, and `inout` parameters. Default

values cannot be provided. Variadic parameters can be used if you name the variadic parameter and place it last in the parameter list. Tuples can also be used as parameter types and return types.

The example below shows a closure expression version of the `backwards` function from earlier:

```
reversed = sort(names, { (s1: String, s2: String) -> Bool in
    return s1 > s2
    })
```

Note that the declaration of parameters and return type for this inline closure is identical to the declaration from the `backwards` 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:

```
reversed = sort(names, { (s1: String, s2: String) -> Bool in return s1
                > s2 } )
```

This illustrates that the overall call to the `sort` function has remained the same. A pair of parentheses still wrap the entire set of arguments for the function. However, one of those arguments is now an inline closure.

Inferring Type From Context

Because the sorting closure is passed as an argument to a function, Swift can infer the types of its parameters and the type of the value it returns from the type of the `sort` function's second parameter. This parameter is expecting a function of type `(String, String) -> Bool`. This means that the `String`, `String`, and `Bool` 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:

```
reversed = sort(names, { s1, s2 in return s1 > s2 } )
```

It is always possible to infer parameter types and return type when passing a closure to a function as an inline closure expression. As a result, you rarely need to write an inline closure in its fullest form.

Nonetheless, you can 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 `sort` function, 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:

```
reversed = sort(names, { s1, s2 in s1 > s2 } )
```

Here, the function type of the `sort` function's second 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:

```
reversed = sort(names, { $0 > $1 } )
```

Here, `$0` and `$1` refer to the closure's first and second `String` arguments.

Operator Functions

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 function that has two parameters of type `String`, and returns a value of type `Bool`. This exactly matches the function type needed for the `sort` function's second parameter. Therefore, you can simply pass in the greater-than operator, and Swift will infer that you want to use its string-specific implementation:

```
reversed = sort(names, >)
```

For more about operator functions, see Operator Functions.

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 a closure expression that is written outside of (and *after*) the parentheses of the function call it supports:

```
func someFunctionThatTakesAClosure(closure: () -> ()) {
    // function body goes here
}

// here's how you call this function without using a trailing closure:

someFunctionThatTakesAClosure({
    // closure's body goes here
    })
```

```
// here's how you call this function with a trailing closure
        instead:

someFunctionThatTakesAClosure() {
    // trailing closure's body goes here
}
```

> NOTE
>
> If a closure expression is provided as the function'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's name when you call the function.

The string-sorting closure from the Closure Expression Syntax section above can be written outside of the `sort` function's parentheses as a trailing closure:

```
reversed = sort(names) { $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"]`:

```
let digitNames = [
```

```
    0: "Zero", 1: "One", 2: "Two",   3: "Three", 4: "Four",
    5: "Five", 6: "Six", 7: "Seven", 8: "Eight", 9: "Nine"
]
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. Note that the call to `numbers.map` does not need to include any parentheses after `map`, because the `map` method has only one parameter, and that parameter is provided as a trailing closure:

```
let strings = numbers.map {
    (var number) -> String in
    var output = ""
    while number > 0 {
        output = digitNames[number % 10]! + output
        number /= 10
    }
    return output
}
// strings is inferred to be of type String[]
// its value is ["OneSix", "FiveEight", "FiveOneZero"]
```

The `map` function 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 closure's `number` parameter is defined as a *variable parameter*, as described in Constant and Variable Parameters, so that the parameter's value can be modified within the closure body, rather than declaring a new local variable and assigning the passed `number` value to it. 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.

> 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 /= 10` 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` function.

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` function'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.

The simplest form of a closure in Swift 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 `makeIncrementor`, which contains a nested function called `incrementor`. The nested `incrementor` function captures two values, `runningTotal` and `amount`, from its surrounding context. After capturing these values, `incrementor` is returned by `makeIncrementor` as a closure that increments `runningTotal` by `amount` each time it is called.

```
func makeIncrementor(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0
    func incrementor() -> Int {
        runningTotal += amount
        return runningTotal
    }
    return incrementor
}
```

The return type of `makeIncrementor` 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 Function Types as Return Types.

The `makeIncrementor` function defines an integer variable called `runningTotal`, to store the current running total of the incrementor that will be returned. This variable is initialized with a value of `0`.

The `makeIncrementor` function has a single `Int` parameter with an external name of `forIncrement`, and a local name of `amount`. The argument value passed to this parameter specifies how much `runningTotal` should be incremented by each time the returned incrementor function is called.

`makeIncrementor` defines a nested function called `incrementor`, which performs the actual incrementing. This function simply adds `amount` to `runningTotal`, and returns the result.

When considered in isolation, the nested `incrementor` function might seem unusual:

```
func incrementor() -> Int {
    runningTotal += amount
```

```
    return runningTotal
}
```

The `incrementor` function doesn't have any parameters, and yet it refers to `runningTotal` and `amount` from within its function body. It does this by capturing the *existing* values of `runningTotal` and `amount` from its surrounding function and using them within its own function body.

Because it does not modify `amount`, `incrementor` actually captures and stores a *copy* of the value stored in `amount`. This value is stored along with the new `incrementor` function.

However, because it modifies the `runningTotal` variable each time it is called, `incrementor` captures a *reference* to the current `runningTotal` variable, and not just a copy of its initial value. Capturing a reference ensures sure that `runningTotal` does not disappear when the call to `makeIncrementor` ends, and ensures that `runningTotal` will continue to be available the next time that the incrementor function is called.

> NOTE
>
> Swift determines what should be captured by reference and what should be copied by value. You don't need to annotate `amount` or `runningTotal` to say that they can be used within the nested `incrementor` function. Swift also handles all memory management involved in disposing of `runningTotal` when it is no longer needed by the incrementor function.

Here's an example of `makeIncrementor` in action:

```
let incrementByTen = makeIncrementor(forIncrement: 10)
```

This example sets a constant called `incrementByTen` to refer to an incrementor 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 another incrementor, it will have its own stored reference to a new, separate `runningTotal` variable. In the example below, `incrementBySeven` captures a reference to a new `runningTotal` variable, and this variable is unconnected to the one captured by `incrementByTen`:

```
let incrementBySeven = makeIncrementor(forIncrement: 7)
incrementBySeven()
// returns a value of 7
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 will refer to the same closure:

```
let alsoIncrementByTen = incrementByTen
alsoIncrementByTen()
// returns a value of 50
```

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 do not have to provide a value for each member of the enumeration. If a value (known as a "raw" value) *is* provided for each enumeration member, the value can be a string, a character, or a value of any integer or floating-point type.

Alternatively, enumeration members can specify associated values of *any* type to be stored along with each different member value, much as unions or variants do in other languages. You can define a common set of related members 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 member value; can be extended to expand their functionality beyond their original implementation; and can conform to protocols to provide standard functionality.

For more on these capabilities, see Properties, Methods, Initialization, Extensions, and Protocols.

Enumeration Syntax

You introduce enumerations with the `enum` keyword and place their entire definition within a pair of braces:

```
enum SomeEnumeration {
    // enumeration definition goes here
}
```

Here's an example for the four main points of a compass:

```
enum CompassPoint {
    case North
    case South
    case East
    case West
}
```

The values defined in an enumeration (such as `North`, `South`, `East`, and `West`) are the *member values* (or *members*) of that enumeration. The `case` keyword indicates that a new line of member values is about to be defined.

> NOTE
>
> Unlike C and Objective-C, Swift enumeration members are not assigned a default integer value when they are created. In the `CompassPoints` example above, `North`, `South`, `East` and `West` do not implicitly equal `0`, `1`, `2` and `3`. Instead, the different enumeration members are fully-fledged values in their own right, with an explicitly-defined type of `CompassPoint`.

Multiple member values can appear on a single line, separated by commas:

```
enum Planet {
    case Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune
}
```

Each enumeration definition defines a brand new type. Like other types in Swift, their names (such as `CompassPoint` and `Planet`) should 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 is 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:
    println("Lots of planets have a north")
case .South:
    println("Watch out for penguins")
case .East:
    println("Where the sun rises")
case .West:
    println("Where the skies are blue")
}
// prints "Watch out for penguins"
```

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 Control Flow, a `switch` statement must be exhaustive when considering an enumeration's members. If the `case` for `.West` is omitted, this code does not compile, because it does not consider the complete list of `CompassPoint` members. Requiring exhaustiveness ensures that enumeration members are not accidentally omitted.

When it is not appropriate to provide a `case` for every enumeration member, you can provide a `default` case to cover any members that are not addressed explicitly:

```
let somePlanet = Planet.Earth
switch somePlanet {
case .Earth:
    println("Mostly harmless")
default:
    println("Not a safe place for humans")
}
// prints "Mostly harmless"
```

Associated Values

The examples in the previous section show how the members 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 is sometimes useful to be able to store *associated values* of other types alongside these member values. This enables you to store additional custom information along with the member value, and permits this information to vary each time you use that member 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 member 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-A format, which uses the numbers `0` to `9`. Each barcode has a "number system" digit, followed by ten "identifier" 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 would be convenient for an inventory tracking system to be able to store UPC-A barcodes as a tuple of three 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:

```
enum Barcode {
    case UPCA(Int, Int, Int)
    case QRCode(String)
```

```
}
```

This can be read as:

“Define an enumeration type called `Barcode`, which can take either a value of `UPCA` with an associated value of type (`Int`, `Int`, `Int`), or a value of `QRCode` with an associated value of type `String`.”

This definition does not 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.UPCA` or `Barcode.QRCode`.

New barcodes can then be created using either type:

```
var productBarcode = Barcode.UPCA(8, 85909_51226, 3)
```

This example creates a new variable called `productBarcode` and assigns it a value of `Barcode.UPCA` with an associated tuple value of `(8, 8590951226, 3)`. The provided “identifier” value has an underscore within its integer literal—`85909_51226`—to make it easier to read as a barcode.

The same product can be assigned a different type of barcode:

```
productBarcode = .QRCode("ABCDEFGHIJKLMNOP")
```

At this point, the original `Barcode.UPCA` and its integer values are replaced by the new `Barcode.QRCode` and its string value. Constants and variables of type `Barcode` can store either a `.UPCA` or a `.QRCode` (together with their associated values), but they can only store one of them at any given time.

The different barcode types can be checked using a switch statement, as before. This time, however, the associated values can be 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 {
case .UPCA(let numberSystem, let identifier, let check):
    println("UPC-A with value of \(numberSystem), \(identifier), \
```

```
                (check).")
case .QRCode(let productCode):
    println("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

If all of the associated values for a enumeration member are extracted as constants, or if all are extracted as variables, you can place a single `var` or `let` annotation before the member name, for brevity:

```
switch productBarcode {
case let .UPCA(numberSystem, identifier, check):
    println("UPC-A with value of \(numberSystem), \(identifier), \
                (check).")
case let .QRCode(productCode):
    println("QR code with value of \(productCode).")
}
// prints "QR code with value of ABCDEFGHIJKLMNOP."
```

Raw Values

The barcode example in Associated Values shows how members of an enumeration can declare that they store associated values of different types. As an alternative to associated values, enumeration members 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 members:

```
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 Strings and Characters.

Note that 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 member is always the same. Associated values are set when you create a new constant or variable based on one of the enumeration's members, and can be different each time you do so.

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. When integers are used for raw values, they auto-increment if no value is specified for some of the enumeration members.

The enumeration below is a refinement of the earlier `Planet` enumeration, with raw integer values to represent each planet's order from the sun:

```
enum Planet: Int {
    case Mercury = 1, Venus, Earth, Mars, Jupiter, Saturn, Uranus,
               Neptune
}
```

Auto-incrementation means that `Planet.Venus` has a raw value of `2`, and so on.

Access the raw value of an enumeration member with its `toRaw` method:

```
let earthsOrder = Planet.Earth.toRaw()
// earthsOrder is 3
```

Use an enumeration's `fromRaw` method to try to find an enumeration member with a particular raw value. This example identifies Uranus from its raw value of `7`:

```
let possiblePlanet = Planet.fromRaw(7)
// possiblePlanet is of type Planet? and equals Planet.Uranus
```

Not all possible `Int` values will find a matching planet, however. Because of this, the `fromRaw` method

returns an *optional* enumeration member. In the example above, `possiblePlanet` is of type `Planet?`, or "optional `Planet`."

If you try to find a Planet with a position of `9`, the optional `Planet` value returned by `fromRaw` will be `nil`:

```
let positionToFind = 9
if let somePlanet = Planet.fromRaw(positionToFind) {
    switch somePlanet {
    case .Earth:
        println("Mostly harmless")
    default:
        println("Not a safe place for humans")
    }
} else {
    println("There isn't a planet at position \
        (positionToFind)")
}
// prints "There isn't a planet at position 9"
```

This example uses optional binding to try to access a planet with a raw value of `9`. The statement `if let somePlanet = Planet.fromRaw(9)` retrieves an optional `Planet`, and sets `somePlanet` to the contents of that optional `Planet` if it can be retrieved. In this case, it is not possible to retrieve a planet with a position of `9`, and so the `else` branch is executed instead.

Classes and Structures

Classes and *structures* 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 classes and structures by using exactly the same syntax as for constants, variables, and functions.

Unlike other programming languages, Swift does not require you to create separate interface and implementation files for custom classes and structures. In Swift, you define a class or a structure 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 classes and structures are much closer in functionality than in other languages, and much of this chapter describes functionality that can apply to instances of *either* a class or a structure type. Because of this, the more general term *instance* is used.

Comparing Classes and Structures

Classes and structures 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 Properties, Methods, Subscripts, Initialization, Extensions, and Protocols.

Classes have additional capabilities that structures do not:

- 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 Inheritance, Type Casting, Initialization, and Automatic Reference Counting.

> NOTE
>
> Structures are always copied when they are passed around in your code, and do not use reference counting.

Definition Syntax

Classes and structures have a similar definition syntax. You introduce classes with the `class` keyword and structures with the `struct` keyword. Both place their entire definition within a pair of braces:

```
class SomeClass {
    // class definition goes here
}
struct SomeStructure {
    // structure definition goes here
}
```

> NOTE
>
> Whenever you define a new class or structure, you effectively define a brand new Swift type. Give types `UpperCamelCase` names (such as `SomeClass` and `SomeStructure` here) to match the capitalization of standard Swift types (such as `String`, `Int`, and `Bool`). Conversely, always 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:

```
struct Resolution {
    var width = 0
    var height = 0
}
class VideoMode {
    var resolution = Resolution()
    var interlaced = false
    var frameRate = 0.0
    var name: String?
}
```

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 class or structure. 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 "non-interlaced 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 is of an optional type.

Class and Structure Instances

The `Resolution` structure definition and the `VideoMode` class definition only describe what a `Resolution` or `VideoMode` will look like. They themselves do not 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:

```
let someResolution = Resolution()
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:

```
println("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 sub-properties, such as the `width` property in the `resolution` property of a `VideoMode`:

```
println("The width of someVideoMode is \
               (someVideoMode.resolution.width)")
```

```
// prints "The width of someVideoMode is 0"
```

You can also use dot syntax to assign a new value to a variable property:

```
someVideoMode.resolution.width = 1280
println("The width of someVideoMode is now \
              (someVideoMode.resolution.width)")
// prints "The width of someVideoMode is now 1280"
```

> NOTE
>
> Unlike Objective-C, Swift enables you to set sub-properties of a structure property directly. In the last example above, the `width` property of the `resolution` property of `someVideoMode` is set directly, without your needing to set the entire `resolution` property to a new value.

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 do not receive a default memberwise initializer. Initializers are described in more detail in Initialization.

Structures and Enumerations Are Value Types

A *value type* is a type that is *copied* when it is assigned to a variable or constant, or when it is 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.

Consider this example, which uses the `Resolution` structure from the previous example:

```
let hd = Resolution(width: 1920, height: 1080)
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`:

```
println("cinema is now \(cinema.width) pixels wide")
// prints "cinema is now 2048 pixels wide"
```

However, the `width` property of the original `hd` instance still has the old value of `1920`:

```
println("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 is two completely separate instances, which just happened to contain the same numeric values. Because they are separate instances, setting the width of `cinema` to `2048` doesn't affect the width stored in `hd`.

The same behavior applies to enumerations:

```
enum CompassPoint {
    case North, South, East, West
}
var currentDirection = CompassPoint.West
let rememberedDirection = currentDirection
currentDirection = .East
if rememberedDirection == .West {
    println("The remembered direction is still .West")
}
// prints "The remembered direction is still .West"
```

When `rememberedDirection` is assigned the value of `currentDirection`, it is actually set to a copy of that value. Changing the value of `currentDirection` thereafter does not 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 instead.

Here's an example, using the `VideoMode` class defined above:

```
let tenEighty = VideoMode()
tenEighty.resolution = hd
tenEighty.interlaced = true
tenEighty.name = "1080i"
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 is set to be interlaced, and is given a name of `"1080i"`. Finally, it is set to a frame rate of `25.0` frames per second.

Next, `tenEighty` is assigned to a new constant, called `alsoTenEighty`, and the frame rate of `alsoTenEighty` is modified:

```
let alsoTenEighty = tenEighty
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.

Checking the `frameRate` property of `tenEighty` shows that it correctly reports the new frame rate of `30.0` from the underlying `VideoMode` instance:

```
println("The frameRate property of tenEighty is now \
             (tenEighty.frameRate)")
// prints "The frameRate property of tenEighty is now 30.0"
```

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 do not actually change. `tenEighty` and `alsoTenEighty` themselves do not "store" the `VideoMode` instance—instead, they both *refer* to a `VideoMode` instance behind the scenes. It is 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 is possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same is not true for structures and enumerations, because they are value types and 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 if 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:

```
if tenEighty === alsoTenEighty {
    println("tenEighty and alsoTenEighty refer to the same Resolution instance.")
}
// prints "tenEighty and alsoTenEighty refer to the same Resolution instance."
```

Note that “identical to” (represented by three equals signs, or ===) does not 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 classes and structures, it is your responsibility to decide what qualifies as two instances being “equal”. The process of defining your own implementations of the “equal to” and “not equal to” operators is described in Equivalence Operators.

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 is not a direct pointer to an address in memory, and does not 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.

Choosing Between Classes and Structures

You can use both classes and structures to define custom data types to use as the building blocks of your program's code.

However, structure instances are always passed by *value*, and class instances are always passed by *reference*. This means that they are suited to different kinds of tasks. As you consider the data constructs and functionality that you need for a project, decide whether each data construct should be defined as a class or as a structure.

As a general guideline, consider creating a structure when one or more of these conditions apply:

- The structure's primary purpose is to encapsulate a few relatively simple data values.
- It is reasonable to expect that the encapsulated values will be copied rather than referenced when you assign or pass around an instance of that structure.
- Any properties stored by the structure are themselves value types, which would also be expected to be copied rather than referenced.
- The structure does not need to inherit properties or behavior from another existing type.

Examples of good candidates for structures include:

- The size of a geometric shape, perhaps encapsulating a `width` property and a `height` property, both of type `Double`.
- A way to refer to ranges within a series, perhaps encapsulating a `start` property and a

`length` property, both of type `Int`.

A point in a 3D coordinate system, perhaps encapsulating `x`, `y` and `z` properties, each of type `Double`.

In all other cases, define a class, and create instances of that class to be managed and passed by reference. In practice, this means that most custom data constructs should be classes, not structures.

Assignment and Copy Behavior for Collection Types

Swift's `Array` and `Dictionary` types are implemented as structures. However, arrays have slightly different copying behavior from dictionaries and other structures when they are assigned to a constant or variable, and when they are passed to a function or method.

The behavior described for `Array` and `Dictionary` below is different again from the behavior of `NSArray` and `NSDictionary` in Foundation, which are implemented as classes, not structures. `NSArray` and `NSDictionary` instances are always assigned and passed around as a reference to an existing instance, rather than as a copy.

> NOTE
>
> The descriptions below refer to the "copying" of arrays, dictionaries, strings, and other values. Where copying is mentioned, the behavior you see in your code will always be as if a copy took place. However, Swift only performs an *actual* copy behind the scenes when it is absolutely necessary to do so. Swift manages all value copying to ensure optimal performance, and you should not avoid assignment to try to preempt this optimization.

Assignment and Copy Behavior for Dictionaries

Whenever you assign a `Dictionary` instance to a constant or variable, or pass a `Dictionary` instance as an argument to a function or method call, the dictionary is *copied* at the point that the assignment or call takes

place. This process is described in Structures and Enumerations Are Value Types.

If the keys and/or values stored in the `Dictionary` instance are value types (structures or enumerations), they too are copied when the assignment or call takes place. Conversely, if the keys and/or values are reference types (classes or functions), the references are copied, but not the class instances or functions that they refer to. This copy behavior for a dictionary's keys and values is the same as the copy behavior for a structure's stored properties when the structure is copied.

The example below defines a dictionary called `ages`, which stores the names and ages of four people. The `ages` dictionary is then assigned to a new variable called `copiedAges` and is copied when this assignment takes place. After the assignment, `ages` and `copiedAges` are two separate dictionaries.

```
var ages = ["Peter": 23, "Wei": 35, "Anish": 65, "Katya": 19]
var copiedAges = ages
```

The keys for this dictionary are of type `String`, and the values are of type `Int`. Both types are value types in Swift, and so the keys and values are also copied when the dictionary copy takes place.

You can prove that the `ages` dictionary has been copied by changing an age value in one of the dictionaries and checking the corresponding value in the other. If you set the value for `"Peter"` in the `copiedAges` dictionary to `24`, the `ages` dictionary still returns the old value of `23` from before the copy took place:

```
copiedAges["Peter"] = 24
println(ages["Peter"])
// prints "23"
```

Assignment and Copy Behavior for Arrays

The assignment and copy behavior for Swift's `Array` type is more complex than for its `Dictionary` type. `Array` provides C-like performance when you work with an array's contents and copies an array's contents only when copying is necessary.

If you assign an `Array` instance to a constant or variable, or pass an `Array` instance as an argument to a

function or method call, the contents of the array are *not* copied at the point that the assignment or call takes place. Instead, both arrays share the same sequence of element values. When you modify an element value through one array, the result is observable through the other.

For arrays, copying only takes place when you perform an action that has the potential to modify the *length* of the array. This includes appending, inserting, or removing items, or using a ranged subscript to replace a range of items in the array. If and when array copying does take place, the copy behavior for an array's contents is the same as for a dictionary's keys and values, as described in Assignment and Copy Behavior for Dictionaries.

The example below assigns a new array of `Int` values to a variable called `a`. This array is also assigned to two further variables called `b` and `c`:

```
var a = [1, 2, 3]
var b = a
var c = a
```

You can retrieve the first value in the array with subscript syntax on either `a`, `b`, or `c`:

```
println(a[0])
// 1
println(b[0])
// 1
println(c[0])
// 1
```

If you set an item in the array to a new value with subscript syntax, all three of `a`, `b`, and `c` will return the new value. Note that the array is not copied when you set a new value with subscript syntax, because setting a single value with subscript syntax does not have the potential to change the array's length:

```
a[0] = 42
println(a[0])
// 42
println(b[0])
// 42
println(c[0])
```

```
// 42
```

However, if you append a new item to `a`, you *do* modify the array's length. This prompts Swift to create a new copy of the array at the point that you append the new value. Henceforth, `a` is a separate, independent copy of the array.

If you change a value in `a` after the copy is made, `a` will return a different value from `b` and `c`, which both still reference the original array contents from before the copy took place:

```
a.append(4)
a[0] = 777
println(a[0])
// 777
println(b[0])
// 42
println(c[0])
// 42
```

Ensuring That an Array Is Unique

It can be useful to ensure that you have a unique copy of an array before performing an action on that array's contents, or before passing that array to a function or method. You ensure the uniqueness of an array reference by calling the `unshare` method on a variable of array type. (The `unshare` method cannot be called on a constant array.)

If multiple variables currently refer to the same array, and you call the `unshare` method on one of those variables, the array is copied, so that the variable has its own independent copy of the array. However, no copying takes place if the variable is already the only reference to the array.

At the end of the previous example, `b` and `c` both reference the same array. Call the `unshare` method on `b` to make it become a unique copy:

```
b.unshare()
```

If you change the first value in `b` after calling the `unshare` method, all three arrays will now report a different value:

```
b[0] = -105
println(a[0])
// 777
println(b[0])
// -105
println(c[0])
// 42
```

Checking Whether Two Arrays Share the Same Elements

Check whether two arrays or subarrays share the same storage and elements by comparing them with the identity operators (`===` and `!==`).

The example below uses the “identical to” operator (`===`) to check whether `b` and `c` still share the same array elements:

```
if b === c {
    println("b and c still share the same array elements.")
} else {
    println("b and c now refer to two independent sets of array
             elements.")
}
// prints "b and c now refer to two independent sets of array
             elements."
```

Alternatively, use the identity operators to check whether two subarrays share the same elements. The example below compares two identical subarrays from `b` and confirms that they refer to the same elements:

```
if b[0...1] === b[0...1] {
    println("These two subarrays share the same elements.")
```

```
} else {
    println("These two subarrays do not share the same elements.")
}
// prints "These two subarrays share the same elements."
```

Forcing a Copy of an Array

Force an explicit copy of an array by calling the array's `copy` method. This method performs a shallow copy of the array and returns a new array containing the copied items.

The example below defines an array called `names`, which stores the names of seven people. A new variable called `copiedNames` is set to the result of calling the `copy` method on the `names` array:

```
var names = ["Mohsen", "Hilary", "Justyn", "Amy", "Rich", "Graham",
             "Vic"]
var copiedNames = names.copy()
```

You can prove that the `names` array has been copied by changing an item in one of the arrays and checking the corresponding item in the other. If you set the first item in the `copiedNames` array to `"Mo"` rather than `"Mohsen"`, the `names` array still returns the old value of `"Mohsen"` from before the copy took place:

```
copiedNames[0] = "Mo"
println(names[0])
// prints "Mohsen"
```

> NOTE
>
> If you simply need to be sure that your reference to an array's contents is the only reference in existence, call the `unshare` method, not the `copy` method. The `unshare` method does not make a copy of the array unless it is necessary to do so. The `copy` method always copies the array, even if it is already unshared.

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.

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 Default Property Values. 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 Modifying Constant Properties During Initialization.

The example below defines a structure called `FixedLengthRange`, which describes a range of integers whose range length cannot be changed once 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:

```
let rangeOfFourItems = FixedLengthRange(firstValue: 0, length: 4)
// this range represents integer values 0, 1, 2, and 3
rangeOfFourItems.firstValue = 6
// this will report an error, even thought firstValue is a variable
              property
```

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` attribute before its declaration.

> NOTE
>
> You must always declare a lazy property as a variable (with the `var` keyword), because its initial value may 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 `DataImporter` and `DataManager`, 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 non-trivial amount of time to
              initialize.
    */
    var fileName = "data.txt"
    // the DataImporter class would provide data importing
              functionality here
}

class DataManager {
    @lazy var importer = DataImporter()
    var data = String[]()
    // the DataManager class would provide data management
        functionality here
}

let manager = DataManager()
manager.data += "Some data"
```

```
manager.data += "Some more data"
// the DataImporter instance for the importer property has not
        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 non-trivial 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` attribute, 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:

```
println(manager.importer.fileName)
// the DataImporter instance for the importer property has now been
        created
// prints "data.txt"
```

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.

```
struct Point {
    var x = 0.0, y = 0.0
}
struct Size {
    var width = 0.0, height = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
    var center: Point {
    get {
        let centerX = origin.x + (size.width / 2)
        let centerY = origin.y + (size.height / 2)
        return Point(x: centerX, y: centerY)
    }
    set(newCenter) {
        origin.x = newCenter.x - (size.width / 2)
        origin.y = newCenter.y - (size.height / 2)
    }
    }
}
var square = Rect(origin: Point(x: 0.0, y: 0.0),
    size: Size(width: 10.0, height: 10.0))
let initialSquareCenter = square.center
square.center = Point(x: 15.0, y: 15.0)
```

```
println("square.origin is now at (\(square.origin.x), \
        (square.origin.y))")
// prints "square.origin is now at (10.0, 10.0)"
```

This example defines three structures for working with geometric shapes:

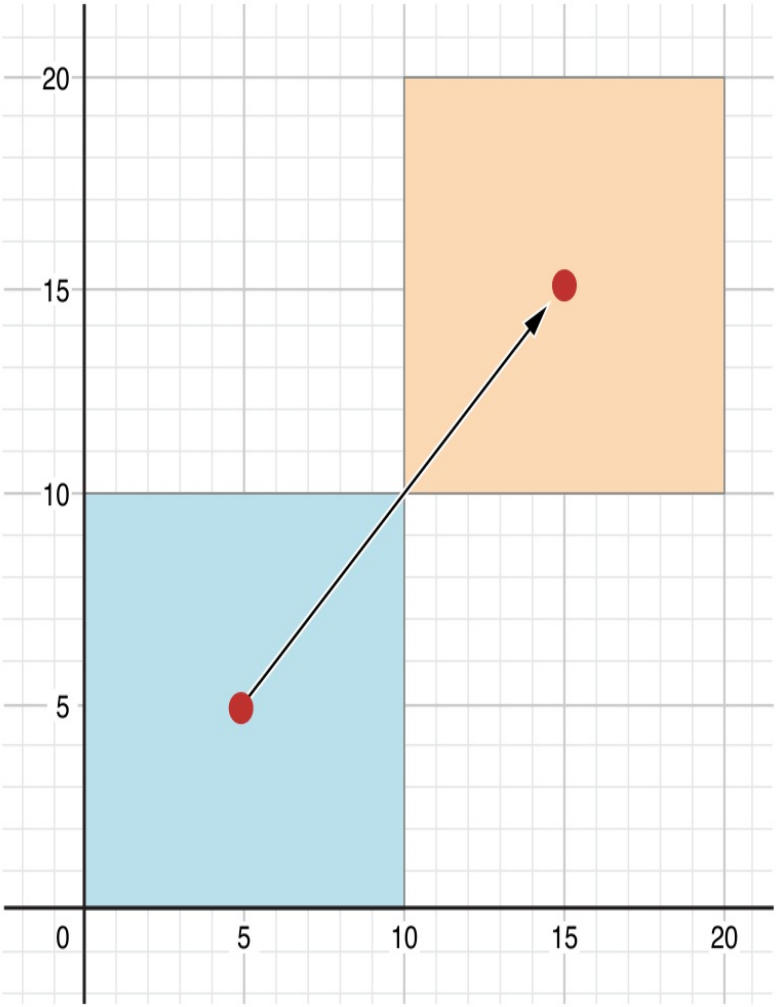
- `Point` encapsulates an `(x, y)` coordinate.
- `Size` encapsulates a `width` and a `height`.
- `Rect` defines a rectangle by an origin point and a size.

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 preceding example 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 `x` 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 does not 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, which takes advantage of this shorthand notation:

```
struct AlternativeRect {
    var origin = Point()
    var size = Size()
    var center: Point {
    get {
        let centerX = origin.x + (size.width / 2)
        let centerY = origin.y + (size.height / 2)
        return Point(x: centerX, y: centerY)
    }
      set {
          origin.x = newValue.x - (size.width / 2)
          origin.y = newValue.y - (size.height / 2)
      }
      }
}
```

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 `var` keyword, because their value is not fixed. The `let` 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 {
    var width = 0.0, height = 0.0, depth = 0.0
    var volume: Double {
    return width * height * depth
    }
}
let fourByFiveByTwo = Cuboid(width: 4.0, height: 5.0, depth: 2.0)
println("the volume of fourByFiveByTwo is \(fourByFiveByTwo.volume)")
// prints "the volume of fourByFiveByTwo is 40.0"
```

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, apart from lazy stored properties. You can also add property observers to any inherited property (whether stored or computed) by overriding the property within a subclass. Property overriding is described in Overriding.

> NOTE
>
> You don't need to define property observers for non-overridden computed properties, because you can observe and respond to changes to their value from directly within the computed property's setter.

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 is 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 choose not to write the parameter name and parentheses within your implementation, the parameter will still be made available with a default parameter name of `newValue`.

Similarly, if you implement a `didSet` observer, it will be passed a constant parameter containing the old property value. You can name the parameter if you wish, or use the default parameter name of `oldValue`.

> NOTE
>
> `willSet` and `didSet` observers are not called when a property is first initialized. They are only called when the property's value is set outside of an initialization context.

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.

```
class StepCounter {
    var totalSteps: Int = 0 {
    willSet(newTotalSteps) {
        println("About to set totalSteps to \(newTotalSteps)")
    }
    didSet {
        if totalSteps > oldValue  {
            println("Added \(totalSteps - oldValue) steps")
```

```
        }
    }
    }
}
let stepCounter = StepCounter()
stepCounter.totalSteps = 200
// About to set totalSteps to 200
// Added 200 steps
stepCounter.totalSteps = 360
// About to set totalSteps to 360
// Added 160 steps
stepCounter.totalSteps = 896
// About to set totalSteps to 896
// 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 assign a value to a property within its own `didSet` observer, the new value that you assign will replace the one that was just set.

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 rather than store a value, and are written in the same way as computed properties.

> NOTE
>
> Global constants and variables are always computed lazily, in a similar manner to Lazy Stored Properties. Unlike lazy stored properties, global constants and variables do not need to be marked with the `@lazy` attribute.
>
> 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).

For value types (that is, structures and enumerations), you can define stored and computed type properties. For classes, you can define computed type properties only.

Stored type properties for value types 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.

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 for value types with the `static` keyword, and type properties for class types with the `class` keyword. The example below shows the syntax for stored and computed type properties:

```
struct SomeStructure {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
    // return an Int value here
    }
}
```

```
enum SomeEnumeration {
    static var storedTypeProperty = "Some value."
    static var computedTypeProperty: Int {
        // return an Int value here
        }
}
class SomeClass {
    class var computedTypeProperty: Int {
    // return an Int value here
    }
}
```

> 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.

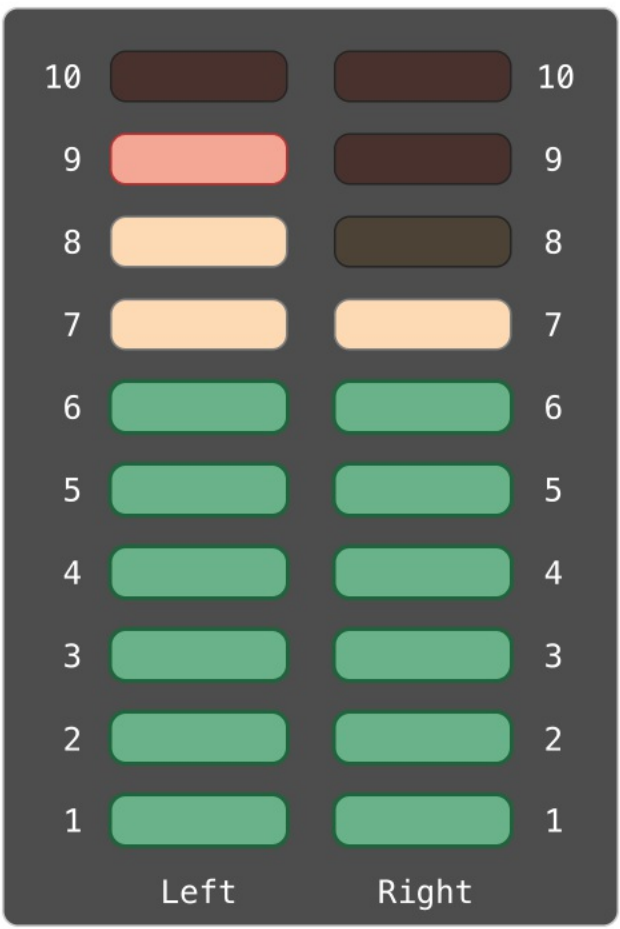
Querying and Setting Type Properties

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:

```
println(SomeClass.computedTypeProperty)
// prints "42"

println(SomeStructure.storedTypeProperty)
// prints "Some value."
SomeStructure.storedTypeProperty = "Another value."
println(SomeStructure.storedTypeProperty)
// prints "Another value."
```

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
    static var maxInputLevelForAllChannels = 0
    var currentLevel: Int = 0 {
    didSet {
        if currentLevel > AudioChannel.thresholdLevel {
            // cap the new audio level to the threshold level
            currentLevel = AudioChannel.thresholdLevel
        }
          if currentLevel > AudioChannel.maxInputLevelForAllChannels {
              // store this as the new overall maximum input level
              AudioChannel.maxInputLevelForAllChannels = currentLevel
          }
      }
      }
}
```

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` static 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`:

```
leftChannel.currentLevel = 7
println(leftChannel.currentLevel)
// prints "7"
println(AudioChannel.maxInputLevelForAllChannels)
// 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
println(rightChannel.currentLevel)
// prints "10"
println(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 Functions.

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:

```
class Counter {
    var count = 0
    func increment() {
        count++
    }
    func incrementBy(amount: Int) {
```

```
        count += amount
    }
    func reset() {
        count = 0
    }
}
```

The `Counter` class defines three instance methods:

- `increment` increments the counter by `1`.
- `incrementBy(amount: Int)` increments the counter by an 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:

```
let counter = Counter()
// the initial counter value is 0
counter.increment()
// the counter's value is now 1
counter.incrementBy(5)
// the counter's value is now 6
counter.reset()
// the counter's value is now 0
```

Local and External Parameter Names for Methods

Function parameters can have both a local name (for use within the function's body) and an external name (for use when calling the function), as described in External Parameter Names. The same is true for method parameters, because methods are just functions that are associated with a type. However, the default behavior of local names and external names is different for functions and methods.

Methods in Swift are very similar to their counterparts in Objective-C. As in Objective-C, the name of a method in Swift typically refers to the method's first parameter using a preposition such as `with`, `for`, or `by`, as seen in the `incrementBy` method from the preceding `Counter` class example. The use of a preposition enables the method to be read as a sentence when it is called. Swift makes this established method naming convention easy to write by using a different default approach for method parameters than it uses for function parameters.

Specifically, Swift gives the *first* parameter name in a method a local parameter name by default, and gives the second and subsequent parameter names both local *and* external parameter names by default. This convention matches the typical naming and calling convention you will be familiar with from writing Objective-C methods, and makes for expressive method calls without the need to qualify your parameter names.

Consider this alternative version of the `Counter` class, which defines a more complex form of the `incrementBy` method:

```
class Counter {
    var count: Int = 0
    func incrementBy(amount: Int, numberOfTimes: Int) {
        count += amount * numberOfTimes
    }
}
```

This `incrementBy` method has two parameters—`amount` and `numberOfTimes`. By default, Swift treats `amount` as a local name only, but treats `numberOfTimes` as both a local *and* an external name. You call the method as follows:

```
let counter = Counter()
counter.incrementBy(5, numberOfTimes: 3)
// counter value is now 15
```

You don't need to define an external parameter name for the first argument value, because its purpose is clear from the function name `incrementBy`. The second argument, however, is qualified by an external parameter name to make its purpose clear when the method is called.

This default behavior effectively treats the method as if you had written a hash symbol (`#`) before the `numberOfTimes` parameter:

```
func incrementBy(amount: Int, #numberOfTimes: Int) {
    count += amount * numberOfTimes
}
```

The default behavior described above mean that method definitions in Swift are written with the same grammatical style as Objective-C, and are called in a natural, expressive way.

Modifying External Parameter Name Behavior for Methods

Sometimes it's useful to provide an external parameter name for a method's first parameter, even though this is not the default behavior. You can either add an explicit external name yourself, or you can prefix the first parameter's name with a hash symbol to use the local name as an external name too.

Conversely, if you do not want to provide an external name for the second or subsequent parameter of a method, override the default behavior by using an underscore character (`_`) as an explicit external parameter name for that parameter.

The self Property

Every instance of a type has an implicit property called `self`, which is exactly equivalent to the instance itself. You use this implicit `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++
}
```

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 implicit `self` property to distinguish between the parameter name and the property name.

Here, `self` disambiguates between a method parameter called `x` and an instance property that is also called `x`:

```
struct Point {
    var x = 0.0, y = 0.0
    func isToTheRightOfX(x: Double) -> Bool {
        return self.x > x
    }
}
let somePoint = Point(x: 4.0, y: 5.0)
if somePoint.isToTheRightOfX(1.0) {
    println("This point is to the right of the line where x == 1.0")
}
// prints "This point is to the right of the line where x == 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 moveByX(deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
var somePoint = Point(x: 1.0, y: 1.0)
somePoint.moveByX(2.0, y: 3.0)
println("The point is now at (\(somePoint.x), \(somePoint.y))")
// prints "The point is now at (3.0, 4.0)"
```

The `Point` structure above defines a mutating `moveByX` 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 Stored Properties of Constant Structure Instances:

```
let fixedPoint = Point(x: 3.0, y: 3.0)
fixedPoint.moveByX(2.0, y: 3.0)
// 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 moveByX(deltaX: Double, y deltaY: Double) {
```

```
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

This version of the mutating `moveByX` method creates a brand 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 member from the same enumeration:

```
enum TriStateSwitch {
    case Off, Low, High
    mutating func next() {
        switch self {
        case Off:
            self = Low
        case Low:
            self = High
        case High:
            self = Off
        }
    }
}
var ovenLight = TriStateSwitch.Low
ovenLight.next()
// ovenLight is now equal to .High
ovenLight.next()
// ovenLight is now equal to .Off
```

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 are called 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 for classes by writing the keyword `class` before the method's `func` keyword, and type methods for structures and enumerations by writing the keyword `static` before the method's `func` keyword.

> 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. For structures and enumerations, this means that you can use `self` to disambiguate between static properties and static 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 static properties by using the static 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 static 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.

```
struct LevelTracker {
    static var highestUnlockedLevel = 1
    static func unlockLevel(level: Int) {
        if level > highestUnlockedLevel { highestUnlockedLevel = level
              }
    }
    static func levelIsUnlocked(level: Int) -> Bool {
        return level <= highestUnlockedLevel
    }
    var currentLevel = 1
      mutating func advanceToLevel(level: Int) -> Bool {
          if LevelTracker.levelIsUnlocked(level) {
              currentLevel = level
              return true
          } else {
              return false
          }
      }
}
```

The `LevelTracker` structure keeps track of the highest level that any player has unlocked. This value is stored in a static property called `highestUnlockedLevel`.

`LevelTracker` also defines two type functions to work with the `highestUnlockedLevel` property. The first is a type function called `unlockLevel`, which updates the value of `highestUnlockedLevel` whenever a new level is unlocked. The second is a convenience type function called `levelIsUnlocked`, which returns `true` if a particular level number is already unlocked. (Note that these type methods can access

the `highestUnlockedLevel` static property without your needing to write it as `LevelTracker.highestUnlockedLevel`.)

In addition to its static 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 `advanceToLevel`. Before updating `currentLevel`, this method checks whether the requested new level is already unlocked. The `advanceToLevel` method returns a Boolean value to indicate whether or not it was actually able to set `currentLevel`.

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()
    let playerName: String
    func completedLevel(level: Int) {
        LevelTracker.unlockLevel(level + 1)
        tracker.advanceToLevel(level + 1)
    }
    init(name: String) {
        playerName = name
    }
}
```

The `Player` class creates a new instance of `LevelTracker` to track that player's progress. It also provides a method called `completedLevel`, 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 `advanceToLevel` is ignored, because the level is known to have been unlocked by the call to `LevelTracker.unlockLevel` on the previous line.)

You can create a instance of the `Player` class for a new player, and see what happens when the player completes level one:

```
var player = Player(name: "Argyrios")
player.completedLevel(1)
println("highest unlocked level is now \
             (LevelTracker.highestUnlockedLevel)")
// prints "highest unlocked level is now 2"
```

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.advanceToLevel(6) {
    println("player is now on level 6")
} else {
    println("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 `someArray[index]` and elements in a `Dictionary` instance as `someDictionary[key]`.

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
    }
}
```

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 drop the `get` keyword for read-only subscripts:

```
subscript(index: Int) -> Int {
    // return an appropriate subscript value here
}
```

Here's an example of a read-only subscript implementation, which defines a `TimesTable` structure to represent an *n*-times-table of integers:

```
struct TimesTable {
    let multiplier: Int
    subscript(index: Int) -> Int {
        return multiplier * index
    }
}
let threeTimesTable = TimesTable(multiplier: 3)
println("six times three is \(threeTimesTable[6])")
// 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 braces, 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 `Dictionary<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 Accessing and Modifying a Dictionary.

> NOTE
>
> Swift’s `Dictionary` type implements its key-value subscripting as a subscript that takes and receives 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 variable parameters and variadic parameters, but cannot 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 braces 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 {
    let rows: Int, columns: Int
    var grid: Double[]
    init(rows: Int, columns: Int) {
        self.rows = rows
        self.columns = columns
        grid = Array(count: rows * columns, repeatedValue: 0.0)
    }
    func indexIsValidForRow(row: Int, column: Int) -> Bool {
        return row >= 0 && row < rows && column >= 0 && column < columns
    }
    subscript(row: Int, column: Int) -> Double {
        get {
            assert(indexIsValidForRow(row, column: column), "Index out of range")
            return grid[(row * columns) + column]
        }
        set {
            assert(indexIsValidForRow(row, column: column), "Index out of range")
            grid[(row * columns) + column] = newValue
```

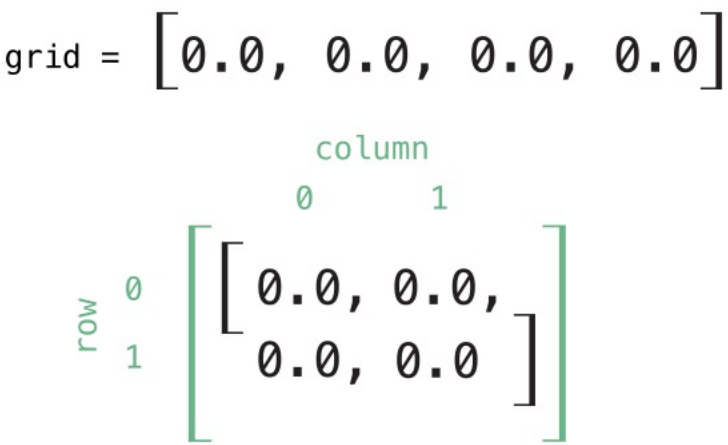
```
20              }
21          }
22      }
```

`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 and Initializing an Array.

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 preceding example 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:

```
matrix[0, 1] = 1.5
```

```
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`):

$$\begin{bmatrix} 0.0 & 1.5 \\ 3.2 & 0.0 \end{bmatrix}$$

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`, which checks whether the requested `row` or `column` is outside the bounds of the matrix:

```
func indexIsValidForRow(row: Int, column: Int) -> Bool {
    return row >= 0 && row < rows && column >= 0 && column < columns
}
```

An assertion is triggered if you try to access a subscript that is outside of the matrix bounds:

```
let someValue = matrix[2, 2]
// this triggers an assert, because [2, 2] is outside of the matrix
               bounds
```

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 `Vehicle`. This base class declares two properties (`numberOfWheels` and `maxPassengers`) that are universal to all vehicles. These properties are used by a method called `description`, which returns a `String` description of the vehicle's characteristics:

```
class Vehicle {
    var numberOfWheels: Int
    var maxPassengers: Int
    func description() -> String {
        return "\(numberOfWheels) wheels; up to \(maxPassengers)
              passengers"
    }
    init() {
        numberOfWheels = 0
        maxPassengers = 1
    }
}
```

The `Vehicle` class also defines an *initializer* to set up its properties. Initializers are described in detail in Initialization, but a brief introduction is required here in order to illustrate how inherited properties can be modified by subclasses.

You use initializers to create a new instance of a type. Although initializers are not methods, they are written in a very similar syntax to instance methods. An initializer prepares a new instance for use, and ensures that all properties of the instance have valid initial values.

In its simplest form, an initializer is like an instance method with no parameters, written using the `init` keyword:

```
init() {
    // perform some initialization here
}
```

To create a new instance of `Vehicle`, call this initializer with *initializer syntax*, written as `TypeName` followed by empty parentheses:

```
let someVehicle = Vehicle()
```

The initializer for `Vehicle` sets some initial property values (`numberOfWheels = 0` and `maxPassengers = 1`) for an arbitrary vehicle.

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 vehicle.

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 refine. You can also add new characteristics to the subclass.

To indicate that a class has a superclass, write the superclass name after the original class name, separated by a colon:

```
class SomeClass: SomeSuperclass {
    // class definition goes here
}
```

The next example defines a second, more specific vehicle called `Bicycle`. This new class is based on the existing capabilities of `Vehicle`. You indicate this by placing the name of the class the subclass builds upon (`Vehicle`) after its own name (`Bicycle`), separated by a colon.

This can be read as:

“Define a new class called `Bicycle`, which inherits the characteristics of `Vehicle`”:

```
class Bicycle: Vehicle {
    init() {
        super.init()
        numberOfWheels = 2
    }
}
```

`Bicycle` is a subclass of `Vehicle`, and `Vehicle` is the superclass of `Bicycle`. The new `Bicycle` class automatically gains all characteristics of `Vehicle`, such as its `maxPassengers` and `numberOfWheels`

properties. You can tailor those characteristics and add new ones to better match the requirements of the `Bicycle` class.

The `Bicycle` class also defines an initializer to set up its tailored characteristics. The initializer for `Bicycle` calls `super.init()`, the initializer for the `Bicycle` class's superclass, `Vehicle`, and ensures that all of the inherited properties are initialized by `Vehicle` before `Bicycle` tries to modify them.

> NOTE
>
> Unlike Objective-C, initializers are not inherited by default in Swift. For more information, see Initializer Inheritance and Overriding.

The default value of `maxPassengers` provided by `Vehicle` is already correct for a bicycle, and so it is not changed within the initializer for `Bicycle`. The original value of `numberOfWheels` is not correct, however, and is replaced with a new value of `2`.

As well as inheriting the properties of `Vehicle`, `Bicycle` also inherits its methods. If you create an instance of `Bicycle`, you can call its inherited `description` method to see how its properties have been updated:

```
let bicycle = Bicycle()
println("Bicycle: \(bicycle.description())")
// Bicycle: 2 wheels; up to 1 passengers
```

Subclasses can themselves be subclassed:

```
class Tandem: Bicycle {
    init() {
        super.init()
        maxPassengers = 2
    }
}
```

This example creates a subclass of `Bicycle` for a two-seater bicycle known as a "tandem". `Tandem` inherits the two properties from `Bicycle`, which in turn inherits these properties from `Vehicle`. `Tandem` doesn't change the number of wheels—it's still a bicycle, after all—but it does update `maxPassengers` to have the correct value for a tandem.

> NOTE
>
> Subclasses are only allowed to modify *variable* properties of superclasses during initialization. You can't modify inherited constant properties of subclasses.

Creating an instance of `Tandem` and printing its description shows how its properties have been updated:

```
let tandem = Tandem()
println("Tandem: \(tandem.description())")
// Tandem: 2 wheels; up to 2 passengers
```

Note that the `description` method is also inherited by `Tandem`. Instance methods of a class are inherited by any and all subclasses of that class.

Overriding

A subclass can provide its own custom implementation of an instance method, class method, instance 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 class method to provide a tailored or alternative implementation of the method within your subclass.

The following example defines a new subclass of `Vehicle` called `Car`, which overrides the `description` method it inherits from `Vehicle`:

```
class Car: Vehicle {
    var speed: Double = 0.0
    init() {
        super.init()
```

```
        maxPassengers = 5
        numberOfWheels = 4
    }
    override func description() -> String {
        return super.description() + "; "
                + "traveling at \(speed) mph"
    }
}
```

`Car` declares a new stored `Double` property called `speed`. This property defaults to `0.0`, meaning "zero miles per hour". `Car` also has a custom initializer, which sets the maximum number of passengers to `5`, and the default number of wheels to `4`.

`Car` overrides its inherited `description` method by providing a method with the same declaration as the `description` method from `Vehicle`. The overriding method definition is prefixed with the `override` keyword.

Rather than providing a completely custom implementation of `description`, the overriding method actually starts by calling `super.description` to retrieve the description provided by `Vehicle`. It then appends some additional information about the car's current speed.

If you create a new instance of `Car`, and print the output of its `description` method, you can see that the description has indeed changed:

```
let car = Car()
println("Car: \(car.description())")
// Car: 4 wheels; up to 5 passengers; traveling at 0.0 mph
```

Overriding Properties

You can override an inherited instance or class 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 its 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 `super.someProperty` from the getter, as in the `SpeedLimitedCar` example below.

The following example defines a new class called `SpeedLimitedCar`, which is a subclass of `Car`. The `SpeedLimitedCar` class represents a car that has been fitted with a speed-limiting device, which prevents the car from traveling faster than 40mph. You implement this limitation by overriding the inherited `speed` property:

```
class SpeedLimitedCar: Car {
    override var speed: Double  {
    get {
        return super.speed
    }
    set {
```

```
        super.speed = min(newValue, 40.0)
    }
    }
}
```

Whenever you set the `speed` property of a `SpeedLimitedCar` instance, the property's setter implementation checks the new value and limits it to 40mph. It does this by setting the underlying `speed` property of its superclass to be the smaller of `newValue` and `40.0`. The smaller of these two values is determined by passing them to the `min` function, which is a global function provided by the Swift standard library. The `min` function takes two or more values and returns the smallest one of those values.

If you try to set the `speed` property of a `SpeedLimitedCar` instance to more than 40mph, and then print the output of its `description` method, you see that the speed has been limited:

```
let limitedCar = SpeedLimitedCar()
limitedCar.speed = 60.0
println("SpeedLimitedCar: \(limitedCar.description())")
// SpeedLimitedCar: 4 wheels; up to 5 passengers; traveling at 40.0
                mph
```

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 the inherited property changes, regardless of how that property was originally implemented. For more information on property observers, see Property Observers.

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 `willSet` or `didSet` implementation as part of an override.

Note also that you cannot provide both an overriding setter and an overriding property observer. 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. `AutomaticCar` also provides a custom `description` method to print the current gear.

```
class AutomaticCar: Car {
    var gear = 1
    override var speed: Double {
    didSet {
        gear = Int(speed / 10.0) + 1
    }
    }
    override func description() -> String {
        return super.description() + " in gear \(gear)"
    }
}
```

Whenever you set the `speed` property of an `AutomaticCar` instance, the property's `didSet` observer automatically sets the `gear` property to an appropriate choice of gear for the new speed. Specifically, the property observer chooses a gear which is the new `speed` value divided by `10`, rounded down to the nearest integer, plus `1`. A speed of `10.0` produces a gear of `1`, and a speed of `35.0` produces a gear of `4`:

```
let automatic = AutomaticCar()
automatic.speed = 35.0
println("AutomaticCar: \(automatic.description())")
// AutomaticCar: 4 wheels; up to 5 passengers; traveling at 35.0 mph
                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` attribute before its introducer keyword (such as `@final var`, `@final func`, `@final class func`, and `@final subscript`).

Any attempts to override a final method, property, or subscript in a subclass are 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` attribute before the `class` keyword in its class definition (`@final class`). Any attempts to subclass a final class will be 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 to 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 Deinitialization.

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.

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 {
    var temperature: Double
    init() {
        temperature = 32.0
    }
}
var f = Fahrenheit()
println("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 when expressed in the Fahrenheit scale).

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 modifying 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 `Celsius`, which stores temperatures expressed in the Celsius scale. The `Celsius` structure implements two custom initializers called `init(fromFahrenheit:)` and `init(fromKelvin:)`, which initialize a new instance of the structure with a value from a different temperature scale:

```
struct Celsius {
    var temperatureInCelsius: Double = 0.0
    init(fromFahrenheit fahrenheit: Double) {
        temperatureInCelsius = (fahrenheit - 32.0) / 1.8
```

```
    }
    init(fromKelvin kelvin: Double) {
        temperatureInCelsius = kelvin - 273.15
    }
}
let boilingPointOfWater = Celsius(fromFahrenheit: 212.0)
// boilingPointOfWater.temperatureInCelsius is 100.0
let freezingPointOfWater = Celsius(fromKelvin: 273.15)
// freezingPointOfWater.temperatureInCelsius is 0.0
```

The first initializer has a single initialization parameter with an external name of `fromFahrenheit` and a local name of `fahrenheit`. The second initializer has a single initialization parameter with an external name of `fromKelvin` and a local name of `kelvin`. Both initializers convert their single argument into a value in the Celsius scale and store this value in a property called `temperatureInCelsius`.

Local and External Parameter Names

As with function and method parameters, initialization parameters can have both a local name for use within the initializer's body and an external name 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 external name for *every* parameter in an initializer if you don't provide an external name yourself. This automatic external name is the same as the local name, as if you had written a hash symbol before every initialization parameter.

> NOTE
>
> If you do not want to provide an external name for a parameter in an initializer, provide an underscore (_) as an explicit external name for that parameter to override the default behavior described above.

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`:

```
struct Color {
    let red = 0.0, green = 0.0, blue = 0.0
    init(red: Double, green: Double, blue: Double) {
        self.red   = red
        self.green = green
        self.blue  = blue
    }
}
```

Whenever you create a new `Color` instance, you call its initializer using external names for each of the three color components:

```
let magenta = Color(red: 1.0, green: 0.0, blue: 1.0)
```

Note that it is not possible to call this initializer without using the external names. External names must always be used in an intializer if they are defined, and omitting them is a compile-time error:

```
let veryGreen = Color(0.0, 1.0, 0.0)
// this reports a compile-time error - external names are required
```

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`:

```
class SurveyQuestion {
    var text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        println(text)
    }
}
let cheeseQuestion = SurveyQuestion(text: "Do you like
        cheese?")
cheeseQuestion.ask()
// prints "Do you like cheese?"
cheeseQuestion.response = "Yes, I do like cheese."
```

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.

Modifying Constant Properties During Initialization

You can modify the value of a constant property at any point during initialization, as long as it is set to a definite value by the time initialization finishes.

> NOTE
>
> For class instances, a constant property can only be modified during initialization 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 {
    let text: String
    var response: String?
    init(text: String) {
        self.text = text
    }
    func ask() {
        println(text)
    }
}
let beetsQuestion = SurveyQuestion(text: "How about beets?")
beetsQuestion.ask()
// prints "How about beets?"
beetsQuestion.response = "I also like beets. (But not with
        cheese.)"
```

Default Initializers

Swift provides a *default initializer* for any structure or base 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

In addition to the default initializers mentioned above, structure types automatically receive a *memberwise initializer* if they provide default values for all of their stored properties and do not define any of their own custom initializers.

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`.

Because both stored properties have a default value, 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)
```

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 only call `self.init` 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 structure initializer, if it is a structure) for that type. This constraint prevents a situation in which you provide a more complex initializer that performs additional essential setup is circumvented by someone accidentally using one of the automatic initializers instead.

> 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
}
struct Point {
    var x = 0.0, y = 0.0
}
```

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:

```
struct Rect {
    var origin = Point()
    var size = Size()
    init() {}
    init(origin: Point, size: Size) {
        self.origin = origin
        self.size = size
    }
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size:
        size)
    }
}
```

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 `{}`, and does not perfom any initialization. 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:

```
let basicRect = Rect()
```

```
// 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:

```
let originRect = Rect(origin: Point(x: 2.0, y: 2.0),
    size: Size(width: 5.0, height: 5.0))
// originRect's origin is (2.0, 2.0) and its size is (5.0, 5.0)
```

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:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

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 Extensions.

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 Automatic Initializer Inheritance 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.

Initializer Chaining

To simplify the relationships between designated and convenience initializers, Swift applies the following three rules for delegation calls between initializers:

Rule 1

Designated initializers must call a designated initializer from their immediate superclass.

Rule 2

Convenience initializers must call another initializer available in the *same* class.

Rule 3

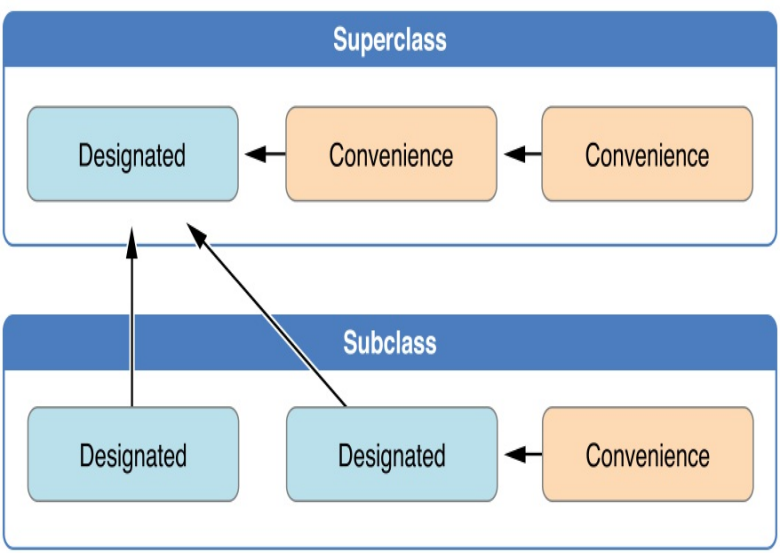
Convenience initializers must ultimately end up calling a designated initializer.

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:



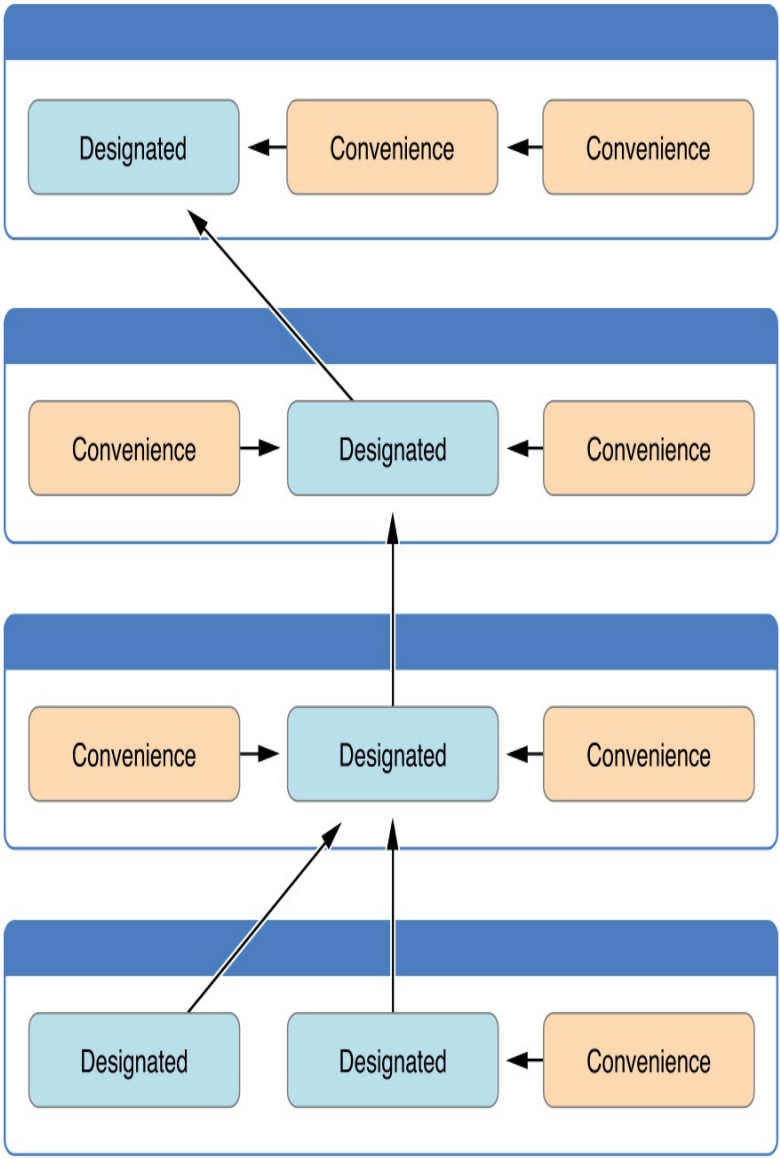
Here, the superclass has a single designated initializer and two convenience initializers. One convenience initializer calls another convenience initializer, which in turn calls the single designated initializer. This satisfies

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 class's implementation.

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 two-phase 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 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.

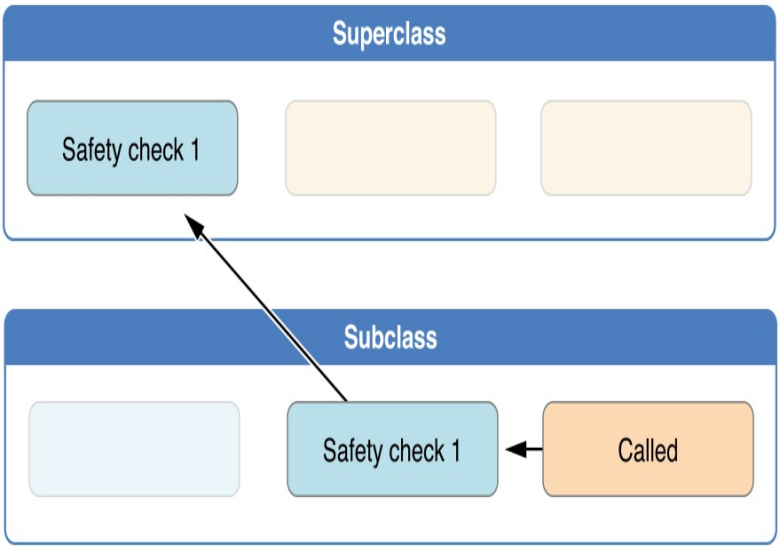
Once the top of the chain is reached, and the final class in the chain has ensured that all of its stored properties have a value, the instance's memory is considered to be fully initialized, and phase 1 is complete.

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:



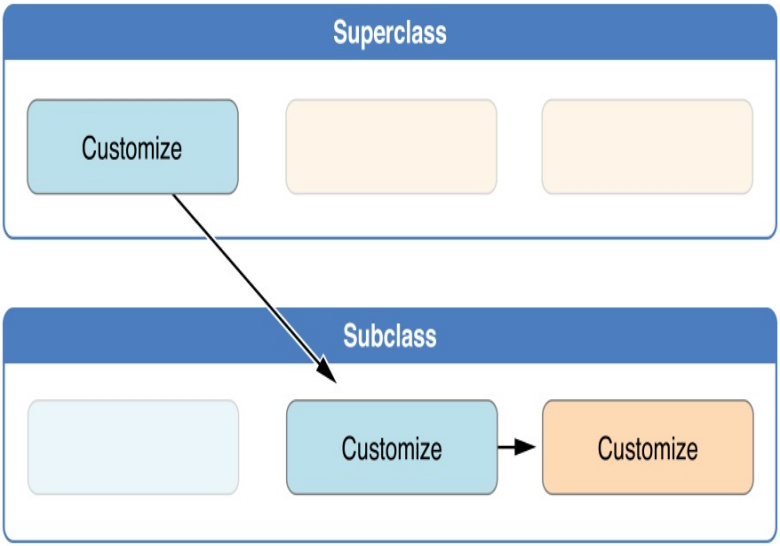
In this example, initialization begins with a call to a convenience initializer on the subclass. This convenience initializer cannot yet modify any properties. It delegates across to a designated initializer from the same class.

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 not inherit their superclass initializers by default. Swift's approach prevents a situation in which a simple initializer from a superclass is automatically inherited by a more specialized subclass and is used to create a new instance of the subclass that is not fully or correctly initialized.

If you want your custom subclass to present one or more of the same initializers as its superclass—perhaps to perform some customization during initialization—you can provide an overriding implementation of the same

initializer within your custom subclass.

If the initializer you are overriding is a *designated* initializer, you can override its implementation in your subclass and call the superclass version of the initializer from within your overriding version.

If the initializer you are overriding is a *convenience* initializer, your override must call another designated initializer from its own subclass, as per the rules described above in Initializer Chaining.

> NOTE
>
> Unlike methods, properties, and subscripts, you do not need to write the `override` keyword when overriding an initializer.

Automatic Initializer Inheritance

As mentioned above, subclasses do not 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.

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` keyword placed before the `init` keyword, separated by a space:

```
convenience init( parameters ) {
    statements
}
```

Designated and Convenience Initializers in Action

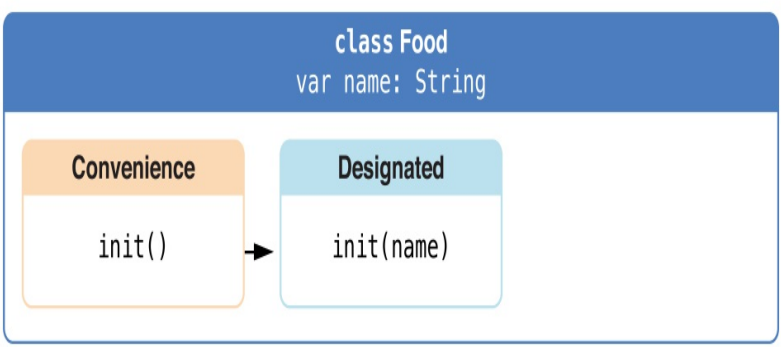
The following example shows designated initializers, convenience initializers, and automatic initializer

inheritance in action. This example defines a hierarchy of three classes called `Food`, `RecipeIngredient`, and `ShoppingListItem`, 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
    }
    convenience init() {
        self.init(name: "[Unnamed]")
    }
}
```

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:

```
let namedMeat = Food(name: "Bacon")
// 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.

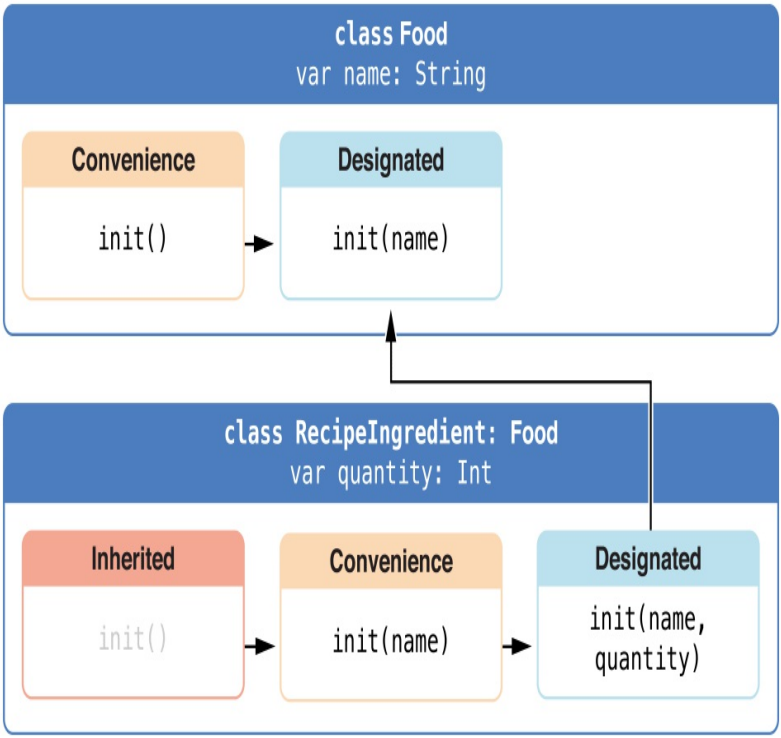
The `Food` class also provides a *convenience* initializer, `init()`, with no arguments. The `init()` initializer provides a default placeholder name for a new food by delegating across to the `Food` class's `init(name: String)` with a `name` value of `[Unnamed]`:

```
let mysteryMeat = Food()
// 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 {
    var quantity: Int
    init(name: String, quantity: Int) {
        self.quantity = quantity
        super.init(name: name)
    }
    convenience init(name: String) {
        self.init(name: name, quantity: 1)
    }
}
```

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.

Note that the `init(name: String)` convenience initializer provided by `RecipeIngredient` takes the same parameters as the `init(name: String)` *designated* initializer from `Food`. Even though `RecipeIngredient` provides this 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:

```
let oneMysteryItem = RecipeIngredient()
let oneBacon = RecipeIngredient(name: "Bacon")
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.

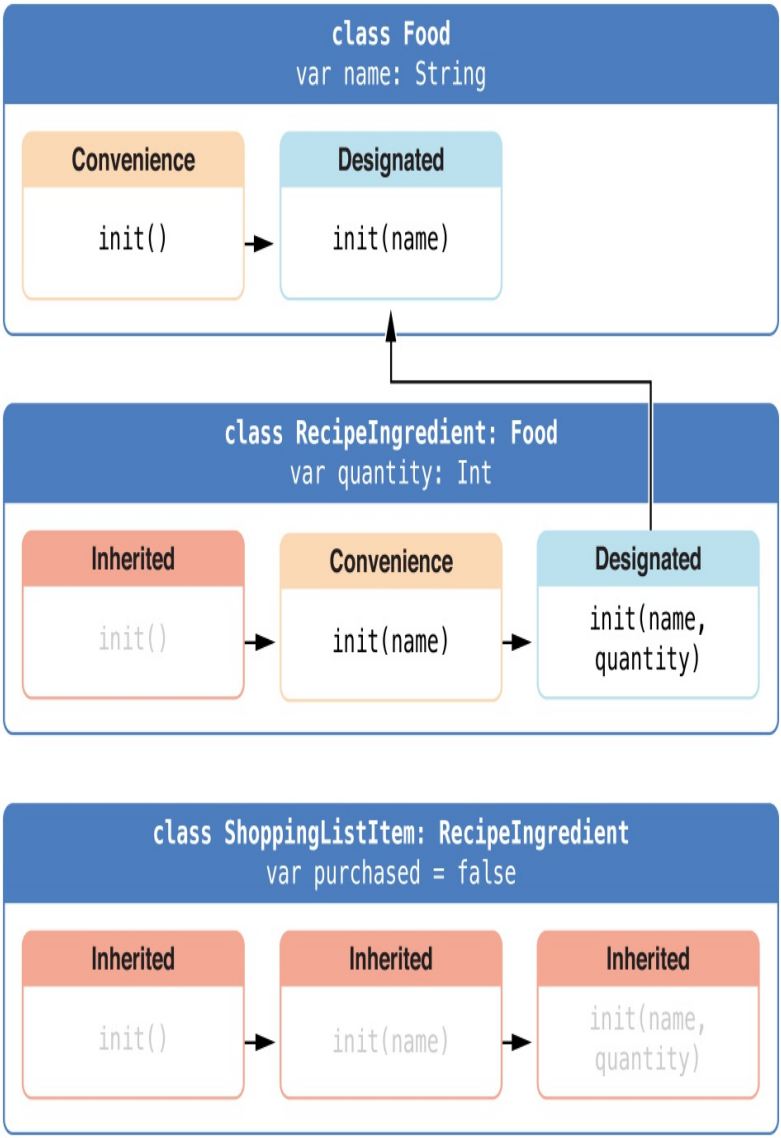
Every item in the shopping list starts out as "unpurchased". To represent this fact, `ShoppingListItem` introduces a Boolean property called `purchased`, with a default value of `false`. `ShoppingListItem` also adds a computed `description` property, which provides a textual description of a `ShoppingListItem` instance:

```
class ShoppingListItem: RecipeIngredient {
    var purchased = false
    var description: String {
    var output = "\(quantity) x \(name.lowercaseString)"
        output += purchased ? " ✔" : " ✘"
        return output
    }
}
```

> 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:

```
var breakfastList = [
```

```
    ShoppingListItem(),
    ShoppingListItem(name: "Bacon"),
    ShoppingListItem(name: "Eggs", quantity: 6),
]
breakfastList[0].name = "Orange juice"
breakfastList[0].purchased = true
for item in breakfastList {
    println(item.description)
}
// 1 x orange juice ✔
// 1 x bacon ✘
// 6 x eggs ✘
```

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.

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:

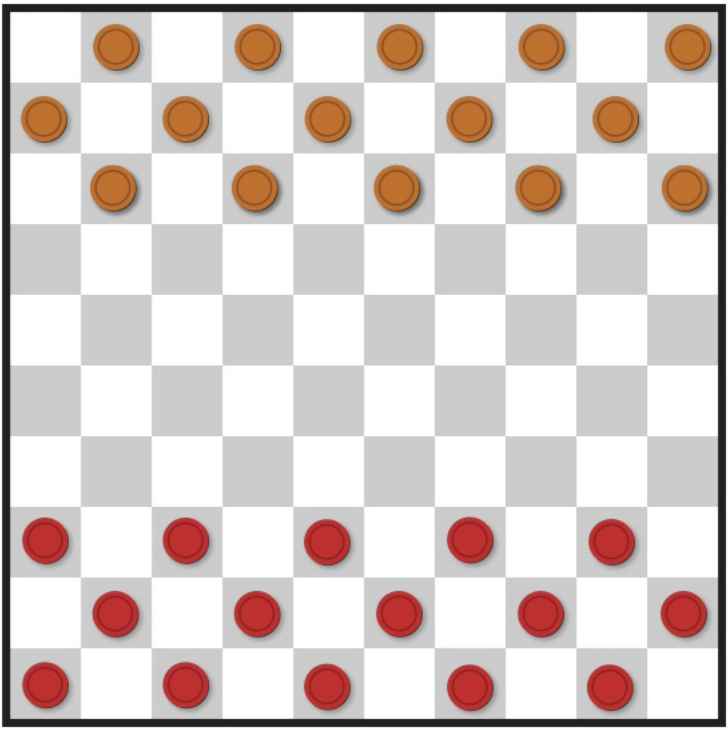
```
class SomeClass {
    let someProperty: SomeType = {
```

```
        // 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 `Checkerboard`, which models a board for the game of *Checkers* (also known as *Draughts*):



The game of *Checkers* is played on a ten-by-ten board, with alternating black and white squares. To represent this game board, the `Checkerboard` structure has a single property called `boardColors`, which is an array of 100 `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 Checkerboard {
    let boardColors: Bool[] = {
        var temporaryBoard = Bool[]()
        var isBlack = false
        for i in 1...10 {
            for j in 1...10 {
```

```
                temporaryBoard.append(isBlack)
                isBlack = !isBlack
            }
                isBlack = !isBlack
            }
            return temporaryBoard
            }()
        func squareIsBlackAtRow(row: Int, column: Int) -> Bool {
            return boardColors[(row * 10) + column]
        }
}
```

Whenever a new `Checkerboard` 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 `squareIsBlackAtRow` utility function:

```
let board = Checkerboard()
println(board.squareIsBlackAtRow(0, column: 1))
// prints "true"
println(board.squareIsBlackAtRow(9, column: 9))
// prints "false"
```

Deinitialization

A *deinitializer* is called immediately before a class instance is deallocated. You write deinitializers with the `deinit` keyword, similar to how intializers 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 Automatic Reference Counting. Typically you don't need to perform manual clean-up when your instances are deallocated. However, when you are working with your own resources, you might need to perform some additional clean-up 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` structure 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 structure with static properties and methods to store and manage its current state:

```
struct Bank {
    static var coinsInBank = 10_000
    static func vendCoins(var numberOfCoinsToVend: Int) -> Int {
        numberOfCoinsToVend = min(numberOfCoinsToVend, coinsInBank)
        coinsInBank -= numberOfCoinsToVend
        return numberOfCoinsToVend
    }
    static func receiveCoins(coins: Int) {
        coinsInBank += coins
    }
}
```

`Bank` keeps track of the current number of coins it holds with its `coinsInBank` property. It also offers two methods—`vendCoins` and `receiveCoins`—to handle the distribution and collection of coins.

`vendCoins` 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). `vendCoins` declares `numberOfCoinsToVend` as a variable parameter, so that the number can be modified within the method's body without the need to declare a new variable. It returns an integer value to indicate the actual number of coins that were provided.

The `receiveCoins` 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:

```
class Player {
    var coinsInPurse: Int
```

```
    init(coins: Int) {
        coinsInPurse = Bank.vendCoins(coins)
    }
    func winCoins(coins: Int) {
        coinsInPurse += Bank.vendCoins(coins)
    }
    deinit {
        Bank.receiveCoins(coinsInPurse)
    }
}
```

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 `winCoins` 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)
println("A new player has joined the game with \
             (playerOne!.coinsInPurse) coins")
// prints "A new player has joined the game with 100 coins"
println("There are now \(Bank.coinsInBank) coins left in the bank")
// prints "There are now 9900 coins left in the bank"
```

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 `playerOne`. 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 `winCoins` method is called:

```
playerOne!.winCoins(2_000)
println("PlayerOne won 2000 coins & now has \(playerOne!.coinsInPurse)
```

```
                coins")
// prints "PlayerOne won 2000 coins & now has 2100 coins"
println("The bank now only has \(Bank.coinsInBank) coins left")
// prints "The bank now only has 7900 coins left"
```

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
println("PlayerOne has left the game")
// prints "PlayerOne has left the game"
println("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 `playerOne` variable to `nil`, meaning "no `Player` instance." At the point that this happens, the `playerOne` 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.

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.

> NOTE
>
> Reference counting only applies 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 {
    let name: String
    init(name: String) {
        self.name = name
        println("\(name) is being initialized")
    }
    deinit {
        println("\(name) is being deinitialized")
    }
}
```

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:

```
reference1 = nil
reference2 = nil
```

ARC does not deallocate the `Person` instance until the third and final strong reference is broken, at which point it is clear that you are no longer using the `Person` instance:

```
reference3 = nil
// 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 is no longer needed.

However, it is 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 is 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
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { println("\(name) is being deinitialized") }
}

class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    var tenant: Person?
    deinit { println("Apartment #\(number) is being
        deinitialized") }
}
```

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 `number` property of type `Int` 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.

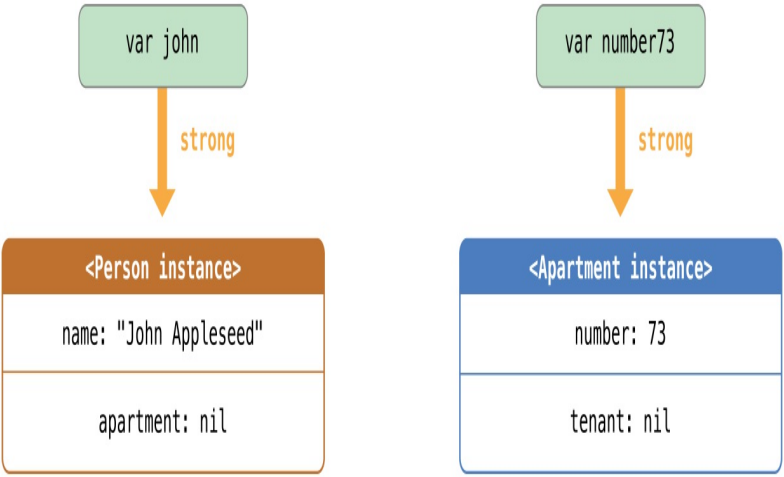
This next code snippet defines two variables of optional type called `john` and `number73`, which will be set to a specific `Apartment` and `Person` instance below. Both of these variables have an initial value of `nil`, by virtue of being optional:

```
var john: Person?
var number73: Apartment?
```

You can now create a specific `Person` instance and `Apartment` instance and assign these new instances to the `john` and `number73` variables:

```
john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)
```

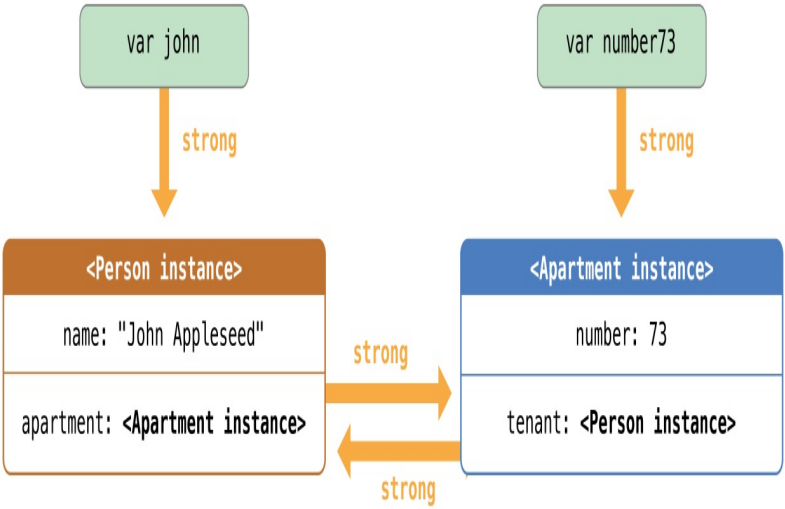
Here's how the strong references look after creating and assigning these two instances. The `john` variable now has a strong reference to the new `Person` instance, and the `number73` variable has a strong reference to the new `Apartment` instance:



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 `number73` optional variables, so that the properties of those instances can be set:

```
1  john!.apartment = number73
2  number73!.tenant = john
```

Here's how the strong references look after you link the two instances together:

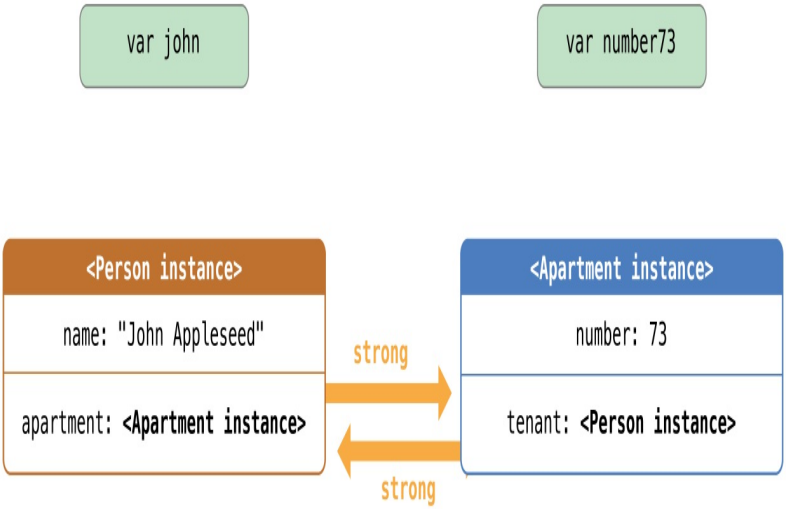


Unfortunately, linking these two instances creates a strong reference cycle between them. The `Person` instance now has a strong reference to the `Apartment` instance, and the `Apartment` instance has a strong reference to the `Person` instance. Therefore, when you break the strong references held by the `john` and `number73` variables, the reference counts do not drop to zero, and the instances are not deallocated by ARC:

```
1 john = nil
2 number73 = 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 `number73` 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 whenever it is valid for that reference to become `nil` at some point during its lifetime. Conversely, use an unowned reference when you know that the reference will never be `nil` once it has been set during initialization.

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.

Use a weak reference to avoid reference cycles whenever it is possible for that reference to have "no value" at some point in its life. If the reference will *always* have a value, use an unowned reference instead, as described in Unowned References. In the `Apartment` example above, it is 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.

> NOTE
>
> Weak references must be declared as variables, to indicate that their value can change at runtime. A weak reference cannot be declared as a constant.

Because weak references are allowed to have "no value", you must declare every weak reference as having an optional type. Optional types are the preferred way to represent the possibility for "no value" in Swift.

Because a weak reference does not keep a strong hold on the instance it refers to, it is 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. 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.

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:

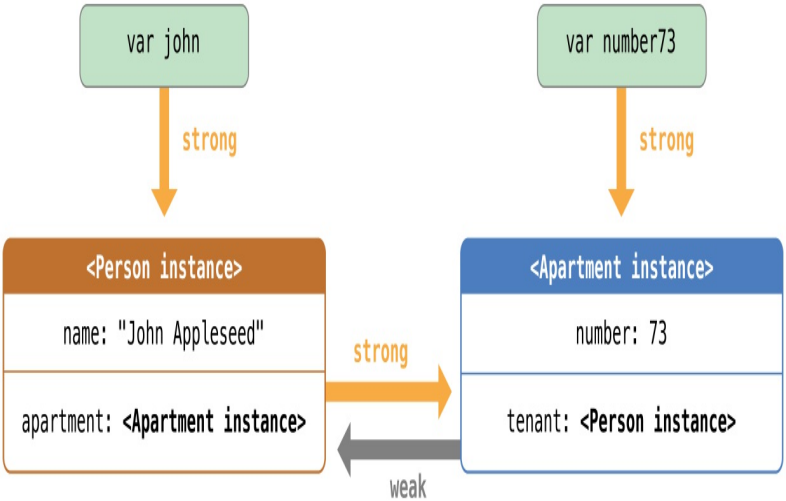
```
class Person {
    let name: String
    init(name: String) { self.name = name }
    var apartment: Apartment?
    deinit { println("\(name) is being deinitialized") }
```

```
}

class Apartment {
    let number: Int
    init(number: Int) { self.number = number }
    weak var tenant: Person?
    deinit { println("Apartment #\(number) is being
        deinitialized") }
}
```

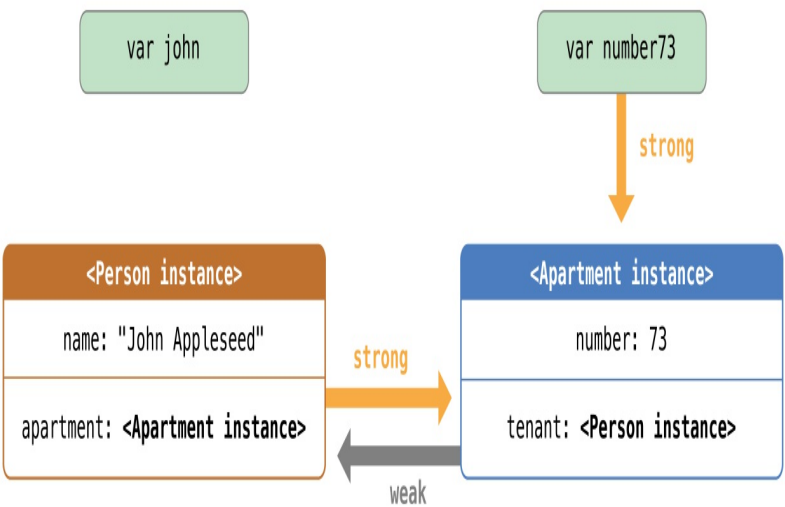
The strong references from the two variables (`john` and `number73`) and the links between the two instances are created as before:

```
var john: Person?
var number73: Apartment?

john = Person(name: "John Appleseed")
number73 = Apartment(number: 73)

john!.apartment = number73
number73!.tenant = john
```

Here's how the references look now that you've linked the two instances together:



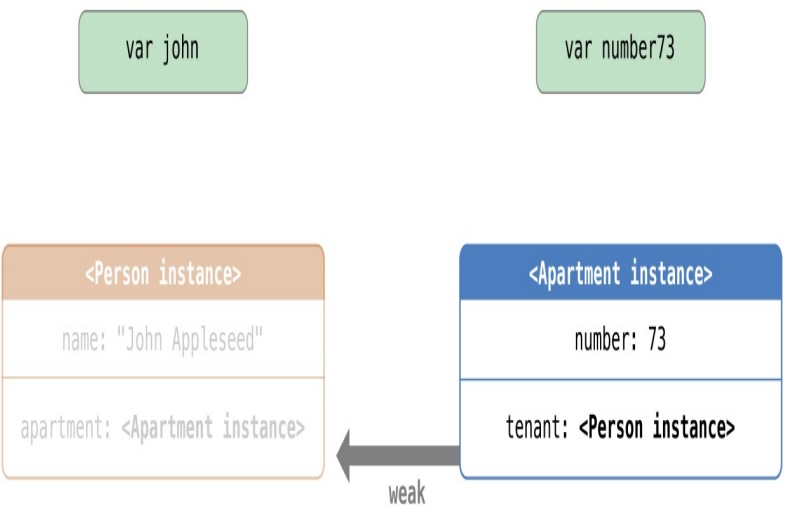
The `Person` instance still has a strong reference to the `Apartment` instance, but the `Apartment` instance now has a *weak* reference to the `Person` instance. This means that when you break the strong reference held by the `john` variables, there are no more strong references to the `Person` instance:



Because there are no more strong references to the `Person` instance, it is deallocated:

```
john = nil
// prints "John Appleseed is being deinitialized"
```

The only remaining strong reference to the `Apartment` instance is from the `number73` variable. If you break *that* strong reference, there are no more strong references to the `Apartment` instance:



Because there are no more strong references to the `Apartment` instance, it too is deallocated:

```
number73 = nil
// prints "Apartment #73 is being deinitialized"
```

The final two code snippets above show that the deinitializers for the `Person` instance and `Apartment` instance print their “deinitialized” messages after the `john` and `number73` variables are set to `nil`. This proves that the reference cycle has been broken.

Unowned References

Like weak references, an *unowned reference* does not keep a strong hold on the instance it refers to. Unlike a weak reference, however, an unowned reference is assumed to *always* have a value. Because of this, an unowned reference is always defined as a non-optional type. You indicate an unowned reference by placing the `unowned` keyword before a property or variable declaration.

Because an unowned reference is non-optional, you don't need to unwrap the unowned reference each time it is used. An unowned reference can always be accessed directly. However, ARC cannot set the reference to `nil` when the instance it refers to is deallocated, because variables of a non-optional type cannot be set to `nil`.

> NOTE
>
> If you try to access an unowned reference after the instance that it references is deallocated, you will trigger a runtime error. Use unowned references only when you are sure that the reference will *always* refer to an instance.
>
> Note also that Swift guarantees your app will crash if you try to access an unowned reference after the instance it references is deallocated. You will never encounter unexpected behavior in this situation. Your app will always crash reliably, although you should, of course, prevent it from doing so.

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. To represent this, the `Customer` class has an optional `card` property, but the `CreditCard` class has a 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:

```
class Customer {
    let name: String
    var card: CreditCard?
    init(name: String) {
        self.name = name
    }
    deinit { println("\(name) is being deinitialized") }
}

class CreditCard {
    let number: Int
    unowned let customer: Customer
    init(number: Int, customer: Customer) {
        self.number = number
        self.customer = customer
    }
    deinit { println("Card #\(number) is being deinitialized")
        }
}
```

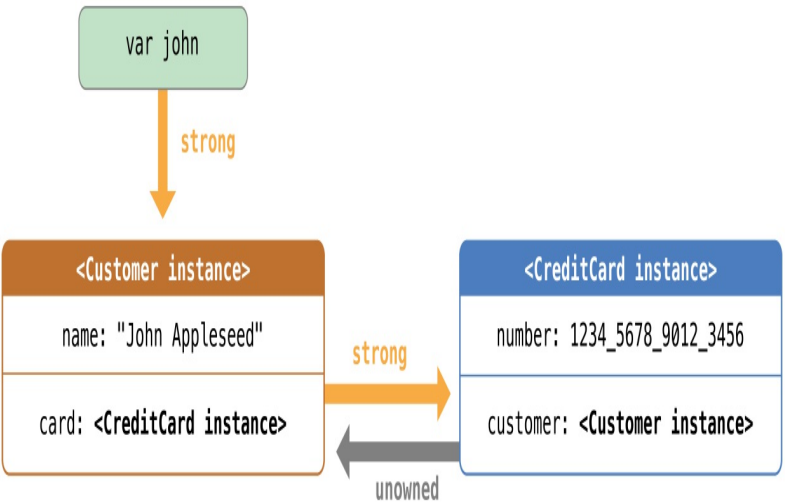
This next code snippet defines an optional `Customer` variable called `john`, which will be used to store a reference to a specific customer. This variable has an initial value of nil, by virtue of being optional:

```
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:

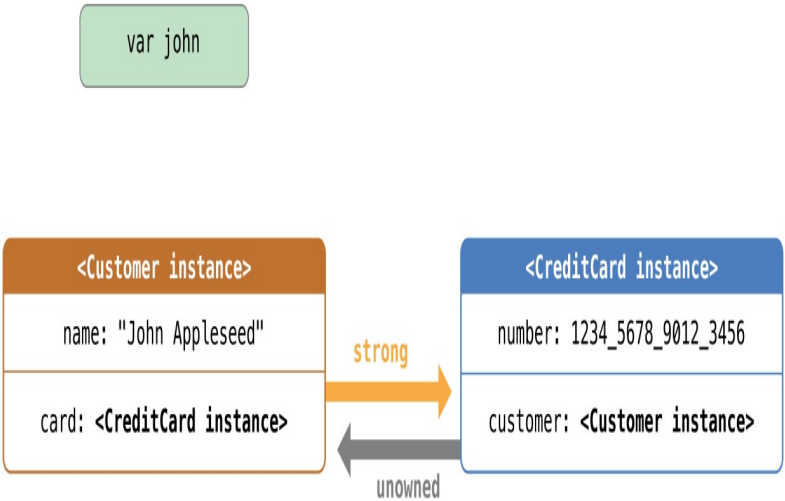
```
john = Customer(name: "John Appleseed")
john!.card = CreditCard(number: 1234_5678_9012_3456, customer: john!)
```

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:



Because there are no more strong references to the `Customer` instance, it is 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`.

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 is 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 is 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, `Country` and `City`, 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 `Country` class has a `capitalCity` property, and the `City` class has a `country` property:

```
class Country {
    let name: String
    let capitalCity: City!
    init(name: String, capitalName: String) {
        self.name = name
        self.capitalCity = City(name: capitalName, country: self)
    }
}

class City {
    let name: String
    unowned let country: Country
    init(name: String, country: Country) {
        self.name = name
        self.country = country
    }
}
```

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:

```
1 var country = Country(name: "Canada", capitalName: "Ottawa")
2 println("\(country.name)'s capital city is called \
              (country.capitalCity.name)")
3 // prints "Canada's capital city is called Ottawa"
```

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 is 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 {

    let name: String
    let text: String?

    @lazy var asHTML: () -> String = {
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
```

```
            self.text = text
        }

        deinit {
            println("\(name) is being deinitialized")
        }

    }
```

The `HTMLElement` class defines a `name` property, which indicates the name of the element, such as `"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 `"<p>some text</p>"` or `"<p />"`, 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.

> NOTE
>
> The `asHTML` property is declared as a lazy property, because it is 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.

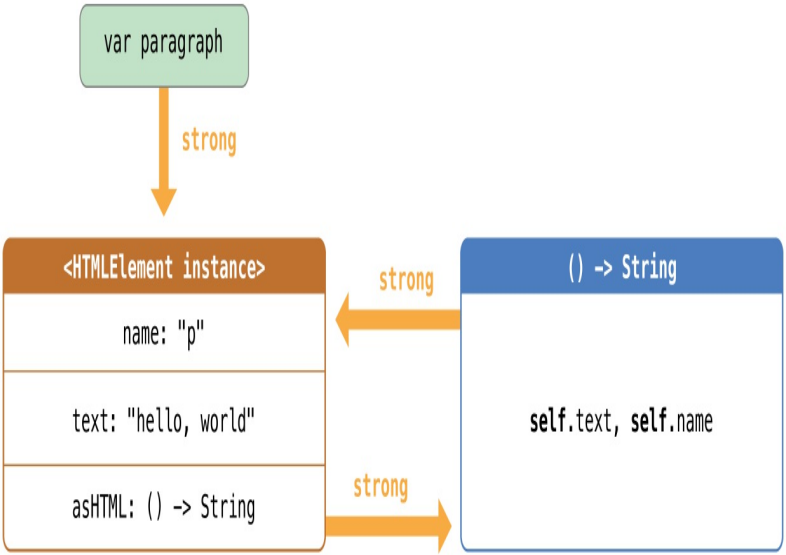
The `HTMLElement` class provides a single initializer, which takes a `name` argument and (if desired) a `text` argument to initialize a new element. The class also defines a deinitializer, which prints a message to show when an `HTMLElement` instance is deallocated.

Here's how you use the `HTMLElement` class to create and print a new instance:

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello,
              world")
println(paragraph!.asHTML())
// prints "<p>hello, world</p>"
```

> 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 Capturing Values.)

> 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:

```
1  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 `someInstance`). 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:

```
@lazy var someClosure: (Int, String) -> String = {
    [unowned self] (index: Int, stringToProcess: String) -> String in
    // closure body goes here
}
```

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:

```
@lazy var someClosure: () -> String = {
    [unowned self] in
    // closure body goes here
}
```

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 earlier. Here's how you write the `HTMLElement` class to avoid the cycle:

```
class HTMLElement {

    let name: String
    let text: String?

```

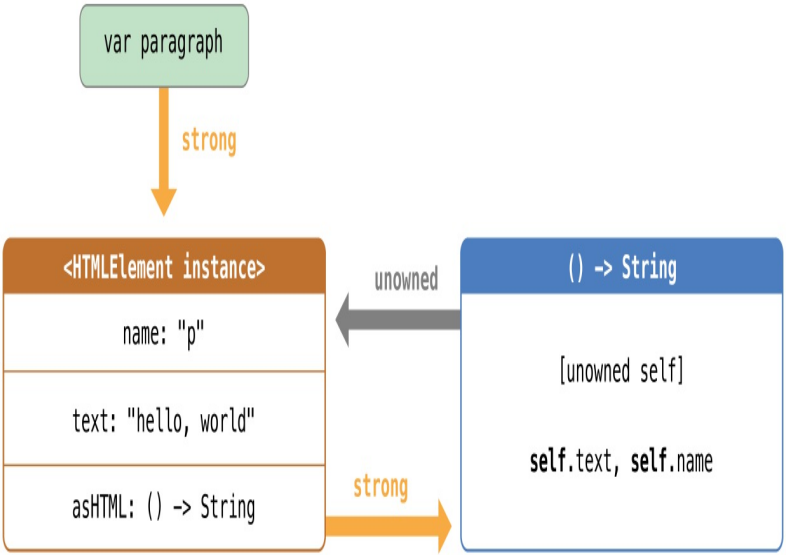
```
    @lazy var asHTML: () -> String = {
        [unowned self] in
        if let text = self.text {
            return "<\(self.name)>\(text)</\(self.name)>"
        } else {
            return "<\(self.name) />"
        }
    }

    init(name: String, text: String? = nil) {
        self.name = name
        self.text = text
    }

    deinit {
        println("\(name) is being deinitialized")
    }

}
```

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:

```
var paragraph: HTMLElement? = HTMLElement(name: "p", text: "hello,
            world")
println(paragraph!.asHTML())
// prints "<p>hello, world</p>"
```

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"
```

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:

```
let roomCount = john.residence!.numberOfRooms
// 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 {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("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`.

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 {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("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 {
    return rooms.count
    }
    subscript(i: Int) -> Room {
        return rooms[i]
    }
    func printNumberOfRooms() {
          println("The number of rooms is \(numberOfRooms)")
      }
      var address: Address?
  }
```

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-only subscript, which starts by asserting that the index passed to the subscript is valid. If the index is valid, the subscript returns 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 {
    var buildingName: String?
    var buildingNumber: String?
    var street: String?
    func buildingIdentifier() -> String? {
        if buildingName {
            return buildingName
        } else if buildingNumber {
```

```
            return buildingNumber
        } else {
            return nil
        }
    }
}
```

The `Address` class also provides a method called `buildingIdentifier`, which has a return type of `String?`. This method checks the `buildingName` and `buildingNumber` properties and returns `buildingName` if it has a value, or `buildingNumber` if it has a value, or `nil` if neither property has a value.

Calling Properties Through Optional Chaining

As demonstrated in Optional Chaining as an Alternative to Forced Unwrapping, you can use optional chaining to access a property on an optional value, and to check if that property access is successful. You cannot, however, set a property's value through optional chaining.

Use the classes defined above to create a new `Person` instance, and try to access its `numberOfRooms` property as before:

```
let john = Person()
if let roomCount = john.residence?.numberOfRooms {
    println("John's residence has \(roomCount) room(s).")
} else {
    println("Unable to retrieve the number of rooms.")
}
// prints "Unable to retrieve the number of rooms."
```

Because `john.residence` is `nil`, this optional chaining call fails in the same way as before, without error.

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() {
    println("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 Functions Without Return Values.

If you call this method on an optional value with optional chaining, the method's return type will be `Void?`, not `Void`, because return values are always of an optional type when called through optional chaining. This enables you to use an `if` statement to check whether it was possible to call the `printNumberOfRooms` method, even though the method does not itself define a return value. The implicit return value from the `printNumberOfRooms` will be equal to `Void` if the method was called succesfully through optional chaining, or `nil` if was not:

```
if john.residence?.printNumberOfRooms() {
    println("It was possible to print the number of rooms.")
} else {
    println("It was not possible to print the number of rooms.")
}
// prints "It was not possible to print the number of rooms."
```

Calling Subscripts Through Optional Chaining

You can use optional chaining to try to retrieve a value from a subscript on an optional value, and to check whether that subscript call is successful. You cannot, however, set a subscript through optional chaining.

> NOTE
>
> When you access a subscript on an optional value through optional chaining, you place the question mark *before* the subscript's braces, 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 {
    println("The first room name is \(firstRoomName).")
} else {
    println("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.

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()
johnsHouse.rooms += Room(name: "Living Room")
johnsHouse.rooms += Room(name: "Kitchen")
john.residence = johnsHouse

if let firstRoomName = john.residence?[0].name {
    println("The first room name is \(firstRoomName).")
} else {
```

```
    println("Unable to retrieve the first room name.")
}
// prints "The first room name is Living Room."
```

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 {
    println("John's street name is \(johnsStreet).")
} else {
    println("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.street.address`, and set an an actual value for the address's `street` property, you can access the value of property through the multi-level optional chaining:

```
let johnsAddress = Address()
johnsAddress.buildingName = "The Larches"
johnsAddress.street = "Laurel Street"
john.residence!.address = johnsAddress

if let johnsStreet = john.residence?.address?.street {
    println("John's street name is \(johnsStreet).")
} else {
    println("Unable to retrieve the address.")
}
// prints "John's street name is Laurel Street."
```

Note the use of an exclamation mark during the assignment of an address instance to `john.residence.address`. The `john.residence` property has an optional type, and so you need to unwrap its actual value with an exclamation mark before accessing the residence's `address` property.

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 let buildingIdentifier =
               john.residence?.address?.buildingIdentifier() {
    println("John's building identifier is \(buildingIdentifier).")
}
// prints "John's building identifier is The Larches."
```

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 upper =
               john.residence?.address?.buildingIdentifier()?.uppercase
               {
    println("John's uppercase building identifier is \(upper).")
}
// prints "John's uppercase building identifier is THE LARCHES."
```

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.

Type Casting

Type casting is a way to check the type of an instance, and/or to treat that instance as if it is 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
    }
}
```

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 {
    var director: String
    init(name: String, director: String) {
        self.director = director
        super.init(name: name)
    }
}

class Song: MediaItem {
    var artist: String
    init(name: String, artist: String) {
        self.artist = artist
        super.init(name: name)
    }
}
```

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:

```
let library = [
    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
    } else if item is Song {
        ++songCount
    }
}

println("Media library contains \(movieCount) movies and \
        (songCount) songs")
// 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 the *type cast operator* (`as`).

Because downcasting can fail, the type cast operator comes in two different forms. The optional 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 optional 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 optional form of the type cast operator (`as?`) to check the downcast each time through the loop:

```
for item in library {
    if let movie = item as? Movie {
        println("Movie: '\(movie.name)', dir. \(movie.director)")
    } else if let song = item as? Song {
        println("Song: '\(song.name)', by \(song.artist)")
    }
}

```

```
// Movie: 'Casablanca', dir. Michael Curtiz
// Song: 'Blue Suede Shoes', by Elvis Presley
// Movie: 'Citizen Kane', dir. Orson Welles
// Song: 'The One And Only', by Chesney Hawkes
// Song: 'Never Gonna Give You Up', by Rick Astley
```

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 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 two `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 type aliases for working with non-specific types:

`AnyObject` can represent an instance of any class type.

`Any` can represent an instance of any type at all, apart from function types.

> NOTE
>
> 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.

AnyObject

When working with Cocoa APIs, it is common to receive an array with a type of `AnyObject[]`, or "an array of values of any object type". This is because Objective-C does not have explicitly typed arrays. However, you can often be confident about the type of objects contained in such an array just from the information you know about the API that provided the array.

In these situations, you can use the forced version of the type cast operator (`as`) to downcast each item in the array to a more specific class type than `AnyObject`, without the need for optional unwrapping.

The example below defines an array of type `AnyObject[]` and populates this array with three instances of the `Movie` class:

```
let someObjects: AnyObject[] = [
    Movie(name: "2001: A Space Odyssey", director: "Stanley Kubrick"),
    Movie(name: "Moon", director: "Duncan Jones"),
    Movie(name: "Alien", director: "Ridley Scott")
]
```

Because this array is known to contain only `Movie` instances, you can downcast and unwrap directly to a non-optional `Movie` with the forced version of the type cast operator (`as`):

```
for object in someObjects {
    let movie = object as Movie
    println("Movie: '\(movie.name)', dir. \(movie.director)")
}
// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick
// Movie: 'Moon', dir. Duncan Jones
// Movie: 'Alien', dir. Ridley Scott
```

For an even shorter form of this loop, downcast the `someObjects` array to a type of `Movie[]` instead of downcasting each item:

```
for movie in someObjects as Movie[] {
    println("Movie: '\(movie.name)', dir. \(movie.director)")
}
// Movie: '2001: A Space Odyssey', dir. Stanley Kubrick
// Movie: 'Moon', dir. Duncan Jones
// Movie: 'Alien', dir. Ridley Scott
```

Any

Here's an example of using `Any` to work with a mix of different types, including non-class types. The example creates an array called `things`, which can store values of type `Any`:

```
var things = Any[]()

things.append(0)
things.append(0.0)
things.append(42)
things.append(3.14159)
things.append("hello")
```

```
things.append((3.0, 5.0))
things.append(Movie(name: "Ghostbusters", director: "Ivan Reitman"))
```

The `things` array contains two `Int` values, two `Double` values, a `String` value, a tuple of type `(Double, Double)`, and the movie "Ghostbusters", directed by Ivan Reitman.

You can use the `is` and `as` operators in a `switch` statement's cases to discover the specific type of a constant or variable that is known only to be of type `Any` or `AnyObject`. 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:

```
for thing in things {
    switch thing {
    case 0 as Int:
        println("zero as an Int")
    case 0 as Double:
        println("zero as a Double")
    case let someInt as Int:
        println("an integer value of \(someInt)")
    case let someDouble as Double where someDouble > 0:
        println("a positive double value of \(someDouble)")
    case is Double:
        println("some other double value that I don't want to
        print")
    case let someString as String:
        println("a string value of \"\(someString)\"")
    case let (x, y) as (Double, Double):
        println("an (x, y) point at \(x), \(y)")
    case let movie as Movie:
        println("a movie called '\(movie.name)', dir. \
        (movie.director)")
    default:
        println("something else")
    }
}
```

```
23
24 // zero as an Int
25 // zero as a Double
26 // an integer value of 42
27 // a positive double value of 3.14159
28 // a string value of "hello"
29 // an (x, y) point at 3.0, 5.0
30 // a movie called 'Ghostbusters', dir. Ivan Reitman
```

NOTE

The cases of a `switch` statement use the forced version of the type cast operator (`as`, not `as?`) to check and cast to a specific type. This check is always safe within the context of a `switch` case statement.

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 `BlackJack` 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 `Values`, which is nested within the `Rank` enumeration:

```
struct BlackjackCard {

    // nested Suit enumeration
    enum Suit: Character {
        case Spades = "♠", Hearts = "♡", Diamonds = "♢", Clubs = "♣"
    }

    // nested Rank enumeration
    enum Rank: Int {
        case Two = 2, Three, Four, Five, Six, Seven, Eight,
             Nine, Ten
        case Jack, Queen, King, Ace
        struct Values {
            let first: Int, second: Int?
```

```
        }
        var values: Values {
        switch self {
        case .Ace:
            return Values(first: 1, second: 11)
        case .Jack, .Queen, .King:
            return Values(first: 10, second: nil)
        default:
            return Values(first: self.toRaw(), second: nil)
            }
        }
    }

    // BlackjackCard properties and methods
    let rank: Rank, suit: Suit
    var description: String {
    var output = "suit is \(suit.toRaw()),"
        output += " value is \(rank.values.first)"
        if let second = rank.values.second {
            output += " or \(second)"
        }
        return output
    }
}
```

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)
println("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 members by their member 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:

```
let heartsSymbol = BlackjackCard.Suit.Hearts.toRaw()
// 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, or enumeration 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 properties and computed static 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

> 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.

Extension Syntax

Declare extensions with the `extension` keyword:

```
extension SomeType {
    // new functionality to add to SomeType goes here
}
```

An extension can extend an existing type to make it adopt one or more protocols. Where this is the case, the protocol names are written in exactly the same way as for a class or structure:

```
extension SomeType: SomeProtocol, AnotherProtocol {
    // implementation of protocol requirements goes here
}
```

Adding protocol conformance in this way is described in Adding Protocol Conformance with an Extension.

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 `Double` type, to provide basic support for working with distance units:

```
extension Double {
    var km: Double { return self * 1_000.0 }
    var m: Double { return self }
    var cm: Double { return self / 100.0 }
    var mm: Double { return self / 1_000.0 }
    var ft: Double { return self / 3.28084 }
}
let oneInch = 25.4.mm
println("One inch is \(oneInch) meters")
   // prints "One inch is 0.0254 meters"
   let threeFeet = 3.ft
   println("Three feet is \(threeFeet) meters")
   // prints "Three feet is 0.914399970739201 meters"
```

These computed properties express that a `Double` 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 3.28024 feet in a meter, and so the `ft` computed property divides the underlying `Double` value by `3.28024`, 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:

```
let aMarathon = 42.km + 195.m
println("A marathon is \(aMarathon) meters long")
// 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.

> NOTE
>
> 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 would not 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.

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 {
    var x = 0.0, y = 0.0
}
struct Rect {
    var origin = Point()
    var size = Size()
}
```

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 Default Initializers. These initializers can be used to create new `Rect` instances:

```
let defaultRect = Rect()
```

```
let memberwiseRect = Rect(origin: Point(x: 2.0, y: 2.0),
    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:

```
extension Rect {
    init(center: Point, size: Size) {
        let originX = center.x - (size.width / 2)
        let originY = center.y - (size.height / 2)
        self.init(origin: Point(x: originX, y: originY), size: size)
    }
}
```

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:

```
let centerRect = Rect(center: Point(x: 4.0, y: 4.0),
    size: Size(width: 3.0, height: 3.0))
// centerRect's origin is (2.5, 2.5) and its size is (3.0, 3.0)
```

> 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:

```
extension Int {
    func repetitions(task: () -> ()) {
        for i in 0..self {
            task()
        }
    }
}
```

The `repetitions` method takes a single argument of type `() -> ()`, which indicates a function that has no parameters and does not return a value.

After defining this extension, you can call the `repetitions` method on any integer number to perform a task that many number of times:

```
3.repetitions({
    println("Hello!")
    })
// Hello!
// Hello!
// Hello!
```

Use trailing closure syntax to make the call more succinct:

```
3.repetitions {
    println("Goodbye!")
}
// Goodbye!
// Goodbye!
// Goodbye!
```

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:

```
extension Int {
    mutating func square() {
        self = self * self
    }
}
var someInt = 3
someInt.square()
// 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:

```
extension Int {
    subscript(digitIndex: Int) -> Int {
        var decimalBase = 1
            for _ in 1...digitIndex {
                decimalBase *= 10
```

```
            }
            return (self / decimalBase) % 10
    }
}
    746381295[0]
    // returns 5
    746381295[1]
    // returns 9
    746381295[2]
    // returns 2
    746381295[8]
    // 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 zeroes to the left:

```
746381295[9]
// returns 0, as if you had requested:
0746381295[9]
```

Nested Types

Extensions can add new nested types to existing classes, structures and enumerations:

```
extension Character {
    enum Kind {
        case Vowel, Consonant, Other
    }
    var kind: Kind {
    switch String(self).lowercaseString {
    case "a", "e", "i", "o", "u":
        return .Vowel
    case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
```

```
        "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
            return .Consonant
        default:
            return .Other
            }
        }
    }
```

This example adds a new nested enumeration to `Character`. This enumeration, called `Kind`, expresses the kind of letter that a particular character represents. Specifically, it expresses whether the character is a vowel or a consonant in a standard Latin script (without taking into account accents or regional variations), or whether it is another kind of character.

This example also adds a new computed instance property to `Character`, called `kind`, which returns the appropriate `Kind` enumeration member for that character.

The nested enumeration can now be used with `Character` values:

```
func printLetterKinds(word: String) {
    println("'\(word)' is made up of the following kinds of letters:")
    for character in word {
        switch character.kind {
        case .Vowel:
            print("vowel ")
        case .Consonant:
            print("consonant ")
        case .Other:
                print("other ")
            }
        }
        print("\n")
    }
    printLetterKinds("Hello")
    // 'Hello' is made up of the following kinds of letters:
    // consonant vowel consonant consonant vowel
```

This function, `printLetterKinds`, takes an input `String` value and iterates over its characters. For each character, it considers the `kind` computed property for that character, and prints an appropriate description of that kind. The `printLetterKinds` function can then be called to print the kinds of letters in an entire word, as shown here for the word `"Hello"`.

> NOTE
>
> `character.kind` is already known to be of type `Character.Kind`. Because of this, all of the `Character.Kind` member values can be written in shorthand form inside the `switch` statement, such as `.Vowel` rather than `Character.Kind.Vowel`.

Protocols

A *protocol* defines a blueprint of methods, properties, and other requirements that suit a particular task or piece of functionality. The protocol doesn't actually provide an implementation for any of these requirements—it only describes what an implementation will look like. 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.

Protocols can require that conforming types have specific instance properties, instance methods, type methods, operators, and subscripts.

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 cannot 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 is valid for it also to be 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 set }` 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 `class` keyword when you define them in a protocol. This is true even though type property requirements are prefixed with the `static` keyword when implemented by a structure or enumeration:

```
protocol AnotherProtocol {
    class 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 defines any kind of thing that has a fully-qualified name. It doesn't specify what *kind* of thing it must be—it only specifies that the thing must be able to provide a full name for itself. It specifies this requirement by stating 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:

```
class Starship: FullyNamed {
    var prefix: String?
    var name: String
    init(name: String, prefix: String? = nil) {
        self.name = name
        self.prefix = prefix
    }
    var fullName: String {
    return (prefix ? prefix! + " " : "") + name
      }
```

```
}
var ncc1701 = Starship(name: "Enterprise", prefix: "USS")
// ncc1701.fullName is "USS Enterprise"
```

This class implements the `fullName` property requirement as a computed read-only 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.

> NOTE
>
> Protocols use the same syntax as normal methods, but are not allowed to specify default values for method parameters.

As with type property requirements, you always prefix type method requirements with the `class` keyword when they are defined in a protocol. This is true even though type method requirements are prefixed with the `static` keyword when implemented by a structure or enumeration:

```
protocol SomeProtocol {
    class 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 is called. (Although it is not specified as part of the protocol, it is assumed that this value will be a number between `0.0` and `1.0` inclusive.)

The `RandomNumberGenerator` protocol does not 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 {
        lastRandom = ((lastRandom * a + c) % m)
        return lastRandom / m
    }
}
let generator = LinearCongruentialGenerator()
println("Here's a random number: \(generator.random())")
// prints "Here's a random number: 0.37464991998171"
println("And another one: \(generator.random())")
// prints "And another one: 0.729023776863283"
```

Mutating Method Requirements

It is 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/or any properties of that instance. This process is described in Modifying Value Types from Within Instance Methods.

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 do not 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 is 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 `OnOffSwitch`. This enumeration toggles between two states, indicated by the enumeration cases `On` and `Off`. The enumeration's `toggle` implementation is

marked as `mutating`, to match the `Togglable` protocol's requirements:

```
enum OnOffSwitch: Togglable {
    case Off, On
    mutating func toggle() {
        switch self {
        case Off:
            self = On
        case On:
            self = Off
        }
    }
}
var lightSwitch = OnOffSwitch.Off
lightSwitch.toggle()
// lightSwitch is now equal to .On
```

Protocols as Types

Protocols do not actually implement any functionality themselves. Nonetheless, any protocol you create will become a fully-fledged type for use in your code.

Because it is a type, 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 `Dice`, which represents an *n*-sided dice for use in a board game. `Dice` instances have an integer property called `sides`, which represents how many sides they have, and a property called `generator`, 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.

`Dice` also has an initializer, to set up its initial state. This initializer has a 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 is 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:

```
var d6 = Dice(sides: 6, generator: LinearCongruentialGenerator())
for _ in 1...5 {
    println("Random dice roll is \(d6.roll())")
}
// Random dice roll is 3
// Random dice roll is 5
// Random dice roll is 4
// Random dice roll is 5
// 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 {
    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 by any type to track the progress of a `DiceGame`.

Here's a version of the *Snakes and Ladders* game originally introduced in Control Flow. 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:

```
class SnakesAndLadders: DiceGame {
    let finalSquare = 25
    let dice = Dice(sides: 6, generator:
              LinearCongruentialGenerator())
    var square = 0
    var board: Int[]
    init() {
        board = Int[](count: finalSquare + 1, repeatedValue: 0)
        board[03] = +08; board[06] = +11; board[09] = +09; board[10] =
              +02
        board[14] = -10; board[19] = -11; board[22] = -02; board[24] =
              -08
    }
    var delegate: DiceGameDelegate?
    func play() {
        square = 0
        delegate?.gameDidStart(self)
        gameLoop: while square != finalSquare {
            let diceRoll = dice.roll()
            delegate?.game(self, didStartNewTurnWithDiceRoll:
            diceRoll)
            switch square + diceRoll {
            case finalSquare:
                break gameLoop
            case let newSquare where newSquare > finalSquare:
                continue gameLoop
```

```
            default:
                square += diceRoll
                square += board[square]
            }
        }
        delegate?.gameDidEnd(self)
    }
}
```

For a description of the *Snakes and Ladders* gameplay, see the Break section of the Control Flow chapter.

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 does not need to change after initialization, and the protocol only requires that it is gettable.)

The *Snakes and Ladders* game board setup takes place within the class's `init()` initializer. All game logic is moved into the protocol's `play` method, which uses the protocol's required `dice` 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 is 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.

`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:

```
class DiceGameTracker: DiceGameDelegate {
    var numberOfTurns = 0
    func gameDidStart(game: DiceGame) {
        numberOfTurns = 0
        if game is SnakesAndLadders {
            println("Started a new game of Snakes and Ladders")
        }
        println("The game is using a \(game.dice.sides)-sided dice")
    }
    func game(game: DiceGame, didStartNewTurnWithDiceRoll
        diceRoll: Int) {
        ++numberOfTurns
        println("Rolled a \(diceRoll)")
    }
    func gameDidEnd(game: DiceGame) {
        println("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 `gameDidStart` shown above uses the `game` parameter to print some introductory information about the game that is about to be played. The `game` parameter has a type of `DiceGame`, not `SnakesAndLadders`, and so `gameDidStart` can access and use only methods and properties that are implemented as part of the `DiceGame` 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 `game` is actually an instance of `SnakesAndLadders` behind the scenes, and prints an appropriate message if so.

`gameDidStart` also accesses the `dice` property of the passed `game` parameter. Because `game` is known to conform to the `DiceGame` protocol, it is 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:

```
let tracker = DiceGameTracker()
let game = SnakesAndLadders()
game.delegate = tracker
game.play()
// Started a new game of Snakes and Ladders
// The game is using a 6-sided dice
// Rolled a 3
// Rolled a 5
// Rolled a 4
   // Rolled a 5
   // 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 do not 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 `TextRepresentable`, 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 {
    func asText() -> String
}
```

The `Dice` class from earlier can be extended to adopt and conform to `TextRepresentable`:

```
extension Dice: TextRepresentable {
    func asText() -> 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`:

```
let d12 = Dice(sides: 12, generator: LinearCongruentialGenerator())
println(d12.asText())
// prints "A 12-sided dice"
```

Similarly, the `SnakesAndLadders` game class can be extended to adopt and conform to the `TextRepresentable` protocol:

```
extension SnakesAndLadders: TextRepresentable {
    func asText() -> String {
        return "A game of Snakes and Ladders with \(finalSquare)
              squares"
    }
}
println(game.asText())
// prints "A game of Snakes and Ladders with 25 squares"
```

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
    func asText() -> String {
        return "A hamster named \(name)"
    }
}
extension Hamster: TextRepresentable {}
```

Instances of `Hamster` can now be used wherever `TextRepresentable` is the required type:

```
let simonTheHamster = Hamster(name: "Simon")
let somethingTextRepresentable: TextRepresentable = simonTheHamster
println(somethingTextRepresentable.asText())
// prints "A hamster named Simon"
```

> NOTE
>
> Types do not 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 Protocols as Types. This example creates an array of `TextRepresentable` things:

```
let things: TextRepresentable[] = [game, d12, simonTheHamster]
```

It is now possible to iterate over the items in the array, and print each item's textual representation:

```
for thing in things {
    println(thing.asText())
}
// 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 is 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 is of type `TextRepresentable`, and anything that is `TextRepresentable` is known to have an `asText` method, it is safe to call `thing.asText` 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 {
    func asPrettyText() -> String
}
```

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 an instance method called `asPrettyText` that returns a `String`.

The `SnakesAndLadders` class can be extended to adopt and conform to `PrettyTextRepresentable`:

```
extension SnakesAndLadders: PrettyTextRepresentable {
    func asPrettyText() -> String {
        var output = asText() + ":\n"
        for index in 1...finalSquare {
            switch board[index] {
            case let ladder where ladder > 0:
                output += "▲ "
            case let snake where snake < 0:
                output += "▼ "
            default:
                output += "○ "
            }
        }
        return output
    }
}
```

This extension states that it adopts the `PrettyTextRepresentable` protocol and provides an implementation of the `asPrettyText` method for the `SnakesAndLadders` type. Anything that is `PrettyTextRepresentable` must also be `TextRepresentable`, and so the `asPrettyText` implementation starts by calling the `asText` method 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 an emoji representation for each square:

> If the square's value is greater than `0`, it is the base of a ladder, and is represented by ▲.
>
> If the square's value is less than `0`, it is the head of a snake, and is represented by ▼.
>
> Otherwise, the square's value is `0`, and it is a "free" square, represented by ○.

The method implementation can now be used to print a pretty text description of any `SnakesAndLadders`

instance:

```
println(game.asPrettyText())
// A game of Snakes and Ladders with 25 squares:
// ○ ○ ▲ ○ ○ ▲ ○ ○ ▲ ▲ ○ ○ ○ ▼ ○ ○ ○ ○ ▼ ○ ○ ▼ ○ ▼ ○
```

Protocol Composition

It can be useful to require a type to conform to multiple protocols at once. You can combine multiple protocols into a single requirement with a *protocol composition*. Protocol compositions have the form `protocol<SomeProtocol, AnotherProtocol>`. You can list as many protocols within the pair of angle brackets (`<>`) as you need, separated by commas.

Here's an example that combines two protocols called `Named` and `Aged` into a single protocol composition requirement on a function parameter:

```
protocol Named {
    var name: String { get }
}
protocol Aged {
    var age: Int { get }
}
struct Person: Named, Aged {
    var name: String
    var age: Int
}
func wishHappyBirthday(celebrator: protocol<Named, Aged>) {
    println("Happy birthday \(celebrator.name) - you're \
        (celebrator.age)!")
}
let birthdayPerson = Person(name: "Malcolm", age: 21)
wishHappyBirthday(birthdayPerson)
// prints "Happy birthday Malcolm - you're 21!"
```

This example defines a protocol called `Named`, with a single requirement for a gettable `String` property called `name`. It also defines a protocol called `Aged`, with a single requirement for a gettable `Int` property called `age`. Both of these protocols are adopted by a structure called `Person`.

The example also defines a function called `wishHappyBirthday`, which takes a single parameter called `celebrator`. The type of this parameter is `protocol<Named, Aged>`, which means "any type that conforms to both the `Named` and `Aged` protocols." It doesn't matter what 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` function. Because `Person` conforms to both protocols, this is a valid call, and the `wishHappyBirthday` function is able to print its birthday greeting.

> NOTE
>
> Protocol compositions do not define a new, permanent protocol type. Rather, they define a temporary local protocol that has the combined requirements of all protocols in the composition.

Checking for Protocol Conformance

You can use the `is` and `as` operators described in Type Casting 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 does not.

The `as?` version of the downcast operator returns an optional value of the protocol's type, and this value is `nil` if the instance does not 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 does not succeed.

This example defines a protocol called `HasArea`, with a single property requirement of a gettable `Double` property called `area`:

```
@objc protocol HasArea {
    var area: Double { get }
}
```

NOTE

You can check for protocol conformance only if your protocol is marked with the `@objc` attribute, as seen for the `HasArea` protocol above. This attribute indicates that the protocol should be exposed to Objective-C code and is described in *Using Swift with Cocoa and Objective-C*. Even if you are not interoperating with Objective-C, you need to mark your protocols with the `@objc` attribute if you want to be able to check for protocol conformance.

Note also that `@objc` protocols can be adopted only by classes, and not by structures or enumerations. If you mark your protocol as `@objc` in order to check for conformance, you will be able to apply that protocol only to class types.

Here are two classes, `Circle` and `Country`, both of which conform to the `HasArea` protocol:

```
class Circle: HasArea {
    let pi = 3.1415927
    var radius: Double
    var area: Double { return pi * radius * radius }
    init(radius: Double) { self.radius = radius }
}
class Country: HasArea {
    var area: Double
    init(area: Double) { self.area = area }
}
```

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 does not conform to the `HasArea` protocol:

```
class Animal {
    var legs: Int
    init(legs: Int) { self.legs = legs }
}
```

The `Circle`, `Country` and `Animal` classes do not have a shared base class. Nonetheless, they are all classes, and so instances of all three types can be used to initialize an array that stores values of type `AnyObject`:

```
let objects: AnyObject[] = [
    Circle(radius: 2.0),
    Country(area: 243_610),
    Animal(legs: 4)
]
```

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 {
    if let objectWithArea = object as? HasArea {
        println("Area is \(objectWithArea.area)")
    } else {
        println("Something that doesn't have an area")
    }
```

```
}
// Area is 12.5663708
// Area is 243610.0
   // Something that doesn't have an area
```

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 `objectWithArea`. The `objectWithArea` constant is known to be of type `HasArea`, and so its `area` property can be accessed and printed in a type-safe way.

Note that the underlying objects are not changed by the casting process. They continue to be a `Circle`, a `Country` and an `Animal`. However, at the point that they are stored in the `objectWithArea` constant, they are 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 do not have to be implemented by types that conform to the protocol. Optional requirements are prefixed by the `@optional` keyword as part of the protocol's definition.

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. For information on optional chaining, see Optional Chaining.

You check for an implementation of an optional requirement by writing a question mark after the name of the requirement when it is called, such as `someOptionalMethod?(someArgument)`. Optional property requirements, and optional method requirements that return a value, will always return an optional value of the appropriate type when they are accessed or called, to reflect the fact that the optional requirement may not have been implemented.

> NOTE
>
> Optional protocol requirements can only be specified if your protocol is marked with the `@objc`

attribute. Even if you are not interoperating with Objective-C, you need to mark your protocols with the `@objc` attribute if you want to specify optional requirements.

Note also that `@objc` protocols can be adopted only by classes, and not by structures or enumerations. If you mark your protocol as `@objc` in order to specify optional requirements, you will only be able to apply that protocol to class types.

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:

```
@objc protocol CounterDataSource {
    @optional func incrementForCount(count: Int) -> Int
    @optional var fixedIncrement: Int { get }
}
```

The `CounterDataSource` protocol defines an optional method requirement called `incrementForCount` 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 are 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?`:

```
@objc class Counter {
    var count = 0
```

```
    var dataSource: CounterDataSource?
    func increment() {
        if let amount = dataSource?.incrementForCount?(count) {
            count += amount
        } else if let amount = dataSource?.fixedIncrement? {
            count += amount
        }
    }
}
```

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 `incrementForCount` method on its data source. The `increment` method uses optional chaining to try to call `incrementForCount`, and passes the current `count` value as the method's single argument.

Note *two* levels of optional chaining at play here. Firstly, it is possible that `dataSource` may be `nil`, and so `dataSource` has a question mark after its name to indicate that `incrementForCount` should only be called if `dataSource` is non-nil. Secondly, even if `dataSource` *does* exist, there is no guarantee that it implements `incrementForCount`, because it is an optional requirement. This is why `incrementForCount` is also written with a question mark after its name.

Because the call to `incrementForCount` can fail for either of these two reasons, the call returns an *optional* `Int` value. This is true even though `incrementForCount` is defined as returning a non-optional `Int` value in the definition of `CounterDataSource`.

After calling `incrementForCount`, 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 is *not* possible to retrieve a value from the `incrementForCount` method—either because `dataSource` is nil, or because the data source does not implement `incrementForCount`—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, and so its name is also written using optional

chaining with a question mark on the end, to indicate that the attempt to access the property's value can fail. As before, the returned 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 is queried. It does this by implementing the optional `fixedIncrement` property requirement:

```
class ThreeSource: CounterDataSource {
    let fixedIncrement = 3
}
```

You can use an instance of `ThreeSource` as the data source for a new `Counter` instance:

```
var counter = Counter()
counter.dataSource = ThreeSource()
for _ in 1...4 {
    counter.increment()
    println(counter.count)
}
// 3
// 6
// 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: CounterDataSource {
    func incrementForCount(count: Int) -> Int {
        if count == 0 {
```

```
            return 0
        } else if count < 0 {
            return 1
        } else {
            return -1
        }
    }
}
```

The `TowardsZeroSource` class implements the optional `incrementForCount` 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
counter.dataSource = TowardsZeroSource()
for _ in 1...5 {
    counter.increment()
    println(counter.count)
}
// -3
// -2
// -1
// 0
// 0
```

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 this 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, non-generic function called `swapTwoInts`, which swaps two `Int` values:

```
func swapTwoInts(inout a: Int, inout b: 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)
println("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:

```
func swapTwoStrings(inout a: String, inout b: String) {
    let temporaryA = a
    a = b
    b = temporaryA
}

func swapTwoDoubles(inout a: Double, inout b: Double) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

You may have noticed that the bodies of the `swapTwoInts`, `swapTwoStrings`, and `swapTwoDoubles` functions are identical. The only difference is the type of the values that they accept (`Int`, `String`, and `Double`).

It would be much more useful, and considerably more flexible, to write a single function that could swap two values of *any* type. This is the kind of problem that generic code can solve. (A generic version of these functions is defined below.)

NOTE

In all three functions, it is important that the types of `a` and `b` are defined to be the same as each other. If `a` and `b` were not of the same type, it would not be possible to swap their values. Swift is a type-safe language, and does not allow (for example) a variable of type `String` and a variable of

type `Double` to swap values with each other. Attempting to do so would be reported as 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>(inout a: T, inout b: 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(inout a: Int, inout b: Int)
func swapTwoValues<T>(inout a: T, inout b: T)
```

The generic version of the function uses a *placeholder* type name (called `T`, 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 `T` must be, but it *does* say that both `a` and `b` must be of the same type `T`, whatever `T` represents. The actual type to use in place of `T` will be determined each time the `swapTwoValues` function is called.

The other difference is that the generic function's name (`swapTwoValues`) is followed by the placeholder type name (`T`) inside angle brackets (`<T>`). The brackets tell Swift that `T` is a placeholder type name within the `swapTwoValues` function definition. Because `T` is a placeholder, Swift does not look for an actual type called `T`.

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:

```
var someInt = 3
var anotherInt = 107
swapTwoValues(&someInt, &anotherInt)
// someInt is now 107, and anotherInt is now 3

var someString = "hello"
var anotherString = "world"
swapTwoValues(&someString, &anotherString)
// someString is now "world", and anotherString is now "hello"
```

> 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 specified, a type parameter can be used 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 placeholder type represented by 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 simple cases where a generic function or generic type needs to refer to a single placeholder type (such as the `swapTwoValues` generic function above, or a generic collection that stores a single type, such as `Array`), it is traditional to use the single-character name `T` for the type parameter. However, you are can use any valid identifier as the type parameter name.

If you are defining more complex generic functions, or generic types with multiple parameters, it can be useful to provide more descriptive type parameter names. For example, Swift's `Dictionary` type has two type parameters—one for its keys and one for its values. If you were writing `Dictionary` yourself, you might name these two type parameters `KeyType` and `ValueType` to remind you of their purpose as you use them within your generic code.

> NOTE
>
> Always give type parameters `UpperCamelCase` names (such as `T` and `KeyType`) to indicate that they are a placeholder for a *type*, not a value.

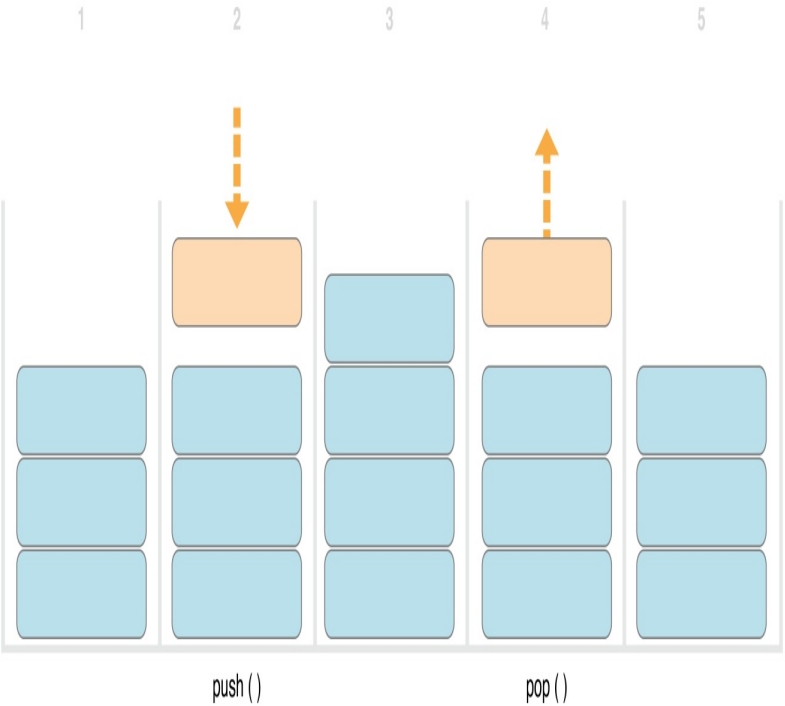
Generic Types

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 / pop behavior for a stack:



1. There are currently three values on the stack.
2. A fourth value is “pushed” on to 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 removed, or “popped”.
5. After popping a value, the stack once again holds three values.

Here’s how to write a non-generic version of a stack, in this case for a stack of `Int` values:

```
struct IntStack {
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
```

```
    mutating func pop() -> Int {
        return items.removeLast()
    }
}
```

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<T> {
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
}
```

Note how the generic version of `Stack` is essentially the same as the non-generic version, but with a placeholder type parameter called `T` instead of an actual type of `Int`. This type parameter is written within a pair of angle brackets (`<T>`) immediately after the structure's name.

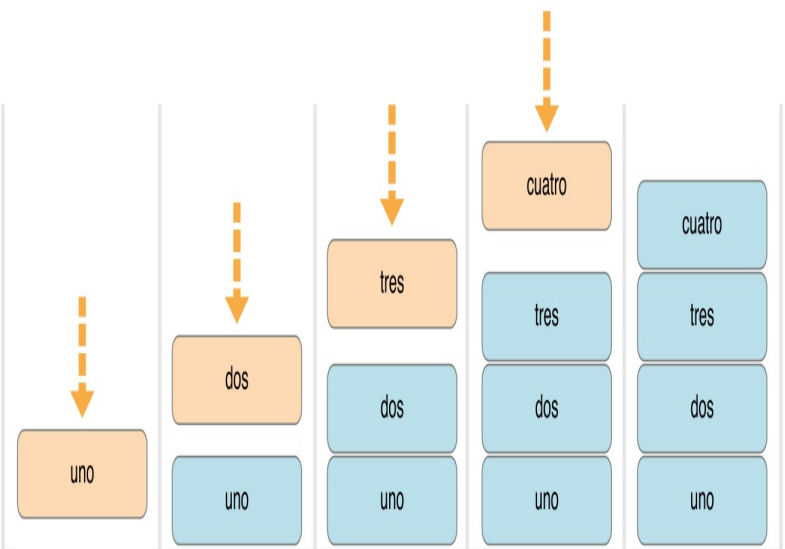
`T` defines a placeholder name for "some type `T`" to be provided later on. This future type can be referred to as "`T`" anywhere within the structure's definition. In this case, `T` is used as a placeholder in three places:

- To create a property called `items`, which is initialized with an empty array of values of type `T`
- To specify that the `push` method has a single parameter called `item`, which must be of type `T`
- To specify that the value returned by the `pop` method will be a value of type `T`

You create instances of `Stack` in a similar way to `Array` and `Dictionary`, by writing the actual type to be used for this specific stack within angle brackets after the type name when creating a new instance with initializer syntax:

```
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:

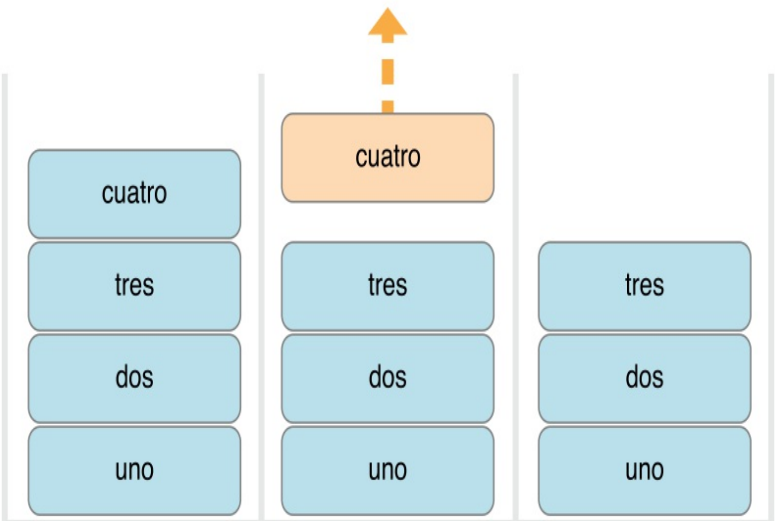


Popping a value from the stack returns and removes the top value, `"cuatro"`:

```
let fromTheTop = stackOfStrings.pop()
```

```
// fromTheTop is equal to "cuatro", and the stack now contains 3
              strings
```

Here's how the stack looks after popping its top value:



Because it is 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`.

Type Constraints

The `swapTwoValues` function and the `Stack` type can work with any type. However, it is 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 explicit 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):

```
func someFunction<T: SomeClass, U: SomeProtocol>(someT: T, someU: U) {
    // function body goes here
}
```

The hypothetical function above has two type parameters. The first type parameter, `T`, has a type constraint that requires `T` to be a subclass of `SomeClass`. The second type parameter, `U`, has a type constraint that requires `U` to conform to the protocol `SomeProtocol`.

Type Constraints in Action

Here's a non-generic function called `findStringIndex`, which is given a `String` value to find and an array of `String` values within which to find it. The `findStringIndex` function returns an optional `Int` value, which will be the index of the first matching string in the array if it is found, or `nil` if the string cannot be found:

```
func findStringIndex(array: String[], valueToFind: String) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

The `findStringIndex` function can be used to find a string value in an array of strings:

```
let strings = ["cat", "dog", "llama", "parakeet", "terrapin"]
if let foundIndex = findStringIndex(strings, "llama") {
    println("The index of llama is \(foundIndex)")
}
// 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 called `findIndex`, by replacing any mention of strings with values of some type `T` instead.

Here's how you might expect a generic version of `findStringIndex`, called `findIndex`, 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 does not compile, for reasons explained after the example:

```
func findIndex<T>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
```

```
}
```

This function does not 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 is not something that Swift can guess for you. Because of this, it is not 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` function, because it is 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>(array: T[], valueToFind: T) -> Int? {
    for (index, value) in enumerate(array) {
        if value == valueToFind {
            return index
        }
    }
    return nil
}
```

The single type parameter for `findIndex` is written as `T: Equatable`, which means "any type `T` that conforms to the `Equatable` protocol."

The `findIndex` function now compiles successfully and can be used with any type that is `Equatable`, such as `Double` or `String`:

```
let doubleIndex = findIndex([3.14159, 0.1, 0.25], 9.3)
// doubleIndex is an optional Int with no value, because 9.3 is not in
               the array
```

```
let stringIndex = findIndex(["Mike", "Malcolm", "Andrea"], "Andrea")
// stringIndex is an optional Int containing a value of 2
```

Associated Types

When defining a protocol, it is sometimes useful to declare one or more *associated types* as part of the protocol's definition. An associated type gives a placeholder name (or *alias*) to a type that is used as part of the protocol. The actual type to use for that associated type is not specified until the protocol is adopted. Associated types are specified with the `typealias` keyword.

Associated Types in Action

Here's an example of a protocol called `Container`, which declares an associated type called `ItemType`:

```
protocol Container {
    typealias ItemType
    mutating func append(item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { 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 are 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 `Container` protocol declares an associated type called `ItemType`, written as `typealias ItemType`. The protocol does not define what `ItemType` is an alias *for*—that information is left for any conforming type to provide. Nonetheless, the `ItemType` alias provides a way to refer to the type of the items in a `Container`, and to define a type for use with the `append` method and subscript, to ensure that the expected behavior of any `Container` is enforced.

Here's a version of the non-generic `IntStack` type from earlier, adapted to conform to the `Container` protocol:

```
struct IntStack: Container {
    // original IntStack implementation
    var items = Int[]()
    mutating func push(item: Int) {
        items.append(item)
    }
    mutating func pop() -> Int {
        return items.removeLast()
    }
    // conformance to the Container protocol
    typealias ItemType = Int
    mutating func append(item: Int) {
        self.push(item)
    }
```

```
    var count: Int {
    return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

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 `ItemType` to use is a type of `Int`. The definition of `typealias ItemType = Int` turns the abstract type of `ItemType` 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 `ItemType` 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 `ItemType` 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 ItemType = Int` line from the code above, everything still works, because it is clear what type should be used for `ItemType`.

You can also make the generic `Stack` type conform to the `Container` protocol:

```
struct Stack<T>: Container {
    // original Stack<T> implementation
    var items = T[]()
    mutating func push(item: T) {
        items.append(item)
    }
    mutating func pop() -> T {
        return items.removeLast()
    }
    // conformance to the Container protocol
    mutating func append(item: T) {
        self.push(item)
```

```
    }
    var count: Int {
    return items.count
    }
    subscript(i: Int) -> T {
        return items[i]
    }
}
```

This time, the placeholder type parameter `T` is used as the type of the `append` method's `item` parameter and the return type of the subscript. Swift can therefore infer that `T` is the appropriate type to use as the `ItemType` 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 Adding Protocol Conformance with an Extension. 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 `ItemType`, just as for the generic `Stack` type above. After defining this extension, you can use any `Array` as a `Container`.

Where Clauses

Type constraints, as described in Type Constraints, enable you to define requirements on the type parameters associated with a generic function or type.

It can also be useful to define requirements for associated types. You do this by defining *where clauses* as part of a type parameter list. A where clause enables you to require that an associated type conforms to a certain protocol, and/or that certain type parameters and associated types be the same. You write a where clause by placing the `where` keyword immediately after the list of type parameters, followed by one or more constraints for associated types, and/or one or more equality relationships between types and associated types.

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 do not.

The two containers to be checked do not 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 where clauses:

```
func allItemsMatch<
    C1: Container, C2: Container
    where C1.ItemType == C2.ItemType, C1.ItemType: Equatable>
    (someContainer: C1, anotherContainer: C2) -> Bool {

        // check that both containers contain the same number of items
        if someContainer.count != anotherContainer.count {
            return false
        }

         // check each pair of items to see if they are
        equivalent
         for i in 0..someContainer.count {
             if someContainer[i] != anotherContainer[i] {
                 return false
             }
         }

         // all items match, so return true
```

```
        return true

    }
```

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 placeholder type parameters for two container types to be determined when the function is called.

The function's type parameter list places the following requirements on the 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 `ItemType` for `C1` must be the same as the `ItemType` for `C2` (written as `C1.ItemType == C2.ItemType`).
- The `ItemType` for `C1` must conform to the `Equatable` protocol (written as `C1.ItemType: Equatable`).

The third and fourth requirements are defined as part of a where clause, and are written after the `where` keyword as part of the function's type parameter list.

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 are 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 are exactly the same type as the items in `someContainer`.

These requirements enable the `allItemsMatch` function to compare the two containers, even if they are 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 is 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-closed range operator (`..`). For each item, the function checks whether the item from `someContainer` is not equal to the corresponding item in `anotherContainer`. If the two items are not equal, then the two containers do not 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>()
stackOfStrings.push("uno")
stackOfStrings.push("dos")
stackOfStrings.push("tres")

var arrayOfStrings = ["uno", "dos", "tres"]

if allItemsMatch(stackOfStrings, arrayOfStrings) {
    println("All items match.")
} else {
    println("Not all items match.")
}
// prints "All items match."
```

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.

Advanced Operators

In addition to the operators described in Basic Operators, 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 just 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 just like any of the predefined operators, and you can even extend existing types to support the custom operators you define.

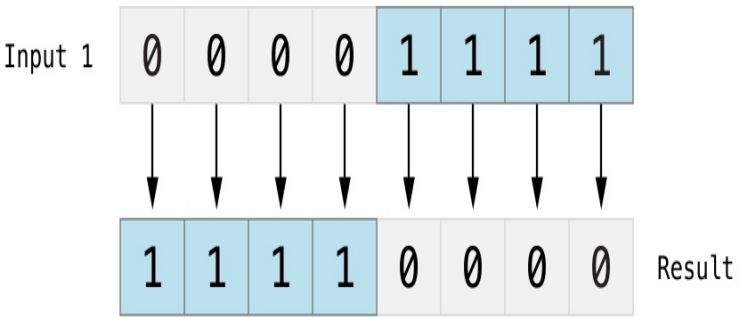
Bitwise Operators

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:

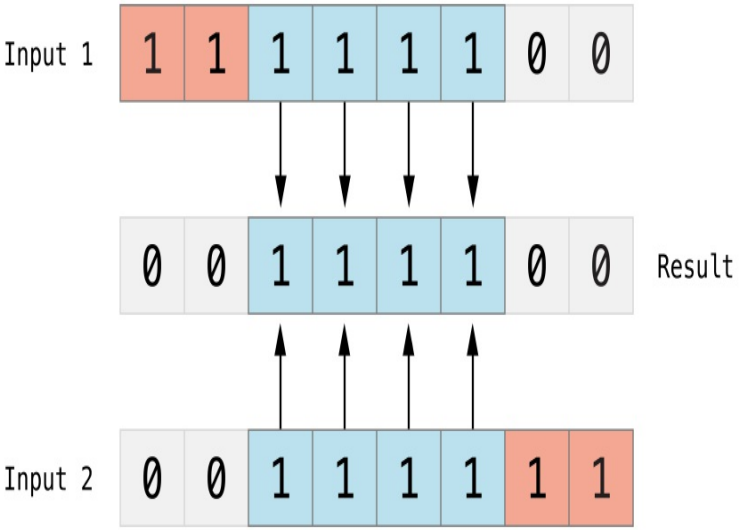
```
let initialBits: UInt8 = 0b00001111
let invertedBits = ~initialBits  // equals 11110000
```

`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. Zeroes become ones, and ones become zeroes. 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:

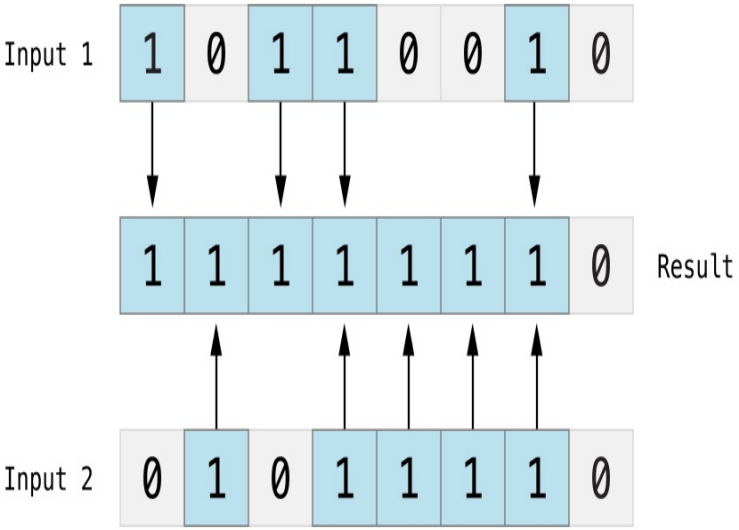


In the example below, the values of `firstSixBits` and `lastSixBits` both have four middle bits equal to `1`. The bitwise AND operator combines them to make the number `00111100`, which is equal to an unsigned decimal value of `60`:

```
let firstSixBits: UInt8 = 0b11111100
let lastSixBits: UInt8  = 0b00111111
let middleFourBits = firstSixBits & lastSixBits  // equals 00111100
```

Bitwise OR Operator

The *bitwise OR operator* (|) 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:

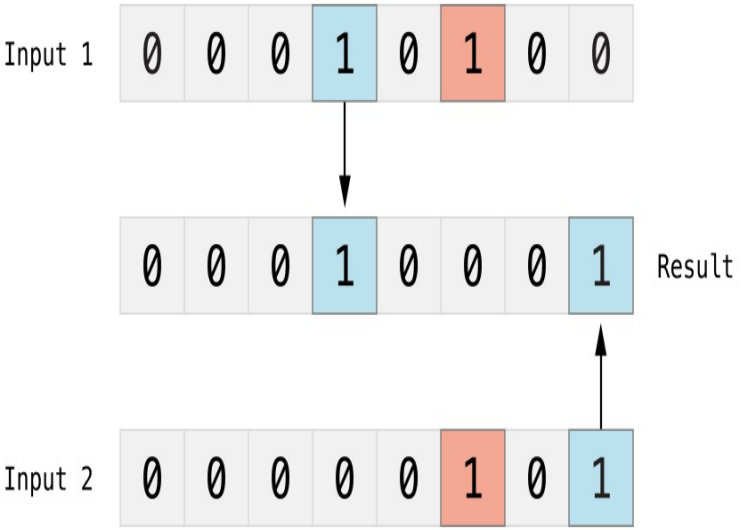


In the example below, the values of `someBits` and `moreBits` have different bits set to `1`. The bitwise OR operator combines them to make the number `11111110`, which equals an unsigned decimal of `254`:

```
let someBits: UInt8 = 0b10110010
let moreBits: UInt8 = 0b01011110
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:

```
let firstBits: UInt8 = 0b00010100
let otherBits: UInt8 = 0b00000101
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 number 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.

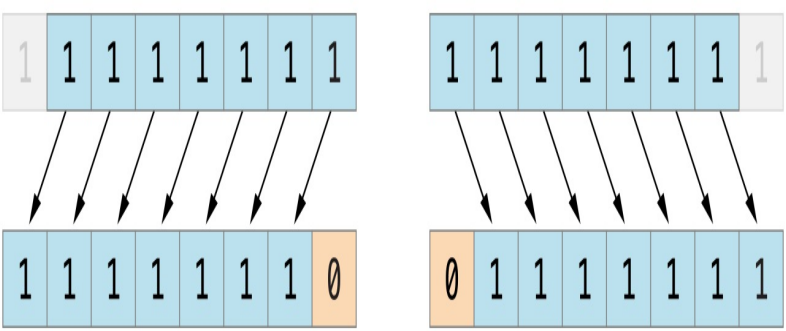
Shifting Behavior for Unsigned Integers

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. Zeroes 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 `11111111` shifted to the right by `1` place). Blue numbers are shifted, gray numbers are discarded, and orange zeroes are inserted:



Here's how bit shifting looks in Swift code:

```
let shiftBits: UInt8 = 4   // 00000100 in binary
shiftBits << 1             // 00001000
shiftBits << 2             // 00010000
shiftBits << 5             // 10000000
shiftBits << 6             // 00000000
shiftBits >> 2             // 00000001
```

You can use bit shifting to encode and decode values within other data types:

```
let pink: UInt32 = 0xCC6699
let redComponent = (pink & 0xFF0000) >> 16    // redComponent is 0xCC,
                or 204
let greenComponent = (pink & 0x00FF00) >> 8   // greenComponent is
                0x66, or 102
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 zeroes 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 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 `0x0000FF`, 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`.

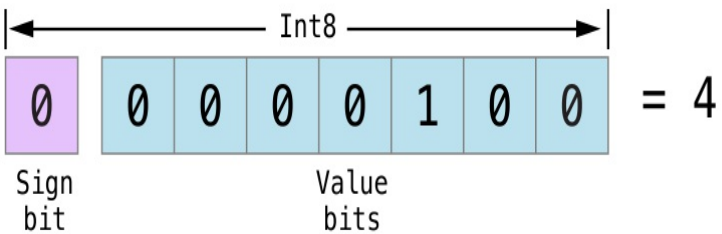
Shifting Behavior for Signed Integers

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.

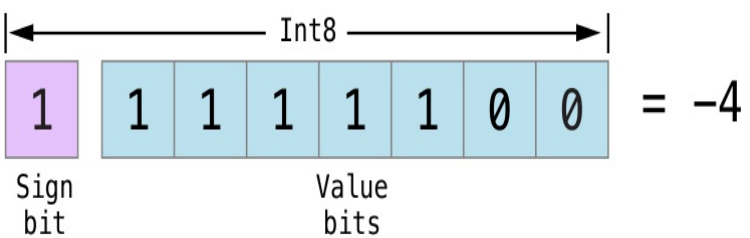
The remaining bits (known as the *value bits*) store the actual value. Positive numbers are stored in exactly the same way as for unsigned integers, counting upwards from `0`. Here's how the bits inside an `Int8` look for the number `4`:



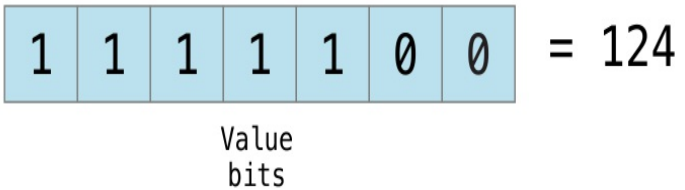
The sign bit is `0` (meaning "positive"), and the seven value bits are just the number `4`, written in binary notation.

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`:

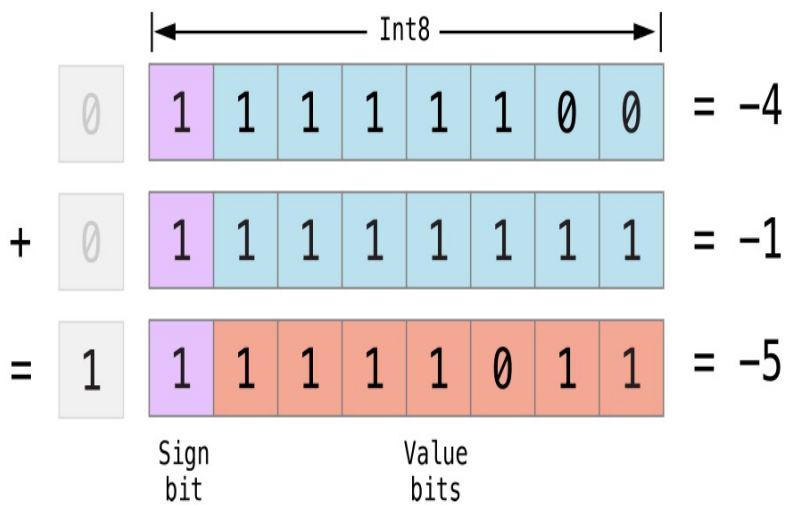


This time, the sign bit is `1` (meaning "negative"), and the seven value bits have a binary value of `124` (which is `128 - 4`):



The 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.

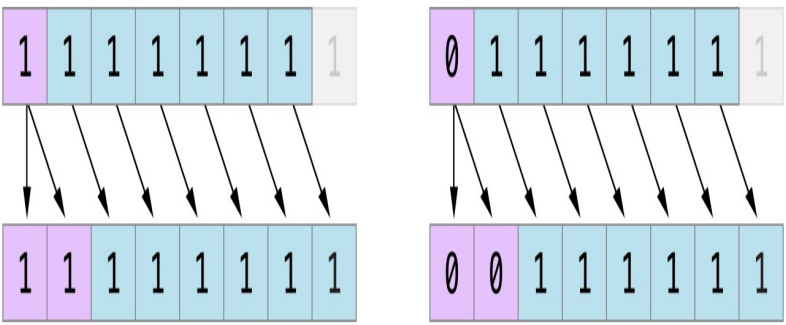
First, you can add `-1` to `-4`, simply by performing a standard binary addition of all eight bits (including the sign bit), and discarding anything that doesn't fit in the eight bits once you're done:



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 number between `-32768` and `32767`. Trying to set a `UInt16` constant or variable to a number outside of this range causes an error:

```
1 var potentialOverflow = Int16.max
```

```
// potentialOverflow equals 32767, which is the largest 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 five 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 (`&*`)

Overflow division (`&/`)

Overflow remainder (`&%`)

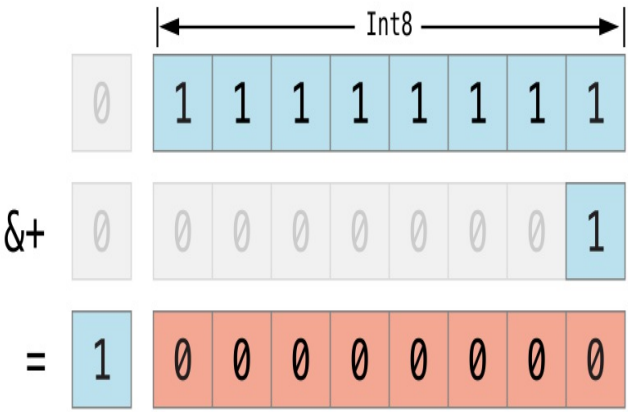
Value Overflow

Here's an example of what happens when an unsigned value is allowed to overflow, using the overflow addition operator (`&+`):

```
var willOverflow = UInt8.max
// willOverflow equals 255, which is the largest value a UInt8 can
                hold
willOverflow = willOverflow &+ 1
// willOverflow is now equal to 0
```

The variable `willOverflow` is initialized with the largest 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

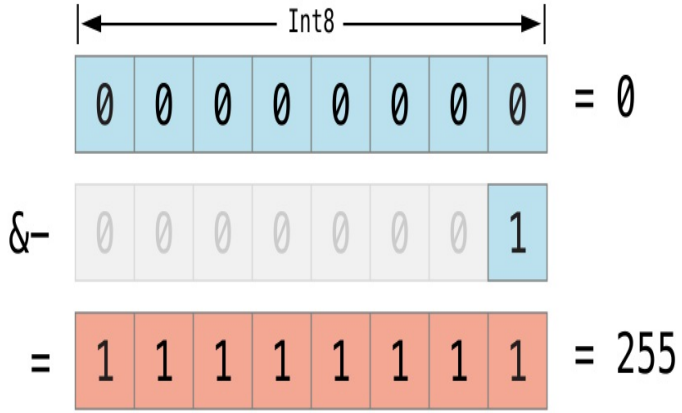
the diagram below. The value that remains within the bounds of the `UInt8` after the overflow addition is `00000000`, or zero:



Value Underflow

Numbers can also become too small to fit in their type's maximum bounds. Here's an example.

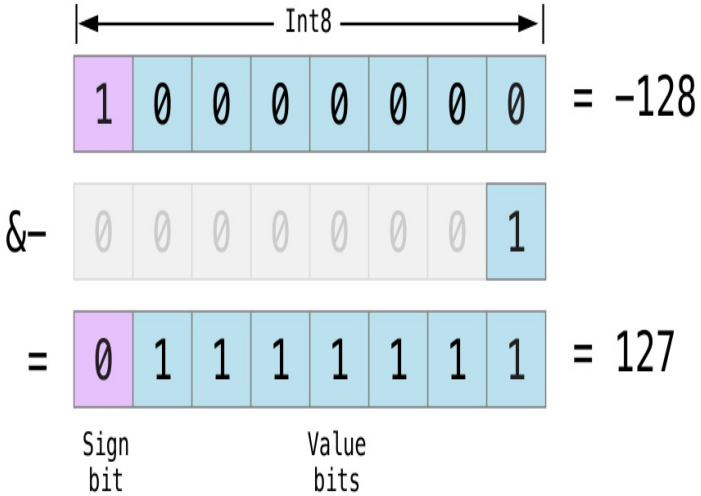
The *smallest* value that a UInt8 can hold is `0` (which is `00000000` in eight-bit binary form). If you subtract `1` from `00000000` using the overflow subtraction operator, the number will overflow back round to `11111111`, or `255` in decimal:



Here's how that looks in Swift code:

```
var willUnderflow = UInt8.min
// willUnderflow equals 0, which is the smallest value a UInt8 can
              hold
willUnderflow = willUnderflow &- 1
// willUnderflow is now equal to 255
```

A similar underflow occurs for signed integers. All subtraction for signed integers is performed as straight binary subtraction, with the sign bit included as part of the numbers being subtracted, as described in Bitwise Left and Right Shift Operators. The smallest number that an `Int8` can hold is `-128`, which is `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 largest positive value that an `Int8` can hold:



Here's the same thing in Swift code:

```
var signedUnderflow = Int8.min
// signedUnderflow equals -128, which is the smallest value an Int8
               can hold
signedUnderflow = signedUnderflow &- 1
// signedUnderflow is now equal to 127
```

The end result of the overflow and underflow behavior described above is that for both signed and unsigned integers, overflow always wraps around from the largest valid integer value back to the smallest, and underflow always wraps around from the smallest value to the largest.

Division by Zero

Dividing a number by zero (`i / 0`), or trying to calculate remainder by zero (`i % 0`), causes an error:

```
let x = 1
let y = x / 0
```

However, the overflow versions of these operators (`&/` and `&%`) return a value of zero if you divide by zero:

```
let x = 1
let y = x &/ 0
// y is equal to 0
```

Precedence and Associativity

Operator *precedence* gives some operators higher priority than others; these operators are calculated first.

Operator *associativity* defines how operators of the same precedence are grouped together (or *associated*)—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. Here's an example. Why does the following expression equal `4`?

```
2 + 3 * 4 % 5
// this equals 4
```

Taken strictly from left to right, you might expect this to read as follows:

> 2 plus 3 equals 5;
>
> 5 times 4 equals 20;
>
> 20 remainder 5 equals 0

However, the actual answer is `4`, not `0`. Higher-precedence operators are evaluated before lower-precedence ones. In Swift, as in C, the multiplication operator (`*`) and the remainder operator (`%`) have a higher precedence than the addition operator (`+`). As a result, they are both evaluated before the addition is considered.

However, multiplication and remainder have the *same* precedence as each other. To work out the exact

evaluation order to use, you also need to consider their associativity. Multiplication and remainder 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 `12`, so this is equivalent to:

```
2 + (12 % 5)
```

`(12 % 5)` is `2`, so this is equivalent to:

```
2 + 2
```

This calculation yields the final answer of `4`.

For a complete list of Swift operator precedences and associativity rules, see Expressions.

> 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 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 Functions

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 function* to add together instances of the `Vector2D` structure:

```
struct Vector2D {
    var x = 0.0, y = 0.0
}
@infix func + (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y + right.y)
}
```

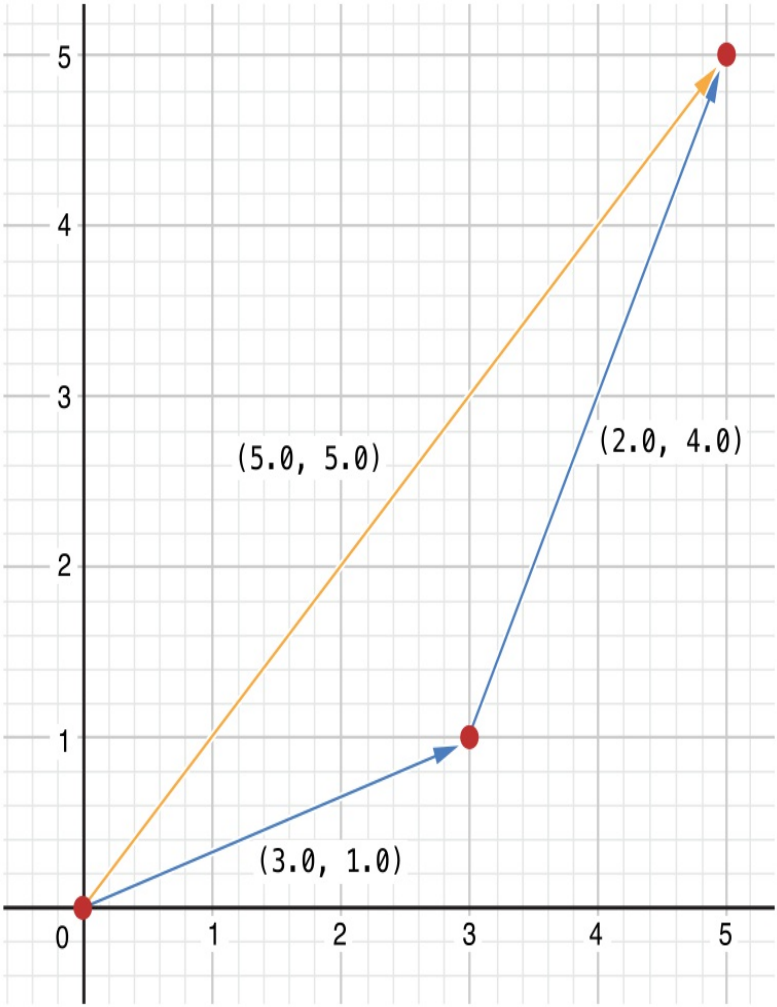
The operator function is defined as a global function called `+`, which takes two input parameters of type `Vector2D` and returns a single output value, also of type `Vector2D`. You implement an infix operator by writing the `@infix` attribute before the `func` keyword when declaring the operator function.

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 function 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 function is defined globally, rather than as a method on the `Vector2D` structure, so that it can be used as an infix operator between existing `Vector2D` instances:

```
let vector = Vector2D(x: 3.0, y: 1.0)
let anotherVector = Vector2D(x: 2.0, y: 4.0)
let combinedVector = vector + anotherVector
// 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 `i++`).

You implement a prefix or postfix unary operator by writing the `@prefix` or `@postfix` attribute before the `func` keyword when declaring the operator function:

```
@prefix func - (vector: Vector2D) -> Vector2D {
    return Vector2D(x: -vector.x, y: -vector.y)
}
```

The example above implements the unary minus operator (`-a`) for `Vector2D` instances. The unary minus operator is a prefix operator, and so this function has to be qualified with the `@prefix` attribute.

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:

```
let positive = Vector2D(x: 3.0, y: 4.0)
let negative = -positive
// negative is a Vector2D instance with values of (-3.0, -4.0)
let alsoPositive = -negative
// 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. Operator functions that implement compound assignment must be qualified with the `@assignment` attribute. You must also mark a compound assignment operator's left input parameter as `inout`, because the parameter's value will be modified directly from within the operator function.

The example below implements an addition assignment operator function for `Vector2D` instances:

```
@assignment func += (inout left: Vector2D, right: Vector2D) {
```

```
    left = left + right
}
```

Because an addition operator was defined earlier, you don't need to reimplement the addition process here. Instead, the addition assignment operator function takes advantage of the existing addition operator function, 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)
```

You can combine the `@assignment` attribute with either the `@prefix` or `@postfix` attribute, as in this implementation of the prefix increment operator (`++a`) for `Vector2D` instances:

```
@prefix @assignment func ++ (inout vector: Vector2D) -> Vector2D {
    vector += Vector2D(x: 1.0, y: 1.0)
    return vector
}
```

The prefix increment operator function above takes advantage of the addition assignment operator defined earlier. It adds a `Vector2D` with `x` and `y` values of `1.0` to the `Vector2D` on which it is called, and returns the result:

```
var toIncrement = Vector2D(x: 3.0, y: 4.0)
let afterIncrement = ++toIncrement
// toIncrement now has values of (4.0, 5.0)
// afterIncrement also has values of (4.0, 5.0)
```

> NOTE
>
> It is not possible to overload the default assignment operator (`=`). Only the compound assignment

operators can be overloaded. Similarly, the ternary conditional operator (`a ? b : c`) cannot be overloaded.

Equivalence Operators

Custom classes and structures do not receive a default implementation of the *equivalence operators*, known as the "equal to" operator (`==`) and "not equal to" operator (`!=`). It is not possible for Swift to guess what would qualify as "equal" for your own custom types, because the meaning of "equal" depends on the roles that those types play in your code.

To use the equivalence operators to check for equivalence of your own custom type, provide an implementation of the operators in the same way as for other infix operators:

```
@infix func == (left: Vector2D, right: Vector2D) -> Bool {
    return (left.x == right.x) && (left.y == right.y)
}
@infix func != (left: Vector2D, right: Vector2D) -> Bool {
    return !(left == right)
}
```

The above example implements an "equal to" operator (`==`) to check if 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. The example also implements the "not equal to" operator (`!=`), which simply returns the inverse of the result of the "equal to" operator.

You can now use these operators to check whether two `Vector2D` instances are equivalent:

```
let twoThree = Vector2D(x: 2.0, y: 3.0)
let anotherTwoThree = Vector2D(x: 2.0, y: 3.0)
if twoThree == anotherTwoThree {
    println("These two vectors are equivalent.")
```

```
}
// prints "These two vectors are equivalent."
```

Custom Operators

You can declare and implement your own *custom operators* in addition to the standard operators provided by Swift. Custom operators can be defined only with the characters `/ = - + * % < > ! & | ^ . ~`.

New operators are declared at a global level using the `operator` keyword, and can be declared as `prefix`, `infix` or `postfix`:

```
operator prefix +++ {}
```

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 incrementer” 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:

```
@prefix @assignment func +++ (inout vector: Vector2D) -> Vector2D {
    vector += vector
    return vector
}
```

This implementation of `+++` is very similar to the implementation of `++` for `Vector2D`, except that this operator function adds the vector to itself, rather than adding `Vector2D(1.0, 1.0)`:

```
var toBeDoubled = Vector2D(x: 1.0, y: 4.0)
let afterDoubling = +++toBeDoubled
// toBeDoubled now has values of (2.0, 8.0)
// afterDoubling also has values of (2.0, 8.0)
```

Precedence and Associativity for Custom Infix Operators

Custom `infix` operators can also specify a *precedence* and an *associativity*. See Precedence and Associativity for an explanation of how these two characteristics affect an infix operator's interaction with other infix operators.

The possible values for `associativity` are `left`, `right`, and `none`. Left-associative operators associate to the left if written next to other left-associative operators of the same precedence. Similarly, right-associative operators associate to the right if written next to other right-associative operators of the same precedence. Non-associative operators cannot be written next to other operators with the same precedence.

The `associativity` value defaults to `none` if it is not specified. The `precedence` value defaults to `100` if it is not specified.

The following example defines a new custom `infix` operator called `+-`, with `left` associativity and a precedence of `140`:

```
operator infix +- { associativity left precedence 140 }
func +- (left: Vector2D, right: Vector2D) -> Vector2D {
    return Vector2D(x: left.x + right.x, y: left.y - right.y)
}
let firstVector = Vector2D(x: 1.0, y: 2.0)
let secondVector = Vector2D(x: 3.0, y: 4.0)
let plusMinusVector = firstVector +- secondVector
// plusMinusVector is a Vector2D instance with values of (4.0, -2.0)
```

This operator adds together the `x` 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 associativity and precedence values (`left` and `140`) as default additive infix operators such as `+` and `-`. For a complete list of the default Swift operator precedence and associativity settings, see Expressions.

Language Reference

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 → { *getter-clause* *setter-clause*$_{opt}$ }
getter-setter-block → { *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.

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 the end of the line. Multiline comments begin with `/*` and end with `*/`. Nesting is allowed, but the comment markers must be balanced.

Identifiers

Identifiers begin with an upper case or lower case letter A through Z, an underscore (_), a noncombining alphanumeric Unicode character in the Basic Multilingual Plane, or a character outside the Basic Multilingual Plan 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 are not 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.

GRAMMAR OF AN IDENTIFIER

identifier → *identifier-head* *identifier-characters* *opt*
identifier → ` *identifier-head* *identifier-characters* *opt* `
identifier → *implicit-parameter-name*
identifier-list → *identifier* | *identifier* , *identifier-list*

identifier-head → Upper- or lowercase letter A through Z
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+CFFFD
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*

Keywords

The following keywords are reserved and may not be used as identifiers, unless they're escaped with backticks, as described above in Identifiers.

Keywords used in declarations: `class`, `deinit`, `enum`, `extension`, `func`, `import`, `init`, `let`, `protocol`, `static`, `struct`, `subscript`, `typealias`, and `var`.

Keywords used in statements: `break`, `case`, `continue`, `default`, `do`, `else`, `fallthrough`, `if`, `in`, `for`, `return`, `switch`, `where`, and `while`.

Keywords used in expressions and types: `as`, `dynamicType`, `is`, `new`, `super`, `self`, `Self`, `Type`, `__COLUMN__`, `__FILE__`, `__FUNCTION__`, and `__LINE__`.

Keywords reserved in particular contexts: `associativity`, `didSet`, `get`, `infix`, `inout`, `left`, `mutating`, `none`, `nonmutating`, `operator`, `override`, `postfix`, `precedence`, `prefix`, `right`, `set`, `unowned`, `unowned(safe)`, `unowned(unsafe)`, `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 an integer, floating-point number, or string type. The following are examples of literals:

```
42              // Integer literal
3.14159         // Floating-point literal
"Hello, world!" // String literal
```

GRAMMAR OF A LITERAL

literal → *integer-literal* *floating-point-literal* *string-literal*

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 are ignored and therefore don't affect the value of the literal. Integer literals can begin with leading zeros (`0`), but are likewise ignored and don't affect the base or value of the literal.

Unless otherwise specified, the default 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 Integers.

GRAMMAR OF AN INTEGER LITERAL

integer-literal → *binary-literal*
integer-literal → *octal-literal*
integer-literal → *decimal-literal*
integer-literal → *hexadecimal-literal*

binary-literal → `0b` *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* *opt*

octal-literal → `0o` *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 → *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*

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 sequence of decimal digits that indicates what power of 10 the value preceding the `e` is multiplied by. For example, `1.25e2` represents 1.25×10^2, which evaluates to `125.0`. Similarly, `1.25e-2` represents 1.25×10^{-2}, which evaluates to `0.0125`.

Hexadecimal floating-point literals consist of a `0x` 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 sequence of decimal digits that indicates what power of 2 the value preceding the `p` is multiplied by. For example, `0xFp2` represents 15×2^2, which evaluates to `60`. Similarly, `0xFp-2` represents 15×2^{-2}, which evaluates to `3.75`.

Unlike with integer literals, negative floating-point numbers are expressed by applying the unary minus operator (`-`) to a floating-point literal, as in `-42.0`. The result is an expression, not a floating-point integer literal.

Underscores (`_`) are allowed between digits for readability, but are ignored and therefore don't affect the value of the literal. Floating-point literals can begin with leading zeros (`0`), but are likewise ignored and don't affect the base or value of the literal.

Unless otherwise specified, the default type of a floating-point literal is the Swift standard library type `Double`, which represents a 64-bit floating-point number. The Swift standard library also defines a `Float` 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-literal opt
hexadecimal-exponent → floating-point-p sign opt hexadecimal-literal

floating-point-e → e E
floating-point-p → p P
sign → + −
```

String Literals

A string literal is a sequence of characters surrounded by double quotes, with the following form:

```
" characters "
```

String literals cannot contain an unescaped double quote (`"`), an unescaped backslash (`\`), a carriage return, or a line feed.

Special characters can be included in string literals using the following escape sequences:

Null Character (`\0`)

Backslash (`\\`)

Horizontal Tab (`\t`)

Line Feed (`\n`)

Carriage Return (`\r`)

Double Quote (`\"`)

Single Quote (`\'`)

Characters can also be expressed by `\x` followed by two hexadecimal digits, `\u` followed by four hexadecimal digits, or `\U` followed by eight hexadecimal digits. The digits in these escape sequences identify a Unicode codepoint.

The value of an expression can be inserted into a string literal by placing the expression in parentheses after a backslash (`\`). The interpolated expression must not contain an unescaped double quote (`"`), an unescaped

backslash (\), a carriage return, or a line feed. The expression must evaluate to a value of a type that the `String` class has an initializer for.

For example, all the following string literals have the same value:

```
"1 2 3"
"1 2 \(3)"
"1 2 \(1 + 2)"
var x = 3; "1 2 \(x)"
```

The default type of a string literal is `String`. The characters that make up a string are of type `Character`. For more information about the `String` and `Character` types, see Strings and Characters.

GRAMMAR OF A STRING LITERAL

string-literal → " *quoted-text* "
quoted-text → *quoted-text-item* *quoted-text* *opt*
quoted-text-item → *escaped-character*
quoted-text-item → \(*expression*)
quoted-text-item → Any Unicode extended grapheme cluster except " , \ , U+000A, or U+000D

escaped-character → \0 \\ \t \n \r \" \'
escaped-character → \x *hexadecimal-digit* *hexadecimal-digit*
escaped-character → \u *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit*
escaped-character → \U *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit*

Operators

The Swift standard library defines a number of operators for your use, many of which are discussed in Basic Operators and Advanced Operators. The present section describes which characters can be used as operators.

Operators are made up of one or more of the following characters: /, =, −, +, !, *, %, <, >, &, |, ^, ~, and .. That said, the tokens =, ->, //, /*, */, ., and the unary prefix operator & are reserved. These tokens can't be overloaded, nor can they be used to define 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 is 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 is 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 is 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 is 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 is one caveat to the rules above. If the `!` or `?` operator has no whitespace on the left, it is treated as a postfix operator, regardless of whether it has whitespace on the right. To use the `?` operator as syntactic sugar for the `Optional` type, it must not have whitespace on the left. To use it in the 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 is 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 Custom Operators and Operator Declaration. To learn how to overload existing operators, see Operator Functions.

GRAMMAR OF OPERATORS

operator → *operator-character* *operator* *opt*
operator-character → `/` `=` `-` `+` `!` `*` `%` `<` `>` `&` `|` `^` `~` `.`

binary-operator → *operator*
prefix-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 is 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 are 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 instance, 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)`.

This chapter discusses the types defined in the Swift language itself and describes the type inference behavior of Swift.

> GRAMMAR OF A TYPE
>
> *type* → *array-type* *function-type* *type-identifier* *tuple-type* *optional-type* *implicitly-unwrapped-optional-type* *protocol-composition-type* *metatype-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:

```
let someTuple: (Double, Double) = (3.14159, 2.71828)
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* *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 does not 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 zero or more 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.

`Void` is a typealias for the the empty tuple type, `()`. If there is only one element inside the parentheses, the type is simply the type of that element. For example, the type of `(Int)` is `Int`, not `(Int)`. As a result, you can label a tuple element only when the tuple type has two or more elements.

GRAMMAR OF A TUPLE TYPE

tuple-type → (*tuple-type-body* *opt*)
tuple-type-body → *tuple-type-element-list* ... *opt*
tuple-type-element-list → *tuple-type-element* | *tuple-type-element* , *tuple-type-element-list*
tuple-type-element → *attributes* *opt* inout *opt* *type* | inout *opt* *element-name* *type-annotation*
element-name → *identifier*

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
```

Because the *parameter type* and the *return type* can be a tuple type, function types support functions and methods that take multiple paramaters and return multiple values.

You can apply the `auto_closure` attribute to a function type that has a parameter type of `()` and that returns the type of an expression (see Type Attributes). An autoclosure function captures an implicit closure over the

specified expression, instead of the expression itself. The following example uses the `auto_closure` attribute in defining a very simple assert function:

```
func simpleAssert(condition: @auto_closure () -> Bool, message:
                  String) {
    if !condition() {
        println(message)
    }
}
let testNumber = 5
simpleAssert(testNumber % 2 == 0, "testNumber isn't an even number.")
// prints "testNumber isn't an even number."
```

A function type can have a variadic parameter as the last 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 Variadic Parameters.

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.

The type of a curried function is equivalent to a nested function type. For example, the type of the curried function `addTwoNumbers()()` below is `Int -> Int -> Int`:

```
func addTwoNumbers(a: Int)(b: Int) -> Int {
    return a + b
}
addTwoNumbers(4)(5) // Returns 9
```

The function types of a curried function are grouped from right to left. For instance, 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 return an `Int`. For example, you can rewrite the curried function `addTwoNumbers()()` as the following nested function:

```
func addTwoNumbers(a: Int) -> (Int -> Int) {
    func addTheSecondNumber(b: Int) -> Int {
        return a + b
    }
    return addTheSecondNumber
}
addTwoNumbers(4)(5) // Returns 9
```

GRAMMAR OF A FUNCTION TYPE

function-type → *type* -> *type*

Array Type

The Swift language uses square brackets (`[]`) immediately after the name of a type as syntactic sugar for the named type `Array<T>`, which is defined in the Swift standard library. In other words, the following two declarations are equivalent:

```
let someArray: String[] = ["Alex", "Brian", "Dave"]
let someArray: Array<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 using square brackets as well: `someArray[0]` refers to the element at index 0, `"Alex"`.

As the above example also shows, you can use square brackets to create an array using an array literal. 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[] = []
```

You can create multidimensional arrays by chaining multiple sets of square brackets to the name of the base type of the elements. 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* [ ] *array-type* [ ]

Optional Type

The Swift language defines the postfix `?` as syntactic sugar for the named type `Optional<T>`, 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 `Optional<T>` is an enumeration with two cases, `None` and `Some(T)`, 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. When declaring an optional type, be sure to use parentheses to properly scope the `?` operator. As an example, to declare an optional array of integers, write the type annotation as `(Int[])?`; writing `Int[]?` produces an error.

If you don't provide an initial value when you declare an optional variable or property, its value automatically defaults to `nil`.

Optionals conform to the `LogicValue` protocol and therefore may occur in a Boolean context. In that context, if an instance of an optional type `T?` contains any value of type `T` (that is, it's value is `Optional.Some(T)`), the optional type evaluates to `true`. Otherwise, it evaluates to `false`.

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 `ImplicitlyUnwrappedOptional<T>`, which is defined in the Swift standard library. In other words, the following two declarations are equivalent:

```
var implicitlyUnwrappedString: String!
var implicitlyUnwrappedString: ImplicitlyUnwrappedOptional<String>
```

In both cases, the variable `implicitlyUnwrappedString` is declared to have the type of an implicitly unwrapped optional string. Note that no whitespace may appear between the type and the `!`.

You can use implicitly unwrapped optionals in all the same places in your code that you can use optionals. For instance, 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, it's value automatically defaults to `nil`.

Because the value of an implicitly unwrapped optional is automatically unwrapped when you use it, there's no need to use the `!` operator to unwrap it. That said, if you try to use an implicitly unwrapped optional that has a value of `nil`, you'll get a runtime error.

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 Implicitly Unwrapped Optionals.

> GRAMMAR OF AN IMPLICITLY UNWRAPPED OPTIONAL TYPE
>
> *implicitly-unwrapped-optional-type* → *type* !

Protocol Composition Type

A protocol composition type describes a type that conforms to each protocol in a list of specified protocols. Protocol composition types may be used in type annotations and in generic parameters.

Protocol composition types have the following form:

```
protocol< Protocol 1 , Protocol 2 >
```

A protocol composition type allows you to specify a value whose type conforms to the requirements of multiple protocols without having to explicitly define a new, named protocol that inherits from each protocol you want the type to conform to. For example, specifying a protocol composition type `protocol<ProtocolA, ProtocolB, ProtocolC>` is effectively the same as defining a new protocol `ProtocolD` that inherits from `ProtocolA`, `ProtocolB`, and `ProtocolC`, but without having to introduce a new name.

Each item in a protocol composition list must be either the name of protocol or a type alias of a protocol composition type. If the list is empty, it specifies the empty protocol composition type, which every type conforms to.

GRAMMAR OF A PROTOCOL COMPOSITION TYPE

protocol-composition-type → `protocol` `<` *protocol-identifier-list*$_{opt}$ `>`
protocol-identifier-list → *protocol-identifier* | *protocol-identifier* `,` *protocol-identifier-list*
protocol-identifier → *type-identifier*

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 use a `dynamicType` expression with an instance of a type to access that instance's runtime type as a value, as the following example shows:

```
class SomeBaseClass {
    class func printClassName() {
        println("SomeBaseClass")
    }
}
class SomeSubClass: SomeBaseClass {
    override class func printClassName() {
        println("SomeSubClass")
    }
```

```
}
let someInstance: SomeBaseClass = SomeSubClass()
// someInstance is of type SomeBaseClass at compile time, but
// someInstance is of type SomeSubClass at runtime
someInstance.dynamicType.printClassName()
// prints "SomeSubClass"
```

GRAMMAR OF A METATYPE TYPE

metatype-type → *type* `.` `Type` *type* `.` `Protocol`

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 comma-separated list of type identifiers.

Class types may 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 does not inherit from another class, the list may begin with a protocol instead. For an extended discussion and several examples of class inheritance, see Inheritance.

Other named types may only inherit from or conform to a list of protocols. Protocol types may 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 may 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 Raw Values.

GRAMMAR OF A TYPE INHERITANCE CLAUSE

type-inheritance-clause → `:` *type-inheritance-list*
type-inheritance-list → *type-identifier* | *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 omit the type completely and simply write `var x = 0`—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 instance, 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 `x` in `var x: Int = 0` is inferred by first checking the type of `0` and then passing this type information up to the root (the variable `x`).

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 type `Float` instead of type `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* → *prefix-expression* *binary-expressions* *opt*
> *expression-list* → *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.

The Swift standard library provides the following prefix operators:

++ Increment

−− Decrement

! Logical NOT

~ Bitwise NOT

+ Unary plus

− Unary minus

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

In addition to the standard library operators listed above, 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 In-Out Parameters.

GRAMMAR OF A PREFIX EXPRESSION

prefix-expression → *prefix-operator* *opt* *postfix-expression*
prefix-expression → *in-out-expression*
in-out-expression → & *identifier*

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
```

The Swift standard library provides the following binary operators:

Exponentiative (No associativity, precedence level 160)

<< Bitwise left shift

>> Bitwise right shift

Multiplicative (Left associative, precedence level 150)

* Multiply

/ Divide

% Remainder

&* Multiply, ignoring overflow

&/ Divide, ignoring overflow

&% Remainder, ignoring overflow

`&` Bitwise AND

Additive (Left associative, precedence level 140)

`+` Add

`−` Subtract

`&+` Add with overflow

`&−` Subtract with overflow

`|` Bitwise OR

`^` Bitwise XOR

Range (No associativity, precedence level 135)

`..` Half-closed range

`...` Closed range

Cast (No associativity, precedence level 132)

`is` Type check

`as` Type cast

Comparative (No associativity, precedence level 130)

`<` Less than

`<=` Less than or equal

`>` Greater than

`>=` Greater than or equal

`==` Equal

`!=` Not equal

`===` Identical

!== Not identical

~= Pattern match

Conjunctive (Left associative, precedence level 120)

&& Logical AND

Disjunctive (Left associative, precedence level 110)

|| Logical OR

Ternary Conditional (Right associative, precedence level 100)

?: Ternary conditional

Assignment (Right associative, precedence level 90)

= Assign

*= Multiply and assign

/= Divide and assign

%= Remainder and assign

+= Add and assign

-= Subtract and assign

<<= Left bit shift and assign

>>= Right bit shift and assign

&= Bitwise AND and assign

^= Bitwise XOR and assign

|= Bitwise OR and assign

&&= Logical AND and assign

||= Logical OR and assign

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

> 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* *prefix-expression*
> *binary-expression* → *conditional-operator* *prefix-expression*
> *binary-expression* → *type-casting-operator*
> *binary-expressions* → *binary-expression* *binary-expressions* *opt*

Assignment Operator

The *assigment 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:

```
(a, _, (b, c)) = ("test", 9.45, (12, 3))
// 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 Ternary Conditional Operator.

GRAMMAR OF A CONDITIONAL OPERATOR

conditional-operator → ? *expression* :

Type-Casting Operators

There are two type-casting operators, the `as` operator and the `is` operator. They have the following form:

```
expression as type
expression as? type
expression is type
```

The `as` operator performs a cast of the *expression* to the specified *type*. It behaves as follows:

If conversion to the specified *type* is guaranteed to succeed, the value of the *expression* is returned as an instance of the specified *type*. An example is casting from a subclass to a superclass.

If conversion to the specified *type* is guaranteed to fail, a compile-time error is raised.

Otherwise, if it's not known at compile time whether the conversion will succeed, the type of the cast expresion is 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`. An example is casting from a superclass to a subclass.

```
class SomeSuperType {}
class SomeType: SomeSuperType {}
class SomeChildType: SomeType {}
let s = SomeType()

let x = s as SomeSuperType  // known to succeed; type is SomeSuperType
let y = s as Int            // known to fail; compile-time error
let z = s as SomeChildType  // might fail at runtime; type is
              SomeChildType?
```

Specifying a type with `as` provides the same information to the compiler as a type annotation, as shown in the following example:

```
let y1 = x as SomeType  // Type information from 'as'
let y2: SomeType = x    // Type information from an annotation
```

The `is` operator checks at runtime to see whether the *expression* is of the specified *type*. If so, it returns `true`; otherwise, it returns `false`.

The check must not be known to be true or false at compile time. The following are invalid:

```
"hello" is String
"hello" is Int
```

For more information about type casting and to see more examples that use the type-casting operators, see Type Casting.

GRAMMAR OF A TYPE-CASTING OPERATOR

type-casting-operator → `is` *type* `as` `?` *opt* *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.

GRAMMAR OF A PRIMARY EXPRESSION

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 → *implicit-member-expression*
primary-expression → *wildcard-expression*

Literal Expression

A *literal expression* consists of either an ordinary literal (such as a string or a number), an array or dictionary literal, or one of the following special literals:

Literal	Type	Value
		The name

`__FILE__`	String	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.

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.

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. An empty array literal is written as an empty pair of brackets (`[]`). The value of an array literal has type `T[]`, where `T` is the type of the expressions inside it. If there are expressions of multiple types, `T` is their closest common supertype.

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. An empty dictionary literal is written as a colon inside a pair of brackets (`[:]`) to distinguish it from an empty array literal. The value of a dictionary literal has type `Dictionary<KeyType, ValueType>`, where `KeyType` is the type of its key expressions and `ValueType` is the type of its value expressions. If there are expressions of multiple types, `KeyType` and `ValueType` are the closest common supertype for their respective values.

GRAMMAR OF A LITERAL EXPRESSION

literal-expression → *literal*
literal-expression → *array-literal* *dictionary-literal*
literal-expression → `__FILE__` `__LINE__` `__COLUMN__` `__FUNCTION__`

array-literal → `[` *array-literal-items* *opt* `]`
array-literal-items → *array-literal-item* `,` *opt* *array-literal-item* `,` *array-literal-items*
array-literal-item → *expression*

dictionary-literal → `[` *dictionary-literal-items* `]` `[` `:` `]`
dictionary-literal-items → *dictionary-literal-item* `,` *opt* *dictionary-literal-item* `,` *dictionary-literal-items*
dictionary-literal-item → *expression* `:` *expression*

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 static or class 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 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 moveByX(deltaX: Double, y deltaY: Double) {
        self = Point(x: x + deltaX, y: y + deltaY)
    }
}
```

GRAMMAR OF A SELF EXPRESSION

self-expression → `self`
self-expression → `self` `.` *identifier*
self-expression → `self` `[` *expression* `]`
self-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` `[` *expression* `]`
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 function declarations, closures contain statements which they execute, and they capture values from their 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 Function Declaration.

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 is also considered when performing type inference on the surrounding expression.

The following closure expressions are equivalent:

```
myFunction {
    (x: Int, y: Int) -> Int in
    return x + y
}

myFunction {
    (x, y) in
    return x + y
}

myFunction { return $0 + $1 }

myFunction { $0 + $1 }
```

For information about passing a closure as an argument to a function, see Function Call Expression.

A closure expression can explicitly specify the values that it captures from the surrounding scope using a *capture list*. A capture list is written as a comma separated list 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.

Each entry in the capture list can be marked as `weak` or `unowned` to capture a weak or unowned reference to the value.

```
myFunction { print(self.title) }                    // strong capture
myFunction { [weak self] in print(self!.title) }    // weak capture
myFunction { [unowned self] in print(self.title) }  // unowned capture
```

You can also bind arbitrary expression to named values in the capture list. The expression is evaluated when the closure is formed, and 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 Closure Expressions.

GRAMMAR OF A CLOSURE EXPRESSION

closure-expression → { *closure-signature*opt *statements* }

closure-signature → *parameter-clause* *function-result*opt `in`
closure-signature → *identifier-list* *function-result*opt `in`
closure-signature → *capture-list* *parameter-clause* *function-result*opt `in`
closure-signature → *capture-list* *identifier-list* *function-result*opt `in`
closure-signature → *capture-list* `in`

capture-list → `[` *capture-specifier* *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 class 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 → . *identifier*

Parenthesized Expression

A *parenthesized 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 ,
    ... )
```

Use parenthesized expressions to create tuples and to pass arguments to a function call. If there is only one value inside the parenthesized expression, the type of the parenthesized expression is the type of that value. For example, the type of the parenthesized expression `(1)` is `Int`, not `(Int)`.

GRAMMAR OF A PARENTHESIZED EXPRESSION

parenthesized-expression → (*expression-element-list* *opt*)
expression-element-list → *expression-element* | *expression-element* , *expression-element-list*
expression-element → *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 `x` and 20 is ignored:

```
(x, _) = (10, 20)
// x is 10, 20 is ignored
```

GRAMMAR OF A WILDCARD EXPRESSION

wildcard-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.

The Swift standard library provides the following postfix operators:

`++` Increment

`--` Decrement

For information about the behavior of these operators, see Basic Operators and Advanced Operators.

GRAMMAR OF A POSTFIX EXPRESSION

postfix-expression → *primary-expression*
postfix-expression → *postfix-expression* *postfix-operator*
postfix-expression → *function-call-expression*
postfix-expression → *initializer-expression*
postfix-expression → *explicit-member-expression*
postfix-expression → *postfix-self-expression*
postfix-expression → *dynamic-type-expression*
postfix-expression → *subscript-expression*
postfix-expression → *forced-value-expression*

Function Call Expression

A *function call expression* consists of a function name followed by a comma-separated 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:

```
function name ( argument name 1 : argument value 1 ,
        argument name 2 : argument value 2 )
```

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, {$0 == 13})
someFunction(x) {$0 == 13}
```

If the trailing closure is the function's only argument, the parentheses can be omitted.

```
// someFunction 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* *parenthesized-expression*
function-call-expression → *postfix-expression* *parenthesized-expression* *opt* *trailing-closure*
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. Unlike functions, an initializer can't be used as a value. For example:

```
var x = SomeClass.someClassFunction // ok
var y = SomeClass.init              // error
```

You also use an initializer expression to delegate to the initializer of a superclass.

```
class SomeSubClass: SomeSuperClass {
    init() {
        // subclass initialization goes here
        super.init()
    }
}
```

GRAMMAR OF AN INITIALIZER EXPRESSION

initializer-expression → *postfix-expression* . init

Explicit Member Expression

A *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.

GRAMMAR OF AN EXPLICIT MEMBER EXPRESSION

explicit-member-expression → *postfix-expression* `.` *decimal-digit*
explicit-member-expression → *postfix-expression* `.` *identifier* *generic-argument-clause* *opt*

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`

`type` `.self`

The first form evaluates to the value of the *expression*. For example, `x.self` evaluates to `x`.

The second form evaluates to the value of the *type*. Use this form to access a type as a value. For example, because `SomeClass.self` evaluates to the `SomeClass` type itself, you can pass it to a function or method that accepts a type-level argument.

GRAMMAR OF A SELF EXPRESSION

postfix-self-expression → *postfix-expression* `.` `self`

Dynamic Type Expression

A `dynamicType` expression consists of an expression, immediately followed by `.dynamicType`. It has the following form:

`expression` `.dynamicType`

The *expression* can't be the name of a type. The entire `dynamicType` expression evaluates to the value of the runtime type of the *expression*, as the following example shows:

```
class SomeBaseClass {
    class func printClassName() {
        println("SomeBaseClass")
    }
}
class SomeSubClass: SomeBaseClass {
    override class func printClassName() {
        println("SomeSubClass")
    }
```

```
}
let someInstance: SomeBaseClass = SomeSubClass()
// someInstance is of type SomeBaseClass at compile time, but
// someInstance is of type SomeSubClass at runtime
someInstance.dynamicType.printClassName()
// prints "SomeSubClass"
```

GRAMMAR OF A DYNAMIC TYPE EXPRESSION

dynamic-type-expression → *postfix-expression* `.` `dynamicType`

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* `[` *expression-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 nonoptional type. Otherwise, a runtime error is raised.

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` ?

On its own, the postfix `?` operator simply returns the value of its argument as an optional.

Postfix expressions that contain an optional-chaining expression are evaluated in a special way. If 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 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 both `.property` and `.performAction()`, and 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.

```
if let unwrappedC = c {
    result = unwrappedC.property.performAction()
}
```

GRAMMAR OF AN OPTIONAL-CHAINING EXPRESSION

optional-chaining-expression → *postfix-expression* ?

Statements

In Swift, there are two kinds of statements: simple statements and control flow statements. Simple statements are the most common and consist of either an expression or a declaration. Control flow statements are used to control the flow of execution in a program. There are three types of control flow statements in Swift: 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. Each type of control flow statement is described in detail below.

A semicolon (`;`) can optionally appear after any statement and is used to separate multiple statements if they appear on the same line.

> GRAMMAR OF A STATEMENT
>
> *statement* → *expression* `;` *opt*
> *statement* → *declaration* `;` *opt*
> *statement* → *loop-statement* `;` *opt*
> *statement* → *branch-statement* `;` *opt*
> *statement* → *labeled-statement*
> *statement* → *control-transfer-statement* `;` *opt*
> *statements* → *statement* *statements* *opt*

Loop Statements

Loop statements allow a block of code to be executed repeatedly, depending on the conditions specified in the loop. Swift has four loop statements: a `for` statement, a `for-in` statement, a `while` statement, and a `do-while` statement.

Control flow in a loop statement can be changed by a `break` statement and a `continue` statement and is discussed in Break Statement and Continue Statement below.

> GRAMMAR OF A LOOP STATEMENT

```
loop-statement → for-statement
loop-statement → for-in-statement
loop-statement → while-statement
loop-statement → do-while-statement
```

For Statement

A `for` statement allows a block of code to be executed repeatedly while incrementing a counter, as long as a condition remains true.

A `for` statement has the following form:

```
for initialization; condition; increment {
    statements
}
```

The semicolons between the *initialization*, *condition*, and *increment* are required. The braces around the *statements* in the body of the loop are also required.

A `for` statement is executed as follows:

1. The *initialization* is evaluated only once. It is typically used to declare and initialize any variables that are needed for the remainder of the loop.
2. The *condition* expression is evaluated.

   If `true`, the program executes the *statements*, and execution continues to step 3. If `false`, the program does not execute the *statements* or the *increment* expression, and the program is finished executing the `for` statement.
3. The *increment* expression is evaluated, and execution returns to step 2.

Variables defined within the *initialization* are valid only within the scope of the `for` statement itself.

The value of the *condition* expression must have a type that conforms to the `LogicValue` protocol.

GRAMMAR OF A FOR STATEMENT

for-statement → `for` *for-init* *opt* `;` *expression* *opt* `;` *expression* *opt* *code-block*
for-statement → `for` `(` *for-init* *opt* `;` *expression* *opt* `;` *expression* *opt* `)` *code-block*

for-init → *variable-declaration* *expression-list*

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 `Sequence` protocol.

A `for-in` statement has the following form:

```
for item in collection {
    statements
}
```

The `generate` method is called on the *collection* expression to obtain a value of a generator type—that is, a type that conforms to the `Generator` protocol. The program begins executing a loop by calling the `next` method on the stream. If the value returned is not `None`, 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` *pattern* `in` *expression* *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 have a type that conforms to the `LogicValue` protocol. The condition can also be an optional binding declaration, as discussed in Optional Binding.

> GRAMMAR OF A WHILE STATEMENT
>
> *while-statement* → `while` *while-condition* *code-block*
> *while-condition* → *expression* | *declaration*

Do-While Statement

A `do-while` statement allows a block of code to be executed one or more times, as long as a condition remains true.

A `do-while` statement has the following form:

```
do {
```

```
    statements
} while condition
```

A `do-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 `do-while` statement.

Because the value of the *condition* is evaluated after the *statements* are executed, the *statements* in a `do-while` statement are executed at least once.

The value of the *condition* must have a type that conforms to the `LogicValue` protocol. The condition can also be an optional binding declaration, as discussed in Optional Binding.

> GRAMMAR OF A DO-WHILE STATEMENT
>
> *do-while-statement* → `do` *code-block* `while` *while-condition*

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 two branch statements: an `if` statement and a `switch` statement.

Control flow in a `switch` statement can be changed by a `break` statement and is discussed in Break Statement below.

> GRAMMAR OF A BRANCH STATEMENT
>
> *branch-statement* → *if-statement*
> *branch-statement* → *switch-statement*

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 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 have a type that conforms to the `LogicValue` protocol. The condition can also be an optional binding declaration, as discussed in Optional Binding.

GRAMMAR OF AN IF STATEMENT

if-statement → `if` *if-condition* *code-block* *else-clause* *opt*
if-condition → *expression* *declaration*
else-clause → `else` *code-block* `else` *if-statement*

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 is very flexible. For instance, 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 Switch in the Control Flow chapter.

A `switch` case can optionally contain a guard expression after each pattern. A *guard expression* is introduced by the keyword `where` 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 guard expression 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 guard expression evaluates to `true`. For instance, 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)`.

```
1  case let (x, y) where x == y:
```

As the above example shows, patterns in a case can also bind constants using the keyword `let` (they can also bind variables using the keyword `var`). These constants (or variables) can then be referenced in a corresponding guard expression and throughout the rest of the code within the scope of the case. That said, if the case contains multiple patterns that match the control expression, none of those patterns can contain constant or variable bindings.

A `switch` statement can also include a default case, introduced by the keyword `default`. 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 instance, when the control expression's type is `Int`), you can include a default case to satisfy the requirement.

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 keyword `fallthrough`, in the case from which you want execution to continue. For more information about the `fallthrough` statement, see Fallthrough Statement below.

> GRAMMAR OF A SWITCH STATEMENT
>
> *switch-statement* → `switch` *expression* `{` *switch-cases* *opt* `}`
> *switch-cases* → *switch-case* *switch-cases* *opt*
> *switch-case* → *case-label* *statements* *default-label* *statements*
> *switch-case* → *case-label* `;` *default-label* `;`
>
> *case-label* → `case` *case-item-list* `:`
> *case-item-list* → *pattern* *guard-clause* *opt* *pattern* *guard-clause* *opt* `,` *case-item-list*
> *default-label* → `default` `:`
>
> *guard-clause* → `where` *guard-expression*
> *guard-expression* → *expression*

Labeled Statement

You can prefix a loop statement or a `switch` 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 Labeled Statements in the Control Flow chapter.

> GRAMMAR OF A LABELED STATEMENT
>
> *labeled-statement* → *statement-label* *loop-statement* | *statement-label* *switch-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 four control transfer statements: a `break` statement, a `continue` statement, a `fallthrough` statement, and a `return` 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*

Break Statement

A `break` statement ends program execution of a loop or a `switch` statement. A `break` statement can consist of only the keyword `break`, or it can consist of the keyword `break` 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 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.

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 Break and Labeled Statements in the Control Flow chapter.

> 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 keyword `continue`, or it can consist of the keyword `continue` 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 Continue and Labeled Statements in the Control Flow chapter.

> GRAMMAR OF A CONTINUE STATEMENT
>
> *continue-statement* → `continue` *label-name* *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 the Control Flow chapter.

> GRAMMAR OF A FALLTHROUGH STATEMENT
>
> *fallthrough-statement* → `fallthrough`

Return Statement

A `return` statement occurs only 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 keyword `return`, or it can consist of the keyword `return` followed by an expression, as shown below.

```
return
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.

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*

Declarations

A *declaration* introduces a new name or construct into your program. For example, you use declarations to introduce functions and methods, variables and constants, and to define new, named enumeration, structure, class, and protocol types. You can also use a declaration to extend the 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 → *function-declaration*
declaration → *enum-declaration*
declaration → *struct-declaration*
declaration → *class-declaration*
declaration → *protocol-declaration*
declaration → *initializer-declaration*
declaration → *deinitializer-declaration*
declaration → *extension-declaration*
declaration → *subscript-declaration*
declaration → *operator-declaration*
declarations → *declaration* *declarations* *opt*

declaration-specifiers → *declaration-specifier* *declaration-specifiers* *opt*
declaration-specifier → `class` `mutating` `nonmutating` `override` `static` `unowned` `unowned(safe)` `unowned(unsafe)` `weak`

Module Scope

The module scope defines the code that's visible to other code in Swift source files that are part of the same module. The top-level code in a Swift source file consists of zero or more statements, declarations, and

expressions. Variables, constants, and other named declarations that are declared at the top-level of a source file are visible to code in every source file that is part of the same module.

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` `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 keyword `let` 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 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:

```
println("The first number is \(firstNumber).")
// prints "The first number is 10."
println("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 Type Inference.

To declare a static constant property, mark the declaration with the `static` keyword. Static properties are discussed in Type Properties.

For more information about constants and for guidance about when to use them, see Constants and Variables and Stored Properties.

GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → *attributes* opt *declaration-specifiers* opt `let` *pattern-initializer-list*

pattern-initializer-list → *pattern-initializer* | *pattern-initializer* `,` *pattern-initializer-list*
pattern-initializer → *pattern* *initializer* opt
initializer → `=` *expression*

Variable Declaration

A *variable declaration* introduces a variable named value into your program and is declared using the keyword `var`.

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 Protocol Property Declaration.

You can override a property in a subclass by prefixing the subclass's property declaration with the `override` keyword, as described in Overriding.

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 required in all variable declarations that include observers, regardless of the context in which they are declared.

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 Property Observers.

Class and Static Variable Properties

To declare a class computed property, mark the declaration with the `class` keyword. To declare a static variable property, mark the declaration with the `static` keyword. Class and static 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* *type-annotation* *initializer* *opt* *willSet-didSet-block*
>
> *variable-declaration-head* → *attributes* *opt* *declaration-specifiers* *opt* `var`
> *variable-name* → *identifier*

```
getter-setter-block → { getter-clause setter-clause opt }
getter-setter-block → { setter-clause getter-clause }
getter-clause → attributes opt get code-block
setter-clause → attributes opt set setter-name opt code-block
setter-name → ( identifier )

getter-setter-keyword-block → { getter-keyword-clause setter-keyword-clause opt }
getter-setter-keyword-block → { setter-keyword-clause getter-keyword-clause }
getter-keyword-clause → attributes opt get
setter-keyword-clause → attributes opt set

willSet-didSet-block → { willSet-clause didSet-clause opt }
willSet-didSet-block → { didSet-clause willSet-clause }
willSet-clause → attributes opt willSet setter-name opt code-block
didSet-clause → attributes opt didSet setter-name opt code-block
```

Type Alias Declaration

A *type alias declaration* introduces a named alias of an existing type into your program. Type alias declarations begin with the keyword `typealias` 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.

See also Protocol Associated Type Declaration.

GRAMMAR OF A TYPE ALIAS DECLARATION

```
typealias-declaration → typealias-head typealias-assignment
typealias-head → typealias typealias-name
typealias-name → identifier
typealias-assignment → = type
```

Function Declaration

A :newTerm`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 keyword `func` 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. By default, the parameters to a function are constants. Write `var` in front of a parameter's name to make it a variable, scoping any changes made to the variable just to the function body, or write `inout` to make those changes also apply to the argument that was passed in the caller's scope. For a discussion of in-out parameters, see In-Out Parameters.

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*. For a discussion of nested functions, see Nested Functions.

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`

For function parameters, the parameter name is used within the function body, but is not used when calling the function. For method parameters, the parameter name is used as within the function body, and is also used as a label for the argument when calling the method. The name of a method's first parameter is used only within the function body, like the parameter of a function. For example:

```
func f(x: Int, y: String) -> String {
    return y + String(x)
}
f(7, "hello")  // x and y have no name

class C {
    func f(x: Int, y: String) -> String {
        return y + String(x)
    }
}
let c = C()
c.f(7, y: "hello")  // x has no name, y has a name
```

You can override the default behavior for how parameter names are used with one of the following forms:

`external parameter name` `local parameter name` : `parameter type`

`parameter name` : `parameter type`

_ `local parameter name` : `parameter type`

A second name before the local parameter name gives the parameter an external name, which can be different than the local parameter name. The external parameter name must be used when the function is called. The corresponding argument must have the external name in function or method calls.

A hash symbol (#) before a parameter name indicates that the name should be used as both an external and a local parameter name. It has the same meaning as writing the local parameter name twice. The corresponding argument must have this name in function or method calls.

An underscore (_) before a local parameter name gives that parameter no name to be used in function calls. The corresponding argument must have no name in function or method calls.

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
```

A parameter named with an underscore (_) is explicitly ignored an 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, which must be its last parameter. 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 Variadic Parameters.

A parameter with an equals sign (=) and an expression after its type is understood to have a default value of the given expression. If the parameter is omitted when calling the function, the default value is used instead. If the parameter is not omitted, it must have its name in the function call. For example, `f()` and `f(x: 7)` are both valid calls to a function with a single default parameter named `x`, but `f(7)` is invalid because it provides a value without a name.

Special Kinds of Methods

Methods on an enumeration or a structure that modify `self` must be marked with the `mutating` keyword at the start of the function declaration.

Methods that override a superclass method must be marked with the `override` keyword at the start of the function declaration. It is an error to override a method without the `override` keyword or to use the `override` keyword 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` attribute for enumerations and structures or the `class` attribute for classes.

Curried Functions and Methods

Curried functions and methods have the following form:

```
func function name(parameters)(parameters) -> return type
    {
    statements
}
```

A function declared this way is understood as a function whose return type is another function. For example, the following two declarations are equivalent:

```
func addTwoNumbers(a: Int)(b: Int) -> Int {
    return a + b
}
func addTwoNumbers(a: Int) -> (Int -> Int) {
    func addTheSecondNumber(b: Int) -> Int {
        return a + b
    }
    return addTheSecondNumber
}

addTwoNumbers(4)(5) // Returns 9
```

Multiple levels of currying are allowed.

GRAMMAR OF A FUNCTION DECLARATION

function-declaration → *function-head* *function-name* *generic-parameter-clause* *opt* *function-signature* *function-body*

function-head → *attributes* *opt* *declaration-specifiers* *opt* `func`
function-name → *identifier* *operator*

function-signature → *parameter-clauses* *function-result* *opt*
function-result → `->` *attributes* *opt* *type*
function-body → *code-block*

parameter-clauses → *parameter-clause* *parameter-clauses* *opt*
parameter-clause → `(` `)` | `(` *parameter-list* `...` *opt* `)`
parameter-list → *parameter* | *parameter* `,` *parameter-list*
parameter → `inout` *opt* `let` *opt* `#` *opt* *parameter-name* *local-parameter-name* *opt* *type-annotation* *default-argument-clause* *opt*
parameter → `inout` *opt* `var` `#` *opt* *parameter-name* *local-parameter-name* *opt* *type-annotation* *default-argument-clause* *opt*
parameter → *attributes* *opt* *type*
parameter-name → *identifier* | `_`
local-parameter-name → *identifier* | `_`
default-argument-clause → `=` *expression*

Enumeration Declaration

An *enumeration declaration* introduces a named enumeration type into your program.

Enumeration declarations have two basic forms and are declared using the keyword `enum`. 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, static methods, initializers, type aliases, and even other enumeration, structure, and class declarations. Enumeration declarations can't contain destructor or protocol declarations.

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 {
    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 keyword `case` 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. For more information and to see examples of cases with associated value types, see Associated Values.

Enumerations with Raw Cases Values

The following form declares an enumeration type that contains enumeration cases of the same basic type:

```
enum enumeration name: raw value type {
```

```
        case enumeration case 1 = raw value 1
        case enumeration case 2 = raw value 2
    }
```

In this form, each case block consists of the keyword `case`, 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 a literal integer, floating-point number, character, or string.

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.

```
enum ExampleEnum: Int {
    case A, B, C = 5, D
}
```

In the above example, the value of `ExampleEnum.A` is `0` and the value of `ExampleEnum.B` is `1`. And because the value of `ExampleEnum.C` is explicitly set to `5`, the value of `ExampleEnum.D` is automatically incremented from `5` and is therefore `6`.

The raw value of an enumeration case can be accessed by calling its `toRaw` method, as in `ExampleEnum.B.toRaw()`. You can also use a raw value to find a corresponding case, if there is one, by calling the `fromRaw` method, 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 `EnumerationType.EnumerationCase`. When the enumeration type can be inferred from context, you can omit it (the dot is still required), as described in Enumeration Syntax and Implicit Member Expression.

To check the values of enumeration cases, use a `switch` statement, as shown in Matching Enumeration Values with a Switch Statement. The enumeration type is pattern-matched against the enumeration case patterns in the case blocks of the `switch` statement, as described in Enumeration Case Pattern.

GRAMMAR OF AN ENUMERATION DECLARATION

enum-declaration → *attributes* opt *union-style-enum* *attributes* opt *raw-value-style-enum*

union-style-enum → *enum-name* *generic-parameter-clause* opt { *union-style-enum-members* opt }
union-style-enum-members → *union-style-enum-member* *union-style-enum-members* opt
union-style-enum-member → *declaration* *union-style-enum-case-clause*
union-style-enum-case-clause → *attributes* opt `case` *union-style-enum-case-list*
union-style-enum-case-list → *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 → *identifier*
enum-case-name → *identifier*

raw-value-style-enum → *enum-name* *generic-parameter-clause* opt : *type-identifier* { *raw-value-style-enum-members* opt }
raw-value-style-enum-members → *raw-value-style-enum-member* *raw-value-style-enum-members* opt
raw-value-style-enum-member → *declaration* *raw-value-style-enum-case-clause*
raw-value-style-enum-case-clause → *attributes* opt `case` *raw-value-style-enum-case-list*
raw-value-style-enum-case-list → *raw-value-style-enum-case* *raw-value-style-enum-case* , *raw-value-style-enum-case-list*
raw-value-style-enum-case → *enum-case-name* *raw-value-assignment* opt
raw-value-assignment → = *literal*

Structure Declaration

A *structure declaration* introduces a named structure type into your program. Structure declarations are declared using the keyword `struct` 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, static properties, instance methods, static methods, initializers, type aliases, and even other structure, class, and enumeration declarations. Structure declarations can't contain destructor or protocol declarations. For a discussion and several examples of structures that include various kinds of declarations, see Classes and Structures.

Structure types can adopt any number of protocols, but can't inherit from classes, enumerations, or other structures.

There are three ways create an instance of a previously declared structure:

- Call one of the initializers declared within the structure, as described in Initializers.
- If no initializers are declared, call the structure's memberwise initializer, as described in Memberwise Initializers for Structure Types.
- 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 Accessing Properties.

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.

GRAMMAR OF A STRUCTURE DECLARATION

struct-declaration → *attributes* *opt* `struct` *struct-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *struct-body*
struct-name → *identifier*
struct-body → `{` *declarations* *opt* `}`

Class Declaration

A *class declaration* introduces a named class type into your program. Class declarations are declared using the keyword `class` 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, class methods, initializers, a single destructor method, 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 Classes and Structures.

A class type can inherit from only one parent class, its *superclass*, but can adopt any number of protocols. The *superclass* appears first in the **type-inheritance-clause**, followed by any *adopted protocols*.

As discussed in Initializer Declaration, classes can have designated and convenience initializers. When you declare either kind of initializer, you can require any subclass to override it by marking the initializer with the `required` attribute. 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, and initializers of its superclass. Overridden methods and properties must be marked with the `override` keyword.

Although properties and methods declared in the *superclass* are inherited by the current class, designated initializers declared in the *superclass* are not. That said, if the current class overrides all of the superclass's designated initializers, it inherits the superclass's convenience initializers. Swift classes do not inherit from a universal base class.

There are two ways 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.

GRAMMAR OF A CLASS DECLARATION

class-declaration → *attributes* *opt* `class` *class-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *class-body*
class-name → *identifier*
class-body → `{` *declarations* *opt* `}`

Protocol Declaration

A *protocol declaration* introduces a named protocol type into your program. Protocol declarations are declared using the keyword `protocol` 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. 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 Protocol Composition Type and Protocol Composition.

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 `optional` attribute to specify that their implementation by a conforming type is optional. The `optional` attribute can be applied 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` attribute 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.

To restrict the adoption of a protocol to class types only, mark the entire protocol declaration with the `class_protocol` attribute. Any protocol that inherits from a protocol marked with the `class_protocol` attribute can likewise be adopted only by a class type.

> NOTE
>
> If a protocol is already marked with the `objc` attribute, the `class_protocol` attribute is implicitly applied to that protocol; there's no need to mark the protocol with the `class_protocol` attribute 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 Protocols as Types. 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 Delegation.

GRAMMAR OF A PROTOCOL DECLARATION

protocol-declaration → *attributes* opt `protocol` *protocol-name* *type-inheritance-clause* opt *protocol-body*
protocol-name → *identifier*
protocol-body → `{` *protocol-member-declarations* opt `}`

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-declarations → *protocol-member-declaration* *protocol-member-declarations* opt

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.

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 `class` keyword. Classes that implement this method also declare the method with the `class` keyword. Structures that implement it must declare the method with the `static` keyword instead. If you're implementing the method in an extension, use the `class` keyword if you're extending a class and the `static` keyword if you're extending a structure.

See also Function Declaration.

> GRAMMAR OF A PROTOCOL METHOD DECLARATION
>
> *protocol-method-declaration* → *function-head* *function-name* *generic-parameter-clause* *opt* *function-signature*

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.

See also Initializer Declaration.

> GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION
>
> *protocol-initializer-declaration* → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause*

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 property 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.

See also Subscript Declaration.

> GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION
>
> *protocol-subscript-declaration* → *subscript-head* *subscript-result* *getter-setter-keyword-block*

Protocol Associated Type Declaration

Protocols declare associated types using the keyword `typealias`. An associated type provides an alias for a type that is used as part of a protocol's declaration. Accosiated types are similiar to type paramters 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 Associated Types.

See also Type Alias Declaration.

GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION

protocol-associated-type-declaration → *typealias-head* *type-inheritance-clause* *opt* *typealias-assignment* *opt*

Initializer Declaration

An *initializer declaration* introduces an initializer for a class, structure, or enumeration into your program. Initializer declarations are declared using the keyword `init` 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 Initialization.

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, prefix the initializer declaration with the context-sensitive keyword `convenience`.

```
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` attribute to require that every subclass implement the initializer. Because designated initializers are not inherited by subclasses, they must be implemented directly. Required convenience initializers can be either implemented explicitly or inherited when the subclass directly implements all of the superclass's designated initializers (or overrides the designated initializers with convenience initializers). Unlike methods, properties, and subscripts, you don't need to mark overridden initializers with the `override` keyword.

To see examples of initializers in various type declarations, see Initialization.

GRAMMAR OF AN INITIALIZER DECLARATION

initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause* *initializer-body*
initializer-head → *attributes* *opt* `convenience` *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.

GRAMMAR OF A DEINITIALIZER DECLARATION

deinitializer-declaration → *attributes* *opt* `deinit` *code-block*

Extension Declaration

An *extension declaration* allows you to extend the behavior of existing class, structure, and enumeration types. Extension declarations begin with the keyword `extension` and have the following form:

```
extension type : adopted protocols {
    declarations
}
```

The body of an extension declaration contains zero or more *declarations*. These *declarations* can include computed properties, computed static properties, instance methods, static and class methods, initializers, subscript declarations, and even class, structure, and enumeration declarations. Extension declarations can't contain destructor or protocol declarations, store properties, property observers, or other extension declarations. For a discussion and several examples of extensions that include various kinds of declarations, see Extensions.

Extension declarations can add protocol conformance to an existing class, structure, and enumeration type in the *adopted protocols*. Extension declarations can't add class inheritance to an existing class, and therefore the **type-inheritance-clause** in an extension declaration contains only a list of protocols.

Properties, methods, and initializers of an existing type can't be overridden in an extension of that type.

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.

> GRAMMAR OF AN EXTENSION DECLARATION
>
> *extension-declaration* → `extension` *type-identifier* *type-inheritance-clause* *opt* *extension-body*
> *extension-body* → `{` *declarations* *opt* `}`

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 keyword `subscript` 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`. That type of the *setter name* must be 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 overridden subscript declaration with the `override` keyword.

You can also declare subscripts in the context of a protocol declaration, as described in Protocol Subscript Declaration.

For more information about subscripting and to see examples of subscript declarations, see Subscripts.

> GRAMMAR OF A SUBSCRIPT DECLARATION
>
> *subscript-declaration* → *subscript-head* *subscript-result* *code-block*
> *subscript-declaration* → *subscript-head* *subscript-result* *getter-setter-block*
> *subscript-declaration* → *subscript-head* *subscript-result* *getter-setter-keyword-block*

subscript-head → *attributes* *opt* `subscript` *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 contextual keyword `operator`.

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 including the contextual keyword `infix`, `prefix`, or `postfix` between `operator` and the name of the operator. 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:

```
operator infix operator name {
    precedence precedence level
    associativity associativity
}
```

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, associativity, or both.

The *precedence* of an operator specifies how tightly an operator binds to its operands in the absence of grouping parentheses. You specify the precedence of an operator by writing the contextual keyword `precedence` followed by the *precedence level*. The *precedence level* can be any whole number (decimal integer) from 0 to 255; unlike decimal integer literals, it can't contain any underscore characters. Although the precedence level is a specific number, it is significant only relative to another operator. That is, when two operators compete with each other for their operands, such as in the expression `2 + 3 * 5`, the operator with the higher precedence

level binds more tightly to its operands.

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 the contextual keyword `associativity` followed by the *associativity*, which is one of the contextual keywords `left`, `right`, or `none`. Operators that are left-associative group left-to-right. For example, the subtraction operator (`−`) is left-associative, and therefore 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, `1 < 2 < 3` is not a valid expression.

Infix operators that are declared without specifying a precedence or associativity are initialized with a precedence level of 100 and an associativity of `none`.

The following form declares a new prefix operator:

```
operator prefix operator name {}
```

A *prefix operator* is a unary operator that is written immediately before its operand, such as the prefix increment operator (`++`) is in the expression `++i`.

Prefix operators declarations don't specify a precedence level. Prefix operators are nonassociative.

The following form declares a new postfix operator:

```
operator postfix operator name {}
```

A *postfix operator* is a unary operator that is written immediately after its operand, such as the postfix increment operator (`++`) is in the expression `i++`.

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 function that has the same name as the

operator. 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 → `operator prefix` *operator* `{ }`
postfix-operator-declaration → `operator postfix` *operator* `{ }`
infix-operator-declaration → `operator infix` *operator* `{` *infix-operator-attributes*$_{opt}$ `}`

infix-operator-attributes → *precedence-clause*$_{opt}$ *associativity-clause*$_{opt}$
precedence-clause → `precedence` *precedence-level*
precedence-level → Digit 0 through 255
associativity-clause → `associativity` *associativity*
associativity → `left right none`

Attributes

Attributes provide more information about a declaration or type. There are two kinds of attributes in Swift, those that apply to declarations and those that apply to types. For instance, the `required` attribute—when applied to a designated or convenience initializer declaration of a class—indicates that every subclass must implement that initializer. And the `noreturn` attribute—when applied to a function or method type—indicates that the function or method doesn't return to its caller.

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. However, you can also apply the `noreturn` attribute to a function or method *type*.

`assignment`
: Apply this attribute to functions that overload a compound assignment operator. Functions that overload a compound assignment operator must mark their initial input parameter as `inout`. For an example of how to use the `assignment` attribute, see Compound Assignment Operators.

`class_protocol`
: Apply this attribute to a protocol to indicate that the protocol can be adopted by class types only.

If you apply the `objc` attribute to a protocol, the `class_protocol` attribute is implicitly applied to that protocol; there's no need to mark the protocol with the `class_protocol` attribute explicitly.

`exported`

Apply this attribute to an import declaration to export the imported module, submodule, or declaration from the current module. If another module imports the current module, that other module can access the items exported by the current module.

`final`

Apply this attribute 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 that class member can't be overridden in any subclass.

`lazy`

Apply this attribute 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` attribute, see Lazy Stored Properties.

`noreturn`

Apply this attribute to a function or method declaration to indicate that the corresponding type of that function or method, `T`, is `@noreturn T`. You can mark a function or method type with this attribute to indicate that the function or method doesn't return to its caller.

You can override a function or method that is not marked with the `noreturn` attribute with a function or method that is. That said, you can't override a function or method that is marked with the `noreturn` attribute with a function or method that is not. Similar rules apply when you implement a protocol method in a conforming type.

`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 `copyWithZone` method—instead of the value of the property itself. The type of the property must conform to the `NSCopying` protocol.

The `NSCopying` attribute behaves in a way similar to the Objective-C `copy` property attribute.

`NSManaged`

Apply this attribute to a stored variable property of a class that inherits from `NSManagedObject` to indicate that the storage and implementation of the property are provided dynamically by Core Data at runtime based on the associated entity description.

`objc`

Apply this attribute to any declaration that can be represented in Objective-C—for example, non-nested classes, protocols, properties and methods (including getters and setters) of classes and protocols, initializers, deinitializers, and subscripts. The `objc` attribute tells the compiler that a declaration is available to use in Objective-C code.

If you apply the `objc` attribute to a class or protocol, it's implicitly applied to the members of that class or protocol. The compiler also implicitly adds the `objc` attribute to a class that inherits from another class marked with the `objc` attribute. Protocols marked with the `objc` attribute can't inherit from protocols that aren't.

The `objc` attribute optionally accepts a single attribute argument, which consists of an identifier. Use this attribute when you want to expose a different name to Objective-C for the entity the `objc` attribute applies to. You can use this argument to name classes, protocols, methods, getters, setters, and initializers. 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.

```
@objc
class ExampleClass {
    var enabled: Bool {
    @objc(isEnabled) get {
        // Return the appropriate value
    }
    }
}
```

`optional`

Apply this attribute 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` attribute 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` attribute 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 attribute to a designated or convenience initializer of a class to indicate that every subclass must implement that initializer.

Required designated initializers must be implemented explicitly. Required convenience initializers can be either implemented explicitly or inherited when the subclass directly implements all of the superclass's designated initializers (or when the subclass overrides the designated initializers with convenience initializers).

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`, `IBDesignable`, `IBInspectable`, and `IBOutlet`. 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` attribute to method declarations of a class and the `IBDesignable` attribute to class declarations.

Type Attributes

You can apply type attributes to types only. However, you can also apply the `noreturn` attribute to a function or method *declaration*.

`auto_closure`

This attribute is used to delay the evaluation of an expression by automatically wrapping that expression in a closure with no arguments. Apply this attribute to a function or method type that takes no arguments and that returns the type of the expression. For an example of how to use the `auto_closure` attribute, see Function Type.

`noreturn`

Apply this attribute to the type of a function or method to indicate that the function or method doesn't return to its caller. You can also mark a function or method declaration with this attribute to indicate that the corresponding type of that function or method, `T`, is `@noreturn T`.

GRAMMAR OF AN ATTRIBUTE

attribute → `@` *attribute-name* *attribute-argument-clause* *opt*
attribute-name → *identifier*
attribute-argument-clause → `(` *balanced-tokens* *opt* `)`
attributes → *attribute* *attributes* *opt*

balanced-tokens → *balanced-token* *balanced-tokens* *opt*
balanced-token → `(` *balanced-tokens* *opt* `)`
balanced-token → `[` *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 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, patterns occur in variable and constant declarations (on their left-hand side), in `for`-`in` statements, and in `switch` statements (in their case labels). Although any pattern can occur in the case labels of a `switch` statement, in the other contexts, only wildcard patterns, identifier patterns, and patterns containing those two patterns can occur.

You can specify a type annotation for a wildcard pattern, an identifier pattern, and a tuple pattern to constraint the pattern to match only values of a certain type.

> GRAMMAR OF A PATTERN
>
> *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* → *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 keyword `let`; those that bind to the name of variable begin with the keyword `var`.

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.

```
let point = (3, 2)
switch point {
    // Bind x and y to the elements of point.
case let (x, y):
    println("The point is at (\(x), \(y)).")
}
// 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`. To constrain only some elements of a tuple pattern, provide type annotations directly to those individual elements. For example, the tuple pattern in `let (x: String, y)` matches any two-element tuple type, as long as the first element is of type `String`.

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, 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:

```
let points = [(0, 0), (1, 0), (1, 1), (2, 0), (2, 1)]
// This code isn't valid.
for (x, 0) in points {
```

```
    /* ... */
}
```

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:

```
let a = 2         // a: Int = 2
let (a) = 2       // a: Int = 2
let (a): Int = 2 // a: Int = 2
```

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 → *pattern*

Enumeration Case Pattern

An *enumeration case pattern* matches a case of an existing enumeration type. Enumeration case patterns appear only in `switch` statement case labels.

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.

GRAMMAR OF AN ENUMERATION CASE PATTERN

enum-case-pattern → *type-identifier* *opt* . *enum-case-name* *tuple-pattern* *opt*

Type-Casting Patterns

There are two type-casting patterns, the `is` pattern and the `as` pattern. Both type-casting patterns appear 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 left-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-casting-pattern* → *is-pattern* | *as-pattern*
> *is-pattern* → `is` *type*
> *as-pattern* → *pattern* `as` *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 an integer value with a range of integers in an `Range` object, as the following example shows:

```
let point = (1, 2)
switch point {
```

```
case (0, 0):
    println("(0, 0) is at the origin.")
case (-2...2, -2...2):
    println("(\(point.0), \(point.1)) is near the origin.")
default:
    println("The point is at (\(point.0), \(point.1)).")
}
// prints "(1, 2) is near the origin."
```

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
func ~=(pattern: String, value: Int) -> Bool {
    return pattern == "\(value)"
}
switch point {
case ("0", "0"):
    println("(0, 0) is at the origin.")
case ("-2...2", "-2...2"):
    println("(\(point.0), \(point.1)) is near the origin.")
default:
    println("The point is at (\(point.0), \(point.1)).")
}
// prints "(1, 2) is near the origin."
```

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, 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 one of the following forms:

```
< generic parameter list >
< generic parameter list where requirements >
```

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 instance, `T`, `U`, `V`, `KeyType`, `ValueType`, 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 instance, in the generic function below, the generic parameter `T: Comparable`

indicates that any type argument substituted for the type parameter `T` must conform to the `Comparable` protocol.

```
func simpleMin<T: Comparable>(x: T, y: T) -> T {
    if x < y {
        return y
    }
    return x
}
```

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.

```
simpleMin(17, 42) // T is inferred to be Int
simpleMin(3.14159, 2.71828) // T is inferred to be Double
```

Where Clauses

You can specify additional requirements on type parameters and their associated types by including a `where` clause after the *generic parameter list*. A `where` clause consists of the keyword `where`, followed by a comma-separated list of one or more *requirements*.

The *requirements* in a `where` clause specify that a type parameter inherits from a class or conforms to a protocol or protocol composition. Although the `where` clause provides syntactic sugar for expressing simple constraints on type parameters (for instance, `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 instance, you can express the constraints that a generic type `T` inherits from a class `C` and conforms to a protocol `P` as `<T where T: C, T: P>`.

As mentioned above, you can constrain the associated types of type parameters to conform to protocols. For example, the generic parameter clause `<T: Generator where T.Element: Equatable>` specifies

that `T` conforms to the `Generator` protocol and the associated type of `T`, `T.Element`, conforms to the `Equatable` protocol (`T` has the associated type `Element` because `Generator` declares `Element` and `T` conforms to `Generator`).

You can also specify the requirement that two types be identical, using the `==` operator. For example, the generic parameter clause `<T: Generator, U: Generator where T.Element == U.Element>` expresses the constraints that `T` and `U` conform to the `Generator` protocol and that their associated types must be identical.

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 in the generic parameter clause. When you call an overloaded generic function or initializer, the compiler uses these constraints to resolve which overloaded function or initializer to invoke.

You can subclass a generic class, but the subclass must also be a generic class.

> GRAMMAR OF A GENERIC PARAMETER CLAUSE
>
> *generic-parameter-clause* → < *generic-parameter-list* *requirement-clause*$_{opt}$ >
> *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*
>
> *requirement-clause* → where *requirement-list*
> *requirement-list* → *requirement* | *requirement* , *requirement-list*
> *requirement* → *conformance-requirement* | *same-type-requirement*
>
> *conformance-requirement* → *type-identifier* : *type-identifier*
> *conformance-requirement* → *type-identifier* : *protocol-composition-type*
> *same-type-requirement* → *type-identifier* == *type-identifier*

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. As an example, the Swift standard library defines a generic dictionary type as:

```
struct Dictionary<KeyType: Hashable, ValueType>: Collection,
                DictionaryLiteralConvertible {
    /* ... */
}
```

The specialized version of the generic `Dictionary` type, `Dictionary<String, Int>` is formed by replacing the generic parameters `KeyType: Hashable` and `ValueType` 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 `where` clause. In the example above, the `KeyType` 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 `T` in `Array<T>` with a specialized version of an array, `Array<Int>`, to form an array whose elements are themselves arrays of integers.

```
let arrayOfArrays: Array<Array<Int>> = [[1, 2, 3], [4, 5, 6], [7, 8,
                9]]
```

As mentioned in Generic Parameter Clause, 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* → < *generic-argument-list* >
> *generic-argument-list* → *generic-argument* *generic-argument* , *generic-argument-list*
> *generic-argument* → *type*

Summary of the Grammar

Statements

GRAMMAR OF A STATEMENT

statement → *expression* `;` *opt*
statement → *declaration* `;` *opt*
statement → *loop-statement* `;` *opt*
statement → *branch-statement* `;` *opt*
statement → *labeled-statement*
statement → *control-transfer-statement* `;` *opt*
statements → *statement* *statements* *opt*

GRAMMAR OF A LOOP STATEMENT

loop-statement → *for-statement*
loop-statement → *for-in-statement*
loop-statement → *while-statement*
loop-statement → *do-while-statement*

GRAMMAR OF A FOR STATEMENT

for-statement → `for` *for-init* *opt* `;` *expression* *opt* `;` *expression* *opt* *code-block*
for-statement → `for` `(` *for-init* *opt* `;` *expression* *opt* `;` *expression* *opt* `)` *code-block*

for-init → *variable-declaration* *expression-list*

GRAMMAR OF A FOR-IN STATEMENT

for-in-statement → `for` *pattern* `in` *expression* *code-block*

GRAMMAR OF A WHILE STATEMENT

while-statement → `while` *while-condition* *code-block*
while-condition → *expression* *declaration*

GRAMMAR OF A DO-WHILE STATEMENT

do-while-statement → `do` *code-block* `while` *while-condition*

GRAMMAR OF A BRANCH STATEMENT

branch-statement → *if-statement*
branch-statement → *switch-statement*

GRAMMAR OF AN IF STATEMENT

if-statement → `if` *if-condition* *code-block* *else-clause* *opt*
if-condition → *expression* *declaration*
else-clause → `else` *code-block* `else` *if-statement*

GRAMMAR OF A SWITCH STATEMENT

switch-statement → `switch` *expression* `{` *switch-cases* *opt* `}`
switch-cases → *switch-case* *switch-cases* *opt*
switch-case → *case-label* *statements* *default-label* *statements*
switch-case → *case-label* `;` *default-label* `;`

case-label → `case` *case-item-list* `:`
case-item-list → *pattern* *guard-clause* *opt* *pattern* *guard-clause* *opt* `,` *case-item-list*
default-label → `default` `:`

guard-clause → `where` *guard-expression*
guard-expression → *expression*

GRAMMAR OF A LABELED STATEMENT

labeled-statement → *statement-label* *loop-statement* *statement-label* *switch-statement*
statement-label → *label-name* `:`
label-name → *identifier*

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*

GRAMMAR OF A BREAK STATEMENT

break-statement → `break` *label-name* *opt*

GRAMMAR OF A CONTINUE STATEMENT

continue-statement → `continue` *label-name* *opt*

GRAMMAR OF A FALLTHROUGH STATEMENT

fallthrough-statement → `fallthrough`

GRAMMAR OF A RETURN STATEMENT

return-statement → `return` *expression* *opt*

Generic Parameters and Arguments

GRAMMAR OF A GENERIC PARAMETER CLAUSE

generic-parameter-clause → `<` *generic-parameter-list* *requirement-clause* *opt* `>`
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*

requirement-clause → `where` *requirement-list*
requirement-list → *requirement* | *requirement* `,` *requirement-list*
requirement → *conformance-requirement* | *same-type-requirement*

conformance-requirement → *type-identifier* `:` *type-identifier*
conformance-requirement → *type-identifier* `:` *protocol-composition-type*
same-type-requirement → *type-identifier* `==` *type-identifier*

GRAMMAR OF A GENERIC ARGUMENT CLAUSE

generic-argument-clause → `<` *generic-argument-list* `>`
generic-argument-list → *generic-argument* | *generic-argument* `,` *generic-argument-list*
generic-argument → *type*

Declarations

GRAMMAR OF A DECLARATION

declaration → *import-declaration*
declaration → *constant-declaration*
declaration → *variable-declaration*
declaration → *typealias-declaration*
declaration → *function-declaration*
declaration → *enum-declaration*
declaration → *struct-declaration*
declaration → *class-declaration*
declaration → *protocol-declaration*
declaration → *initializer-declaration*
declaration → *deinitializer-declaration*
declaration → *extension-declaration*
declaration → *subscript-declaration*
declaration → *operator-declaration*
declarations → *declaration* *declarations* *opt*

declaration-specifiers → *declaration-specifier* *declaration-specifiers* *opt*
declaration-specifier → `class` `mutating` `nonmutating` `override` `static` `unowned` `unowned(safe)` `unowned(unsafe)` `weak`

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 → *attributes* *opt* `import` *import-kind* *opt* *import-path*

import-kind → `typealias` `struct` `class` `enum` `protocol` `var` `func`
import-path → *import-path-identifier* *import-path-identifier* `.` *import-path*
import-path-identifier → *identifier* *operator*

GRAMMAR OF A CONSTANT DECLARATION

constant-declaration → *attributes* *opt* *declaration-specifiers* *opt* `let` *pattern-initializer-list*

pattern-initializer-list → *pattern-initializer* *pattern-initializer* `,` *pattern-initializer-list*
pattern-initializer → *pattern* *initializer* *opt*
initializer → `=` *expression*

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* *type-annotation* *initializer* *opt* *willSet-didSet-block*

variable-declaration-head → *attributes* *opt* *declaration-specifiers* *opt* `var`
variable-name → *identifier*

getter-setter-block → `{` *getter-clause* *setter-clause* *opt* `}`
getter-setter-block → `{` *setter-clause* *getter-clause* `}`
getter-clause → *attributes* *opt* `get` *code-block*
setter-clause → *attributes* *opt* `set` *setter-name* *opt* *code-block*
setter-name → `(` *identifier* `)`

getter-setter-keyword-block → `{` *getter-keyword-clause* *setter-keyword-clause* *opt* `}`
getter-setter-keyword-block → `{` *setter-keyword-clause* *getter-keyword-clause* `}`
getter-keyword-clause → *attributes* *opt* `get`
setter-keyword-clause → *attributes* *opt* `set`

willSet-didSet-block → `{` *willSet-clause* *didSet-clause* *opt* `}`
willSet-didSet-block → `{` *didSet-clause* *willSet-clause* `}`
willSet-clause → *attributes* *opt* `willSet` *setter-name* *opt* *code-block*
didSet-clause → *attributes* *opt* `didSet` *setter-name* *opt* *code-block*

GRAMMAR OF A TYPE ALIAS DECLARATION

typealias-declaration → *typealias-head* *typealias-assignment*
typealias-head → `typealias` *typealias-name*
typealias-name → *identifier*
typealias-assignment → `=` *type*

GRAMMAR OF A FUNCTION DECLARATION

function-declaration → *function-head* *function-name* *generic-parameter-clause* *opt* *function-signature* *function-body*

function-head → *attributes* *opt* *declaration-specifiers* *opt* `func`
function-name → *identifier* *operator*

function-signature → *parameter-clauses* *function-result* *opt*
function-result → `->` *attributes* *opt* *type*

function-body → *code-block*

parameter-clauses → *parameter-clause* *parameter-clauses* *opt*
parameter-clause → ( ) (*parameter-list* ... *opt*)
parameter-list → *parameter* *parameter* , *parameter-list*
parameter → inout *opt* let *opt* # *opt* *parameter-name* *local-parameter-name* *opt* *type-annotation* *default-argument-clause* *opt*
parameter → inout *opt* var # *opt* *parameter-name* *local-parameter-name* *opt* *type-annotation* *default-argument-clause* *opt*
parameter → *attributes* *opt* *type*
parameter-name → *identifier* _
local-parameter-name → *identifier* _
default-argument-clause → = *expression*

GRAMMAR OF AN ENUMERATION DECLARATION

enum-declaration → *attributes* *opt* *union-style-enum* *attributes* *opt* *raw-value-style-enum*

union-style-enum → *enum-name* *generic-parameter-clause* *opt* { *union-style-enum-members* *opt* }
union-style-enum-members → *union-style-enum-member* *union-style-enum-members* *opt*
union-style-enum-member → *declaration* *union-style-enum-case-clause*
union-style-enum-case-clause → *attributes* *opt* case *union-style-enum-case-list*
union-style-enum-case-list → *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 → *identifier*
enum-case-name → *identifier*

raw-value-style-enum → *enum-name* *generic-parameter-clause* *opt* : *type-identifier* { *raw-value-style-enum-members* *opt* }
raw-value-style-enum-members → *raw-value-style-enum-member* *raw-value-style-enum-members* *opt*
raw-value-style-enum-member → *declaration* *raw-value-style-enum-case-clause*
raw-value-style-enum-case-clause → *attributes* *opt* case *raw-value-style-enum-case-list*
raw-value-style-enum-case-list → *raw-value-style-enum-case* *raw-value-style-enum-case* , *raw-value-style-enum-case-list*
raw-value-style-enum-case → *enum-case-name* *raw-value-assignment* *opt*
raw-value-assignment → = *literal*

GRAMMAR OF A STRUCTURE DECLARATION

struct-declaration → *attributes* *opt* struct *struct-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *struct-body*
struct-name → *identifier*
struct-body → { *declarations* *opt* }

GRAMMAR OF A CLASS DECLARATION

class-declaration → *attributes* *opt* `class` *class-name* *generic-parameter-clause* *opt* *type-inheritance-clause* *opt* *class-body*
class-name → *identifier*
class-body → `{` *declarations* *opt* `}`

GRAMMAR OF A PROTOCOL DECLARATION

protocol-declaration → *attributes* *opt* `protocol` *protocol-name* *type-inheritance-clause* *opt* *protocol-body*
protocol-name → *identifier*
protocol-body → `{` *protocol-member-declarations* *opt* `}`

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-declarations → *protocol-member-declaration* *protocol-member-declarations* *opt*

GRAMMAR OF A PROTOCOL PROPERTY DECLARATION

protocol-property-declaration → *variable-declaration-head* *variable-name* *type-annotation* *getter-setter-keyword-block*

GRAMMAR OF A PROTOCOL METHOD DECLARATION

protocol-method-declaration → *function-head* *function-name* *generic-parameter-clause* *opt* *function-signature*

GRAMMAR OF A PROTOCOL INITIALIZER DECLARATION

protocol-initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause*

GRAMMAR OF A PROTOCOL SUBSCRIPT DECLARATION

protocol-subscript-declaration → *subscript-head* *subscript-result* *getter-setter-keyword-block*

GRAMMAR OF A PROTOCOL ASSOCIATED TYPE DECLARATION

protocol-associated-type-declaration → *typealias-head* *type-inheritance-clause* *opt* *typealias-assignment* *opt*

GRAMMAR OF AN INITIALIZER DECLARATION

initializer-declaration → *initializer-head* *generic-parameter-clause* *opt* *parameter-clause* *initializer-body*
initializer-head → *attributes* *opt* `convenience` *opt* `init`
initializer-body → *code-block*

GRAMMAR OF A DEINITIALIZER DECLARATION

deinitializer-declaration → *attributes* *opt* `deinit` *code-block*

GRAMMAR OF AN EXTENSION DECLARATION

extension-declaration → `extension` *type-identifier* *type-inheritance-clause* *opt* *extension-body*
extension-body → `{` *declarations* *opt* `}`

GRAMMAR OF A SUBSCRIPT DECLARATION

subscript-declaration → *subscript-head* *subscript-result* *code-block*
subscript-declaration → *subscript-head* *subscript-result* *getter-setter-block*
subscript-declaration → *subscript-head* *subscript-result* *getter-setter-keyword-block*
subscript-head → *attributes* *opt* `subscript` *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 → `operator prefix` *operator* `{ }`
postfix-operator-declaration → `operator postfix` *operator* `{ }`
infix-operator-declaration → `operator infix` *operator* `{` *infix-operator-attributes* *opt* `}`

infix-operator-attributes → *precedence-clause* *opt* *associativity-clause* *opt*
precedence-clause → `precedence` *precedence-level*
precedence-level → Digit 0 through 255
associativity-clause → `associativity` *associativity*
associativity → `left right none`

Patterns

GRAMMAR OF A PATTERN

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 → *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` *pattern* `let` *pattern*

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 → *pattern*

GRAMMAR OF AN ENUMERATION CASE PATTERN

enum-case-pattern → *type-identifier* *opt* `.` *enum-case-name* *tuple-pattern* *opt*

GRAMMAR OF A TYPE CASTING PATTERN

type-casting-pattern → *is-pattern* *as-pattern*
is-pattern → `is` *type*
as-pattern → *pattern* `as` *type*

GRAMMAR OF AN EXPRESSION PATTERN

expression-pattern → *expression*

Attributes

GRAMMAR OF AN ATTRIBUTE

attribute → @ *attribute-name* *attribute-argument-clause* *opt*
attribute-name → *identifier*
attribute-argument-clause → (*balanced-tokens* *opt*)
attributes → *attribute* *attributes* *opt*

balanced-tokens → *balanced-token* *balanced-tokens* *opt*
balanced-token → (*balanced-tokens* *opt*)
balanced-token → [*balanced-tokens* *opt*]
balanced-token → { *balanced-tokens* *opt* }
balanced-token → Any identifier, keyword, literal, or operator
balanced-token → Any punctuation except (,) , [,] , { , or }

Expressions

GRAMMAR OF AN EXPRESSION

expression → *prefix-expression* *binary-expressions* *opt*
expression-list → *expression* *expression* , *expression-list*

GRAMMAR OF A PREFIX EXPRESSION

prefix-expression → *prefix-operator* *opt* *postfix-expression*
prefix-expression → *in-out-expression*
in-out-expression → & *identifier*

GRAMMAR OF A BINARY EXPRESSION

binary-expression → *binary-operator* *prefix-expression*
binary-expression → *assignment-operator* *prefix-expression*
binary-expression → *conditional-operator* *prefix-expression*
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 → `?` *expression* `:`

GRAMMAR OF A TYPE-CASTING OPERATOR

type-casting-operator → `is` *type* `as` `?` *opt* *type*

GRAMMAR OF A PRIMARY EXPRESSION

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 → *implicit-member-expression*
primary-expression → *wildcard-expression*

GRAMMAR OF A LITERAL EXPRESSION

literal-expression → *literal*
literal-expression → *array-literal* *dictionary-literal*
literal-expression → `__FILE__` `__LINE__` `__COLUMN__` `__FUNCTION__`

array-literal → `[` *array-literal-items* *opt* `]`
array-literal-items → *array-literal-item* `,` *opt* *array-literal-item* `,` *array-literal-items*
array-literal-item → *expression*

dictionary-literal → `[` *dictionary-literal-items* `]` `[` `:` `]`
dictionary-literal-items → *dictionary-literal-item* `,` *opt* *dictionary-literal-item* `,` *dictionary-literal-items*
dictionary-literal-item → *expression* `:` *expression*

GRAMMAR OF A SELF EXPRESSION

self-expression → `self`
self-expression → `self` `.` *identifier*
self-expression → `self` `[` *expression* `]`
self-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` `[` *expression* `]`
superclass-initializer-expression → `super` `.` `init`

GRAMMAR OF A CLOSURE EXPRESSION

closure-expression → `{` *closure-signature*$_{opt}$ *statements* `}`

closure-signature → *parameter-clause* *function-result*$_{opt}$ `in`
closure-signature → *identifier-list* *function-result*$_{opt}$ `in`
closure-signature → *capture-list* *parameter-clause* *function-result*$_{opt}$ `in`
closure-signature → *capture-list* *identifier-list* *function-result*$_{opt}$ `in`
closure-signature → *capture-list* `in`

capture-list → `[` *capture-specifier* *expression* `]`
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 → `(` *expression-element-list*$_{opt}$ `)`
expression-element-list → *expression-element* *expression-element* `,` *expression-element-list*
expression-element → *expression* *identifier* `:` *expression*

GRAMMAR OF A WILDCARD EXPRESSION

wildcard-expression → `_`

GRAMMAR OF A POSTFIX EXPRESSION

postfix-expression → *primary-expression*
postfix-expression → *postfix-expression* *postfix-operator*
postfix-expression → *function-call-expression*
postfix-expression → *initializer-expression*
postfix-expression → *explicit-member-expression*
postfix-expression → *postfix-self-expression*
postfix-expression → *dynamic-type-expression*
postfix-expression → *subscript-expression*
postfix-expression → *forced-value-expression*
postfix-expression → *optional-chaining-expression*

GRAMMAR OF A FUNCTION CALL EXPRESSION

function-call-expression → *postfix-expression* *parenthesized-expression*
function-call-expression → *postfix-expression* *parenthesized-expression* *opt* *trailing-closure*
trailing-closure → *closure-expression*

GRAMMAR OF AN INITIALIZER EXPRESSION

initializer-expression → *postfix-expression* `.` `init`

GRAMMAR OF AN EXPLICIT MEMBER EXPRESSION

explicit-member-expression → *postfix-expression* `.` *decimal-digit*
explicit-member-expression → *postfix-expression* `.` *identifier* *generic-argument-clause* *opt*

GRAMMAR OF A SELF EXPRESSION

postfix-self-expression → *postfix-expression* `.` `self`

GRAMMAR OF A DYNAMIC TYPE EXPRESSION

dynamic-type-expression → *postfix-expression* `.` `dynamicType`

GRAMMAR OF A SUBSCRIPT EXPRESSION

subscript-expression → *postfix-expression* `[` *expression-list* `]`

GRAMMAR OF A FORCED-VALUE EXPRESSION

forced-value-expression → *postfix-expression* `!`

GRAMMAR OF AN OPTIONAL-CHAINING EXPRESSION

optional-chaining-expression → *postfix-expression* `?`

Lexical Structure

GRAMMAR OF AN IDENTIFIER

identifier → *identifier-head* *identifier-characters* *opt*
identifier → ` *identifier-head* *identifier-characters* *opt* `
identifier → *implicit-parameter-name*

identifier-list → *identifier* *identifier* , *identifier-list*

identifier-head → Upper- or lowercase letter A through Z
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+CFFFD
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*

GRAMMAR OF A LITERAL

literal → *integer-literal* *floating-point-literal* *string-literal*

GRAMMAR OF AN INTEGER LITERAL

integer-literal → *binary-literal*
integer-literal → *octal-literal*
integer-literal → *decimal-literal*
integer-literal → *hexadecimal-literal*

binary-literal → `0b` *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* *opt*

octal-literal → `0o` *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 → *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*

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-literal* *opt*
hexadecimal-exponent → *floating-point-p* *sign* *opt* *hexadecimal-literal*

floating-point-e → `e` `E`
floating-point-p → `p` `P`
sign → `+` `-`

GRAMMAR OF A STRING LITERAL

string-literal → `"` *quoted-text* `"`
quoted-text → *quoted-text-item* *quoted-text* *opt*
quoted-text-item → *escaped-character*
quoted-text-item → `\(` *expression* `)`
quoted-text-item → Any Unicode extended grapheme cluster except `"`, `\`, U+000A, or U+000D

escaped-character → `\0` `\\` `\t` `\n` `\r` `\"` `\'`
escaped-character → `\x` *hexadecimal-digit* *hexadecimal-digit*
escaped-character → `\u` *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit*
escaped-character → `\U` *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit* *hexadecimal-digit*

GRAMMAR OF OPERATORS

operator → *operator-character* *operator* *opt*
operator-character → `/` `=` `-` `+` `!` `*` `%` `<` `>` `&` `|` `^` `~` `.`

binary-operator → *operator*
prefix-operator → *operator*
postfix-operator → *operator*

Types

GRAMMAR OF A TYPE

type → *array-type* *function-type* *type-identifier* *tuple-type* *optional-type* *implicitly-unwrapped-optional-type* *protocol-composition-type* *metatype-type*

GRAMMAR OF A TYPE ANNOTATION

type-annotation → `:` *attributes* *opt* *type*

GRAMMAR OF A TYPE IDENTIFIER

type-identifier → *type-name* *generic-argument-clause* *opt* *type-name* *generic-argument-clause* *opt* `.` *type-identifier*
type-name → *identifier*

GRAMMAR OF A TUPLE TYPE

tuple-type → `(` *tuple-type-body* *opt* `)`
tuple-type-body → *tuple-type-element-list* `...` *opt*
tuple-type-element-list → *tuple-type-element* *tuple-type-element* `,` *tuple-type-element-list*
tuple-type-element → *attributes* *opt* `inout` *opt* *type* `inout` *opt* *element-name* *type-annotation*
element-name → *identifier*

GRAMMAR OF A FUNCTION TYPE

function-type → *type* `->` *type*

GRAMMAR OF AN ARRAY TYPE

array-type → *type* `[` `]` *array-type* `[` `]`

GRAMMAR OF AN OPTIONAL TYPE

optional-type → *type* `?`

GRAMMAR OF AN IMPLICITLY UNWRAPPED OPTIONAL TYPE

implicitly-unwrapped-optional-type → *type* `!`

GRAMMAR OF A PROTOCOL COMPOSITION TYPE

protocol-composition-type → `protocol` `<` *protocol-identifier-list*$_{opt}$ `>`
protocol-identifier-list → *protocol-identifier* *protocol-identifier* `,` *protocol-identifier-list*
protocol-identifier → *type-identifier*

GRAMMAR OF A METATYPE TYPE

metatype-type → *type* `.` `Type` *type* `.` `Protocol`

GRAMMAR OF A TYPE INHERITANCE CLAUSE

type-inheritance-clause → `:` *type-inheritance-list*
type-inheritance-list → *type-identifier* *type-identifier* `,` *type-inheritance-list*

Copyright and Notices

> IMPORTANT
>
> This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future seeds of the API or technology.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010