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
```

Escaping Closures

A closure is said to *escape* a function when the closure is passed as an argument to the function, but is called after the function returns. When you declare a function that takes a closure as one of its parameters, you can write `@escaping` before the parameter's type to indicate that the closure is allowed to escape.

One way that a closure can escape is by being stored in a variable that is defined outside the function. As an example, many functions that start an asynchronous operation take a closure argument as a completion handler. The function returns after it starts the operation, but the closure isn't called until the operation is completed—the closure needs to escape, to be called later. For example:

```
var completionHandlers: [() -> Void] = []
func someFunctionWithEscapingClosure(completionHandler: @escaping () ->
  Void) {
    completionHandlers.append(completionHandler)
}
```

The `someFunctionWithEscapingClosure(_:)` function takes a closure as its argument and adds it to an array that's declared outside the function. If you didn't mark the parameter of this function with `@escaping`, you would get a compile-time error.

Marking a closure with `@escaping` means you have to refer to `self` explicitly within the closure. For example, in the code below, the closure passed to `someFunctionWithEscapingClosure(_:)` is an escaping closure, which means it needs to refer to `self` explicitly. In contrast, the closure passed to `someFunctionWithNonescapingClosure(_:)` is a nonescaping closure, which means it can refer to `self` implicitly.

The Basics

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

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

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

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

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

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

Constants and Variables

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

Declaring Constants and Variables

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

```
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 won't 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"
```

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

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

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

Naming Constants and Variables

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

> NOTE
>
> As an optimization, Swift may instead capture and store a *copy* of a value if that value is not mutated by a closure, and if the value is not mutated after the closure is created.
>
> Swift also handles all memory management involved in disposing of variables when they are no longer needed.

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

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

This example sets a constant called `incrementByTen` to refer to an incrementer function that adds `10` to its `runningTotal` variable each time it is called. Calling the function multiple times shows this behavior in action:

```
incrementByTen()
// returns a value of 10
incrementByTen()
// returns a value of 20
incrementByTen()
// returns a value of 30
```

If you create a second incrementer, it will have its own stored reference to a new, separate `runningTotal` variable:

```
let incrementBySeven = makeIncrementer(forIncrement: 7)
incrementBySeven()
// returns a value of 7
```

Calling the original incrementer (`incrementByTen`) again continues to increment its own `runningTotal` variable, and does not affect the variable captured by `incrementBySeven`:

```
incrementByTen()
// returns a value of 40
```

> NOTE
>
> If you assign a closure to a property of a class instance, and the closure captures that instance by referring to the instance or its members, you will create a strong reference cycle between the closure and the instance. Swift uses *capture lists* to break these strong reference cycles. For more information, see Strong Reference Cycles for Closures.

Closures Are Reference Types

In the example above, `incrementBySeven` and `incrementByTen` are constants, but the closures these constants refer to are still able to increment the `runningTotal` variables that they have captured. This is because functions and closures are *reference types*.

A closure can *capture* constants and variables from the surrounding context in which it is defined. The closure can then refer to and modify the values of those constants and variables from within its body, even if the original scope that defined the constants and variables no longer exists.

In Swift, the simplest form of a closure that can capture values is a nested function, written within the body of another function. A nested function can capture any of its outer function's arguments and can also capture any constants and variables defined within the outer function.

Here's an example of a function called `makeIncrementer`, which contains a nested function called `incrementer`. The nested `incrementer()` function captures two values, `runningTotal` and `amount`, from its surrounding context. After capturing these values, `incrementer` is returned by `makeIncrementer` as a closure that increments `runningTotal` by `amount` each time it is called.

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

The return type of `makeIncrementer` is `() -> Int`. This means that it returns a *function*, rather than a simple value. The function it returns has no parameters, and returns an `Int` value each time it is called. To learn how functions can return other functions, see Function Types as Return Types.

The `makeIncrementer(forIncrement:)` function defines an integer variable called `runningTotal`, to store the current running total of the incrementer that will be returned. This variable is initialized with a value of `0`.

The `makeIncrementer(forIncrement:)` function has a single `Int` parameter with an argument label of `forIncrement`, and a parameter name of `amount`. The argument value passed to this parameter specifies how much `runningTotal` should be incremented by each time the returned incrementer function is called. The `makeIncrementer` function defines a nested function called `incrementer`, which performs the actual incrementing. This function simply adds `amount` to `runningTotal`, and returns the result.

When considered in isolation, the nested `incrementer()` function might seem unusual:

```
func incrementer() -> Int {
    runningTotal += amount
    return runningTotal
}
```

The `incrementer()` function doesn't have any parameters, and yet it refers to `runningTotal` and `amount` from within its function body. It does this by capturing a *reference* to `runningTotal` and `amount` from the surrounding function and using them within its own function body. Capturing by reference ensures that `runningTotal` and `amount` do not disappear when the call to `makeIncrementer` ends, and also ensures that `runningTotal` is available the next time the `incrementer` function is called.


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

Constant and variable names can't contain whitespace characters, 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 declare it again with the same name, or change it to store values of a different type. Nor can you change a constant into a variable or a variable into a constant.

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

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

```
var friendlyWelcome = "Hello!"
friendlyWelcome = "Bonjour!"
// friendlyWelcome is now "Bonjour!"
```

Unlike a variable, the value of a constant can't be changed after it's set. Attempting to do so is reported as an error when your code is compiled:

```
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 `print(_:separator:terminator:)` function:

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

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

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

```swift
print("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 nonexecutable text in your code, as a note or reminder to yourself. Comments are ignored by the Swift compiler when your code is compiled.

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

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

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

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

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

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

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

Semicolons

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

```swift
let cat = "🐱"; print(cat)
// Prints "🐱"
```

Integers

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:

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

The `map(_:)` method calls the closure expression once for each item in the array. You do not need to specify the type of the closure's input parameter, `number`, because the type can be inferred from the values in the array to be mapped.

In this example, the variable `number` is initialized with the value of the closure's `number` parameter, so that the value can be modified within the closure body. (The parameters to functions and closures are always constants.) The closure expression also specifies a return type of `String`, to indicate the type that will be stored in the mapped output array.

The closure expression builds a string called `output` each time it is called. It calculates the last digit of `number` by using the remainder operator (`number % 10`), and uses this digit to look up an appropriate string in the `digitNames` dictionary. The closure can be used to create a string representation of any integer greater than zero.

> NOTE
>
> The call to the `digitNames` dictionary's subscript is followed by an exclamation mark (`!`), because dictionary subscripts return an optional value to indicate that the dictionary lookup can fail if the key does not exist. In the example above, it is guaranteed that `number % 10` will always be a valid subscript key for the `digitNames` dictionary, and so an exclamation mark is used to force-unwrap the `String` value stored in the subscript's optional return value.

The string retrieved from the `digitNames` dictionary is added to the *front* of `output`, effectively building a string version of the number in reverse. (The expression `number % 10` gives a value of `6` for `16`, `8` for `58`, and `0` for `510`.)

The `number` variable is then divided by `10`. Because it is an integer, it is rounded down during the division, so `16` becomes `1`, `58` becomes `5`, and `510` becomes `51`.

The process is repeated until `number` is equal to `0`, at which point the `output` string is returned by the closure, and is added to the output array by the `map(_:)` method.

The use of trailing closure syntax in the example above neatly encapsulates the closure's functionality immediately after the function that closure supports, without needing to wrap the entire closure within the `map(_:)` method's outer parentheses.

Capturing Values

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

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

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

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

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

The string-sorting closure from the Closure Expression Syntax section above can be written outside of the `sorted(by:)` method's parentheses as a trailing closure:

```swift
reversedNames = names.sorted() { $0 > $1 }
```

If a closure expression is provided as the function or method's only argument and you provide that expression as a trailing closure, you do not need to write a pair of parentheses `()` after the function or method's name when you call the function:

```swift
reversedNames = names.sorted { $0 > $1 }
```

Trailing closures are most useful when the closure is sufficiently long that it is not possible to write it inline on a single line. As an example, Swift's `Array` type has a `map(_:)` method which takes a closure expression as its single argument. The closure is called once for each item in the array, and returns an alternative mapped value (possibly of some other type) for that item. The nature of the mapping and the type of the returned value is left up to the closure to specify.

After applying the provided closure to each array element, the `map(_:)` method returns a new array containing all of the new mapped values, in the same order as their corresponding values in the original array.

Here's how you can use the `map(_:)` method with a trailing closure to convert an array of `Int` values into an array of `String` values. The array `[16, 58, 510]` is used to create the new array `["OneSix", "FiveEight", "FiveOneZero"]`:

```swift
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.

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:

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

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

Int

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

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

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

UInt

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

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

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

Floating-Point Numbers

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

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

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

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

Type Safety and Type Inference

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

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

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

Because of type inference, Swift requires far fewer type declarations than languages such as C or Objective-C. Constants and variables are still explicitly typed, but much of the work of specifying their type is done for you.

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

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

```
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`:

Here, the function type of the `sorted(by:)` method's argument makes it clear that a `Bool` value must be returned by the closure. Because the closure's body contains a single expression (`s1 > s2`) that returns a `Bool` value, there is no ambiguity, and the `return` keyword can be omitted.

Shorthand Argument Names

Swift automatically provides shorthand argument names to inline closures, which can be used to refer to the values of the closure's arguments by the names `$0`, `$1`, `$2`, and so on.

If you use these shorthand argument names within your closure expression, you can omit the closure's argument list from its definition, and the number and type of the shorthand argument names will be inferred from the expected function type. The `in` keyword can also be omitted, because the closure expression is made up entirely of its body:

```
reversedNames = names.sorted(by: { $0 > $1 } )
```

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

Operator Methods

There's actually an even *shorter* way to write the closure expression above. Swift's `String` type defines its string-specific implementation of the greater-than operator (`>`) as a method that has two parameters of type `String`, and returns a value of type `Bool`. This exactly matches the method type needed by the `sorted(by:)` method. Therefore, you can simply pass in the greater-than operator, and Swift will infer that you want to use its string-specific implementation:

```
reversedNames = names.sorted(by: >)
```

For more about operator method, see Operator Methods.

Trailing Closures

If you need to pass a closure expression to a function as the function's final argument and the closure expression is long, it can be useful to write it as a *trailing closure* instead. A trailing closure is written after the function call's parentheses, even though it is still an argument to the function. When you use the trailing closure syntax, you don't write the argument label for the closure as part of the function call.

```
reversedNames = names.sorted(by: { (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 `backward(_:_:)` function. In both cases, it is written as `(s1: String, s2: String) -> Bool`. However, for the inline closure expression, the parameters and return type are written *inside* the curly braces, not outside of them.

The start of the closure's body is introduced by the `in` keyword. This keyword indicates that the definition of the closure's parameters and return type has finished, and the body of the closure is about to begin.

Because the body of the closure is so short, it can even be written on a single line:

```
reversedNames = names.sorted(by: { (s1: String, s2: String) -> Bool in
  return s1 > s2 } )
```

This illustrates that the overall call to the `sorted(by:)` method has remained the same. A pair of parentheses still wrap the entire argument for the method. However, that argument is now an inline closure.

Inferring Type From Context

Because the sorting closure is passed as an argument to a method, Swift can infer the types of its parameters and the type of the value it returns. The `sorted(by:)` method is being called on an array of strings, so its argument must be 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:

```
reversedNames = names.sorted(by: { s1, s2 in return s1 > s2 } )
```

It is always possible to infer the parameter types and return type when passing a closure to a function or method as an inline closure expression. As a result, you never need to write an inline closure in its fullest form when the closure is used as a function or method argument.

Nonetheless, you can still make the types explicit if you wish, and doing so is encouraged if it avoids ambiguity for readers of your code. In the case of the `sorted(by:)` method, the purpose of the closure is clear from the fact that sorting is taking place, and it is safe for a reader to assume that the closure is likely to be working with `String` values, because it is assisting with the sorting of an array of strings.

Implicit Returns from Single-Expression Closures

Single-expression closures can implicitly return the result of their single expression by omitting the `return` keyword from their declaration, as in this version of the previous example:

```
reversedNames = names.sorted(by: { s1, s2 in s1 > s2 } )
```



```
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. Decimal floats can also have an optional *exponent*, indicated by an uppercase or lowercase `e`; hexadecimal floats must have an exponent, indicated by an uppercase or lowercase `p`.

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

- `1.25e2` means 1.25 x 10^{2}, or `125.0`.
- `1.25e-2` means 1.25 x 10^{-2}, or `0.0125`.

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

- `0xFp2` means 15 x 2^{2}, or `60.0`.
- `0xFp-2` means 15 x 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 zeros 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're known to be nonnegative. Using the default integer type in everyday situations means that integer constants and variables are immediately interoperable in your code and will match the inferred type for integer literal values.

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

Integer Conversion

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

```
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 can't be added together directly, because they're not of the same type. Instead, this example calls `UInt16(one)` to create a new `UInt16` initialized with the value of `one`, and uses this value in place of the original:

```
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's the sum of two `UInt16` values.

Swift's standard library provides a method called `sorted(by:)`, 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 `sorted(by:)` method returns a new array of the same type and size as the old one, with its elements in the correct sorted order. The original array is not modified by the `sorted(by:)` method.

The closure expression examples below use the `sorted(by:)` method to sort an array of `String` values in reverse alphabetical order. Here's the initial array to be sorted:

```
let names = ["Chris", "Alex", "Ewa", "Barry", "Daniella"]
```

The `sorted(by:)` method accepts a closure that takes two arguments of the same type as the array's contents, and returns a `Bool` value to say whether the first value should appear before or after the second value once the values are sorted. The sorting closure needs to return `true` if the first value should appear *before* the second value, and `false` otherwise.

This example is sorting an array of `String` values, and so the sorting closure needs to be a function of type `(String, String) -> Bool`.

One way to provide the sorting closure is to write a normal function of the correct type, and to pass it in as an argument to the `sorted(by:)` method:

```
func backward(_ s1: String, _ s2: String) -> Bool {
    return s1 > s2
}
var reversedNames = names.sorted(by: backward)
// reversedNames is equal to ["Ewa", "Daniella", "Chris", "Barry", "Alex"]
```

If the first string (`s1`) is greater than the second string (`s2`), the `backward(_:_:)` function will return `true`, indicating that `s1` should appear before `s2` in the sorted array. For characters in strings, "greater than" means "appears later in the alphabet than". This means that the letter `"B"` is "greater than" the letter `"A"`, and the string `"Tom"` is greater than the string `"Tim"`. This gives a reverse alphabetical sort, with `"Barry"` being placed before `"Alex"`, and so on.

However, this is a rather long-winded way to write what is essentially a single-expression function (`a > b`). In this example, it would be preferable to write the sorting closure inline, using closure expression syntax.

Closure Expression Syntax

Closure expression syntax has the following general form:

```
{ (parameters) -> return type in
    statements
}
```

The *parameters* in closure expression syntax can be in-out parameters, but they can't have a default value. Variadic parameters can be used if you name the variadic parameter. Tuples can also be used as parameter types and return types.

The example below shows a closure expression version of the `backward(_:_:)` function from above:

Closures

Closures are self-contained blocks of functionality that can be passed around and used in your code. Closures in Swift are similar to blocks in C and Objective-C and to lambdas in other programming languages.

Closures can capture and store references to any constants and variables from the context in which they are defined. This is known as *closing over* those constants and variables. Swift handles all of the memory management of capturing for you.

> NOTE
>
> Don't worry if you are not familiar with the concept of capturing. It is explained in detail below in 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
- Trailing closure syntax

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 or methods that take functions as one or more of their arguments.

Closure expressions are a way to write inline closures in a brief, focused syntax. Closure expressions provide several syntax optimizations for writing closures in a shortened form without loss of clarity or intent. The closure expression examples below illustrate these optimizations by refining a single example of the `sorted(by:)` method over several iterations, each of which expresses the same functionality in a more succinct way.

The Sorted Method

`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.

Floating-point to integer conversion must also be made explicit. 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 don't have an explicit type in and of themselves. Their type is inferred only at the point that they're evaluated by the compiler.

Type Aliases

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

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

```
typealias AudioSample = UInt16
```

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

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

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

Booleans

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

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

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

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

```
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.

Nested functions are hidden from the outside world by default, but can still be called and used by their enclosing function. An enclosing function can also return one of its nested functions to allow the nested function to be used in another scope.

You can rewrite the `chooseStepFunction(backward:)` example above to use and return nested functions:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    func stepForward(input: Int) -> Int { return input + 1 }
    func stepBackward(input: Int) -> Int { return input - 1 }
    return backward ? stepBackward : stepForward
}
var currentValue = -4
let moveNearerToZero = chooseStepFunction(backward: currentValue > 0)
// moveNearerToZero now refers to the nested stepForward() function
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("zero!")
// -4...
// -3...
// -2...
// -1...
// zero!
```

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

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

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

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

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

The example above determines whether a positive or negative step is needed to move a variable called `currentValue` progressively closer to zero. `currentValue` has an initial value of `3`, which means that `currentValue > 0` returns `true`, causing `chooseStepFunction(backward:)` to return the `stepBackward(_:)` function. A reference to the returned function is stored in a constant called `moveNearerToZero`.

Now that `moveNearerToZero` refers to the correct function, it can be used to count to zero:

```swift
print("Counting to zero:")
// Counting to zero:
while currentValue != 0 {
    print("\(currentValue)... ")
    currentValue = moveNearerToZero(currentValue)
}
print("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*.

Tuples

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

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

```swift
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:

```swift
let (statusCode, statusMessage) = http404Error
print("The status code is \(statusCode)")
// Prints "The status code is 404"
print("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:

```swift
let (justTheStatusCode, _) = http404Error
print("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:

```swift
print("The status code is \(http404Error.0)")
// Prints "The status code is 404"
print("The status message is \(http404Error.1)")
// Prints "The status message is Not Found"
```

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

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

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

```
print("The status code is \(http200Status.statusCode)")
// Prints "The status code is 200"
print("The status message is \(http200Status.description)")
// Prints "The status message is OK"
```

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

> NOTE
>
> Tuples are useful for temporary groups of related values. They're 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 Structures and Classes.

Optionals

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

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

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

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

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

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

nil

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

A different function with the same matching type can be assigned to the same variable, in the same way as for nonfunction types:

```
mathFunction = multiplyTwoInts
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 6"
```

As with any other type, you can leave it to Swift to infer the function type when you assign a function to a constant or variable:

```
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) {
    print("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`:

Function Types

Every function has a specific *function type*, made up of the parameter types and the return type of the function.

For example:

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {
    return a * b
}
```

This example defines two simple mathematical functions called `addTwoInts` and `multiplyTwoInts`. These functions each take two `Int` values, and return an `Int` value, which is the result of performing an appropriate mathematical operation.

The type of both of these functions is `(Int, Int) -> Int`. This can be read as:

"A function that has two parameters, both of type `Int`, and that returns a value of type `Int`."

Here's another example, for a function with no parameters or return value:

```
func printHelloWorld() {
    print("hello, world")
}
```

The type of this function is `() -> Void`, or "a function that has no parameters, and returns `Void`."

Using Function Types

You use function types just like any other types in Swift. For example, you can define a constant or variable to be of a function type and assign an appropriate function to that variable:

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

This can be read as:

"Define a variable called `mathFunction`, which has a type of 'a function that takes two `Int` values, and returns an `Int` value.' Set this new variable to refer to the function called `addTwoInts`."

The `addTwoInts(_:_:)` function has the same type as the `mathFunction` variable, and so this assignment is allowed by Swift's type-checker.

You can now call the assigned function with the name `mathFunction`:

```
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"
```



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

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

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

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

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

If Statements and Forced Unwrapping

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

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

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

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

```
if convertedNumber != nil {
    print("convertedNumber has an integer value of \(convertedNumber!).")
}
// Prints "convertedNumber has an integer value of 123."
```

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

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

Optional Binding

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

Write an optional binding for an `if` statement as follows:

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

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

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

This code can be read as:

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

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

You can use both constants and variables with optional binding. If you wanted to manipulate the value of `actualNumber` within the first branch of the `if` statement, you could write `if var actualNumber` instead, and the value contained within the optional would be made available as a variable rather than a constant.

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

In-Out Parameters

Function parameters are constants by default. Trying to change the value of a function parameter from within the body of that function results in a compile-time error. This means that you can't change the value of a parameter by mistake. If you want a function to modify a parameter's value, and you want those changes to persist after the function call has ended, define that parameter as an *in-out parameter* instead.

You write an in-out parameter by placing the `inout` keyword right before a parameter's type. An in-out parameter has a value that is passed *in* to the function, is modified by the function, and is passed back *out* of the function to replace the original value. For a detailed discussion of the behavior of in-out parameters and associated compiler optimizations, see In-Out Parameters.

You can only pass a variable as the argument for an in-out parameter. You cannot pass a constant or a literal value as the argument, because constants and literals cannot be modified. You place an ampersand (`&`) directly before a variable's name when you pass it as an argument to an in-out parameter, to indicate that it can be modified by the function.

> NOTE
>
> In-out parameters cannot have default values, and variadic parameters cannot be marked as `inout`.

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

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

The `swapTwoInts(_:_:)` function simply swaps the value of `b` into `a`, and the value of `a` into `b`. The function performs this swap by storing the value of `a` in a temporary constant called `temporaryA`, assigning the value of `b` to `a`, and then assigning `temporaryA` to `b`.

You can call the `swapTwoInts(_:_:)` function with two variables of type `Int` to swap their values. Note that the names of `someInt` and `anotherInt` are prefixed with an ampersand when they are passed to the `swapTwoInts(_:_:)` function:

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
print("someInt is now \(someInt), and anotherInt is now \(anotherInt)")
// Prints "someInt is now 107, and anotherInt is now 3"
```

The example above shows that the original values of `someInt` and `anotherInt` are modified by the `swapTwoInts(_:_:)` function, even though they were originally defined outside of the function.

> NOTE
>
> In-out parameters are not the same as returning a value from a function. The `swapTwoInts` example above does not define a return type or return a value, but it still modifies the values of `someInt` and `anotherInt`. In-out parameters are an alternative way for a function to have an effect outside of the scope of its function body.

Default Parameter Values

You can define a *default value* for any parameter in a function by assigning a value to the parameter after that parameter's type. If a default value is defined, you can omit that parameter when calling the function.

```
func someFunction(parameterWithoutDefault: Int, parameterWithDefault: Int
  = 12) {
    // If you omit the second argument when calling this function, then
    // the value of parameterWithDefault is 12 inside the function body.
}
someFunction(parameterWithoutDefault: 3, parameterWithDefault: 6) //
  parameterWithDefault is 6
someFunction(parameterWithoutDefault: 4) // parameterWithDefault is 12
```

Place parameters that don't have default values at the beginning of a function's parameter list, before the parameters that have default values. Parameters that don't have default values are usually more important to the function's meaning—writing them first makes it easier to recognize that the same function is being called, regardless of whether any default parameters are omitted.

Variadic Parameters

A *variadic parameter* accepts zero or more values of a specified type. You use a variadic parameter to specify that the parameter can be passed a varying number of input values when the function is called. Write variadic parameters by inserting three period characters (`...`) after the parameter's type name.

The values passed to a variadic parameter are made available within the function's body as an array of the appropriate type. For example, a variadic parameter with a name of `numbers` and a type of `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, 8.25, 18.75)
// returns 10.0, which is the arithmetic mean of these three numbers
```

> NOTE
>
> A function may have at most one variadic parameter.


```
if let firstNumber = Int("4"), let secondNumber = Int("42"), firstNumber <
  secondNumber && secondNumber < 100 {
    print("\(firstNumber) < \(secondNumber) < 100")
}
// Prints "4 < 42 < 100"

if let firstNumber = Int("4") {
    if let secondNumber = Int("42") {
        if firstNumber < secondNumber && secondNumber < 100 {
            print("\(firstNumber) < \(secondNumber) < 100")
        }
    }
}
// Prints "4 < 42 < 100"
```

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

Implicitly Unwrapped Optionals

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

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

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

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

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

```
let possibleString: String? = "An optional string."
let forcedString: String = possibleString! // requires an exclamation mark

let assumedString: String! = "An implicitly unwrapped optional string."
let implicitString: String = assumedString // no need for an exclamation
  mark
```

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

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

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

```
if assumedString != nil {
    print(assumedString!)
}
// Prints "An implicitly unwrapped optional string."
```

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

```
if let definiteString = assumedString {
    print(definiteString)
}
// Prints "An implicitly unwrapped optional string."
```

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

Error Handling

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

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

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

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

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

Function Argument Labels and Parameter Names

Each function parameter has both an *argument label* and a *parameter name*. The argument label is used when calling the function; each argument is written in the function call with its argument label before it. The parameter name is used in the implementation of the function. By default, parameters use their parameter name as their argument label.

```
func someFunction(firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(firstParameterName: 1, secondParameterName: 2)
```

All parameters must have unique names. Although it's possible for multiple parameters to have the same argument label, unique argument labels help make your code more readable.

Specifying Argument Labels

You write an argument label before the parameter name, separated by a space:

```
func someFunction(argumentLabel parameterName: Int) {
    // In the function body, parameterName refers to the argument value
    // for that parameter.
}
```

Here's a variation of the `greet(person:)` function that takes a person's name and hometown and returns a greeting:

```
func greet(person: String, from hometown: String) -> String {
    return "Hello \(person)!  Glad you could visit from \(hometown)."
}
print(greet(person: "Bill", from: "Cupertino"))
// Prints "Hello Bill!  Glad you could visit from Cupertino."
```

The use of argument labels can allow a function to be called in an expressive, sentence-like manner, while still providing a function body that is readable and clear in intent.

Omitting Argument Labels

If you don't want an argument label for a parameter, write an underscore (_) instead of an explicit argument label for that parameter.

```
func someFunction(_ firstParameterName: Int, secondParameterName: Int) {
    // In the function body, firstParameterName and secondParameterName
    // refer to the argument values for the first and second parameters.
}
someFunction(1, secondParameterName: 2)
```

If a parameter has an argument label, the argument *must* be labeled when you call the function.

Because the tuple's member values are named as part of the function's return type, they can be accessed with dot syntax to retrieve the minimum and maximum found values:

```
let bounds = minMax(array: [8, -6, 2, 109, 3, 71])
print("min is \(bounds.min) and max is \(bounds.max)")
// Prints "min is -6 and max is 109"
```

Note that the tuple's members do not need to be named at the point that the tuple is returned from the function, because their names are already specified as part of the function's return type.

Optional Tuple Return Types

If the tuple type to be returned from a function has the potential to have "no value" for the entire tuple, you can use an *optional* tuple return type to reflect the fact that the entire tuple can be `nil`. You write an optional tuple return type by placing a question mark after the tuple type's closing parenthesis, such as `(Int, Int)?` or `(String, Int, Bool)?`.

> NOTE
>
> An optional tuple type such as `(Int, Int)?` is different from a tuple that contains optional types such as `(Int?, Int?)`. With an optional tuple type, the entire tuple is optional, not just each individual value within the tuple.

The `minMax(array:)` function above returns a tuple containing two `Int` values. However, the function does not perform any safety checks on the array it is passed. If the `array` argument contains an empty array, the `minMax(array:)` function, as defined above, will trigger a runtime error when attempting to access `array[0]`.

To handle an empty array safely, write the `minMax(array:)` function with an optional tuple return type and return a value of `nil` when the array is empty:

```
func minMax(array: [Int]) -> (min: Int, max: Int)? {
    if array.isEmpty { return nil }
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

You can use optional binding to check whether this version of the `minMax(array:)` function returns an actual tuple value or `nil`:

```
if let bounds = minMax(array: [8, -6, 2, 109, 3, 71]) {
    print("min is \(bounds.min) and max is \(bounds.max)")
}
// Prints "min is -6 and max is 109"
```


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

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

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

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

```
func makeASandwich() throws {
    // ...
}

do {
    try makeASandwich()
    eatASandwich()
} catch SandwichError.outOfCleanDishes {
    washDishes()
} catch SandwichError.missingIngredients(let ingredients) {
    buyGroceries(ingredients)
}
```

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

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

Throwing, catching, and propagating errors is covered in greater detail in Error Handling.

Assertions and Preconditions

Assertions and *preconditions* are checks that happen at runtime. You use them to make sure an essential condition is satisfied before executing any further code. If the Boolean condition in the assertion or precondition evaluates to `true`, code execution continues as usual. If the condition evaluates to `false`, the current state of the program is invalid; code execution ends, and your app is terminated.

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

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

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

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

Debugging with Assertions

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

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

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

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

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

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

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



```
func printAndCount(string: String) -> Int {
    print(string)
    return string.count
}
func printWithoutCounting(string: String) {
    let _ = printAndCount(string: string)
}
printAndCount(string: "hello, world")
// prints "hello, world" and returns a value of 12
printWithoutCounting(string: "hello, world")
// prints "hello, world" but does not return a value
```

The first function, `printAndCount(string:)`, prints a string, and then returns its character count as an `Int`. The second function, `printWithoutCounting(string:)`, calls the first function, but ignores its return value. When the second function is called, the message is still printed by the first function, but the returned value is not used.

> NOTE
>
> Return values can be ignored, but a function that says it will return a value must always do so. A function with a defined return type cannot allow control to fall out of the bottom of the function without returning a value, and attempting to do so will result in a compile-time error.

Functions with Multiple Return Values

You can use a tuple type as the return type for a function to return multiple values as part of one compound return value.

The example below defines a function called `minMax(array:)`, which finds the smallest and largest numbers in an array of `Int` values:

```
func minMax(array: [Int]) -> (min: Int, max: Int) {
    var currentMin = array[0]
    var currentMax = array[0]
    for value in array[1..<array.count] {
        if value < currentMin {
            currentMin = value
        } else if value > currentMax {
            currentMax = value
        }
    }
    return (currentMin, currentMax)
}
```

The `minMax(array:)` function returns a tuple containing two `Int` values. These values are labeled `min` and `max` so that they can be accessed by name when querying the function's return value.

The body of the `minMax(array:)` function starts by setting two working variables called `currentMin` and `currentMax` to the value of the first integer in the array. The function then iterates over the remaining values in the array and checks each value to see if it is smaller or larger than the values of `currentMin` and `currentMax` respectively. Finally, the overall minimum and maximum values are returned as a tuple of two `Int` values.

The function definition still needs parentheses after the function's name, even though it does not take any parameters. The function name is also followed by an empty pair of parentheses when the function is called.

Functions With Multiple Parameters

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

This function takes a person's name and whether they have already been greeted as input, and returns an appropriate greeting for that person:

```
func greet(person: String, alreadyGreeted: Bool) -> String {
    if alreadyGreeted {
        return greetAgain(person: person)
    } else {
        return greet(person: person)
    }
}
print(greet(person: "Tim", alreadyGreeted: true))
// Prints "Hello again, Tim!"
```

You call the `greet(person:alreadyGreeted:)` function by passing it both a `String` argument value labeled `person` and a `Bool` argument value labeled `alreadyGreeted` in parentheses, separated by commas. Note that this function is distinct from the `greet(person:)` function shown in an earlier section. Although both functions have names that begin with `greet`, the `greet(person:alreadyGreeted:)` function takes two arguments but the `greet(person:)` function takes only one.

Functions Without Return Values

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

```
func greet(person: String) {
    print("Hello, \(person)!")
}
greet(person: "Dave")
// Prints "Hello, Dave!"
```

Because it does not need to return a value, the function's definition does not include the return arrow (–>) or a return type.

> NOTE
>
> Strictly speaking, this version of the `greet(person:)` 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, which is written as `()`.

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

Enforcing Preconditions

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

You write a precondition by calling the `precondition(_:_:file:line:)` function. You pass this function an expression that evaluates to `true` or `false` and a message to display if the result of the condition is `false`. For example:

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

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

> NOTE
>
> If you compile in unchecked mode (`-Ounchecked`), preconditions aren't checked. The compiler assumes that preconditions are always true, and it optimizes your code accordingly. However, the `fatalError(_:file:line:)` function always halts execution, regardless of optimization settings.
>
> You can use the `fatalError(_:file:line:)` function during prototyping and early development to create stubs for functionality that hasn't been implemented yet, by writing `fatalError("Unimplemented")` as the stub implementation. Because fatal errors are never optimized out, unlike assertions or preconditions, you can be sure that execution always halts if it encounters a stub implementation.

Basic Operators

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

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

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

This chapter describes the common operators in Swift. 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 `c!`).
- *Binary* operators operate on two targets (such as `2 + 3`) and are *infix* because they appear in between their two targets.
- *Ternary* operators operate on three targets. Like C, Swift has only one ternary operator, the ternary conditional operator (`a ? b : c`).

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

Assignment Operator

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

```
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:


```
print(greet(person: "Anna"))
// Prints "Hello, Anna!"
print(greet(person: "Brian"))
// Prints "Hello, Brian!"
```

You call the `greet(person:)` function by passing it a `String` value after the `person` argument label, such as `greet(person: "Anna")`. Because the function returns a `String` value, `greet(person:)` can be wrapped in a call to the `print(_:separator:terminator:)` function to print that string and see its return value, as shown above.

> NOTE
>
> The `print(_:separator:terminator:)` function doesn't have a label for its first argument, and its other arguments are optional because they have a default value. These variations on function syntax are discussed below in Function Argument Labels and Parameter Names and Default Parameter Values.

The body of the `greet(person:)` function starts by defining a new `String` constant called `greeting` and setting it to a simple greeting message. This greeting is then passed back out of the function using the `return` keyword. In the line of code that says `return greeting`, the function finishes its execution and returns the current value of `greeting`.

You can call the `greet(person:)` function multiple times with different input values. The example above shows what happens if it is called with an input value of `"Anna"`, and an input value of `"Brian"`. The function returns a tailored greeting in each case.

To make the body of this function shorter, you can combine the message creation and the return statement into one line:

```
func greetAgain(person: String) -> String {
    return "Hello again, " + person + "!"
}
print(greetAgain(person: "Anna"))
// Prints "Hello again, Anna!"
```

Function Parameters and Return Values

Function parameters and return values are extremely flexible in Swift. You can define anything from a simple utility function with a single unnamed parameter to a complex function with expressive parameter names and different parameter options.

Functions Without Parameters

Functions are not required to define input parameters. Here's a function with no input parameters, which always returns the same `String` message whenever it is called:

```
func sayHelloWorld() -> String {
    return "hello, world"
}
print(sayHelloWorld())
// Prints "hello, world"
```

Functions

Functions are self-contained chunks of code that perform a specific task. You give a function a name that identifies what it does, and this name is used to "call" the function to perform its task when needed.

Swift's unified function syntax is flexible enough to express anything from a simple C-style function with no parameter names to a complex Objective-C-style method with names and argument labels for each parameter. Parameters can provide default values to simplify function calls and can be passed as in-out parameters, which modify a passed variable once the function has completed its execution.

Every function in Swift has a type, consisting of the function's parameter types and return type. You can use this type like any other type in Swift, which makes it easy to pass functions as parameters to other functions, and to return functions from functions. Functions can also be written within other functions to encapsulate useful functionality within a nested function scope.

Defining and Calling Functions

When you define a function, you can optionally define one or more named, typed values that the function takes as input, known as *parameters*. You can also optionally define a type of value that the function will pass back as output when it is done, known as its *return type*.

Every function has a *function name*, which describes the task that the function performs. To use a function, you "call" that function with its name and pass it input values (known as *arguments*) that match the types of the function's parameters. A function's arguments must always be provided in the same order as the function's parameter list.

The function in the example below is called `greet(person:)`, because that's what it does—it takes a person's name as input and returns a greeting for that person. To accomplish this, you define one input parameter—a `String` value called `person`—and a return type of `String`, which will contain a greeting for that person:

```
func greet(person: String) -> String {
    let greeting = "Hello, " + person + "!"
    return greeting
}
```

All of this information is rolled up into the function's *definition*, which is prefixed with the `func` keyword. You indicate the function's return type with the *return arrow* `->` (a hyphen followed by a right angle bracket), which is followed by the name of the type to return.

The definition describes what the function does, what it expects to receive, and what it returns when it is done. The definition makes it easy for the function to be called unambiguously from elsewhere in your code:

```
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 don't allow values to overflow by default. You can opt in to value overflow behavior by using Swift's overflow operators (such as `a &+ b`). See Overflow Operators.

The addition operator is also supported for `String` concatenation:

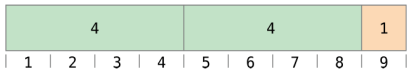
```
"hello, " + "world"  // equals "hello, world"
```

Remainder Operator

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

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

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



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

In Swift, this would be written as:

```
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 x 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 x 2) + 1`

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

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

Inserting –9 and 4 into the equation yields:

`–9 = (4 x –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.

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
```



```
if #available(iOS 10, macOS 10.12, *) {
    // Use iOS 10 APIs on iOS, and use macOS 10.12 APIs on macOS
} else {
    // Fall back to earlier iOS and macOS APIs
}
```

The availability condition above specifies that in iOS, the body of the `if` statement executes only in iOS 10 and later; in macOS, only in macOS 10.12 and later. The last argument, `*`, is required and specifies that on any other platform, the body of the `if` executes on the minimum deployment target specified by your target.

In its general form, the availability condition takes a list of platform names and versions. You use platform names such as `iOS`, `macOS`, `watchOS`, and `tvOS`—for the full list, see Declaration Attributes. In addition to specifying major version numbers like iOS 8 or macOS 10.10, you can specify minor versions numbers like iOS 11.2.6 and macOS 10.13.3.

```
if #available(platform name version, ..., *) {
    statements to execute if the APIs are available
} else {
    fallback statements to execute if the APIs are unavailable
}
```

```swift
func greet(person: [String: String]) {
    guard let name = person["name"] else {
        return
    }

    print("Hello \(name)!")

    guard let location = person["location"] else {
        print("I hope the weather is nice near you.")
        return
    }

    print("I hope the weather is nice in \(location).")
}

greet(person: ["name": "John"])
// Prints "Hello John!"
// Prints "I hope the weather is nice near you."
greet(person: ["name": "Jane", "location": "Cupertino"])
// Prints "Hello Jane!"
// Prints "I hope the weather is nice in Cupertino."
```

If the `guard` statement's condition is met, code execution continues after the `guard` statement's closing brace. Any variables or constants that were assigned values using an optional binding as part of the condition are available for the rest of the code block that the `guard` statement appears in.

If that condition is not met, the code inside the `else` branch is executed. That branch must transfer control to exit the code block in which the `guard` statement appears. It can do this with a control transfer statement such as `return`, `break`, `continue`, or `throw`, or it can call a function or method that doesn't return, such as `fatalError(_:file:line:)`.

Using a `guard` statement for requirements improves the readability of your code, compared to doing the same check with an `if` statement. It lets you write the code that's typically executed without wrapping it in an `else` block, and it lets you keep the code that handles a violated requirement next to the requirement.

Checking API Availability

Swift has built-in support for checking API availability, which ensures that you don't accidentally use APIs that are unavailable on a given deployment target.

The compiler uses availability information in the SDK to verify that all of the APIs used in your code are available on the deployment target specified by your project. Swift reports an error at compile time if you try to use an API that isn't available.

You use an *availability condition* in an `if` or `guard` statement to conditionally execute a block of code, depending on whether the APIs you want to use are available at runtime. The compiler uses the information from the availability condition when it verifies that the APIs in that block of code are available.

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* (+=):

```swift
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 don't return a value. For example, you can't write `let b = a += 2`.

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

Comparison Operators

Swift supports all standard C *comparison operators*:

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

> NOTE
>
> Swift also provides two *identity operators* (`===` and `!==`), which you use to test whether two object references both refer to the same object instance. For more information, see Structures and Classes.

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

```swift
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" {
    print("hello, world")
} else {
    print("I'm sorry \(name), but I don't recognize you")
}
// Prints "hello, world", because name is indeed equal to "world".
```

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

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

```
(1, "zebra") < (2, "apple")   // true because 1 is less than 2; "zebra" and "apple" are not compared
(3, "apple") < (3, "bird")    // true because 3 is equal to 3, and "apple" is less than "bird"
(4, "dog") == (4, "dog")      // true because 4 is equal to 4, and "dog" is equal to "dog"
```

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

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

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

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

Ternary Conditional Operator


```
gameLoop: while square != finalSquare {
    diceRoll += 1
    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]
    }
}
print("Game over!")
```

The dice is rolled at the start of each loop. Rather than moving the player immediately, the loop uses a `switch` statement to consider the result of the move and to determine whether the move is allowed:

- If the dice roll will move the player onto the final square, the game is over. The `break gameLoop` statement transfers control to the first line of code outside of the `while` loop, which ends the game.
- If the dice roll will move the player *beyond* the final square, the move is invalid and the player needs to roll again. The `continue gameLoop` statement ends the current `while` loop iteration and begins the next iteration of the loop.
- In all other cases, the dice roll is a valid move. The player moves forward by `diceRoll` squares, and the game logic checks for any snakes and ladders. The loop then ends, and control returns to the `while` condition to decide whether another turn is required.

> NOTE
>
> If the `break` statement above did not use the `gameLoop` label, it would break out of the `switch` statement, not the `while` statement. Using the `gameLoop` label makes it clear which control statement should be terminated.
>
> It is not strictly necessary to use the `gameLoop` label when calling `continue gameLoop` to jump to the next iteration of the loop. There is only one loop in the game, and therefore no ambiguity as to which loop the `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.

Early Exit

A `guard` statement, like an `if` statement, executes statements depending on the Boolean value of an expression. You use a `guard` statement to require that a condition must be true in order for the code after the `guard` statement to be executed. Unlike an `if` statement, a `guard` statement always has an `else` clause—the code inside the `else` clause is executed if the condition is not true.

To achieve these aims, you can mark a loop statement or conditional statement with a *statement label*. With a conditional statement, you can use a statement label with the `break` statement to end the execution of the labeled statement. With a loop statement, you can use a statement label with the `break` or `continue` statement to end or continue the execution of the labeled statement.

A labeled statement is indicated by placing a label on the same line as the statement's introducer keyword, followed by a colon. Here's an example of this syntax for a `while` loop, although the principle is the same for all loops and `switch` statements:

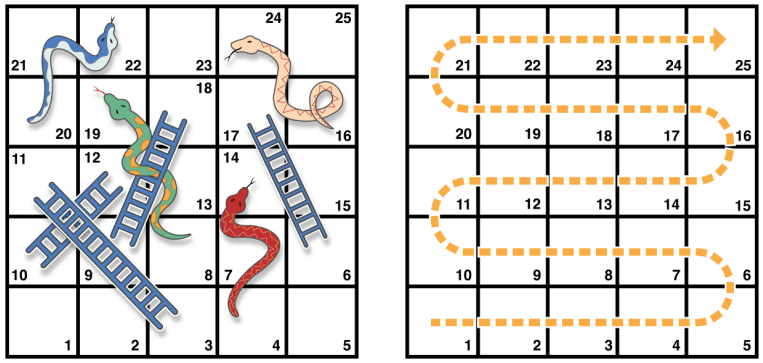
```
label name: while condition {
    statements
}
```

The following example uses the `break` and `continue` statements with a labeled `while` loop for an adapted version of the *Snakes and Ladders* game that you saw earlier in this chapter. This time around, the game has an extra rule:

- To win, you must land *exactly* on square 25.

If a particular dice roll would take you beyond square 25, you must roll again until you roll the exact number needed to land on square 25.

The game board is the same as before.



The values of `finalSquare`, `board`, `square`, and `diceRoll` are initialized in the same way as before:

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

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

The ternary conditional operator is shorthand for the code below:

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

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

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

The example above is shorthand for the code below:

```
let contentHeight = 40
let hasHeader = true
let rowHeight: Int
if hasHeader {
    rowHeight = contentHeight + 50
} else {
    rowHeight = contentHeight + 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, which is more concise than the code used in the second example.

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

Nil-Coalescing Operator

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

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

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

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

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

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

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

var colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is nil, so colorNameToUse is set to the default of
  "red"
```

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

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

```
userDefinedColorName = "green"
colorNameToUse = userDefinedColorName ?? defaultColorName
// userDefinedColorName is not nil, so colorNameToUse is set to "green"
```

Range Operators

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

Closed Range Operator

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

The closed range operator is useful when iterating over a range in which you want all of the values to be used, such as with a `for-in` loop:

In Swift, `switch` statements don't fall through the bottom of each case and into the next one. That is, the entire `switch` statement completes its execution as soon as the first matching case is completed. By contrast, C requires you to insert an explicit `break` statement at the end of every `switch` case to prevent fallthrough. Avoiding default fallthrough means that Swift `switch` statements are much more concise and predictable than their counterparts in C, and thus they avoid executing multiple `switch` cases by mistake.

If you need C-style fallthrough behavior, you can opt in to this behavior on a case-by-case basis with the `fallthrough` keyword. The example below uses `fallthrough` to create a textual description of a number.

```
let integerToDescribe = 5
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."
}
print(description)
// Prints "The number 5 is a prime number, and also an integer."
```

This example declares a new `String` variable called `description` and assigns it an initial value. The function then considers the value of `integerToDescribe` using a `switch` statement. If the value of `integerToDescribe` is one of the prime numbers in the list, the function appends text to the end of `description`, to note that the number is prime. It then uses the `fallthrough` keyword to "fall into" the `default` case as well. The `default` case adds some extra text to the end of the description, and the `switch` statement is complete.

Unless the value of `integerToDescribe` is in the list of known prime numbers, it is not matched by the first `switch` case at all. Because there are no other specific cases, `integerToDescribe` is matched by the `default` case.

After the `switch` statement has finished executing, the number's description is printed using the `print(_:separator:terminator:)` function. In this example, the number `5` is correctly identified as a prime number.

> NOTE
>
> The `fallthrough` keyword does not check the case conditions for the `switch` case that it causes execution to fall into. The `fallthrough` keyword simply causes code execution to move directly to the statements inside the next case (or `default` case) block, as in C's standard `switch` statement behavior.

Labeled Statements

In Swift, you can nest loops and conditional statements inside other loops and conditional statements to create complex control flow structures. However, loops and conditional statements can both use the `break` statement to end their execution prematurely. Therefore, it is sometimes useful to be explicit about which loop or conditional statement you want a `break` statement to terminate. Similarly, if you have multiple nested loops, it can be useful to be explicit about which loop the `continue` statement should affect.

> NOTE
>
> A `switch` case that contains only a comment is reported as a compile-time error. Comments are not statements and do not cause a `switch` case to be ignored. Always use a `break` statement to ignore a `switch` case.

The following example switches on a `Character` value and determines whether it represents a number symbol in one of four languages. For brevity, multiple values are covered in a single `switch` case.

```
let numberSymbol: Character = "三"  // Chinese symbol 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 {
    print("The integer value of \(numberSymbol) is \(integerValue).")
} else {
    print("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.

Because it's not practical to list every possible `Character` value in the example above, a `default` case handles any characters that are not matched. This `default` case does not need to perform any action, and so it is written with a single `break` statement as its body. As soon as the `default` case is matched, the `break` statement ends the `switch` statement's execution, and code execution continues from the `if let` statement.

Fallthrough


```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

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

Half-Open Range Operator

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

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

```
let names = ["Anna", "Alex", "Brian", "Jack"]
let count = names.count
for i in 0..<count {
    print("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's a half-open range. For more about arrays, see Arrays.

One-Sided Ranges

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

```
for name in names[2...] {
    print(name)
}
// Brian
// Jack

for name in names[...2] {
    print(name)
}
// Anna
// Alex
// Brian
```

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

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

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

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

Logical Operators

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

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

Logical NOT Operator

The *logical NOT operator* (`!a`) inverts a Boolean value so that `true` becomes `false`, and `false` becomes `true`.

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

Continue

The `continue` statement tells a loop to stop what it is doing and start again at the beginning of the next iteration through the loop. It says "I am done with the current loop iteration" without leaving the loop altogether.

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

```
let puzzleInput = "great minds think alike"
var puzzleOutput = ""
let charactersToRemove: [Character] = ["a", "e", "i", "o", "u", " "]
for character in puzzleInput {
    if charactersToRemove.contains(character) {
        continue
    } else {
        puzzleOutput.append(character)
    }
}
print(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.

Break

The `break` statement ends execution of an entire control flow statement immediately. The `break` statement can be used inside a `switch` or loop statement when you want to terminate the execution of the `switch` or loop statement earlier than would otherwise be the case.

Break in a Loop Statement

When used inside a loop statement, `break` ends the loop's execution immediately and transfers control to the code after the loop's closing brace (`}`). No further code from the current iteration of the loop is executed, and no further iterations of the loop are started.

Break in a Switch Statement

When used inside a `switch` statement, `break` causes the `switch` statement to end its execution immediately and to transfer control to the code after the `switch` statement's closing brace (`}`).

This behavior can be used to match and ignore one or more cases in a `switch` statement. Because Swift's `switch` statement is exhaustive and does not allow empty cases, it is sometimes necessary to deliberately match and ignore a case in order to make your intentions explicit. You do this by writing the `break` statement as the entire body of the case you want to ignore. When that case is matched by the `switch` statement, the `break` statement inside the case ends the `switch` statement's execution immediately.

```swift
let someCharacter: Character = "e"
switch someCharacter {
case "a", "e", "i", "o", "u":
    print("\(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":
    print("\(someCharacter) is a consonant")
default:
    print("\(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. Finally, the `default` case matches any other character.

Compound cases can also include value bindings. All of the patterns of a compound case have to include the same set of value bindings, and each binding has to get a value of the same type from all of the patterns in the compound case. This ensures that, no matter which part of the compound case matched, the code in the body of the case can always access a value for the bindings and that the value always has the same type.

```swift
let stillAnotherPoint = (9, 0)
switch stillAnotherPoint {
case (let distance, 0), (0, let distance):
    print("On an axis, \(distance) from the origin")
default:
    print("Not on an axis")
}
// Prints "On an axis, 9 from the origin"
```

The `case` above has two patterns: `(let distance, 0)` matches points on the x-axis and `(0, let distance)` matches points on the y-axis. Both patterns include a binding for `distance` and `distance` is an integer in both patterns—which means that the code in the body of the `case` can always access a value for `distance`.

Control Transfer Statements

Control transfer statements change the order in which your code is executed, by transferring control from one piece of code to another. Swift has five control transfer statements:

- `continue`
- `break`
- `fallthrough`
- `return`
- `throw`

The `continue`, `break`, and `fallthrough` statements are described below. The `return` statement is described in Functions, and the `throw` statement is described in Propagating Errors Using Throwing Functions.


```swift
let allowedEntry = false
if !allowedEntry {
    print("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`:

```swift
let enteredDoorCode = true
let passedRetinaScan = false
if enteredDoorCode && passedRetinaScan {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "ACCESS DENIED"
```

Logical OR Operator

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

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

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

```swift
let hasDoorKey = false
let knowsOverridePassword = true
if hasDoorKey || knowsOverridePassword {
    print("Welcome!")
} else {
    print("ACCESS DENIED")
}
// Prints "Welcome!"
```

Combining Logical Operators

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

```swift
if enteredDoorCode && passedRetinaScan || hasDoorKey ||
  knowsOverridePassword {
    print("Welcome!")
} else {
    print("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. The example can be read as:

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

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

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

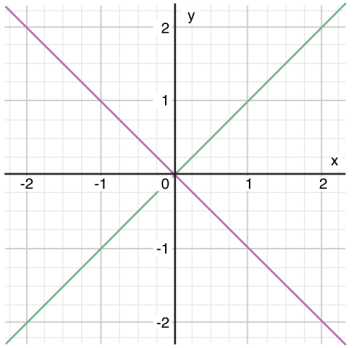
Explicit Parentheses

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

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



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



The `switch` statement determines whether the point is on the green diagonal line where `x == y`, on the purple diagonal line where `x == -y`, or neither.

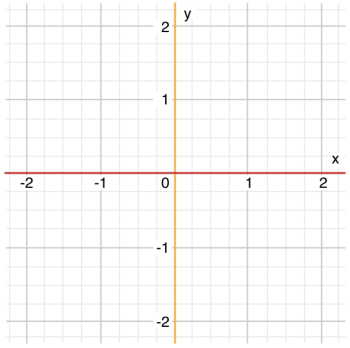
The three `switch` cases declare placeholder constants `x` and `y`, which temporarily take on the two tuple values from `yetAnotherPoint`. These constants are used as part of a `where` clause, to create a dynamic filter. The `switch` case matches the current value of `point` only if the `where` clause's condition evaluates to `true` for that value.

As in the previous example, the final case matches all possible remaining values, and so a `default` case is not needed to make the `switch` statement exhaustive.

Compound Cases

Multiple switch cases that share the same body can be combined by writing several patterns after `case`, with a comma between each of the patterns. If any of the patterns match, then the case is considered to match. The patterns can be written over multiple lines if the list is long. For example:

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



The `switch` statement determines whether the point is on the red x-axis, on the orange y-axis, or elsewhere (on neither axis).

The three `switch` cases declare placeholder constants `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`.

After the temporary constants are declared, they can be used within the case's code block. Here, they are used to print the categorization of the point.

This `switch` statement does not have a `default` case. The final case, `case let (x, y)`, declares a tuple of two placeholder constants that can match any value. Because `anotherPoint` is always a tuple of two values, this case matches all possible remaining values, and a `default` case is not needed to make the `switch` statement exhaustive.

Where

A `switch` case can use a `where` clause to check for additional conditions.

The example below categorizes an (x, y) point on the following graph:

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

Strings and Characters

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

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

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

> NOTE
>
> Swift's `String` type is bridged with Foundation's `NSString` class. Foundation also extends `String` to expose methods defined by `NSString`. This means, if you import Foundation, you can access those `NSString` methods on `String` without casting.
>
> For more information about using `String` with Foundation and Cocoa, see Bridging Between String and NSString.

String Literals

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

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

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

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

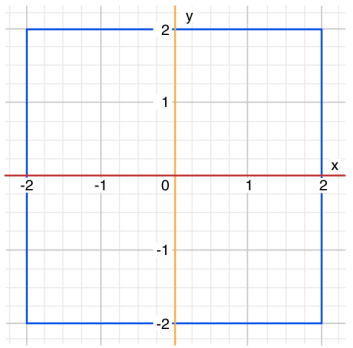
Multiline String Literals

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

You can use tuples to test multiple values in the same `switch` statement. Each element of the tuple can be tested against a different value or interval of values. Alternatively, use the underscore character (_), also known as the wildcard pattern, to match any possible value.

The example below takes an (x, y) point, expressed as a simple tuple of type `(Int, Int)`, and categorizes it on the graph that follows the example.

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



The `switch` statement determines whether the point is at the origin (0, 0), on the red x-axis, on the orange y-axis, inside the blue 4-by-4 box centered on the origin, or outside of the box.

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

Value Bindings

A `switch` case can name the value or values it matches to temporary constants or variables, for use in the body of the case. This behavior is known as *value binding*, because the values are bound to temporary constants or variables within the case's body.

The example below takes an (x, y) point, expressed as a tuple of type `(Int, Int)`, and categorizes it on the graph that follows:

```swift
let anotherCharacter: Character = "a"
switch anotherCharacter {
case "a", "A":
    print("The letter A")
default:
    print("Not the letter A")
}
// Prints "The letter A"
```

For readability, a compound case can also be written over multiple lines. For more information about compound cases, see Compound Cases.

> NOTE
>
> To explicitly fall through at the end of a particular `switch` case, use the `fallthrough` keyword, as described in Fallthrough.

Interval Matching

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

```swift
let approximateCount = 62
let countedThings = "moons orbiting Saturn"
let naturalCount: String
switch approximateCount {
case 0:
    naturalCount = "no"
case 1..<5:
    naturalCount = "a few"
case 5..<12:
    naturalCount = "several"
case 12..<100:
    naturalCount = "dozens of"
case 100..<1000:
    naturalCount = "hundreds of"
default:
    naturalCount = "many"
}
print("There are \(naturalCount) \(countedThings).")
// Prints "There are dozens of moons orbiting Saturn."
```

In the above example, `approximateCount` is evaluated in a `switch` statement. Each `case` compares that value to a number or interval. Because the value of `approximateCount` falls between 12 and 100, `naturalCount` is assigned the value `"dozens of"`, and execution is transferred out of the `switch` statement.

Tuples


```swift
let quotation = """
The White Rabbit put on his spectacles.  "Where shall I begin,
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on
till you come to the end; then stop."
"""
```

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

```swift
let singleLineString = "These are the same."
let multilineString = """
These are the same.
"""
```

When your source code includes a line break inside of a multiline string literal, that line break also appears in the string's value. If you want to use line breaks to make your source code easier to read, but you don't want the line breaks to be part of the string's value, write a backslash (\) at the end of those lines:

```swift
let softWrappedQuotation = """
The White Rabbit put on his spectacles.  "Where shall I begin, \
please your Majesty?" he asked.

"Begin at the beginning," the King said gravely, "and go on \
till you come to the end; then stop."
"""
```

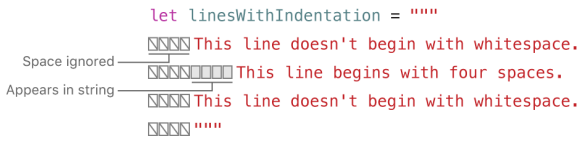
To make a multiline string literal that begins or ends with a line feed, write a blank line as the first or last line. For example:

```swift
let lineBreaks = """

This string starts with a line break.
It also ends with a line break.

"""
```

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



In the example above, even though the entire multiline string literal is indented, the first and last lines in the string don't begin with any whitespace. The middle line has more indentation than the closing quotation marks, so it starts with that extra four-space indentation.

Special Characters in String Literals

String literals can include the following special characters:

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

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

```
let wiseWords = "\"Imagination is more important than knowledge\" –
  Einstein"
// "Imagination is more important than knowledge" – Einstein
let dollarSign = "\u{24}"        // $,  Unicode scalar U+0024
let blackHeart = "\u{2665}"      // ♥,  Unicode scalar U+2665
let sparklingHeart = "\u{1F496}" // 💖, Unicode scalar U+1F496
```

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

```
let threeDoubleQuotationMarks = """
Escaping the first quotation mark \"""
Escaping all three quotation marks \"\"\"
"""
```

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
```

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

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



```
let someCharacter: Character = "z"
switch someCharacter {
case "a":
    print("The first letter of the alphabet")
case "z":
    print("The last letter of the alphabet")
default:
    print("Some other character")
}
// Prints "The last letter of the alphabet"
```

The `switch` statement's first case matches the first letter of the English alphabet, `a`, and its second case matches the last letter, `z`. Because the `switch` must have a case for every possible character, not just every alphabetic character, this `switch` statement uses a `default` case to match all characters other than a and `z`. This provision ensures that the `switch` statement is exhaustive.

No Implicit Fallthrough

In contrast with `switch` statements in C and Objective-C, `switch` statements in Swift do not fall through the bottom of each case and into the next one by default. Instead, the entire `switch` statement finishes its execution as soon as the first matching `switch` case is completed, without requiring an explicit `break` statement. This makes the `switch` statement safer and easier to use than the one in C and avoids executing more than one `switch` case by mistake.

> NOTE
>
> Although `break` is not required in Swift, you can use a `break` statement to match and ignore a particular case or to break out of a matched case before that case has completed its execution. For details, see Break in a Switch Statement.

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": // Invalid, the case has an empty body
case "A":
    print("The letter A")
default:
    print("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.

To make a `switch` with a single case that matches both `"a"` and `"A"`, combine the two values into a compound case, separating the values with commas.

The final `else` clause is optional, however, and can be excluded if the set of conditions does not need to be complete.

```
temperatureInFahrenheit = 72
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
}
```

Because the temperature is neither too cold nor too warm to trigger the `if` or `else if` conditions, no message is printed.

Switch

A `switch` statement considers a value and compares it against several possible matching patterns. It then executes an appropriate block of code, based on the first pattern that matches successfully. A `switch` statement provides an alternative to the `if` statement for responding to multiple potential states.

In its simplest form, a `switch` statement compares a value against one or more values of the same type.

```
switch some value to consider {
case value 1:
    respond to value 1
case value 2,
     value 3:
    respond to value 2 or 3
default:
    otherwise, do something else
}
```

Every `switch` statement consists of multiple possible *cases*, each of which begins with the `case` keyword. In addition to comparing against specific values, Swift provides several ways for each case to specify more complex matching patterns. These options are described later in this chapter.

Like the body of an `if` statement, each `case` is a separate branch of code execution. The `switch` statement determines which branch should be selected. This procedure is known as *switching* on the value that is being considered.

Every `switch` statement must be *exhaustive*. That is, every possible value of the type being considered must be matched by one of the `switch` cases. If it's not appropriate to provide a case for every possible value, you can define a default case to cover any values that are not addressed explicitly. This default case is indicated by the `default` keyword, and must always appear last.

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

String Mutability

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

```
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's passed to a function or method, or when it's assigned to a constant or variable. In each case, a new copy of the existing `String` value is created, and the new copy is passed or assigned, not the original version. Value types are described in Structures and Enumerations Are Value Types.

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

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

Working with Characters

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

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

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

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

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

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

```swift
let catCharacters: [Character] = ["C", "a", "t", "!", "🐱"]
let catString = String(catCharacters)
print(catString)
// Prints "Cat!🐱"
```

Concatenating Strings and Characters

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

```swift
let string1 = "hello"
let string2 = " there"
var welcome = string1 + string2
// welcome now equals "hello there"
```

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

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

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

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

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

Swift provides two ways to add conditional branches to your code: the `if` statement and the `switch` statement. Typically, you use the `if` statement to evaluate simple conditions with only a few possible outcomes. The `switch` statement is better suited to more complex conditions with multiple possible permutations and is useful in situations where pattern matching can help select an appropriate code branch to execute.

If

In its simplest form, the `if` statement has a single `if` condition. It executes a set of statements only if that condition is `true`.

```swift
var temperatureInFahrenheit = 30
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
}
// Prints "It's very cold. Consider wearing a scarf."
```

The example above checks whether the temperature is less than or equal to 32 degrees Fahrenheit (the freezing point of water). If it is, a message is printed. Otherwise, no message is printed, and code execution continues after the `if` statement's closing brace.

The `if` statement can provide an alternative set of statements, known as an *else clause*, for situations when the `if` condition is `false`. These statements are indicated by the `else` keyword.

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

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.

```swift
temperatureInFahrenheit = 90
if temperatureInFahrenheit <= 32 {
    print("It's very cold. Consider wearing a scarf.")
} else if temperatureInFahrenheit >= 86 {
    print("It's really warm. Don't forget to wear sunscreen.")
} else {
    print("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 was added to respond to particularly warm temperatures. The final `else` clause remains, and it prints a response for any temperatures that are neither too warm nor too cold.

```
repeat {
    statements
} while condition
```

Here's the *Snakes and Ladders* example again, written as a `repeat-while` loop rather than a `while` loop. The values of `finalSquare`, `board`, `square`, and `diceRoll` are initialized in exactly the same way as with a `while` loop.

```
let finalSquare = 25
var board = [Int](repeating: 0, count: finalSquare + 1)
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 isn't possible to win the game by moving up a ladder. Therefore, it's safe to check for a snake or a ladder as the first action in the loop.

At the start of the game, the player is on "square zero". `board[0]` always equals `0` and has no effect.

```
repeat {
    // move up or down for a snake or ladder
    square += board[square]
    // roll the dice
    diceRoll += 1
    if diceRoll == 7 { diceRoll = 1 }
    // move by the rolled amount
    square += diceRoll
} while square < finalSquare
print("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's not evaluated until the *end* of the first run through the loop. The structure of the `repeat-while` loop is better suited to this game than the `while` loop in the previous example. In the `repeat-while` loop above, `square += board[square]` is always executed *immediately after* the loop's `while` condition confirms that `square` is still on the board. This behavior removes the need for the array bounds check seen in the `while` loop version of the game described earlier.

Conditional Statements

It is often useful to execute different pieces of code based on certain conditions. You might want to run an extra piece of code when an error occurs, or to display a message when a value becomes too high or too low. To do this, you make parts of your code *conditional*.

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

```
let badStart = """
one
two
"""
let end = """
three
"""
print(badStart + end)
// Prints two lines:
// one
// twothree

let goodStart = """
one
two

"""
print(goodStart + end)
// Prints three lines:
// one
// two
// three
```

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

String Interpolation

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

```
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's included inside the string literal.

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

Unicode

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

Unicode Scalars

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

> NOTE
>
> A Unicode scalar is any Unicode *code point* in the range `U+0000` to `U+D7FF` inclusive or `U+E000` to `U+10FFFF` inclusive. Unicode scalars don't include the Unicode *surrogate pair* code points, which are the code points in the range `U+D800` to `U+DFFF` inclusive.

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

Extended Grapheme Clusters

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

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

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

```
let eAcute: Character = "\u{E9}"                         // é
let combinedEAcute: Character = "\u{65}\u{301}"          // e followed by
  ´
// eAcute is é, combinedEAcute is é
```



```
var square = 0
var diceRoll = 0
while square < finalSquare {
    // roll the dice
    diceRoll += 1
    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]
    }
}
print("Game over!")
```

The example above uses a very simple approach to dice rolling. Instead of generating a random number, it starts with a `diceRoll` value of `0`. Each time through the `while` loop, `diceRoll` is incremented by one and is then checked to see whether it has become too large. Whenever this return value equals `7`, the dice roll has become too large and is reset to a value of `1`. The result is a sequence of `diceRoll` values that is always `1`, `2`, `3`, `4`, `5`, `6`, `1`, `2` and so on.

After rolling the dice, the player moves forward by `diceRoll` squares. It's possible that the dice roll may have moved the player beyond square 25, in which case the game is over. To cope with this scenario, the code checks that `square` is less than the `board` array's `count` property. If `square` is valid, the value stored in `board[square]` is added to the current `square` value to move the player up or down any ladders or snakes.

> NOTE
>
> If this check is not performed, `board[square]` might try to access a value outside the bounds of the `board` array, which would trigger a runtime error.

The current `while` loop execution then ends, and the loop's condition is checked to see if the loop should be executed again. If the player has moved on or beyond square number `25`, the loop's condition evaluates to `false` and the game ends.

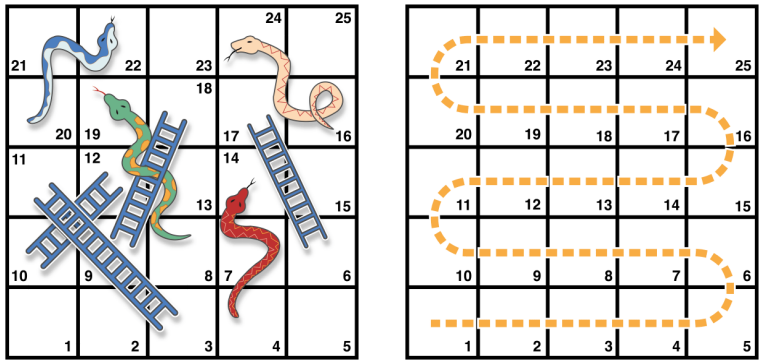
A `while` loop is appropriate in this case, because the length of the game is not clear at the start of the `while` loop. Instead, the loop is executed until a particular condition is satisfied.

Repeat-While

The other variation of the `while` loop, known as the `repeat-while` loop, performs a single pass through the loop block first, *before* considering the loop's condition. It then continues to repeat the loop until the condition is `false`.

> NOTE
>
> The `repeat-while` loop in Swift is analogous to a `do-while` loop in other languages.

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



The rules of the game are as follows:

- The board has 25 squares, and the aim is to land on or beyond square 25.
- The player's starting square is "square zero", which is just off the bottom-left corner of the board.
- Each turn, you roll a six-sided dice and move by that number of squares, following the horizontal path indicated by the dotted arrow above.
- If your turn ends at the bottom of a ladder, you move up that ladder.
- If your turn ends at the head of a snake, you move down that snake.

The game board is represented by an array of `Int` values. Its size is based on a constant called `finalSquare`, which is used to initialize the array and also to check for a win condition later in the example. Because the players start off the board, on "square zero", the board is initialized with 26 zero `Int` values, not 25.

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

Some squares are then set to have more specific values for the snakes and ladders. Squares with a ladder base have a positive number to move you up the board, whereas squares with a snake head have a negative number to move you back down the board.

```
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`). To align the values and statements, the unary plus operator (`+i`) is explicitly used with the unary minus operator (`-i`) and numbers lower than `10` are padded with zeros. (Neither stylistic technique is strictly necessary, but they lead to neater code.)

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

```
let precomposed: Character = "\u{D55C}"                  // 한
let decomposed: Character = "\u{1112}\u{1161}\u{11AB}"   // ㅎ, ㅏ, ㄴ
// precomposed is 한, decomposed is 한
```

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

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

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

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

Counting Characters

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

```
let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
print("unusualMenagerie has \(unusualMenagerie.count) characters")
// Prints "unusualMenagerie has 40 characters"
```

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

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

```
var word = "cafe"
print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in cafe is 4"

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

print("the number of characters in \(word) is \(word.count)")
// Prints "the number of characters in café is 4"
```

NOTE

> Extended grapheme clusters can be composed of multiple Unicode scalars. This means that different characters—and different representations of the same character—can require different amounts of memory to store. Because of this, characters in Swift don't each take up the same amount of memory within a string's representation. As a result, the number of characters in a string can't be calculated without iterating through the string to determine its extended grapheme cluster boundaries. If you are working with particularly long string values, be aware that the `count` property must iterate over the Unicode scalars in the entire string in order to determine the characters for that string.
>
> The count of the characters returned by the `count` property isn't always the same as the `length` property of an `NSString` that contains the same characters. The length of an `NSString` is based on the number of 16-bit code units within the string's UTF-16 representation and not the number of Unicode extended grapheme clusters within the string.

Accessing and Modifying a String

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

String Indices

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

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

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

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

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

```
let greeting = "Guten Tag!"
greeting[greeting.startIndex]
// G
greeting[greeting.index(before: greeting.endIndex)]
// !
greeting[greeting.index(after: greeting.startIndex)]
// u
let index = greeting.index(greeting.startIndex, offsetBy: 7)
greeting[index]
// a
```

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

In some situations, you might not want to use closed ranges, which include both endpoints. Consider drawing the tick marks for every minute on a watch face. You want to draw `60` tick marks, starting with the `0` minute. Use the half-open range operator (`..<`) to include the lower bound but not the upper bound. For more about ranges, see Range Operators.

```
let minutes = 60
for tickMark in 0..<minutes {
    // render the tick mark each minute (60 times)
}
```

Some users might want fewer tick marks in their UI. They could prefer one mark every `5` minutes instead. Use the `stride(from:to:by:)` function to skip the unwanted marks.

```
let minuteInterval = 5
for tickMark in stride(from: 0, to: minutes, by: minuteInterval) {
    // render the tick mark every 5 minutes (0, 5, 10, 15 ... 45, 50, 55)
}
```

Closed ranges are also available, by using `stride(from:through:by:)` instead:

```
let hours = 12
let hourInterval = 3
for tickMark in stride(from: 3, through: hours, by: hourInterval) {
    // render the tick mark every 3 hours (3, 6, 9, 12)
}
```

While Loops

A `while` loop performs a set of statements until a condition becomes `false`. These kinds of loops are best used when the number of iterations is not known before the first iteration begins. Swift provides two kinds of `while` loops:

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

While

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

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

```
while condition {
    statements
}
```

This example plays a simple game of *Snakes and Ladders* (also known as *Chutes and Ladders*):

The contents of a `Dictionary` are inherently unordered, and iterating over them does not guarantee the order in which they will be retrieved. In particular, the order you insert items into a `Dictionary` doesn't define the order they are iterated. For more about arrays and dictionaries, see Collection Types.

You can also use `for-in` loops with numeric ranges. This example prints the first few entries in a five-times table:

```
for index in 1...5 {
    print("\(index) times 5 is \(index * 5)")
}
// 1 times 5 is 5
// 2 times 5 is 10
// 3 times 5 is 15
// 4 times 5 is 20
// 5 times 5 is 25
```

The sequence being iterated over is a range of numbers from `1` to `5`, inclusive, as indicated by the use of the closed range operator (`...`). The value of `index` is set to the first number in the range (`1`), and the statements inside the loop are executed. In this case, the loop contains only one statement, which prints an entry from the five-times table for the current value of `index`. After the statement is executed, the value of `index` is updated to contain the second value in the range (`2`), and the `print(_:separator:terminator:)` function is called again. This process continues until the end of the range is reached.

In the example above, `index` is a constant whose value is automatically set at the start of each iteration of the loop. As such, `index` does not have to be declared before it is used. It is implicitly declared simply by its inclusion in the loop declaration, without the need for a `let` declaration keyword.

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

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

The example above calculates the value of one number to the power of another (in this case, `3` to the power of `10`). It multiplies a starting value of `1` (that is, `3` to the power of `0`) by `3`, ten times, using a closed range that starts with `1` and ends with `10`. For this calculation, the individual counter values each time through the loop are unnecessary—the code simply executes the loop the correct number of times. The underscore character (_) used in place of a loop variable causes the individual values to be ignored and does not provide access to the current value during each iteration of the loop.


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

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

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

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

Inserting and Removing

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

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

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

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

```
welcome.remove(at: welcome.index(before: welcome.endIndex))
// welcome now equals "hello there"

let range = welcome.index(welcome.endIndex, offsetBy: -6)..
  <welcome.endIndex
welcome.removeSubrange(range)
// welcome now equals "hello"
```

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

Substrings

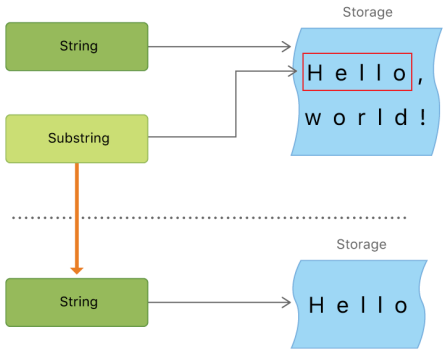
When you get a substring from a string—for example, using a subscript or a method like `prefix(_:)`—the result is an instance of `Substring`, not another string. Substrings in Swift have most of the same methods as strings, which means you can work with substrings the same way you work with strings.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

```
let greeting = "Hello, world!"
let index = greeting.firstIndex(of: ",") ?? greeting.endIndex
let beginning = greeting[..<index]
// beginning is "Hello"

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

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

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



NOTE

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

Comparing Strings

Control Flow

Swift provides a variety of control flow statements. These include `while` loops to perform a task multiple times; `if`, `guard`, and `switch` statements to execute different branches of code based on certain conditions; and statements such as `break` and `continue` to transfer the flow of execution to another point in your code.

Swift also provides a `for-in` loop that makes it easy to iterate over arrays, dictionaries, ranges, strings, and other sequences.

Swift's `switch` statement is considerably more powerful than its counterpart in many C-like languages. Cases can match many different patterns, including interval matches, tuples, and casts to a specific type. Matched values in a `switch` case can be bound to temporary constants or variables for use within the case's body, and complex matching conditions can be expressed with a `where` clause for each case.

For-In Loops

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

This example uses a `for-in` loop to iterate over the items in an array:

```
let names = ["Anna", "Alex", "Brian", "Jack"]
for name in names {
    print("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 the body of the `for-in` loop. In the code example below, the dictionary's keys are decomposed into a constant called `animalName`, and the dictionary's values are decomposed into a constant called `legCount`.

```
let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
for (animalName, legCount) in numberOfLegs {
    print("\(animalName)s have \(legCount) legs")
}
// ants have 6 legs
// cats have 4 legs
// spiders have 8 legs
```

```swift
if let removedValue = airports.removeValue(forKey: "DUB") {
    print("The removed airport's name is \(removedValue).")
} else {
    print("The airports dictionary does not contain a value for DUB.")
}
// Prints "The removed airport's name is Dublin Airport."
```

Iterating Over a Dictionary

You can iterate over the key-value pairs in a dictionary with a `for-in` loop. Each item in the dictionary is returned as a `(key, value)` tuple, and you can decompose the tuple's members into temporary constants or variables as part of the iteration:

```swift
for (airportCode, airportName) in airports {
    print("\(airportCode): \(airportName)")
}
// YYZ: Toronto Pearson
// LHR: London Heathrow
```

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

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

```swift
for airportCode in airports.keys {
    print("Airport code: \(airportCode)")
}
// Airport code: YYZ
// Airport code: LHR

for airportName in airports.values {
    print("Airport name: \(airportName)")
}
// Airport name: Toronto Pearson
// 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:

```swift
let airportCodes = [String](airports.keys)
// airportCodes is ["YYZ", "LHR"]

let airportNames = [String](airports.values)
// airportNames is ["Toronto Pearson", "London Heathrow"]
```

Swift's `Dictionary` type does not have a defined ordering. To iterate over the keys or values of a dictionary in a specific order, use the `sorted()` method on its `keys` or `values` property.

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

String and Character Equality

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

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

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

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

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

// "Voulez-vous un café?" using LATIN SMALL LETTER E and COMBINING ACUTE
  ACCENT
let combinedEAcuteQuestion = "Voulez-vous un caf\u{65}\u{301}?"

if eAcuteQuestion == combinedEAcuteQuestion {
    print("These two strings are considered equal")
}
// Prints "These two strings are considered equal"
```

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

```swift
let latinCapitalLetterA: Character = "\u{41}"

let cyrillicCapitalLetterA: Character = "\u{0410}"

if latinCapitalLetterA != cyrillicCapitalLetterA {
    print("These two characters are not equivalent.")
}
// Prints "These two characters are not equivalent."
```

NOTE

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

Prefix and Suffix Equality

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

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

```
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 += 1
    }
}
print("There are \(act1SceneCount) scenes in Act 1")
// Prints "There are 5 scenes in Act 1"
```

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

```
var mansionCount = 0
var cellCount = 0
for scene in romeoAndJuliet {
    if scene.hasSuffix("Capulet's mansion") {
        mansionCount += 1
    } else if scene.hasSuffix("Friar Lawrence's cell") {
        cellCount += 1
    }
}
print("\(mansionCount) mansion scenes; \(cellCount) cell scenes")
// Prints "6 mansion scenes; 2 cell scenes"
```



```
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 Airport", forKey: "DUB") {
    print("The old value for DUB was \(oldValue).")
}
// Prints "The old value for DUB was Dublin."
```

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

```
if let airportName = airports["DUB"] {
    print("The name of the airport is \(airportName).")
} else {
    print("That airport is not in the airports dictionary.")
}
// Prints "The name of the airport is Dublin Airport."
```

You can use subscript syntax to remove a key-value pair from a dictionary by assigning a value of `nil` for that key:

```
airports["APL"] = "Apple International"
// "Apple International" is not the real airport for APL, so delete it
airports["APL"] = nil
// APL has now been removed from the dictionary
```

Alternatively, remove a key-value pair from a dictionary with the `removeValue(forKey:)` method. This method removes the key-value pair if it exists and returns the removed value, or returns `nil` if no value existed:

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

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

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

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

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

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

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

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

Accessing and Modifying a Dictionary

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

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

```
print("The airports dictionary contains \(airports.count) items.")
// Prints "The airports dictionary contains 2 items."
```

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

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

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

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

Unicode Representations of Strings

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

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

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

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

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

```
let dogString = "Dog‼🐶"
```

UTF-8 Representation

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

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

```swift
for codeUnit in dogString.utf8 {
    print("\(codeUnit) ", terminator: "")
}
print("")
// Prints "68 111 103 226 128 188 240 159 144 182 "
```

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

UTF-16 Representation

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

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

```swift
for codeUnit in dogString.utf16 {
    print("\(codeUnit) ", terminator: "")
}
print("")
// Prints "68 111 103 8252 55357 56374 "
```

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

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

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

Unicode Scalar Representation

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

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

Dictionary Type Shorthand Syntax

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

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

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

Creating an Empty Dictionary

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

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

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

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

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

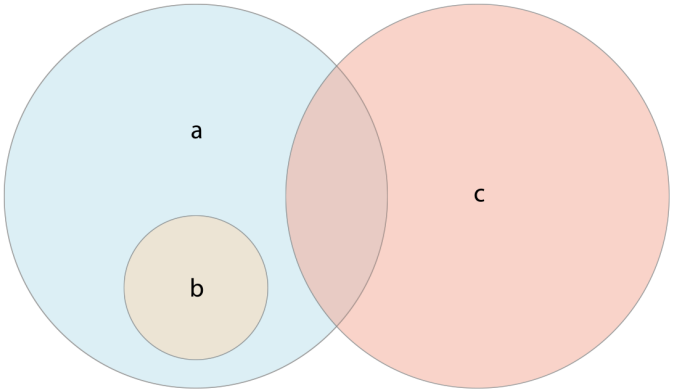
Creating a Dictionary with a Dictionary Literal

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



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

```
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐮", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐦", "🐭"]

houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
farmAnimals.isDisjoint(with: cityAnimals)
// true
```

Dictionaries

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

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

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

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

```
for scalar in dogString.unicodeScalars {
    print("\(scalar.value) ", terminator: "")
}
print("")
// Prints "68 111 103 8252 128054 "
```

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

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

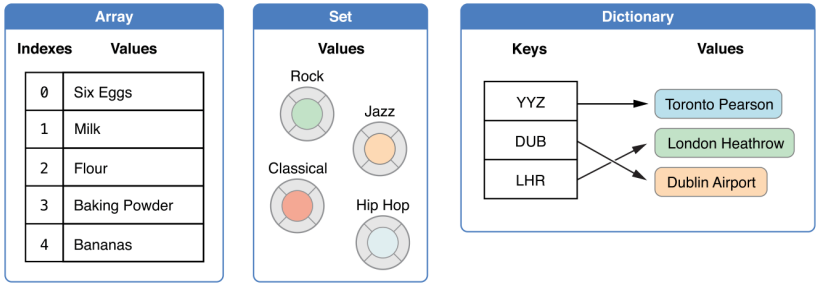
The `value` property of the fifth and final `UnicodeScalar`, `128054`, is a decimal equivalent of the hexadecimal value `1F436`, which represents the Unicode scalar `U+1F436` for the `DOG FACE` character.

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

```
for scalar in dogString.unicodeScalars {
    print("\(scalar) ")
}
// D
// o
// g
// ‼
// 🐶
```

Collection Types

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



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

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

Mutability of Collections

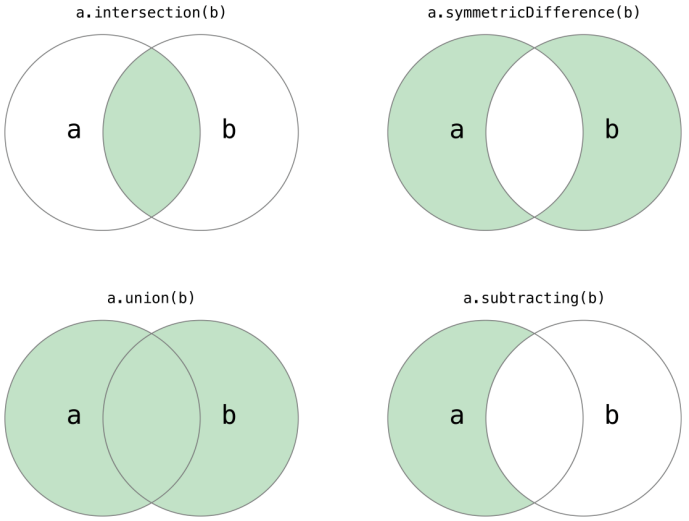
If you create an array, a set, or a dictionary, and assign it to a variable, the collection that is created will be *mutable*. This means that you can change (or *mutate*) the collection after it's created by adding, removing, or changing items in the collection. If you assign an array, a set, or a dictionary to a constant, that collection is *immutable*, and its size and contents cannot be changed.

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

Arrays

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

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



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

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

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

Set Membership and Equality

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

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

```
for genre in favoriteGenres {
    print("\(genre)")
}
// Classical
// Jazz
// Hip hop
```

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

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

```
for genre in favoriteGenres.sorted() {
    print("\(genre)")
}
// Classical
// Hip hop
// Jazz
```

Performing Set Operations

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

Fundamental Set Operations

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

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

Array Type Shorthand Syntax

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

Creating an Empty Array

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

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

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

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

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

Creating an Array with a Default Value

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

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

Creating an Array by Adding Two Arrays Together

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

```swift
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)
// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]

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

Creating an Array with an Array Literal

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

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

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

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

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

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

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

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

```swift
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:

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



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

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

Accessing and Modifying a Set

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

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

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

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

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

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

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

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

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

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

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

Iterating Over a Set

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

Set Type Syntax

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

Creating and Initializing an Empty Set

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

```
var letters = Set<Character>()
print("letters is of type Set<Character> with \(letters.count) items.")
// Prints "letters is of type Set<Character> with 0 items."
```

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

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

```
letters.insert("a")
// letters now contains 1 value of type Character
letters = []
// letters is now an empty set, but is still of type Set<Character>
```

Creating a Set with an Array Literal

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

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

```
var favoriteGenres: Set<String> = ["Rock", "Classical", "Hip hop"]
// favoriteGenres has been initialized with three initial items
```

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

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

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

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

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

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

```
shoppingList.append("Flour")
// shoppingList now contains 3 items, and someone is making pancakes
```

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

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

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

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

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

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

```
shoppingList[0] = "Six eggs"
// the first item in the list is now equal to "Six eggs" rather than
  "Eggs"
```

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

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

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

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

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

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

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

```
let mapleSyrup = shoppingList.remove(at: 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
```

> NOTE
>
> If you try to access or modify a value for an index that is outside of an array's existing bounds, you will trigger a runtime error. You can check that an index is valid before using it by comparing it to the array's `count` property. The largest valid index in an array is `count - 1` because arrays are indexed from zero—however, when `count` is `0` (meaning the array is empty), there are no valid indexes.

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

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

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

```
let apples = shoppingList.removeLast()
// the last item in the array has just been removed
// shoppingList now contains 5 items, and no apples
// 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 {
    print(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```


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

```
for (index, value) in shoppingList.enumerated() {
    print("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-In Loops.

Sets

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

> NOTE
>
> Swift's `Set` type is bridged to Foundation's `NSSet` class.
>
> For more information about using `Set` with Foundation and Cocoa, see Bridging Between Set and NSSet.

Hash Values for Set Types

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

All of Swift's basic types (such as `String`, `Int`, `Double`, and `Bool`) are hashable by default, and can be used as set value types or dictionary key types. Enumeration case values without associated values (as described in Enumerations) are also hashable by default.

> NOTE
>
> You can use your own custom types as set value types or dictionary key types by making them conform to the `Hashable` protocol from Swift's standard library. Types that conform to the `Hashable` protocol must provide a gettable `Int` property called `hashValue`. The value returned by a type's `hashValue` property is not required to be the same across different executions of the same program, or in different programs.
>
> Because the `Hashable` protocol conforms to `Equatable`, conforming types must also provide an implementation of the equals operator (==). The `Equatable` protocol requires any conforming implementation of == to be an equivalence relation. That is, an implementation of == must satisfy the following three conditions, for all values `a`, `b`, and `c`:
>
> - `a == a` (Reflexivity)
> - `a == b` implies `b == a` (Symmetry)
> - `a == b && b == c` implies `a == c` (Transitivity)