# SWIFT

## The Basics

A type-safe language.

### Constants and Variables

The value of a constant can’t be changed once it’s set.

#### Declaring Constants and Variables

Declare constants with the let keyword and variables with the var keyword.

**let maximumAttempts : Int = 10**

**var currentLoginAttempt : Int = 0**

### Integers

Swift provides signed and unsigned integers in 8, 16, 32 and 64 bit forms.

#### Integer Bounds

Can be accessed using min and max properties.

**let minValue = UInt8.min**

**let maxValue = UInt8.max**

#### Int

The integer type, Int, has the same size as the current platform’s native word size.

#### UInt

Unsigned integer type.

### Floating-Point Numbers

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

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

### Numeric Type Conversion

<skipped>

### Type Aliases

Type Aliases define an alternative name for an existing type. They are defined using the typealias keyword.

**typealias AudioSample = UInt16**

### Booleans

Represents logical values of true or false.

### Tuples

Tuples group multiple values into a single compound value. E.g.

**let http404Error = (404, "Not Found")**

You can decompose a tuple’s contents into separate constants or variables, which you the access.

**let (statusCode, statusMessage) = http404Error**

**print("code is \(statusCode)")**

or with index numbers starting at 0.

**print("code is \(http404Error.0)")**

**// Prints "The status code is 404"**

**print("message is \(http404Error.1)")**

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

### Optionals

An optional represents two possibilities; either there is a value, or you can unwrap the optional to access that value, or there isn’t a value at all.

**var serverResponseCode: Int? = 404**

**// the question mark denotes that the type is an optional. This variable can be assigned a value of nil without giving out exceptions.**

## Basic Operators

## Strings and Characters

## Collection Types

There are three primary *collection types*. **Arrays**, **sets** and **dictionaries**.

* **Arrays** are ordered collection of values.
* **Sets** are unordered collection of unique values.
* **Dictionaries** are unordered collections of key-value associations.

They are store values of uniform type. (You cannot insert a value of the wrong type)

### Mutability of Collections

They are all ***mutable***.

If you assign an array, set or dictionary to a constant, that collection is ***immutable***.

### Arrays

Arrays store values of the *same type* in an ordered list. Values don’t have to be unique.

#### Array Type Shorthand Syntax

Array types are written as Array<Element> where Element is the type of values the array is allowed to store.

The short hand form is [Element].

**var someInts = [Int]() // Empty array**

**someInts = [] // valid since type can be inferred.**

#### Creating an Array with a Default Value

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

#### Creating an Array by Adding Two Arrays Together

**var arrDbls = arr1 + arr2**

Creating an Array with an Array Literal

You can initialize an array with an array literal.

**var shoppingList: [String] = ["Eggs", "Milk"] // or**

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

Accessing and Modifying an Array

count property is the number of items.

**shoppingList.count**

isEmpty property checks whether the count is equal to 0.

**shoppingList.isEmpty**

append() method can add a new item to the end of an array. Or the += (addition assignment) operator.

**shoppingList.append(“Flour”)**

**shoppingList += ["Baking Powder"]**

Retrieve the value of an array by using subscript syntax.

**var firstItem = shoppingList[0]**

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

You can also use the subscript syntax to change a range of values at once.

**shoppingList[4...6] = ["Bananas", "Apples"]**

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

**shopList.insert("Maple Syrup", at: 0)**

removeLast() removes the final item from an array.

#### Iterating Over an Array

**for item in shoppingList {**

**print(item)**

**}**

**for (index, value) in shoppingList.enumerated() {**

**print("Item \(index + 1): \(value)")**

**}**

**// Item 1: Six eggs**

### Sets

A **set** stores ***distinct*** values of the same type in a collection with ***no defined ordering***.

#### Hash Values for Set Types

A type must be hashable in order to be stored in a set. A hash value is an int value that’s the same for all objects that compare equally. If a == b, the hash value of a is equal to the hash value of b. Basic types are hashable by default.

* Declaration -> Array<Element>

There’s no shorthand form.

**var letters = Set<Character>()**

#### Creating a Set with an Array Literal

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

If you are initializing a set with an array literal that contains values of just on type, a shorter initialization form could be used.

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

#### Accessing and Modifying as Set

Sets have count, isEmpty, insert(\_:), remove(\_:) members that arrays have.

removeAll() method removes all items.

contains(\_:) method checks whether a set contains a particular item.

**if favoriteGenres.contains("Funk") {**

**print("I get up on the good foot.")**

**}**

#### Iterating over a set

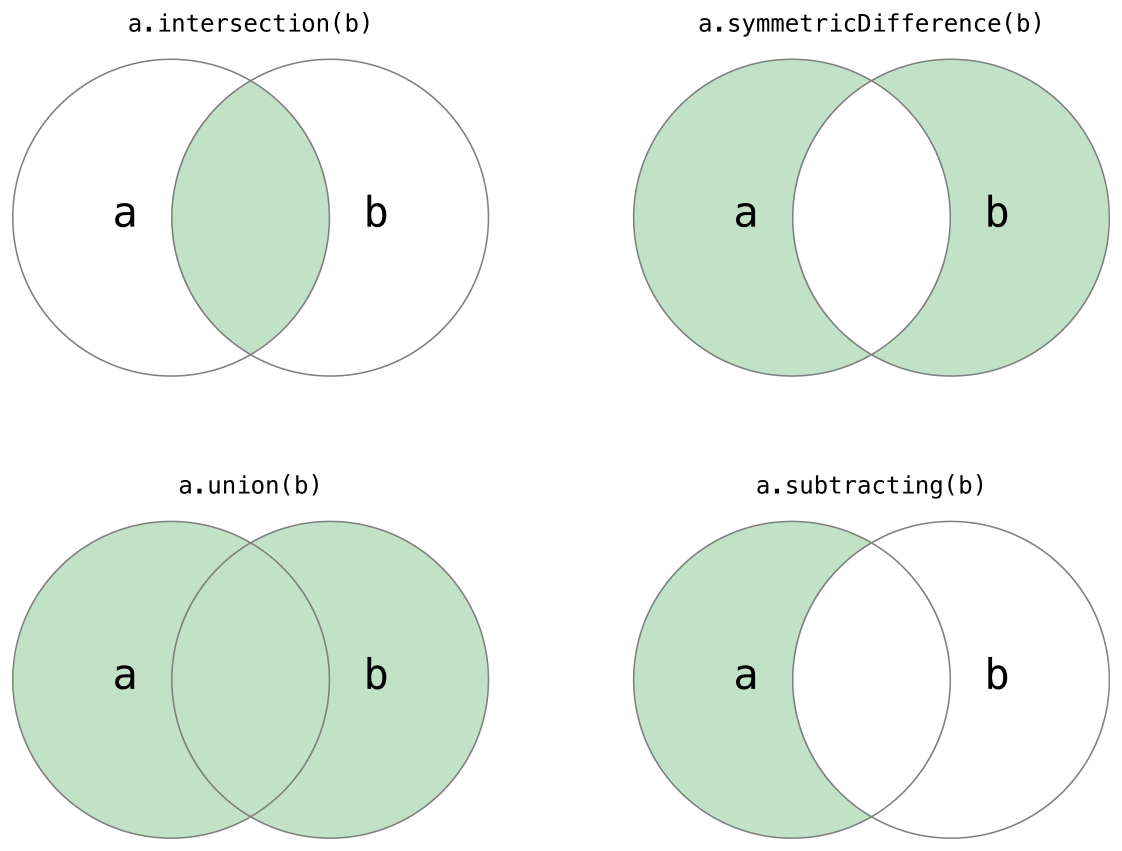
**for genre in favoriteGenres {**

**print("\(genre)")**

**}**

### Performing Set Operations

#### Fundamental Set Operations



* Intersection(\_:) creates a new set with only the values common to both sets.
* symetricDifference(\_:) creates a new set with values in either set but not both.
* Union(\_:) creates a new set with all of the values in both sets.
* 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()**

**// []**

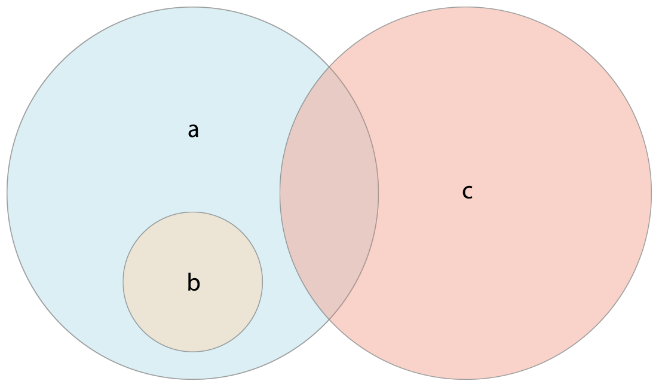
**oddDigits.subtracting(singleDigitPrimeNumbers).sorted()**

**// [1, 9]**

**oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()**

**// [1, 2, 9]**

#### Set Membership and Equality



The set a is a *superset* of b.

The set b is a *subset* of a.

The set b and c are *disjoint*.

* 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.

### Dictionaries

A **dictionary** stores unordered associations between keys of the same type and values of the same type in a collection.

It is written as Dictionary<Key, Value>, or [key: Value]. Key type must be hashable (conform to the Hashable protocol).

Creating an Empty Dictionary

**var namesOfIntegers = [Int: String]()**

**or**

**namesOfIntegers[16] = "sixteen"**

**// namesOfIntegers now contains 1 key-value pair**

**namesOfIntegers = [:]**

**// namesOfIntegers is once again an empty dictionary of type [Int: String]**

or with a dictionary literal.

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

E.g

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

or with type inference

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

#### Accessing and Modifying a Dictionary

count, isEmpty, work in dictionaries as they do in arrays.

updateValue(\_:forKey:) method sets or updates the value of a particular key or use the subscript syntax -:

**airports["LHR"] = "London"**

Remove a key-value pair from a dictionary by assigning a value of nil for that key: or with the removeValue(forKey:) method.

#### Iterating Over a Dictionary

A for-in loop will return dictionary items as a (key, value) tuple.

**for (airportCode, airportName) in airports {**

**print("\(airportCode): \(airportName)")**

**}**

Use the sorted() method on its keys or values property to iterate in a specific order.

## Control Flow

These include **while** loops, **if**, **guard**, **switch**, **break** and **continue** statements. **For-in** loops is provided to iterate over collections.

### For-In Loops

They iterate over a sequence, such as items in an array, ranges of numbers, or characters in a string.

**let names = ["Anna", "Alex", "Brian", "Jack"]**

**for name in names {**

**print("Hello, \(name)!") }**

For loop with dictionaries

**let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]**

**for (animalName, legCount) in numberOfLegs {**

**print("\(animalName)s have \(legCount) legs")**

For-in loops with numeric ranges

**for index in 1...5 {**

**print("\(index) times 5 is \(index \* 5)")**

**}**

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.

**for \_ in 1...10 {**

**answer \*= base**

**}**

You may not want to include closed ranges (include endpoints). Use the half-open range operator **(..<)** to include the lower bound.

**let minutes = 60**

**for tickMark in 0..< minutes{...}**

Use the stride(from:to:by:) function to skip unwanted indexes.

**let minuteInterval = 5**

**for tickMark in**

**stride( from: 0,**

**to: minutes,**

**by: minuteInterval){...}**

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

**let hours = 12**

**let hourInterval = 3**

**for tickMark in**

**stride(from: 3,**

**through: hours,**

**by: hourInterval) {...}**

### While loops

Performs a set of statements until a condition becomes false. There are two kinds.

* 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.

### Conditional Statements

Swift has two ways to add conditional branches to your code.

* If - statements
* switch – statements

#### if

May contain multiple conditions. Evaluates statements if those conditions evaluate to true.

**var temperature = 30**

**if temperature <= 32 {...}**

#### Switch

A switch statement considers a value and compares it against several possible matching patterns.

**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 must be exhaustive i.e. every possible value of the type being considered must be matched by one of the switch cases.

You define a default case to cover any values that aren’t addressed explicitly. The default case must always appear last.

switch statements in Swift don’t fall through the bottom of each case and into the next one by default.

The body of each case ***must*** contain at least one executable statement.

To make a switch statement that covers more than one case, combine the cases into a compound case, separating the values with commas.

#### Interval matching

Values in switch cases can be checked for their inclusion in an interval.

**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"**

**}**

#### Tuples

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

**let somePoint = (1, 1)**

**switch somePoint {**

**case (0, 0):**

**print("\(somePoint) origin")**

**case (\_, 0):**

**print("\(somePoint) x-axis")**

**case (0, \_):**

**print("\(somePoint) y-axis")**

**case (-2...2, -2...2):**

**print("\(somePoint) inside the box")**

**default:**

**print("\(somePoint) outside of the box")**

**}**

Swift allows multiple switch cases to cover the same value or values. If multiple matches are possible, the first matching case is always used.

#### 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 is known as value binding.

**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))")**

**}**

#### Where

A switch case can use a where clause to check for additional conditions. The switch case matches only if the where clause’s condition evaluates to true for that value.

**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")**

**}**

#### 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.

**case "a", "e", "i", "o", "u":**

Example compound case with 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 the patterns in the compound case.

### 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.

* continue
* break
* fallthrough
* return
* throw

#### Continue

Tells a loop to stop and start again at the beginning of the next iteration through the loop.

#### Break

Ends execution of an entire control flow statement immediately.

Break in a loop statement, end’s the loop’s execution immediately and transfers control to the code after the loops closing brace (}).

Break in a switch statement causes the switch statement to end its execution immediately and to transfer control to the code after the switch statement’s closing brace.

#### Fallthrough

In swift, the entire switch statement completes its execution as soon as the first matching case is completed. Statements don’t fall through the bottom of each case and into the next one.

If you need C-style fallthrough behavior, you can opt in to this behavior on a case by case basis with the fallthrough keyword.

#### Labeled Statements

When nesting loops and conditional statements inside other loops and conditional statements, control transfer statements can be ambiguous. It may be useful to be explicit about which loop or conditional statement you want a break statement to terminate.

You can mark these loops or conditions with a ***statement label***. You can then the control flow statements with the labels to disambiguate which statements you are breaking out of.

**gameLoop: while square != finalSquare {...}**

### Early Exit

A guard statement, executes statements depending on the Boolean value of an expression. Use a guard statement to require that a condition must be true in order for the code after the guard statement to be executed.

A guard statement always has an else clause.

**func greet(person: [String: String]) {**

**guard let name=person["name"]else {**

**return**

**}**

**}**

### Check API Availability

A guard statement

## Functions

Every function in Swift has a type, consisting of the function’s parameter types and return types. You can use this type like any other type in Swift.

#### 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.

**func minMax(array: [Int]) -> (min: Int, max: Int) {**

#### 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 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 types closing parenthesis. (int, int)?.

#### Functions with Implicit Return

If the entire body of the function is a single expression, the function implicitly returns that expression.

**func foo(for person: String)-> String {**

**"Hello, " + person + "!"**

**}**

Any function that you write as just one return line can omit the return keyword.

### 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. The parameter name is used in the implementation of the function. By default, parameters use their parameter name as their argument label.

**func someFunction(argumentLabel parameterName: Int) { ... }**

**func greet(person: String, from hometown: String) -> String {**

**return "Hello \(person)! Glad you could visit from \(hometown)."**

**}**

**print(greet(person: "Bill", from: "Cupertino"))**

#### 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.

Default Parameter Values – Skipped

Variadic Parameters – Skipped

#### 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.

* 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 can only pass a variable as the argument for an in-out parameter.

* 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.

In-out parameters can’t have default values

Variadic parameters can’t be marked as inout.

Example.

**func swapTwoInts**

**( \_ a: inout Int,**

**\_ b: inout Int ) { ... }**

**swapTwoInts( &someInt,**

**&anotherInt )**

### Function Types

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

#### Using Function Types

You use function types just like any other types in Swift.

You can define a constant or variable to be of a function types and assign an appropriate function to that variable.

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

Now mathFunction can be used like a function that has that parameter, return type signature.

**mathFunction(2, 3)**

#### Function Types as Parameter Types

You can use a function type as a parameter type to another function.

**func printMathResult(**

**\_ mathFunction: (Int, Int) -> Int,**

**\_ a: Int,**

**\_ b: Int) {**

**print("=> \(mathFunction(a, b))")**

**}**

You can pass any function of that type as the argument for the first parameter.

#### Function Types as Return Types

You do this by writing a complete function type immediately after the return arrow of the returning function.

**func chooseStepFunction(backward: Bool) -> (Int) -> Int {**

### Nested Functions

You can define functions inside the bodies of other functions, known as **nested functions**.