Swift Cheat Sheet

This is not meant to be a beginner's guide or a detailed discussion about Swift; it is meant to be a quick reference to common, high level topics.

- Read the Objective-C (https://github.com/iwasrobbed/Objective-C-CheatSheet) cheatsheet as well.
- To download a PDF version of this, use GitPrint.com (https://gitprint.com/iwasrobbed/Swift-CheatSheet/blob/master/README.md?download)

Note: This was written this fairly quickly, mostly to teach myself Swift, so it still needs a lot of love and there are important sections still missing. Please feel free to edit this document to update or improve upon it, making sure to keep with the general formatting of the document. The list of contributors can be found here (https://github.com/iwasrobbed/Swift-CheatSheet/graphs/contributors) .

If something isn't mentioned here, it's probably covered in detail in one of these:

- Apple: A Swift Tour (https://developer.apple.com/library/mac/documentation/Swift/Conceptual/Swift_Programming_Language/GuidedTour.html)
- Apple: Swift Programming Language (https://developer.apple.com/library/mac/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-XID_467)
- Apple iBooks: Swift Programming Language (https://itunes.apple.com/us/book/swift-programming-language/id881256329?mt=11)
- Apple: Using Swift with Objective-C and Cocoa (https://developer.apple.com/library/ios/documentation/swift/conceptual/buildingcocoaapps/index.html)
- objc.io (http://www.objc.io)
- NSHipster (http://nshipster.com)
- Functional Programming in Swift (http://www.objc.io/books/)

Table of Contents

- Commenting
- Data Types
- Operators
- Operator Overloading
- Declaring Classes
- Declarations
- Literals
- Functions
- Constants and Variables
- Naming Conventions
- Closures
- Generics
- Control Statements
- Extending Classes
- Error Handling
- Passing Information
- User Defaults
- Common Patterns
- Unicode Support

Commenting

Comments should be used to organize code and to provide extra information for future refactoring or for other developers who might be reading your code. Comments are ignored by the compiler so they do not increase the compiled program size.

Two ways of commenting:

//%20This%20is%20an%20inline%20comment%0A%0A/*%20This%20is%20a%20block%20comment%0A%20%20%20and%20it%20can%20span%20multiple%20li

Using `MARK` to organize your code:

//%20MARK%3A%20-
%20Use%20mark%20to%20logically%20organize%20your%20code%0A%0A//%20Declare%20some%20functions%20or%20variables%20here%0A%0A//%20MARK%
%20They%20also%20show%20up%20nicely%20in%20the%20properties/functions%20list%20in%20Xcode%0A%0A//%20Declare%20some%20more%20functions%

Back to top

Data Types

Size

Permissible sizes of data types are determined by how many bytes of memory are allocated for that specific type and whether it's a 32-bit or 64-bit environment. In a 32-bit environment, `1ong` is given 4 bytes, which equates to a total range of `2^(4*8)` (with 8 bits in a byte) or `4294967295`. In a 64-bit

environment, `long` is given 8 bytes, which equates to `2^(8*8)` or `1.84467440737096e19`.

For a complete guide to 64-bit changes, please see the transition document (https://developer.apple.com/library/mac/documentation/Darwin/Conceptual/64bitPorting/transition/transition.html#//apple_ref/doc/uid/TP40001064-CH207-TPXREF101) .

C Primitives

Unless you have a good reason to use C primitives, you should just use the Swift types to ensure compability going foward.

In fact, Swift just aliases C types to a Swift equivalent:

//%20C%20char%20is%20aliased%20as%20an%20Int8%20and%20unsigned%20as%20UInt8%0Alet%20aChar%20%3D%20CChar%28%29%0Alet%20anUnsignedCh
128%20and%20max%3A%20127%0Aprintln%28%22C%20unsigned%20char%20size%3A%20%5C%28sizeofValue%28anUnsignedChar%29%29%20with%20min%3A
32768%20and%20max%3A%2032767%0Aprintln%28%22C%20unsigned%20short%20size%3A%20%5C%28sizeofValue%28unsignedShort%29%29%20with%20min
2147483648%20and%20max%3A%202147483647%0Aprintln%28%22C%20unsigned%20int%20size%3A%20%5C%28sizeofValue%28unsignedInt%29%29%20with%
92233720368547758O8%20and%20max%3A%2092233720368547758O7%0Aprintln%28%22C%20unsigned%20long%20size%3A%20%5C%28sizeofValue%28unsigne
92233720368547758O8%20and%20max%3A%2092233720368547758O7%0Aprintln%28%22C%20unsigned%20long%20long%20size%3A%20%5C%28sizeofValue%2

From the docs (https://developer.apple.com/library/ios/documentation/swift/conceptual/buildingcocoaapps/InteractingWithCAPIs.html) :

C Type	Swift Type
bool	CBool
char, signed char	CChar
unsigned char	CUnsignedChar
short	CShort
unsigned short	CUnsignedShort
int	CInt
unsigned int	CUnsignedInt
long	CLong
unsigned long	CUnsignedLong
long long	CLongLong
unsigned long long	CUnsignedLongLong
wchar_t	CWideChar
char16_t	CChar16
char32_t	CChar32
float	CFloat
double	CDouble

Integers

Integers can be signed or unsigned. When signed, they can be either positive or negative and when unsigned, they can only be positive.

Apple states: *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` [...] is large enough for many integer ranges.*

Fixed width integer types with their accompanying byte sizes as the variable names:

//%20Exact%20integer%20types%0Alet%20aOneByteInt%3A%20Int8%20%3D%20127%0Alet%20aOneByteUnsignedInt%3A%20UInt8%20%3D%20255%0Alet%20aTw

Floating Point

Floats cannot be signed or unsigned.

//%20Single%20precision%20%2832-bit%29%20floating-point.%20Use%20it%20when%20floating-point%20values%20do%20not%20require%2064-bit%20precision.%0Alet%20aFloat%20%3D%20Float%28%29%0Aprintln%28%22Float%20size%3A%20%5C%28sizeofValue%28aFloat%29%29%22%29%0A//%20Floa
bit%29%20floating-point.%20Use%20it%20when%20floating-point%20values%20must%20be%20very%20large%20or%20particularly%20precise.%0Alet%20aDouble%20%3D%20Double%28%29%0Aprintln%28%22Double%20s

Boolean

//%20Boolean%0Alet%20isBool%3A%20Bool%20%3D%20true%20//%20Or%20false

In Objective-C comparative statements, `0` and `nil` were considered `false` and any non-zero/non-nil values were considered `true`. However, this is not the case in Swift. Instead, those objects must conform to the `BooleanType` protocol to be able to compare this way, and you'll need to directly check their value such as `if (x == 0)` or `if (object != nil)`

Primitives

nil : Used to specify a null object pointer. When classes are first initialized, all properties of the class point to `nil`.

Enum & Bitmask Types

Enumeration types can be defined as follows:

//%20Specifying%20a%20typed%20enum%20with%20a%20name%20%28recommended%20way%29%0Aenum%20UITableViewCellStyle%3A%20Int%20%7B%0A%2

Working with Bitmasks

Swift unfortunately doesn't have a nice substitute for the old `NS_OPTIONS` macro for creating bitmasks to compare to. In fact, it's downright ugly. See posts here (http://nshipster.com/rawoptionsettype/) and here (http://stackoverflow.com/a/24066171/308315) .

An example for posterity (using Nate Cook's simple generator (http://natecook.com/blog/2014/07/swift-options-bitmask-generator/)):

struct%20ExampleOptions%20%3A%20RawOptionSetType%20%3CBooleanType%3E%20%7B%0A%20%20%20%20//%20Boilerplate%0A%20%20%20%20typealias%2
%3E%20ExampleOptions%20%7B%20return%20self%28raw%29%20%7D%0A%20%20%20%20var%20rawValue%3A%20UInt%20%7B%20return%20self.value%20%

Then you'd be able to do comparisons like so:

let%20firstOption%20%3D%20ExampleOptions.OptionOne%0Alet%20multipleOptions%3A%20ExampleOptions%20%3D%20firstOption%20%7C%20.OptionTwo%0

Note: As of beta 6, `RawOptionSetType` no longer implements `BooleanType` by default, so standard bitmask checks only work if you manually conform to the `BooleanType` protocol. If you don't conform to that, then you'll have to check your `&` comparisons against `nil` rather than `Bool`.

Type Casting

Sometimes it is necessary to cast an object into a specific class or data type. Examples of this would be casting from a `Float` to an `Int` or from a `UITableViewCell` to a subclass such as `RPTableViewCell`.

Checking Types

Swift uses `is` and `as` both for checking object types as well as conformance to a given protocol.

Operator: is

Checking object type using `is`:

if%20item%20is%20Movie%20%7B%0A%20%20%20%20++movieCount%0A%20%20%20%20println%28%22It%20is%20a%20movie.%22%29%0A%7D%20else%20if

The `is` operator returns `true`if an instance is of that object type, or conforms to the specified protocol, and returns `false` if it does not.

Operators: as? and as

If you want to be able to easily access the data during one of these checks, you can use `as?` to optionally (or `as` to force) unwrap the object when necessary:

for%20item%20in%20library%20%7B%0A%20%20%20%20if%20let%20movie%20%3D%20item%20as%3F%20Movie%20%7B%0A%20%20%20%20%20%20%20%20p

The `as?` version of the downcast operator returns an optional value of the object or protocol's type, and this value is `nil` if the downcast fails or this instance does not conform to the specified protocol.

The `as` version of the downcast operator forces the downcast to the specified object or protocol type and triggers a runtime error if the downcast does not succeed.

Casting from Generic Types

If you're working with `AnyObject` objects given from the Cocoa API, you can use:

for%20movie%20in%20someObjects%20as%20%5BMovie%5D%20%7B%0A%20%20%20%20//%20do%20stuff%0A%7D

If given an array with `Any` objects, you can use a `switch` statement with the type defined for each `case`:

var%20things%20%3D%20%5BAny%5D%28%29%0A%0Afor%20thing%20in%20things%20%7B%0A%20%20%20%20switch%20thing%20%7B%0A%20%20%20%20%20

Basic Casting

Swift also offers some simple methods of casting between it's given data types.

//%20Example%201%3A%0Alet%20aDifferentDataType%3A%20Float%20%3D%203.14%0Alet%20anInt%3A%20Int%20%3D%20Int%28aDifferentDataType%29%0A%0

Back to top

Operators

Swift supports most standard C operators and improves several capabilities to eliminate common coding errors. The assignment operator `=` does not return a value, to prevent it from being mistakenly used when the equal to operator `==` is intended.

Arithmetic operators (`+`, `-`, `*`, `/`, `%` and so forth) detect and disallow value overflow, to avoid unexpected results when working with numbers that become larger or smaller than the allowed value range of the type that stores them.

Arithmetic Operators

Operator	Purpose

Operator	Purpose
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder

Note: Unlike the remainder operator in C and Objective-C, Swift's remainder operator can also operate on floating-point numbers (e.g. `8 % 2.5 // equals 0.5`)

Comparative Operators

Operator	Purpose
==	Equal to
===	Identical to
!=	Not equal to
!==	Not identical to
~=	Pattern match
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Assignment Operators

Operator	Purpose
=	Assign
+=	Addition
-=	Subtraction
*=	Multiplication
/=	Division
%=	Remainder
&=	Bitwise AND
|=	Bitwise Inclusive OR
^=	Exclusive OR
<<=	Shift Left
>>=	Shift Right
&&=	Logical AND
||=	Logical OR

Increment and Decrement Operators

Operator	Purpose
++	Addition
--	Subtraction

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

Logical Operators

Operator	Purpose
!	NOT
&&	Logical AND
||	Logical OR

Range Operators

Operator	Purpose
..<	Half-open range
...	Closed range

Bitwise Operators

Operator	Purpose
&	Bitwise AND
|	Bitwise Inclusive OR
^	Exclusive OR
~	Unary complement (bit inversion)
<<	Shift Left
>>	Shift Right

Overflow and Underflow Operators

Typically, assigning or incrementing an integer, float, or double past it's range would result in a runtime error. However, if you'd instead prefer to safely truncate the number of available bits, you can opt-in to have the variable overflow or underflow using the following operators:

Operator	Purpose
&+	Addition
&-	Subtraction
&*	Multiplication
&/	Division
&%	Remainder

Example for unsigned integers (works similarly for signed):

var%20willOverflow%20%3D%20UInt8.max%0A//%20willOverflow%20equals%20255%2C%20which%20is%20the%20largest%20value%20a%20UInt8%20can%20h
%201%0A//%20willUnderflow%20is%20now%20equal%20to%20255

Another example to show how you can prevent dividing by zero from resulting in infinity:

let%20x%20%3D%201%0Alet%20y%20%3D%20x%20%26/%200%0A//%20y%20is%20equal%20to%200

Other Operators

Operator	Purpose
??	Nil coalescing
?:	Ternary conditional
!	Force unwrap object value
?	Safely unwrap object value

Back to top

Operator Overloading

Swift allows you to overwrite existing operators or define new operators for existing or custom types. For example, this is why in Swift you can join strings

using the + operator, even though it is typically used for math.

Operator overloading is limited to the following symbols, `/ = - + * % < > ! & | ^ . ~`, however you cannot overload the = operator by itself (it must be combined with another symbol).

Operators can be specified as:

- `prefix`: goes before an object such as `++movieCount`
- `infix`: goes between two objects, such as `a + b`
- `postfix`: goes after an object, such as `unwrapMe!`

Examples:

```
//%20Turn%20addition%20into%20subtraction%0A@infix%20func%20+%20%28x%3A%20Int%2C%20y%3A%20Int%29%20-
%3E%20Int%20%7B%0A%20%20%20%20return%20x%20-
%20y%0A%7D%0Avar%20z%20%3D%2010%20+%209%20//%20z%20is%201%0A%0A//%20Create%20an%20entirely%20new%20operator%20for%20the%20Vector2
%3E%20Vector2D%20%7B%0A%20%20%20%20return%20Vector2D%28x%3A%20left.x%20+%20right.x%2C%20y%3A%20left.y%20+%20right.y%29%0A%7D%0Alet
```

If you want to create compound assignment operators, you'll need to use the `@assignment` keyword:

```
@assignment%20func%20+%3D%20%28inout%20left%3A%20Vector2D%2C%20right%3A%20Vector2D%29%20%7B%0A%20%20%20%20left%20%3D%20left%20+%
```

Back to top

Declaring Classes

Classes are typically declared using separate `.swift` files, but multiple classes can also be created within the same file if you'd like to organize it that way.

Unlike Objective-C, there's no need for an interface file (`.h`) in Swift.

The implementation file should contain (in this order):

- Any needed `import` statements
- A `class` declaration which contains any constants or variables necessary for the class
- All public and private functions

Example:

MyClass.swift

```
import%20UIKit%0A%0Aclass%20MyClass%20%7B%0A%20%20%20%20//%20Declare%20any%20constants%20or%20variables%20at%20the%20top%0A%20%20%
%20Class%20Methods%2C%20e.g.%20MyClass.functionName%28%29%0A%0A%20%20%20%20class%20func%20alert%28%29%20%7B%0A%20%20%20%20%20%
%20Instance%20Methods%2C%20e.g.%20myClass.functionName%28%29%0A%0A%20%20%20%20init%28x%3A%20Int%2C%20y%3A%20Int%29%20%7B%0A%20
%20Private%20Methods%0A%0A%20%20%20%20private%20func%20pointLocation%28%29%20-
%3E%20String%20%7B%0A%20%20%20%20%20%20%20%20return%20%22x%3A%20%5C%28x%29%2C%20y%3A%20%5C%28y%29%22%0A%20%20%20%20%7D%
```

Instantiation

When you want to create a new instance of a class, you use the syntax:

```
let%20myClass%20%3D%20MyClass%28x%3A%201%2C%20y%3A%202%29
```

where `x` and `y` are variables that are passed in at the time of instantiation.

Back to top

Declarations

More info here in the docs (https://developer.apple.com/library/ios/documentation/swift/conceptual/Swift_Programming_Language/Declarations.html#//apple_ref/doc/uid/TP40014097-CH34-XID_704) .

Preprocessor

Swift doesn't come with a preprocessor so it only supports a limited number of statements for build time. Things like `#define` have been replaced with global constants defined outside of a class.

Directive	Purpose
#if	An `if` conditional statement
#elif	An `else if` conditional statement
#else	An `else` conditional statement
#endif	An `end if` conditional statement

Imports

Directive	Purpose

Directive	Purpose
import	Imports a framework

Constants & Variables

Directive	Purpose
let	Declares local or global constant
var	Declares a local or global variable
class	Declares a class-level constant or variable
static	Declares a static type

Classes, Structure, Functions and Protocols

Directive	Purpose
typealias	Introduces a named alias of an existing type
enum	Introduces a named enumeration
struct	Introduces a named structure
class	Begins the declaration of a class
init	Introduces an initializer for a class, struct or enum
init?	Produces an optional instance or an implicitly unwrapped optional instance; can return `nil`
deinit	Declares a function called automatically when there are no longer any references to a class object, just before the class object is deallocated
func	Begins the declaration of a function
protocol	Begins the declaration of a formal protocol
static	Defines as type-level within struct or enum
convenience	Delegate the init process to another initializer or to one of the class's designated initializers
extension	Extend the behavior of class, struct, or enum
subscript	Adds subscripting support for objects of a particular type, normally for providing a convenient syntax for accessing elements in a collective, list or sequence
override	Marks overriden initializers

Operators

Directive	Purpose
operator	Introduces a new infix, prefix, or postfix operator

Declaration Modifiers

Directive	Purpose
dynamic	Marks a member declaration so that access is always dynamically dispatched using the Objective-C runtime and never inlined or devirtualized by the compiler
final	Specifies that a class can't be subclassed, or that a property, function, or subscript of a class can't be overridden in any subclass
lazy	Indicates that the property's initial value is calculated and stored at most once, when the property is first accessed
optional	Specifies that a protocol's property, function, or subscript isn't required to be implemented by conforming members
required	Marks the initializer so that every subclass must implement it
weak	Indicates that the variable or property has a weak reference to the object stored as its value

Access Control

Directive	Purpose

Directive	Purpose
public	Indicates the entities are intended for use as API, and can be accessed by any file that imports the module, e.g. as a framework used in several of your projects
private	Indicates the entities are available only from within the source file where they are defined
internal	(Default) Indicates the entities are only available to the entire module that includes the definition, e.g. an app or framework target

Back to top

Literals

Literals are compiler directives which provide a shorthand notation for creating common objects.

Syntax	What it does
`"string"`	Returns a `String` object
`28`	Returns an `Int`
`3.14`, `0xFp2`, `1.25e2`	Returns a `Float` object
`true`, `false`	Returns a `Bool` object
`[]`	Returns an `Array` object
`[keyName:value]`	Returns a `Dictionary` object
`0b`	Returns a binary digit
`0o`	Returns an octal digit
`0x`	Returns a hexadecimal digit

Strings

Special characters can be included:

- Null Character: `\0`
- Backslash: `\\` (can be used to escape a double quote)
- Horizontal Tab: `\t`
- Line Feed: `\n`
- Carriage Return: `\r`
- Double Quote: `\"`
- Single Quote: `\'`
- Unicode scalar: `\u{n}` where n is between one and eight hexadecimal digits

Array Access Syntax

let%20example%20%3D%20%5B%20%22hi%22%2C%20%22there%22%2C%2023%2C%20true%20%5D%0Aprintln%28%22item%20at%20index%200%3A%20%5C%2

Dictionary Access Syntax

let%20example%20%3D%20%5B%20%22hi%22%20%3A%20%22there%22%2C%20%22iOS%22%20%3A%20%22people%22%20%5D%0Aif%20let%20value%20%3D%

Mutability

For mutable literals, declare it with `var`; immutable with `let`.

Back to top

Functions

Declaration Syntax

Functions without a return type use this format:

//%20Does%20not%20return%20anything%20or%20take%20any%20arguments%0Afunc%20doWork%28%29%20%7B%0A%20%20%20%20//%20Code%0A%7D

`class` precedes declarations of class functions:

//%20Call%20on%20a%20class%2C%20e.g.%20MyClass.someClassFunction%28%29%0Aclass%20func%20someClassFunction%28%29%20%7B%0A%20%20%20%

Declare instance functions:

//%20Called%20on%20an%20instance%20of%20a%20class%2C%20e.g.%20myClass.someInstanceFunction%28%29%0Afunc%20doMoreWork%28%29%20%7B%0

Function arguments are declared within the parentheses:

//%20Draws%20a%20point%20at%20the%20given%20x%20and%20y%0Afunc%20drawPoint%28x%3A%20Int%2C%20y%3A%20Int%29

Return types are declared as follows:

//%20Returns%20a%20String%20object%20for%20the%20given%20String%20argument%0Afunc%20sayHelloToMyLilFriend%28lilFriendsName%3A%20String%2
%3E%20String%20%7B%0A%20%20%20%20return%20%22Oh%20hello%2C%20%5C%28lilFriendsName%29.%20Cup%20of%20tea%3F%22%0A%7D

You can have multiple return values:

//%20Returns%20multiple%20objects%0Afunc%20sayHelloToMyLilFriend%28lilFriendsName%3A%20String%29%20-
%3E%20%28msg%3A%20String%2C%20nameLength%3A%20Int%29%20%7B%0A%20%20%20%20return%20%28%22Oh%20hello%2C%20%5C%28lilFriendsName

And those multiple return values can be optional:

func%20sayHelloToMyLilFriend%28lilFriendsName%3A%20String%29%20-%3E%20%28msg%3A%20String%2C%20nameLength%3A%20Int%29%3F

By default, external parameter names aren't given when you call the function, but you can specify that one or more are shown in the method signature by putting a `#` symbol in front of the parameter name:

```
func sayHelloToMyLilFriend(#lilFriendsName: String) {
    // Code
}

sayHelloToMyLilFriend(lilFriendsName:"Rob")
```

You can also specify default values for the parameters:

```
func sayHelloToMyLilFriend(lilFriendsName: String = "Rob") {
    // Code
}

sayHelloToMyLilFriend() // "Oh hello, Rob. Cup of tea?"
sayHelloToMyLilFriend(lilFriendsName:"Jimbob") // "Oh hello, Jimbob. Cup of tea?"
```

Specifying defaults for your parameters forces you to specify the parameter name (e.g. `lilFriendsName:` when you call the function). However, you can opt-out of this by placing an underscore `_` in front of it in the method signature:

func%20sayHelloToMyLilFriend%28_%20lilFriendsName%3A%20String%20%3D%20%22Rob%22%29%20%7B%0A%20%20%20%20//%20Code%0A%7D%0A%0Asay

Swift also supports variadic parameters so you can have an open-ended number of parameters passed in:

func%20sayHelloToMyLilFriends%28lilFriendsName%3A%20String...%29%20%7B%0A%20%20%20%20//%20Code%0A%7D%0A%0AsayHelloToMyLilFriends%28%2

And lastly, you can also use a couple of prefixes to declare input parameters as a variable with `var` or as `inout`.

An in-out parameter has a value that is passed in to the function, is modified by the function, and is passed back out of the function to replace the original value.

You may remember `inout` parameters from Objective-C where you had to sometimes pass in an `&error` parameter to certain methods, where the `&` symbol specifies that you're actually passing in a pointer to the object instead of the object itself. The same applies to Swift's `inout` parameters now as well.

Calling Functions

Functions are called using dot syntax: `myClass.doWork()` or `self.sayHelloToMyLilFriend("Rob Phillips")`

`self` is a reference to the function's containing class.

At times, it is necessary to call a function in the superclass using `super.someMethod()`.

Back to top

Constants and Variables

Declaring a constant or variable allows you to maintain a reference to an object within a class or to pass objects between classes.

Constants are defined with `let` and variables with `var`. By nature, constants are obviously immutable (i.e. cannot be changed once they are instantiated) and variables are mutable.

class%20MyClass%20%7B%0A%20%20%20%20let%20text%20%3D%20%22Hello%22%20//%20Constant%0A%20%20%20%20var%20isComplete%3A%20Bool%20//

There are many ways to declare properties in Swift, so here are a few examples:

var%20myInt%20%3D%201%20//%20inferred%20type%0Avar%20myExplicitInt%3A%20Int%20%3D%201%20//%20explicit%20type%0Avar%20x%20%3D%201%2C%

Access Levels

The default access level for constants and variables is `internal`:

class%20MyClass%20%7B%0A%20%20%20%20//%20Internal%20%28default%29%20properties%0A%20%20%20%20var%20text%3A%20String%0A%20%20%20%2

To declare them publicly, they should also be within a `public` class as shown below:

public%20class%20MyClass%20%7B%0A%20%20%20%20//%20Public%20properties%0A%20%20%20%20public%20var%20text%3A%20String%0A%20%20%20%20

Private variables and constants are declared with the `private` directive:

class%20MyClass%20%7B%0A%20%20%20%20//%20Private%20properties%0A%20%20%20%20private%20var%20text%3A%20String%0A%20%20%20%20private%

Getters and Setters

In Objective-C, variables were backed by getters, setters, and private instance variables created at build time. However, in Swift getters and setters are only used for computed properties and constants actually don't have a getter or setter at all.

The getter is used to read the value, and the setter is used to write the value. The setter clause is optional, and when only a getter is needed, you can omit both clauses and simply return the requested value directly. However, if you provide a setter clause, you must also provide a getter clause.

You can overrride the getter and setter of a property to create computed properties:

//%20Example%20of%20computed%20property%0Avar%20x%3A%20Int%20%7B%0A%20%20%20%20get%20%7B%0A%20%20%20%20%20%20%20%20println%28

Access Callbacks

Swift also has callbacks for when a property will be or was set using `willSet` and `didSet` shown below:

var%20numberOfEdits%20%3D%200%0Avar%20value%3A%20String%20%3D%20%22%22%20%7B%0A%20%20%20%20willSet%20%7B%0A%20%20%20%20%20%

Accessing

Properties can be accessed using dot notation:

self.myVariableOrConstant

Local Variables

Local variables and constants only exist within the scope of a function.

func%20doWork%28%29%20%7B%0A%20%20%20%20let%20localStringVariable%20%3D%20%22Some%20local%20string%20variable.%22%0A%20%20%20%20se

Back to top

Naming Conventions

The general rule of thumb: Clarity and brevity are both important, but clarity should never be sacrificed for brevity.

Functions and Properties

These both use `camelCase` where the first letter of the first word is lowercase and the first letter of each additional word is capitalized.

Class names and Protocols

These both use `CapitalCase` where the first letter of every word is capitalized.

Functions

These should use verbs if they perform some action (e.g. `performInBackground`). You should be able to infer what is happening, what arguments a function takes, or what is being returned just by reading a function signature.

Example:

//%20Correct%0Aoverride%20func%20tableView%28tableView%3A%20UITableView%2C%20cellForRowAtIndexPath%20indexPath%3A%20NSIndexPath%29%20-
%3E%20UITableViewCell%20%7B%0A%20%20%20%20//%20Code%0A%7D%0A%0A//%20Incorrect%20%28not%20expressive%20enough%29%0Aoverride%20func
%3E%20UITableViewCell%20%7B%0A%20%20%20%20//%20Code%0A%7D

Back to top

Closures

Closures in Swift are similar to blocks and are essentially chunks of code, typically organized within a `{}` clause, that are passed between functions or to execute code as a callback within a function. Swift's `func` functions are actually just a special case of a closure in use.

Syntax

%7B%20%28params%29%20-%3E%20returnType%20in%0A%20%20%20%20statements%0A%7D

Examples

//%20Map%20just%20iterates%20over%20the%20array%20and%20performs%20whatever%20is%20in%20the%20closure%20on%20each%20item%0Alet%20peopl
%3E%20String%20in%0A%20%20%20%20%22Oh%20hai%2C%20%5C%28person%29...%22%0A%7D%29%0A//%20Oh%20hai%2C%20Rob%0A//%20Oh%20hai%2C%
%3E%20Bool%20in%0A%20%20%20%20return%20s1%20%3E%20s2%0A%7D%29%0A//%20Or%20on%20a%20single%20line%3A%0Areversed%20%3D%20sorted%
%3E%20Bool%20in%20return%20s1%20%3E%20s2%20%7D%20%29%0A//%20Or%20because%20Swift%20can%20infer%20the%20Bool%20type%3A%0Areversed%
than%20operator%20implementation%20exactly%20matches%20this%20function%20definition%3A%0Areversed%20%3D%20sorted%28names%2C%20%3E%29

If the closure is the last parameter to the function, you can also use the trailing closure pattern. This is especially useful when the closure code is especially long and you'd like some extra space to organize it:

func%20someFunctionThatTakesAClosure%28closure%3A%20%28%29%20-
%3E%20%28%29%29%20%7B%0A%20%20%20%20//%20function%20body%20goes%20here%0A%7D%0A%0A//%20Instead%20of%20calling%20like%20this%3A%0

Capturing Values

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

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

func%20makeIncrementor%28forIncrement%20amount%3A%20Int%29%20-%3E%20%28%29%20-%3E%20Int%20%7B%0A%20%20%20%20var%20runningTotal%20%3D%200%0A%20%20%20%20func%20incrementor%28%29%20-%3E%20Int%20%7B%0A%20%20%20%20%20%20%20%20runningTotal%20+%3D%20amount%0A%20%20%20%20%20%20%20%20return%20runningTotal%0A%2

Swift determines what should be captured by reference and what should be copied by value. You don't need to annotate a variable to say that they can be used within the nested function. Swift also handles all memory management involved in disposing of variables when they are no longer needed by the function.

Back to top

Generics

Coming soon...

Back to top

Control Statements

Swift uses all of the same control statements that other languages have:

If-Else If-Else

if%20someTestCondition%20%7B%0A%20%20%20%20//%20Code%20to%20execute%20if%20the%20condition%20is%20true%0A%7D%20else%20if%20someOther

As you can see, parentheses are optional.

Ternary Operators

The shorthand notation for an `if-else` statement is a ternary operator of the form: `someTestCondition ? doIfTrue : doIfFalse`

Example:

func%20stringForTrueOrFalse%28trueOrFalse%3A%20Bool%29%20-%3E%20String%20%7B%0A%20%20%20%20return%20trueOrFalse%20%3F%20%22True%22%20%3A%20%22False%22%0A%7D

For Loops

Swift enables you to use ranges inside of `for` loops now:

for%20index%20in%201...5%20%7B%0A%20%20%20%20println%28%22%5C%28index%29%20times%205%20is%20%5C%28index%20*%205%29%22%29%0A%7D

Or you can use more traditional loops:

for%20var%20index%20%3D%200%3B%20index%20%3C%205%3B%20++index%20%7B%0A%20%20%20%20println%28%22index%20is%20%5C%28index%29%22

Enumerating arrays & dictionaries

//%20We%20explicitly%20cast%20to%20the%20Movie%20class%20from%20AnyObject%20class%0Afor%20movie%20in%20someObjects%20as%20%5BMovie%5D

If you need to cast to a certain object type, see the earlier discussion about the `as` and `as?` keywords.

While Loop

while%20someTextCondition%20%7B%0A%20%20%20%20//%20Code%20to%20execute%20while%20the%20condition%20is%20true%0A%7D

Do While Loop

do%20%7B%0A%20%20%20%20//%20Code%20to%20execute%20while%20the%20condition%20is%20true%0A%7D%20while%20someTestCondition

Switch

Switch statements are often used in place of `if` statements if there is a need to test if a certain variable matches the value of another constant or variable. For example, you may want to test if an error code integer you received matches an existing constant value or if it's a new error code.

switch%20errorStatusCode%20%7B%0A%20%20%20%20case%20kRPNetworkErrorCode%3A%0A%20%20%20%20%20%20%20%20//%20Code%20to%20execute%2

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

Exiting Loops

Although `break` is not required in Swift, you can still 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.

- `return` : Stops execution and returns to the calling function. It can also be used to return a value from a function.
- `break` : Used to stop execution of a loop.

Back to top

Extending Classes

Coming soon...

Back to top

Error Handling

Coming soon...

Back to top

Passing Information

Coming soon...

Back to top

User Defaults

User defaults are basically a way of storing simple preference values which can be saved and restored across app launches. It is not meant to be used as a data storage layer, like Core Data or sqlite.

Storing Values

let%20userDefaults%20%3D%20NSUserDefaults.standardUserDefaults%28%29%0AuserDefaults.setValue%28%22Some%20Value%22%2C%20forKey%3A%20%22Rl

Always remember to call `synchronize` on the defaults instance to ensure they are saved properly.

Retrieving Values

let%20userDefaults%20%3D%20NSUserDefaults.standardUserDefaults%28%29%0Alet%20someValue%3A%20AnyObject%3F%20%3D%20userDefaults.valueForKey

There are also other convenience functions on `NSUserDefaults` instances such as `boolForKey(...)`, `stringForKey(...)`, etc.

Back to top

Common Patterns

For a comprehensive list of design patterns, as established by the Gang of Four, look here: Design Patterns in Swift (https://github.com/ochococo/Design-Patterns-In-Swift)

Singletons

Singleton's are a special kind of class where only one instance of the class exists for the current process. They are a convenient way to share data between different parts of an app without creating global variables or having to pass the data around manually, but they should be used sparingly since they often create tighter coupling between classes.

To turn a class into a singleton, you use the following implementation where the function name is prefixed with `shared` plus another word which best describes your class. For example, if the class is a network or location manager, you would name the function `sharedManager` instead of `sharedInstance`.

private%20let%20_singletonInstance%20%3D%20MyClass%28%29%0Aclass%20MyClass%20%7B%0A%20%20class%20var%20sharedInstance%3A%20MyClass%2

Explanation: The lazy initializer for a global variable is run as `dispatch_once` the first time that variable is accessed to make sure the initialization is atomic. This ensures it is thread safe, fast, lazy, and also bridged to ObjC for free. More from Apple here (https://developer.apple.com/swift/blog/?id=7) .

Usage: You would get a reference to that singleton class in another class with the following code:

let%20myClass%20%3D%20MyClass.sharedInstance%0AmyClass.doSomething%28%29%0Aprintln%28%22Attribute%20value%20is%20%5C%28myClass.someVar

Back to top

Unicode Support

Although I don't recommend this, Swift will compile even if you use emoji's in your code since it offers Unicode support.

More info from Apple here (https://developer.apple.com/library/ios/documentation/swift/conceptual/Swift_Programming_Language/StringsAndCharacters.html)

Back to top