Glossary of Common Swift Terms

With so many terms for syntax and functionality in Swift it's easy to feel confused sometimes. So, this page acts as a one-stop dictionary of terms, providing definitions for all common terms used in the Swift language.

@

- **@autoclosure:** An attribute attached to function parameters that are closures, which asks Swift to silently wrap any code using it in a closure rather than requiring users to do it by hand. This is used rarely, but it's important in the `assert()` function.
- **@available:** An attribute attached to types or functions that mark them as being available or unavailable to specific versions of Swift or operating systems.
- **@discardableResult:** An attribute attached to methods that return a value, marking the return value as safe to ignore if the caller wants to. When this is not used, Swift will show a warning if you don't do something with the function's return value.
- **@dynamicCallable:** An attribute attached to types to mark them as being directly callable, primarily so that Swift can interact more easily with dynamic languages such as Python.

- **@dynamicMemberLookup:** An attribute attached to types to mark them as being able to handle undefined properties using special methods, primarily so that Swift can interact more easily with dynamic languages such as Python.
- **@escaping:** An attribute attached to function parameters that are closures, which tells Swift the closure will be used after the function has returned. This will in turn cause Swift to store the closure safely so that it doesn't get destroyed prematurely.
- **@objc:** An attribute used to mark methods and properties that must be accessible to Objective-C code. Swift does not make its code accessible to Objective-C by default to avoid making the code larger than it needs to be.
- **@objcMembers:** An attribute used to mark classes where all properties and methods must be accessible to Objective-C code. Swift does not make its code accessible to Objective-C by default to avoid making the code larger than it needs to be.
- **@unknown:** An attribute attached to the default case of **`switch`** blocks that allows code to handle enum cases that may be added at some point in the future, without breaking source compatibility.

a

- **Access control:** A set of keywords that control how properties may be accessed by other code. **`open`**

means the property can be accessed and overridden from anywhere, `public` means the property may be accessed from anywhere but overridden only within the module it came from, `internal` means the property may be accessed from anywhere inside the same module, `fileprivate` means the property may be accessed from anywhere inside the same file, and `private` means the property may be accessed from anywhere inside the same type.

- **ABI:** The description of how the Swift compiler produces binaries: how data structures are laid out in memory and accessed, how functions are called, and so on.
- **API:** The collection of classes, structs, methods, and properties exposed by a library to solve a particular problem. Short for Application Programming Interface.
- **Argument:** The name for a value that is being passed into a function, to be used inside the function. For example, in `sayHello(to: "Paul")` the `to` part is an argument. Many people just say "parameter" rather than "argument", but argument is technically correct.
- **Array:** A sequential collection of values of any type, such as an array of names in a band.
- **Associated type:** A missing type in a protocol that must be specified by whatever type is conforming to the protocol. Associated types allow us to have flexibility when adding conformances: we can say

that to conform to our protocol you must have an array of items, but we don't care what the data type of those items is. Associated types are written as one word in Swift: `associatedtype`.
- **Associated value:** A value that has been added to an enum case to provide some extra meaning. For example, you might have an enum case saying the weather is windy, then add an associated value saying how windy.

b

- **Block:** Can mean any chunk of code that starts with { and ends with } ("a code block"), but "block" is also the Objective-C name for closures.
- **Boolean:** A data type that stores either true or false.
- **Brace:** The name for opening and closing curly brackets, { and }.
- **Bracket:** The name for opening and closing square brackets, [and]
- **Break:** A keyword that exit the current loop. If used with a labeled statement, e.g. `break myLoop`, it will break out of the specified block.

c

- **Capturing values:** The name for the process of closures keeping a reference to values that are used inside the closure but were created outside. This is different from copying: the closure refers to the

original values, not its own copies, so if the original values change then the closure's values change too.

- **CaseIterable:** A Swift protocol that can be applied to enums. If the enum has cases with no associated values, the compiler will generate an `allCases` array that lets you loop over the cases in the enum.
- **Catch:** A keyword that starts a block of code to handle errors. You can specify what kind of errors should be caught, and use a generic "catch all" block to catch everything else. Paired with `do`.
- **CGFloat:** A floating-point number that may be equivalent to a `Double` or `Float` depending on the platform.
- **Class:** A custom data type that can have one or more properties and one or more methods. Unlike structs, classes are reference types.
- **Class inheritance:** The ability for one class to build on another, inheriting all its methods and properties. Some languages allow one class to inherit from multiple parents, but Swift does not.
- **Closure:** An anonymous function that automatically keeps a reference to any values it uses that were declared outside the function.
- **Codable:** A protocol that allows easy conversion between a struct or a class and JSON or XML.
- **Collection:** A Swift protocol that is used by sequences types you can traverse multiple times without destroying them or affecting the collection, such as arrays and dictionaries.

- **Comparable:** A common Swift protocol that says conforming types can be placed into an order using `<`.
- **Compiler directive:** One of several special pieces of code written using a `#`, that act as instructions to the compiler. For example, compiler directives can check whether we're targeting the simulator or not, and compile one of two code variants.
- **Compound assignment operator:** An operator that modifies a value and assigns it back to the original variable at the same time. For example, `score += 1` adds 1 to the current value of `score`.
- **Computed property:** Any property that does not have a simple storage area for its value, and instead is calculated each time the property is accessed by running some code.
- **Condition:** Any check that evaluates to true or false using an `if` statement. You can provide code to run when your condition is true, as well as an `else`
- **Conditional conformance:** The ability to say that a type conforms to a protocol only if specific conditions are met. For example, Swift's arrays conform to `Equatable` only if their element also conforms to `Equatable`.
- **Constant:** Any named piece of data in your code that may not change when your program runs.
- **Continue:** A keyword that exits the current iteration of a loop, causing the loop to start its next iteration

immediately. If used with a labeled statement, e.g. `continue myLoop`, it will continue the specified block.
- **Controller:** A part of your program that handles logic. Part of the Model-View-Controller architecture.

d

- **Data:** A type that holds any kind of binary data.
- **Default case:** A special case for `switch` blocks that will match all other values.
- **Default parameter:** A function parameter that has a default value attached, to allow callers not to provide it and get reasonable behavior. For example, `func checkSettings(debugMode: Bool = true)` can be called as `checkSettings(debugMode: true)` or `checkSettings(debugMode: false)`, but also as `checkSettings()` – missing a `debugMode` value will assume true, because that's the default value.
- **Defer:** A keyword that allows us to schedule work for when the current scope is exited.
- **Deinitializer:** A special method that is called when an instance of a class is being destroyed. These may not accept parameters, and do not exist on structs.
- **Dictionary:** A high-performance, unordered collection of values stored using a key for fast access.
- **Do:** A keyword that starts a block of code that might throw errors. Paired with `catch`.
- **Double:** A high-precision floating-point number,

such as 3.1, or 3.141592654.

e

- **Enum:** A set of named values that are easier to remember and safer than just using strings or integers. For example, you might create an enum of directions, using north, south, east, and west – this is much nicer than using 0, 1, 2, and 3, for example. Short for "enumeration". Pronounced as "ee-numb", but "ee-noom" is an accepted variation.
- **Equatable:** A common Swift protocol that says conforming types can be compared for equality using `==`.
- **Error:** A Swift protocol that our own enums can conform to, which we can then use to throw errors from functions.
- **Expression:** Some code that evaluates to a value. For example, `12 * 12` evaluates to 144.
- **Extension:** A set of additional methods and computed properties that are added to a concrete type, such as `Int`.

f

- **Fallthrough:** A keyword used in `switch` blocks to mean "carry on executing the case immediately following this one."
- **Failable initializer:** An initializer that returns an

optional value, because initialization might have failed for some reason. These are written as `init?()` or `init!()`.

- **Final class:** A class that may not be inherited from by anything else.
- **Float:** A low-precision floating-point number, such as 3.1, or 3.141592654.
- **For loop:** A loop that counts through all values in a sequence, such as an array or a range.
- **Force unwrap:** The process of using the value inside an optional without checking it exists first. If the optional is empty – if it has no value – force unwrapping will crash your code.
- **Framework:** A collection of code that you are using. Frameworks differ from libraries in that frameworks usually take over control of their operation, and call back to you when they need information.
- **Function:** A named section of your code that performs a specific, reusable task. You might pass in parameters to the function to customize how it runs, and it might return one or more values that are the result of its work.
- **Functional programming:** A programming approach that favors sending and receiving immutable data to and from functions, avoiding side effects, and composing functions together.

g

- **Generics:** The ability for one type or function to be used with a variety of data types. For example, Swift's arrays are generic, because you can make an array that stores integers, an array that stores strings, and so on.
- **guard:** A piece of Swift syntax that checks whether a condition is true, and forces you to exit the current scope immediately if it is not. This is commonly used as `guard let`, which checks whether an optional has a value, and, if it does, creates a new constant for that optional's value so it can be used safely. If it has no value, the `guard` condition fails and you must exit the current scope.

h

- **Hashable:** A common Swift protocol that says conforming types can be represented using hash values.
- **Higher-order function:** A function that accepts another function as a parameter, or sends back a function as its return value.

i

- **if let:** A piece of Swift syntax that checks whether an optional has a value, and, if it does, creates a new constant for that optional's value so it can be used safely. If it has no value, the `if` condition fails and you

can run an `else` block instead.

- **Implicitly unwrapped optional:** A special Swift optional that may or may not contain a value, but does not need to be checked before use. If you attempt to use an implicitly unwrapped optional that has no value, your app will crash.
- **Indirect enum:** An enum that is compiled specially so that it can reference itself in its associated values. For example, if you had a linked list enum that needed to reference another linked list item as an associated value, this would need to be indirect.
- **Infinite loop:** A loop that never ends until you say so. This is most commonly done using a boolean variable set to true – you can set it to false as soon as you want the loop to end.
- **Initializer:** A special method that gets run to create an instance of a struct or class. You can have many initializers, and in the case of classes may call parent initializers inside your own initializer.
- **inout parameter:** A function parameter that, when changed inside the function, remains changed outside the function.
- **Integer:** A whole number, such as 5, 55, or 55 million.

k

- **Keypath:** A way of referring to properties without actually reading them.

- **Keyword:** Any word that has specific meaning as a Swift construct, such as `class`, `continue`, and `try`.

l

- **Labeled statement:** A name attached to a specific block of code, such as a loop or a condition, allowing you to break out of it specifically even if you are inside other blocks. For example, if you are inside a loop that's inside a loop that's inside a third loop, you could exit the outermost loop.
- **Lazy:** A keyword that marks a property as being created only when it is first accessed. This is a performance optimization, because it means if the property is never accessed the work required to calculate its value is never done.
- **Library:** A collection of code that you are using. Libraries differ from frameworks in that libraries are just collections of classes, structs, enums, and so on, for you to use however you want.

m

- **Memberwise initializer:** An initialer for structs that is automatically generated by the Swift compiler, requiring that all properties must have a value. If you implement your own initializer inside the struct the memberwise initializer will no longer be generated by Swift.
- **Method:** A function that belongs to a struct or a

class.

- **Model:** A part of your program that stores data. Part of the Model-View-Controller architecture.
- **Multi-line strings:** A string that spans multiple lines. Multi-line strings in Swift must start and end with three double quotes on their own line.
- **Mutating method:** A method on a struct that will change one of the properties of the struct. This must be used because only variable structs can have mutating methods called on them. Mutating methods are not required for classes, where every method can be mutating without the keyword.

n

- **Nested type:** A class or a struct that is defined inside another class or struct.
- **Nil:** Swift's term for missing data. An optional that has no value will be set to nil.
- **Nil coalescing:** An operator in Swift, written as `??`, that uses the value from an optional if it has one, or a default value otherwise.
- **Non-throwing function:** A function that cannot throw errors. These must not be marked using `throws`, and must not use the `throw` keyword.

o

- **Objective-C:** Apple's first programming language, and the forerunner to Swift.

- **Operand:** The values that work alongside an operator. For example, in `2 + 3` the 2 and 3 are operands.
- **Operator:** Any symbol that acts as a function on values to its left and/or right. For example, `+` is an operator that adds two values together.
- **Operator overloading:** The ability for one operator, such as `+` to do multiple things depending on how it's used. For example, `1 + 1` is an integer addition, but `"Hello " + "Paul"` will join the strings together.
- **Optional:** A type that wraps another type, such as `String` or `Int`, but adds the ability to store no value at all. "No value" is different from all regular integer values, including zero. Swift uses optionals heavily as a way of providing runtime safety, and the compiler forces us to use them correctly.
- **Optional chaining:** The ability to use multiple optional methods or properties in a single line of code. If any of them are nil then execution of that line will stop, but if they all succeed then you will get an optional value back. For example, `user?.name?.uppercased()`.
- **Override method:** A class's method that has the same name, parameters, and return type as a method on the class's parent class. The `override` keyword tells the Swift compiler you understand that you are changing the behavior.

p

- **Parameter:** The name for a value that has been passed into a function, to be used inside the function. For example, in `func sayHello(to: String)`, the `to` part is a parameter.
- **Parameter label:** Custom names assigned to function parameters that affect the way they are used externally. For example, in `sayHello(to name: String)`, people calling the function will say `sayHello(to: "Paul")`, but inside the function you would refer to `name`.
- **Parenthesis:** The name for opening and closing rounded brackets, (and).
- **Polymorphism:** The ability for an object to appear as and be used as multiple different types. For example, a `Labrador` class instance could also be used as a `Dog` and `Mammal` if you had defined those as parent classes.
- **Property:** A constant or variable that belongs to a class, struct, or enum.
- **Property observer:** Code added to a property using `willSet` and `didSet`, that gets called whenever a property is being changed.
- **Property wrapper:** An attribute placed around a property that gives it custom abilities. For example, you could write a `@UserDefaults` property wrapper to make loading and saving data to user defaults easier.
- **Protocol:** A list of criteria that a type must conform to, such as property names and methods. Protocols

allow us to treat many objects the same way, because they implement all the behavior required to make the protocol work.

- **Protocol extension:** A set of additional methods and computed properties that are added to a protocol, such as `Equatable`.
- **Protocol inheritance:** The ability for one protocol to inherit requirements from another protocol. For example, the `Comparable` protocol inherits from `Equatable`.
- **Protocol-oriented programming:** An approach to programming that favors code re-use through protocol extensions, providing the benefits of multiple inheritance without some of the complexity. Specifically, protocol-oriented programming (POP) cannot add stored properties to a type, so there is less cruft.

r

- **Range:** Ranges span the distance between two values, up to and sometimes including the final value. This is mostly used with numbers: `1..<4` includes the numbers 1, 2, and 3, whereas the range `1...4` includes the numbers 1, 2, 3, and 4. Ranges can also be made from other data types, such as dates.
- **Raw strings:** The ability to specify custom string delimiters for situations when you want to use backslashes and quote marks without them having

their regular meaning.

- **Raw value:** A simple underlying data type that enum values can be mapped to for the purpose of loading and saving. For example, you might say that the planets Mercury, Venus, and Earth have the integer raw values 1, 2, and 3.
- **Reference type:** Any type that stores its data indirectly in memory, with the variable or constant really just being a pointer (or reference) to that piece of memory. If you point a second variable to a reference type, it will point at the same data in memory as the original reference, so the data is shared.
- **Repeat loop:** A loop that will continue executing as long as its condition is true. If its condition starts false, it will still execute at least once.
- **Rethrowing functions:** A function that uses the `rethrows` keyword so that it throws errors only if the closure it accepts throws errors.
- **Return type:** The type of data that a function says it will return. Swift always enforces this, so if you say you will return a string you must do so.
- **Runtime:** When your code is running, as opposed to compile time, which is when the code is being built.

S

- **Scope:** A region of code where variables and constants are valid. Each time you use an opening

brace you start a new scope: all previous variables and constants remain valid in that scope, but any that are declared inside that scope are only valid until the scope ends with a closing brace.

- **Selector:** A way of referring to functions without actually calling them, usually for the purpose of calling them later on.
- **Serialization:** The process of converting objects into text or binary, and vice versa. For example, converting an array of users into JSON, or converting JSON into an array of users.
- **Set:** A high-performance, unordered collection of values of any type that conforms to the `Hashable` protocol.
- **Shorthand parameter name:** Special parameter names made available inside closures, written as a dollar sign the numbers starting from 0. For example, if a closure accepts a `name` string and an `age` integer, you could refer to the string as `$0` and the age as `$1`. Note: you may not mix and match shorthand syntax with regular syntax.
- **some:** A keyword that uses an opaque return type – return types such as `some View` in SwiftUI to mean "some sort of View will be returned but it doesn't matter which type specifically."
- **Statement:** Some code that performs an action, such as `print(age)`.
- **Static method:** A method that is shared across all instances of a struct or class, rather than unique to

each instance. Because this is not run on a specific instance of a struct or class, it cannot access any properties that are not also marked `static`.

- **Static property:** A property that is shared across all instances of a struct or class, rather than unique to each instance.
- **StaticString:** A specialized form of Swift's `String` that must be hand-typed – you must literally type the string directly into your code rather than using string interpolation.
- **String:** A collection of letters, such as "Hello".
- **String interpolation:** The ability for Swift to insert the values of variables and constants into strings, such as "Hello, \(name)."
- **Struct:** A custom data type that can have one or more properties and one or more methods. Unlike classes, structs are value types.
- **Subscript:** Special methods that are used with collections, and provide easy access to read and write values in the collection. For example, `someArray[3]` is a subscript, as is `someDictionary["name"]`.
- **Switch case:** A system of evaluating multiple conditions in one block of code. For example, you might use `switch age` to evaluate the `age` variable, then have cases for 0 to 10, 10 to 20, 20 to 30, and so on. Switch blocks must be exhaustive in Swift, which means they must have cases to cover all possible values.

- **Syntactic sugar:** The name for special Swift syntax that is designed to mask complexity. The name comes from the idea that it's better to create short and sweet syntax that does the same as longer syntax. For example, `[String]` is syntactic sugar for `Array<String>`.
- **Synthesize:** The name for when the Swift compiler generates code on your behalf. For example, if you say that a custom struct conforms to `Equatable`, and all its properties already conform to `Equatable`, then Swift can synthesize a `==` function for it.

t

- **Ternary operator:** An operator that works with three values, written in Swift as `? :`. For example, `isEnabled ? 10 : 100` will return 10 if `isEnabled` is true, and 100 if it's false.
- **Throwing function:** A function that has the ability to throw errors. These must be marked using the `throws` keyword in Swift, and called using `try`.
- **Trailing closure syntax:** The ability for functions that accept a closure as their final parameter to have that closure specified after the function's parentheses, which usually helps make the function call easier to read.
- **try catch:** The ability to run throwing functions and catch any errors that occur. You must start using a `do` block, then call any throwing methods inside that

using `try`, and finally add one or more `catch` blocks to catch any errors. Writing a `catch` block to catch all errors is sometimes called a Pokémon catch, because "you gotta catch 'em all."

- **Tuple:** A fixed-size collection of values of any type, like an anonymous struct. Pronounced as "tyoo-pull" or "too-pull", or sometimes even "tupple".
- **Type alias:** A type alias is the ability to specify what type should fill an associated type. This is usually not needed, because Swift can figure out the type alias just by looking at properties used with the associated type. Type aliases are written as one word in Swift: `typealias`.
- **Type inference:** The ability for Swift's compiler to figure out what kind of data type should be assigned to each of your variables and constants, so you don't have to explicitly state it.
- **Typecasting:** The ability to treat an object of one type as an object of a different type. This is needed when you know more information about the Swift compiler, such as when you instantiate a view controller from a storyboard.

u

- **Unwrapping optionals:** The process of checking whether there is a value inside an optional, and, if there is, extracting that value so it can be used safely.

V

- **Value type:** Any type that stores its data directly in memory. If you point a second variable to a value type it will always copy the data in its entirety, rather than allowing it to be shared.
- **Variable:** Any named piece of data in your code that may change when your program runs.
- **Variadic function:** A function that takes one or more values of a specific type, separated by commas. The `print()` function is variadic, because you can write `print(1)` to print a single value, or `print(1, 2, 3, 4, 5, 6, 7, 8)` to print many.
- **View:** A part of your program that shows visible results to your user. Part of the Model-View-Controller architecture.
- **Void:** The absence of a value. Used most commonly to refer to the return types of functions that return nothing.

W

- **While loop:** A loop that will continue executing as long as its condition is true. If its condition starts false, it won't execute even once.