Swift Language Quick Reference 5.9

Stephen W. Liddle, Brigham Young University

Updated 2-Nov-23

Declare constant (at global or nested scope):
`let x = 10; let y: Double = 11`
Rely on type inference mechanism if possible
Unicode as identifier: `let π = 3.14159`
Keyword as identifier: `let \`if\` = 5` ***(bad idea!)***
Declare stored variable (global or nested scope):
`var x = 10`
If declared in class, we call it a (stored) property
Computed variable/property has getter and optional setter:
```
var variable-name: type {
  get { … }
  set(setter-name) { … }
  // default setter-name is newValue
}
```
Stored variable/property observers:
```
var variable-name: type = expression {
  willSet(setter-name) { … }
  didSet(setter-name) { … }
  // default setter-names are newValue
  // (in willSet) and oldValue (in didSet)
}
```
`lazy` modifier indicates initialize on first access
`weak` modifier indicates holds weak reference ***(must be optional; could become*** `nil` ***any time)***

Comments: either `// single-line`
or `/*...*/` multi-line, which can nest `/*.../*.../*.../*/`

Types
Numeric types: `Bool`, `Int`, `UInt`, `Double`; also `Float`, `Int8`, `Int16`, `Int32`, `Int64`, and `UInt8|16|32|64`
`Int` ***and*** `Double` ***are preferred, but 32-bit*** `Float` ***and*** `Int` ***variants with indicated bit sizes are available***
Types have useful properties and methods like `.min` and `.max`
Numeric literals have infinite precision
Bases: `0b1111 == 0o17 == 0xf == 15`
`let B = 1_000_000_000 // ignore _s`
Scientific notation: `+48.0 == 4.8e1 == 0xCp2`

String/character types:
Fully Unicode-compliant string/character types
Concatenation operators: +, +=
Repeat initializer: `let dotLead = String(count:4, repeatedValue: Character("."))`
Methods: `isEmpty()`, `hasPrefix()`, `hasSuffix()`, `toInt()`
`String` literal: `"hello"`
`Character` literal: `"A"`
`String` escapes: `\0`, `\\`, `\t`, `\n`, `\r`, `\"`, `\'`, `\u{n}`
`String` interpolation: `"Answer: \(result+5)"`
Multiline string literal: `"""`
```
    The White Rabbit put on his spectacles. \
    "Where shall I begin, please your Majesty?"
    """
```
Leading spaces at the indent level are removed, line break is NOT inserted when backslash (\) ends a line
Extended delimiters turn off escaping: `#"line1\nline2"#`
`String` is a value type; use `let`/`var` to control mutability

Boolean literals: `true`, `false`
Null reference: `nil`
Tuple: (***name1*****:*****value1*****,*** ***name1*****:*****value2***)
`Void` is typealias for `()`, tuple with no elements
One-place tuple is same as its element type:
e.g. `(Int)` is the same as `Int`
Access tuple elements by label or position:
`print("o: (\(origin.x),\(origin.1))")`
Decompose: `let (x, y) = (4, "abc")`
Ignore elements by _: `let (_, y) = (4, "abc")` or assign
`(a, _, (b, c)) = ("x", 9.45, (12, 3))`
`// a is "x", b is 12, c is 3, 9.45 ignored`
Can compare tuples of same type with ==, !=, <, >, <=, >= if operator works on each element of the tuple

Alias for type specification:
```
typealias Point = (x: Int, y: Int)
let origin: Point = (0, 0)
```

Collection types:
`[T]` is syntactic sugar for `Array<T>`
`[K:V]` is syntactic sugar for `Dictionary<K,T>`
Also `Set<K>`
`K` must conform to `Hashable` protocol
Concatenate collections with +
Collections are value types; use `let`/`var` to control mutability

Optionals:
`T?` is syntactic sugar for `Optional<T>`, an `enum` with two cases: `None` and `Some(T)`
Use `!` to unwrap optional, but runtime error if `nil`
`T!` stands for `ImplicitlyUnwrappedOptional<T>`
```
let possibleString: String? = "Optional string"
let assumedString: String! = "Impl. unwrapped"
```
Optional binding with `if let c = o { ... }`
Or `guard let c = o else { /* must exit scope */ }`
Can use `var` instead of `let` if you want a variable
Shorthand `if let x` is equivalent to `if let x = x`

Special literals:
`#file` (`String`, name of file in which it appears)
`#fileID` (`String`, name of file and its module)
`#filePath` (`String`, path to file in which it appears)
`#line` (`Int`, line number on which it appears)
`#column` (`Int`, column # in which it begins)
`#function` (`String`, name of enclosing declaration)
`#dsohandle` (`String`, name of enclosing declaration)

Array and dictionary literals: `[]` and `[:]`
Optional comma allowed after last item in the literal

`self` refers to instance or type based on context
In `mutating` method of a value type, you can assign a new instance to `self`
`super` refers to base class instance or type

`class`
Reference type; initializers (`init`), deinitializer (`deinit`)
Failable initializer marked `init?(...)` may return `nil`
Can use identity operators (===, !==) on references; no universal base class; must write `override` to override a method; `private` superclass members are not accessible; `final` modifier prevents further overriding; `convenience`, `required` initializers; subclass can override getters/setters

`struct`
Value type; like class but without inheritance and pass by value instead of pass by reference; often encapsulates a few relatively simple data values (not reference types)
`mutating`, `nonmutating`, `unowned`, `unowned(safe)`, `unowned(unsafe)`

`enum`
Value type with distinct cases (***member values***)
```
enum WeekendDay { case Sunday, Saturday }
var day = WeekendDay.Sunday
day = .Saturday // shorthand if type known
```
`toRaw()`, `fromRaw()`
Recursive associated type uses `indirect case`

Visibility modifiers:
`private` visible only to the current module, not subclasses
`private(set)` gives internal-level getter, `private` setter
`fileprivate` gives access to types within the same file
No modifier means `internal`, visible within current module and subclasses (generally your whole app is a "module")
`public` visible to the world
`open` world can override (only applies to `class` types)

Protocols: (like Java interface: declares instance/type methods/properties, operators, subscripts that must be possessed by ***conforming*** types; can inherit from one or more other protocols)
```
protocol Name: BaseProtocol1, BaseProtocol2 {
  var settable: Int { get set }
  optional var readable: Int { get }
  init(...)
  func random() -> Double
  class func someTypeMethod()
```

```
  mutating func toggle()
}
```
Protocol composition: `P1 & P2 & ...`
```
protocol Name: class {
  // can only be adopted by class types
}
```
Members marked `optional` need not be implemented
Swift provides synthesized implementations of `Equatable`, `Hashable`, `Comparable` where possible
```
struct Vector: Equatable { var x = 0.0, y = 0.0 }
```
To enforce reference semantics, inherit from `AnyObject`
`optional` is for Objective-C interoperability (mark protocol and `optional` members with `@objc`)

Important protocols:
`Equatable`, `Hashable`, `Comparable`, `Printable`

Types can be nested

Metatype: name.`Type` or **name.**`Protocol`
`Self` keyword
typename.`self` gets actual type object (e.g. class)

name.`dynamicType` gets **name's** runtime type
E.g. instead of `PlayingCard.validSuits()`, `self.dynamicType.validSuits()` in case `self` is an instance of a subclass of `PlayingCard`

Semicolons optional after statements, but can be used to combine multiple statements on a line

Loops:
```
for item in collection { ... }
while condition { ... }
repeat { ... } while condition
```
Can use `case`, `let`, `var` in `for`/`while` statements
We use ranges often (e.g. closed range `i in 1...5`, half-open range `0..<5`, one-sided ranges `..<2` or `0...`)

Branching statements:
```
if condition {
  ...
} else if condition {
  ...
} else {
  ...
}

guard condition else {
  // Must either call noreturn function or
  // use return/break/continue/throw to
  // transfer control out of this scope
}

switch control-expression {
  case pattern1, pattern2:
    ...
  case pattern3 where condition:
    ...
  case ignore-this-pattern:
    break
  default:
    ...
}
```
`default` case must appear in last position
Cases must be exhaustive w.r.t. control expression
Use `fallthrough` to fall through to next case
Can label `if`/`switch` statements with ***name*****:** and use `break` or `continue` with those labels
Can use ranges in cases (`case 1..<5` or `case 5...100`)
For tuples, can use patterns like `case (_, 0)` or value bindings like `case (let x, let y)`

```
switch point {
case let (x, y) where x == y:
    println("Dimensions of point identical")
}

defer {
  // Execute this just before leaving scope
  // E.g. close an open file
}
```

Error handling:
```
func canThrowAnError() throws { ... }

do {
  try condition
  ...
} catch pattern1 {
  ...
} catch pattern2 where condition {
  ...
}
```
`try` ***expression*** (evaluate expression, may throw)
`try?` ***expression*** (return `nil` if expression throws)
`try!` ***Expression*** (runtime error if expression throws)
`throw`, `throws`, `rethrows`

Postfix operators: ++ and --
Postfix self: ***expression*****.**`self` or `type.self`
Dynamic type: *expression*.`dynamicType` evaluates to runtime type of the expression
Subscript expression: *expr*[*index-expressions*]
Can provide `subscript`(***parameters***) `->` ***return-type*** { ... } for a type to overload the `[]` operator
Forced-value expression: *expression*!
Optional-chaining expression: *expression*? interrupts evaluation of chain if expression is `nil`

Nil-coalescing operator: `a ?? b` is shorthand for `a != nil ? a! : b`

`import` [***import-kind***] ***module***[**.*symbol-name***]
Can import whole module or just a specific symbol
Import kinds: `typealias`, `struct`, `class`, `enum`, `protocol`, `var`, `func`

Array operators & functions
`init(count:, repeatedValue:)`
`subscript`, `append()`, `insert(_:, atIndex:)`, `removeAtIndex()`, `removeLast()`, `removeAll(keepCapacity: = false)`, `reserveCapacity()`, `count`, `isEmpty`, `capacity`, `sort()`, `sorted()`, `reverse()`, `filter()`, `map()`, `reduce()`, `+=`

Dictionary operators & functions
`subscript`, `updateValue(_:, forKey:)`, `removeValueForKey()`, `removeAll()`, `count`, `keys`, `values`, `==`, `!=`

Type checking and casting:
`is` (true if ***expression*** can be cast to ***type***)
`as` (cast ***expression*** at compile time)
`as?` (cast ***expression*** or return `nil`)
`as!` (cast or throw runtime error)

Functional programming constructs:
Closure:
```
{ capture-list (parameters) -> return-type in
  ...
}
```
Can infer types from context; implicit return from single-expression closures; shorthand argument names: `$0`, `$1`; operator functions; trailing closures; closures capture values in their context; are reference types
Optional ***capture-list*** `[a]` captures just `a`, not other variables
`[weak self]` or `[unowned self]`
`(elements, ...) (inout x: Int, ...)`
```
map<U>(_ :) -> Array<U>
reduce<U>(_ :, combine: (U, T)->U) -> U
filter(_ :) -> Array<T>
reverse() -> Array<U>
```
Pass `inout` parameter by writing `f(&anInoutVar)`
Return tuple to return multiple values from function
Use external and internal names for parameters:
```
func size(for value: Int, _ value2: Int)
```
Can give default value to parameters with = ***expression***
Can use variadic parameters (i.e. lists of arbitrary length):
```
func average(_ numbers: Double...) -> Double
```
Function types can be used as regular types
Functions can be nested
When closure can escape called function, mark `@escaping`

Implicit member expression: .*member-name*
When type can be inferred, no need to specify the type name to access members

Extensions:
```
extension SomeType: Protocol1, Protocol2 {
  // new definitions go here
}
```
Can add new instance or type computed properties & methods, initializers, subscripts, nested types, and protocol conformance (but cannot override)
Can add default implementation of protocol requirements
Can extend generic type for constrained cases:
```
extension Array where Element: Identifiable {...}
```

Generics: Types or functions can be generic indicated by adding generic parameter list:
```
func myMax<T: Comparable>(_ x: T, _ y: T) -> T
   { return x < y ? y : x }
struct Stack<E> {
    var items: [E] = []
    mutating func push(_ item: E) { /* ... */ }
    mutating func pop() -> E { /* ... */ }
}
```
Can add type constraints in generic parameter list:
```
struct Stack<E: Equatable> { ... }
```
Protocol can specify unknown `associatedtype` that is defined by `typealias` or by when generic type implements conformance to the protocol:
```
protocol Container {
    associatedtype Item
    mutating func append(_ item: Item) { ... }
} // Container requires some Item type
struct IntStack: Container {
    typealias Item = Int
    mutating func append(_ item: Int) { ... }
} // Item is declared to be Int
struct Stack<E>: Container {
    mutating func append(_ item: E) { ... }
} // Compiler infers that Item is Int
```
An empty `extension` could also work if the type being extended already satisfies the protocol requirements and thus the compiler can infer the `associatedtype`, as with:
```
extension Array: Container {}
```
Type constraints and where clauses can be used in many contexts including on `associatedtypes`

Opaque types: Let you hide the specific type(s) used to create a value, like the `body` of a `View`:
```
var body: some View { ... }
```
The compiler does NOT expose the specific subtype of `View` that is actually returned (which it does know), only that it conforms to `View`, thus keeping the context simpler
Protocol with `associatedtype` cannot be used as function return type, but function can return an opaque type

Binary operators (N/L/R for no/left/right associativity)
Some binary operators: (numbers indicate precedence level)

<<, >>	Bitwise shift left, right	N160
*, /, %	Multiply, divide, remainder	L150
&*, &/, &%	* / % ignoring overflow	L150
+, –; &+, &–	Add, subtract; ignoring overflow	L140
|, ^	Bitwise OR, XOR	L140
..<, ...	Half-open range, closed range	N135
is, as	Type check, type cast	N132
<, <=, >=, >	Inequality comparators	N130
==, !=	Equality comparators	N130
===, !==	Identical, not identical (refs.)	N130
~=	Pattern match	N130
&&	Logical AND	L120
||	Logical OR	L110
??	Nil coalescing	R110
?:	Ternary conditional	R100
=, +=, *=, etc.	Assignment	R90

&x arithmetic operators truncate result without runtime exception on overflow/underflow, or return 0 instead of runtime error on &/ division by 0

Custom operator:
```
operator
```
Custom operators begin with /, =, –, +, !, *, %, <, >, &, |, ^, ~ or certain Unicode ranges of characters
`associativity`, `left`, `right`, `none`, `infix`, `prefix`, `postfix`, `precedence`
TBD

Free functions:
`print()`, `println()`, `sort()`, `sorted()`

Selector and key-path expressions:
```
#selector(method name)
#selector(getter: property name)
#selector(setter: property name)
#keyPath(expression)
```

Conditional compilation:
```
#if compilation-condition-1
  ...
#elseif compilation-condition-2
  ...
#else
  ...
#endif
```
Conditions:
```
os(macOS | iOS | watchOS | tvOS | Linux| Windows)
arch(i386 | x86_64 | arm | arm64)
swift(>=x.y|<x.y), e.g. swift(>=2.1)
compiler(>=x.y|<x.y)
canImport(module-name)
targetEnvironment(simulator | macCatalyst)
```
```
#sourceLocation(file: filename, line: line-number)
```
```
if #available(platform-name version, ..., *) {
  ...
} else {
  ...
}
```
Platform names: `iOS`, `iOSApplicationExtension`, `macOS`, `macOSApplicationExtension`, `macCatalyst`, `macCatalystApplicationExtension`, `watchOS`, `tvOS`

Keywords in declarations:
`associatedtype`, `class`, `deinit`, `enum`, `extension`, `func`, `import`, `init`, `inout`, `internal`, `let`, `operator`, `private`, `protocol`, `public`, `static`, `struct`, `subscript`, `typealias`, `var`

Keywords in statements:
`break`, `case`, `continue`, `default`, `defer`, `do`, `else`, `fallthrough`, `for`, `guard`, `if`, `in`, `repeat`, `return`, `switch`, `where`, `while`

Keywords in expressions, types, patterns:
`as`, `Any`, `catch`, `dynamicType`, `false`, `is`, `nil`, `rethrows`, `self`, `Self`, `super`, `throw`, `throws`, `true`, `try`, `_`

Keywords starting with #:
`#available()`, `#colorLiteral(red:green:blue:alpha:)`, `#column`, `#dsohandle`, `#else`, `#elseif`, `#endif`, `#error()`, `#file`, `#fileID`, `#filePath`, `#fileLiteral(resourceName:)`, `#function`, `#if`, `#imageLiteral(resourceName:)`, `#keyPath()`, `#line`, `#selector()`, `#selector(getter:)`, `#selector(setter:)`, `#sourceLocation()`, `#sourceLocation(file:line:)`, `#warning()`

Keywords only in special contexts:
`associativity`, `convenience`, `dynamic`, `didSet`, `final`, `get`, `indirect`, `infix`, `lazy`, `left`, `mutating`, `none`, `nonmutating`, `optional`, `override`, `postfix`, `precedence`, `precedencegroup`, `prefix`, `Protocol`, `required`, `right`, `set`, `Type`, `unowned`, `weak`, `willSet`

Punctuation (can't be used as custom operators):
`( ) [ ] { } . , : ; = @ # & -> ` ? !`

Attributes:
`@autoclosure`, `available`, `objc`, `noescape`, `nonobjc`, `noreturn`, `testable`, `convention`, others

Special parameters include _, type..., and default argument value

```
unavailable, introduced: version, deprecated:
   version, obsoleted: version, *, message: message,
   renamed: new-name, discardableResult,
   GKInspectable, NSApplicationMain, NSCopying,
   NSManaged, UIApplicationMain, IBAction, IBOutlet,
   IBDesignable, IBInspectable, convention [swift |
   block | c]
```

Declaration modifiers:
dynamic, final, lazy, optional, required
convenience, mutating, nonmutating, override
infix, postfix, prefix, static

Automatic Reference Counting and memory allocation:
unowned, unowned(safe), unowned(unsafe), weak
Swift tracks references automatically via reference count
When reference count hits zero, Swift releases the object
There can be reference cycles preventing deallocation, so we introduce weak and unowned references

```
precondition(index > 0, "Index must be greater
than zero.")
preconditionFailure(_:_:file:line:)
fatalError(_:file:line:)
```

#error(diagnostic-message)
#warning(diagnostic-message)

Property wrappers: wrapped value, projected value
Access projected value with $
`Any` type (and `as Any` to assign optional to `Any`)

Concurrency:
async, await, async let, Task, TaskGroup, actor
TBD

Memory safety:
TBD

Playground literals: #colorLiteral, #fileLiteral, #imageLiteral
resultBuilder

NOTE: this is a work in progress—not done!