##### Swift Language Guide (Week 3 Milestone)

* [**The Basics**](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309)
* **Numeric Literals**

Integer literals can be written as:

* A decimal number, with no prefix (没有前綴)
* A binary number, with a 0b prefix
* An octal number, with a 0o prefix
* A hexadecimal number, with a 0x prefix

All of these integer literals have a decimal value of 17:

1. let decimalInteger = 17
2. let binaryInteger = 0b10001 // 17 in binary notation 二進制
3. let octalInteger = 0o21 // 17 in octal notation 八進制
4. let hexadecimalInteger = 0x11 // 17 in hexadecimal notation 十六進制

* **Numeric Type Conversion**

Use the Int type for all general-purpose integer constants and variables in your code, even if they are known to be non-negative. Using the default integer type in everyday situations means that integer constants and variables are immediately interoperable in your code and will match the inferred type for integer literal values.

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

* + **Integer Conversion**
  1. let cannotBeNegative: UInt8 = -1
  2. // UInt8 cannot store negative numbers, and so this will report an error
  3. let tooBig: Int8 = Int8.max + 1
  4. // Int8 cannot store a number larger than its maximum value,
  5. // and so this will also report an error
  + **Integer and Floating-Point Conversion**

Conversions between integer and floating-point numeric types must be made explicit:

* 1. let three = 3
  2. let pointOneFourOneFiveNine = 0.14159
  3. let pi = Double(three) + pointOneFourOneFiveNine
  4. // pi equals 3.14159, and is inferred to be of type Double
* **Type Aliases (型別別名)**

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

* 1. typealias AudioSample = UInt16

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

1. var maxAmplitudeFound = AudioSample.min
2. // maxAmplitudeFound is now 0

* **Tuples**

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

1. let http404Error = (404, "Not Found")
2. // http404Error is of type (Int, String), and equals (404, "Not Found")

You can decompose (分解) a tuple’s contents into separate constants or variables, which you then access as usual:

1. let (statusCode, statusMessage) = http404Error
2. print("The status code is \(statusCode)")
3. // Prints "The status code is 404"
4. print("The status message is \(statusMessage)")
5. // Prints "The status message is Not Found"

If you only need some of the tuple’s values, ignore parts of the tuple with an underscore (\_) when you decompose the tuple:

1. let (justTheStatusCode, \_) = http404Error
2. print("The status code is \(justTheStatusCode)")
3. // Prints "The status code is 404"

Alternatively, access the individual element values in a tuple using index numbers starting at zero:

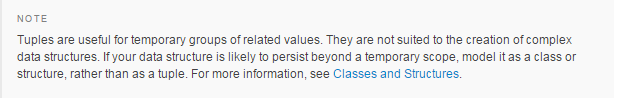
1. print("The status code is \(http404Error.0)")
2. // Prints "The status code is 404"
3. print("The status message is \(http404Error.1)")
4. // Prints "The status message is Not Found"

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

1. let http200Status = (statusCode: 200, description: "OK")

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

1. print("The status code is \(http200Status.statusCode)")
2. // Prints "The status code is 200"
3. print("The status message is \(http200Status.description)")
4. // Prints "The status message is OK"



* [Basic Operators](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/BasicOperators.html#//apple_ref/doc/uid/TP40014097-CH6-ID60)
* Nil Coalescing Operator (空值聚合運算子)

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

The nil coalescing operator is shorthand for the code below:

1. a != nil ? a! : b

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

* [Strings and Characters](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/StringsAndCharacters.html#//apple_ref/doc/uid/TP40014097-CH7-ID285)
* Unicode
* Unicode is an international standard for encoding, representing, and processing text in different writing systems. It enables you to represent almost any character from any language in a standardized form, and to read and write those characters to and from an external source such as a text file or web page. Swift’sString and Character types are fully Unicode-compliant, as described in this section.

### Unicode Scalars

Behind the scenes, Swift’s native String type is built from Unicode scalar values. A Unicode scalar is a unique 21-bit number for a character or modifier, such as U+0061 for LATIN SMALL LETTER A ("a"), or U+1F425for FRONT-FACING BABY CHICK ("🐥").

### Special Characters in String Literals

String literals can include the following special characters:

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

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

* Unicode Representation of Strings

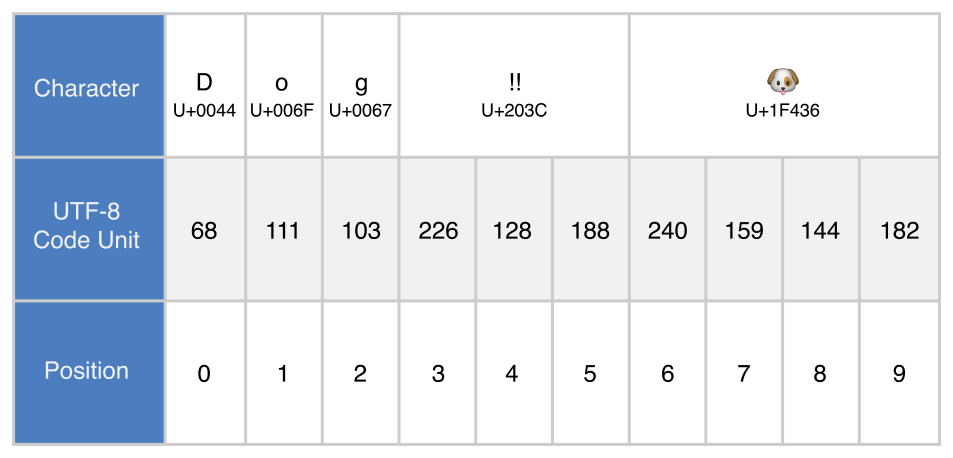
When a Unicode string is written to a text file or some other storage, the Unicode scalars in that string are encoded in one of several Unicode-defined encoding forms. Each form encodes the string in small chunks known as code units. These include the UTF-8 encoding form (which encodes a string as 8-bit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

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

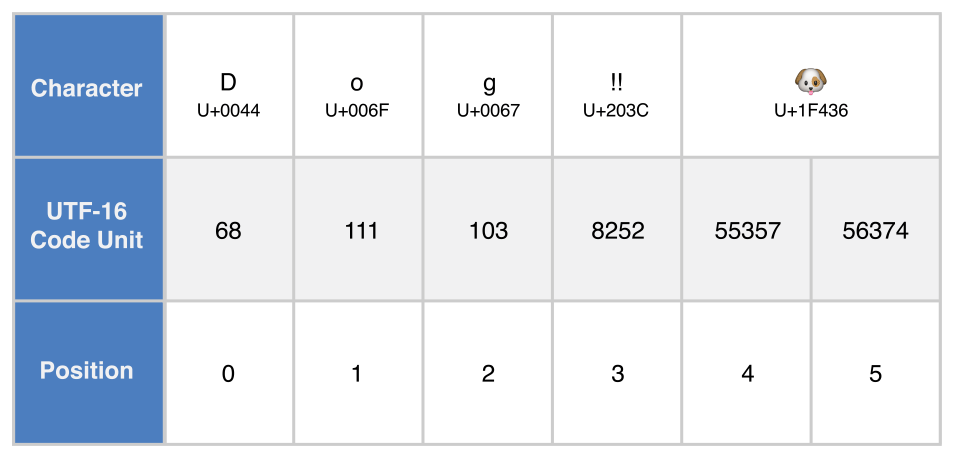
* A collection of UTF-8 code units (accessed with the string’s utf8 property)
* A collection of UTF-16 code units (accessed with the string’s utf16 property)
* A collection of 21-bit Unicode scalar values, equivalent to the string’s UTF-32 encoding form (accessed with the string’s unicodeScalars property)

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

1. let dogString = "Dog‼🐶"

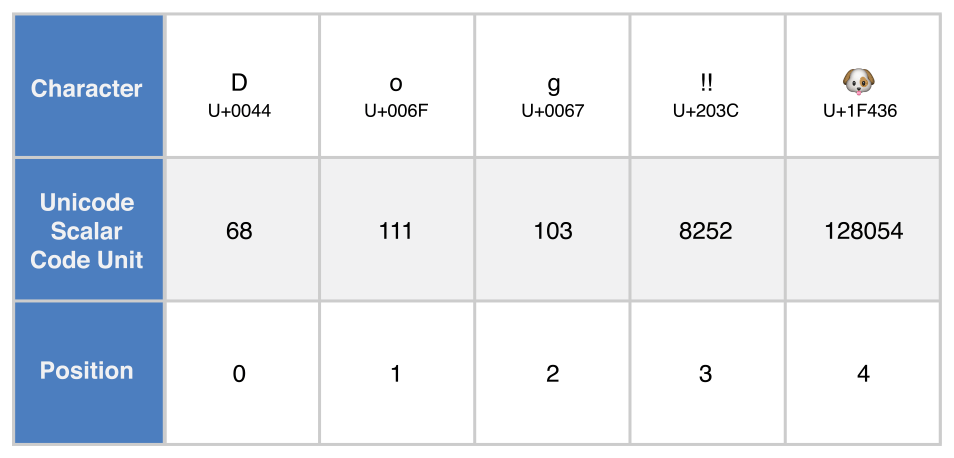


1. for codeUnit in dogString.utf8 {
2. print("\(codeUnit) ", terminator: "")
3. }
4. print("")
5. // 68 111 103 226 128 188 240 159 144 182

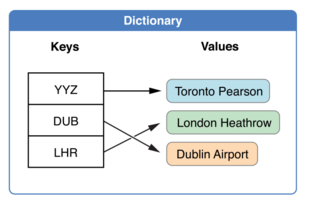


1. for codeUnit in dogString.utf16 {
2. print("\(codeUnit) ", terminator: "")
3. }
4. print("")
5. // 68 111 103 8252 55357 56374

* **Unicode Scalar Representation**



1. for scalar in dogString.unicodeScalars {
2. print("\(scalar.value) ", terminator: "")
3. }
4. print("")
5. // 68 111 103 8252 128054

* [Collection Types](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/CollectionTypes.html#//apple_ref/doc/uid/TP40014097-CH8-ID105)
* Dictionaries (字典)

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

NOTE:Swift’s Dictionary type is bridged to Foundation’s NSDictionary class.

For more information about using Dictionary with Foundation and Cocoa, see [*Using Swift with Cocoa and Objective-C (Swift 2.2)*](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/BuildingCocoaApps/index.html#//apple_ref/doc/uid/TP40014216).

### Dictionary Type Shorthand Syntax

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

NOTE: A dictionary Key type must conform to the Hashable protocol, like a set’s value type.

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

### Creating an Empty Dictionary

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

1. var namesOfIntegers = [Int: String]()
2. // namesOfIntegers is an empty [Int: String] dictionary
3. namesOfIntegers[16] = "sixteen"
4. // namesOfIntegers now contains 1 key-value pair
5. namesOfIntegers = [:]
6. // namesOfIntegers is once again an empty dictionary of type [Int: String]

### Creating a Dictionary with a Dictionary Literal

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

A *key-value pair* is a combination of a key and a value. In a dictionary literal, the key and value in each key-value pair are separated by a colon. The key-value pairs are written as a list, separated by commas, surrounded by a pair of square brackets:

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

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

### Accessing and Modifying a Dictionary

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

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

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

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

1. if airports.isEmpty {
2. print("The airports dictionary is empty.")
3. } else {
4. print("The airports dictionary is not empty.")
5. }
6. // Prints "The airports dictionary is not empty."

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

1. airports["LHR"] = "London"
2. // the airports dictionary now contains 3 items

You can also use subscript syntax to change the value associated with a particular key:

1. airports["LHR"] = "London Heathrow"
2. // the value for "LHR" has been changed to "London Heathrow"

As an alternative to subscripting, use a dictionary’s updateValue(\_:forKey:) method to set or update the value for a particular key. Like the subscript examples above, the updateValue(\_:forKey:) method sets a value for a key if none exists, or updates the value if that key already exists. Unlike a subscript, however, theupdateValue(\_:forKey:) method returns the *old* value after performing an update. This enables you to check whether or not an update took place.

The updateValue(\_:forKey:) method returns an optional value of the dictionary’s value type. For a dictionary that stores String values, for example, the method returns a value of type String?, or “optional String”. This optional value contains the old value for that key if one existed before the update, or nil if no value existed:

1. if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {
2. print("The old value for DUB was \(oldValue).")
3. }
4. // Prints "The old value for DUB was Dublin."

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

1. if let airportName = airports["DUB"] {
2. print("The name of the airport is \(airportName).")
3. } else {
4. print("That airport is not in the airports dictionary.")
5. }
6. // Prints "The name of the airport is Dublin Airport."

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

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

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

1. if let removedValue = airports.removeValueForKey("DUB") {
2. print("The removed airport's name is \(removedValue).")
3. } else {
4. print("The airports dictionary does not contain a value for DUB.")
5. }
6. // Prints "The removed airport's name is Dublin Airport."

### Iterating Over a Dictionary

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

1. for (airportCode, airportName) in airports {
2. print("\(airportCode): \(airportName)")
3. }
4. // YYZ: Toronto Pearson
5. // LHR: London Heathrow

For more about the for-in loop, see [For-In Loops](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html#//apple_ref/doc/uid/TP40014097-CH9-ID121).

You can also retrieve an iterable collection of a dictionary’s keys or values by accessing its keys and valuesproperties:

1. for airportCode in airports.keys {
2. print("Airport code: \(airportCode)")
3. }
4. // Airport code: YYZ
5. // Airport code: LHR
6. for airportName in airports.values {
7. print("Airport name: \(airportName)")
8. }
9. // Airport name: Toronto Pearson
10. // Airport name: London Heathrow

If you need to use a dictionary’s keys or values with an API that takes an Array instance, initialize a new array with the keys or values property:

1. let airportCodes = [String](airports.keys)
2. // airportCodes is ["YYZ", "LHR"]
3. let airportNames = [String](airports.values)
4. // airportNames is ["Toronto Pearson", "London Heathrow"]

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

* Arrays – see previous notes
* [Control Flow](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html#//apple_ref/doc/uid/TP40014097-CH9-ID120)
* Switch (particularly "Tuples", "Value Bindings", and "Where")
* [Functions](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-ID158)
* Functions with Multiple Return Values
* Optional Tuple Return Types
* Function Parameter Names
* [Closures](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Closures.html#//apple_ref/doc/uid/TP40014097-CH11-ID94)
* Closure Expressions
* Trailing Closures
* Capturing Values
* Closures Are Reference Types
* [Enumerations](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Enumerations.html#//apple_ref/doc/uid/TP40014097-CH12-ID145)
* Enumeration Syntax
* Matching Enumeration Values with a Switch Statement
* Associated Values
* Raw Values
* [Classes and Structures](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html#//apple_ref/doc/uid/TP40014097-CH13-ID82)
* Structures and Enumerations Are Value Types
* Classes Are Reference Types
* Choosing Between Classes and Structure
* Assignment and Copy Behaviour for Strings, Arrays, and Dictionaries
* [Properties](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Properties.html#//apple_ref/doc/uid/TP40014097-CH14-ID254)
* Stored Properties
* Lazy Stored Properties
* Computed Properties
* Property Observers
* Global and Local Variables
* Type Properties
* [Methods](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Methods.html#//apple_ref/doc/uid/TP40014097-CH15-ID234)
* Instance Methods
* Type Methods
* [Subscripts](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Subscripts.html#//apple_ref/doc/uid/TP40014097-CH16-ID305)
* Subscript Syntax
* Subscript Usage
* Subscript Options
* [Inheritance](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Inheritance.html#//apple_ref/doc/uid/TP40014097-CH17-ID193)
* Defining a Base Class
* Subclassing
* Overriding
* Preventing Overrides
* [Initialization](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Initialization.html#//apple_ref/doc/uid/TP40014097-CH18-ID203)
* Read the whole Section!
* [Optional Chaining](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/OptionalChaining.html#//apple_ref/doc/uid/TP40014097-CH21-ID245)
* Read the whole Section!
* [Type Casting](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TypeCasting.html#//apple_ref/doc/uid/TP40014097-CH22-ID338)
* Read the whole Section!
* [Nested Types](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TypeCasting.html#//apple_ref/doc/uid/TP40014097-CH22-ID338)
* Read the whole Section!
* [Generics](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Generics.html#//apple_ref/doc/uid/TP40014097-CH26-ID179)
* Read the whole Section!