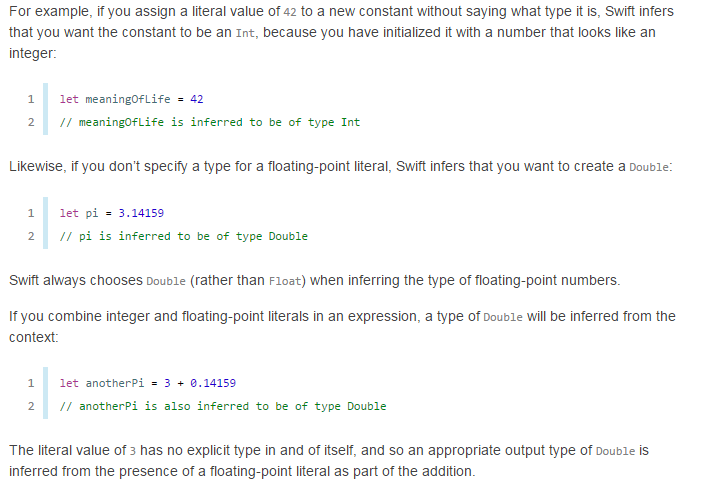
##### Swift Language Guide (Week 1 Milestone)

* [**The Basics**](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID309) **(基礎部份)**
  + **Constants and Variables (常數與變數)**
  1. let maximumNumberOfLoginAttempts = 10
  2. var currentLoginAttempt = 0
  + **Comments (註解)**

1. // this is a single line comment
2. /\* This is also a comment
3. but written over multiple lines \*/
   * **Semicolons (分號)**
   1. let cat = "I’m a cat"; print(cat)
   2. // Prints "I’m a cat"
   * **Integers (整數)**
   1. let minValue = UInt8.min // minValue is equal to 0, and is of type UInt8
   2. let maxValue = UInt8.max // maxValue is equal to 255, and is of type UInt8

* On a 32-bit platform, Int is the same size as Int32.
* On a 64-bit platform, Int is the same size as Int64.
* On a 32-bit platform, UInt is the same size as UInt32. // Unsigned Integer
* On a 64-bit platform, UInt is the same size as UInt64. // Unsigned Integer
  + **Floating-Point Numbers (浮點數)**
  + Double represents a 64-bit floating-point number.
  + Float represents a 32-bit floating-point number.
  + **Type Safety and Type Inference (安全類型與推測類型)**



Floating-point literals can be decimal (with no prefix), or hexadecimal (with a 0x prefix). They must always have a number (or hexadecimal number) on both sides of the decimal point. Decimal floats can also have an optional exponent, indicated by an uppercase or lowercase e; hexadecimal floats must have an exponent, indicated by an uppercase or lowercase p.

For decimal numbers with an exponent of exp, the base number is multiplied by 10exp:

* 1.25e2 means 1.25 x 102, or 125.0.
* 1.25e-2 means 1.25 x 10-2, or 0.0125.

For hexadecimal numbers with an exponent of exp, the base number is multiplied by 2exp:

* 0xFp2 means 15 x 22, or 60.0.
* 0xFp-2 means 15 x 2-2, or 3.75.

All of these floating-point literals have a decimal value of 12.1875:

1. let decimalDouble = 12.1875
2. let exponentDouble = 1.21875e1
3. let hexadecimalDouble = 0xC.3p0

Numeric literals can contain extra formatting to make them easier to read. Both integers and floats can be padded with extra zeros and can contain underscores to help with readability. Neither type of formatting affects the underlying value of the literal:

1. let paddedDouble = 000123.456
2. let oneMillion = 1\_000\_000
3. let justOverOneMillion = 1\_000\_000.000\_000\_1

* + **Booleans (布林值)**

1. let orangesAreOrange = true
2. let turnipsAreDelicious = false
   * **Optionals (可選類型)**

You use optionals in situations where a value may be absent. An optional says:

* There is a value, and it equals x

or

* There isn’t a value at all

Here’s an example of how optionals can be used to cope with the absence of a value. Swift’s Int type has an initializer which tries to convert a String value into an Int value. However, not every string can be converted into an integer. The string "123" can be converted into the numeric value 123, but the string"hello, world" does not have an obvious numeric value to convert to.

The example below uses the initializer to try to convert a String into an Int:

1. let possibleNumber = "123"
2. let convertedNumber = Int(possibleNumber)
3. // convertedNumber is inferred to be of type "Int?", or "optional Int"

### nil

You set an optional variable to a valueless state by assigning it the special value nil:

1. var serverResponseCode: Int? = 404
2. // serverResponseCode contains an actual Int value of 404
3. serverResponseCode = nil
4. // serverResponseCode now contains no value

Note

nil cannot be used with nonoptional constants and variables. If a constant or variable in your code needs to work with the absence of a value under certain conditions, always declare it as an optional value of the appropriate type.

If you define an optional variable without providing a default value, the variable is automatically set to nil for you:

1. var surveyAnswer: String?
2. // surveyAnswer is automatically set to nil

NOTE

Swift’s nil is not the same as nil in Objective-C. In Objective-C, nil is a pointer to a nonexistent object. In Swift, nil is not a pointer—it is the absence of a value of a certain type. Optionals of any type can be set to nil, not just object types.

### If Statements and Forced Unwrapping

You can use an if statement to find out whether an optional contains a value by comparing the optional against nil. You perform this comparison with the “equal to” operator (==) or the “not equal to” operator (!=).

If an optional has a value, it is considered to be “not equal to” nil:

1. if convertedNumber != nil {
2. print("convertedNumber contains some integer value.")
3. }
4. // Prints "convertedNumber contains some integer value."

Once you’re sure that the optional *does* contain a value, you can access its underlying value by adding an exclamation mark (!) to the end of the optional’s name. The exclamation mark effectively says, “I know that this optional definitely has a value; please use it.” This is known as *forced unwrapping* of the optional’s value:

1. if convertedNumber != nil {
2. print("convertedNumber has an integer value of \(convertedNumber!).")
3. }
4. // Prints "convertedNumber has an integer value of 123."

NOTE

Trying to use ! to access a nonexistent optional value triggers a runtime error. Always make sure that an optional contains a non-nil value before using ! to force-unwrap its value.

* **Optional Binding**

You use optional binding to find out whether an optional contains a value, and if so, to make that value available as a temporary constant or variable. Optional binding can be used with if and while statements to check for a value inside an optional, and to extract that value into a constant or variable, as part of a single action. if and while statements are described in more detail in [Control Flow](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html#//apple_ref/doc/uid/TP40014097-CH9-ID120).

Write an optional binding for an if statement as follows:

if let constantName *=* someOptional *{*

statements

}

You can rewrite the possibleNumber example from the [Optionals](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.html#//apple_ref/doc/uid/TP40014097-CH5-ID330) section to use optional binding rather than forced unwrapping:

1. if let actualNumber = Int(possibleNumber) {
2. print("\"\(possibleNumber)\" has an integer value of \(actualNumber)")
3. } else {
4. print("\"\(possibleNumber)\" could not be converted to an integer")
5. }
6. // Prints ""123" has an integer value of 123"

NOTE

Constants and variables created with optional binding in an if statement. are available only within the body of the if statement. In contrast, the constants and variables created with a guard statement are available in the lines of code that follow the guard statement, as described in [Early Exit](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html#//apple_ref/doc/uid/TP40014097-CH9-ID525),

### Implicitly Unwrapped Optionals

* As described above, optionals indicate that a constant or variable is allowed to have “no value”. Optionals can be checked with an if statement to see if a value exists, and can be conditionally unwrapped with optional binding to access the optional’s value if it does exist.
* Sometimes it is clear from a program’s structure that an optional will always have a value, after that value is first set. In these cases, it is useful to remove the need to check and unwrap the optional’s value every time it is accessed, because it can be safely assumed to have a value all of the time.
* These kinds of optionals are defined as implicitly unwrapped optionals. You write an implicitly unwrapped optional by placing an exclamation mark (String!) rather than a question mark (String?) after the type that you want to make optional.
* Implicitly unwrapped optionals are useful when an optional’s value is confirmed to exist immediately after the optional is first defined and can definitely be assumed to exist at every point thereafter. The primary use of implicitly unwrapped optionals in Swift is during class initialization, as described in [Unowned References and Implicitly Unwrapped Optional Properties](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AutomaticReferenceCounting.html" \l "//apple_ref/doc/uid/TP40014097-CH20-ID55).
* An implicitly unwrapped optional is a normal optional behind the scenes, but can also be used like a nonoptional value, without the need to unwrap the optional value each time it is accessed. The following example shows the difference in behavior between an optional string and an implicitly unwrapped optional string when accessing their wrapped value as an explicit String:

1. let possibleString: String? = "An optional string."
2. let forcedString: String = possibleString! // requires an exclamation mark
3. let assumedString: String! = "An implicitly unwrapped optional string."
4. let implicitString: String = assumedString // no need for an exclamation mark

You can think of an implicitly unwrapped optional as giving permission for the optional to be unwrapped automatically whenever it is used. Rather than placing an exclamation mark after the optional’s name each time you use it, you place an exclamation mark after the optional’s type when you declare it.

NOTE

If an implicitly unwrapped optional is nil and you try to access its wrapped value, you’ll trigger a runtime error. The result is exactly the same as if you place an exclamation mark after a normal optional that does not contain a value.

You can still treat an implicitly unwrapped optional like a normal optional, to check if it contains a value:

1. if assumedString != nil {
2. print(assumedString)
3. }
4. // Prints "An implicitly unwrapped optional string."

You can also use an implicitly unwrapped optional with optional binding, to check and unwrap its value in a single statement:

1. if let definiteString = assumedString {
2. print(definiteString)
3. }
4. // Prints "An implicitly unwrapped optional string."

NOTE

Do not use an implicitly unwrapped optional when there is a possibility of a variable becoming nil at a later point. Always use a normal optional type if you need to check for a nil value during the lifetime of a variable.

* [**Basic Operators**](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/BasicOperators.html#//apple_ref/doc/uid/TP40014097-CH6-ID60) **(基本運算子)**
* **Terminology (術語)**

Operators are unary, binary, or ternary: (一次元, 二次元, 或三次元)

* Unary operators operate on a single target (such as -a). Unary prefix operators appear immediately before their target (such as !b), and unary postfix operators appear immediately after their target (such as c!).
* Binary operators operate on two targets (such as 2 + 3) and are infix because they appear in between their two targets.
* Ternary operators operate on three targets. Like C, Swift has only one ternary operator, the ternary conditional operator (a ? b : c).

The values that operators affect are operands. In the expression 1 + 2, the + symbol is a binary operator and its two operands are the values 1 and 2.

* **Assignment Operator (指派運算子)**

The *assignment operator* (a = b) initializes or updates the value of a with the value of b:

1. let b = 10
2. var a = 5
3. a = b
4. // a is now equal to 10

If the right side of the assignment is a tuple with multiple values, its elements can be decomposed into multiple constants or variables at once:

1. let (x, y) = (1, 2)
2. // x is equal to 1, and y is equal to 2

Unlike the assignment operator in C and Objective-C, the assignment operator in Swift does not itself return a value. The following statement is not valid:

1. if x = y {
2. // this is not valid, because x = y does not return a value
3. }

* **Arithmetic Operators (數值運算子)**

Swift supports the four standard *arithmetic operators* for all number types:

* + Addition (+)
  + Subtraction (-)
  + Multiplication (\*)
  + Division (/)

1. 1 + 2 // equals 3
2. 5 - 3 // equals 2
3. 2 \* 3 // equals 6
4. 10.0 / 2.5 // equals 4.0

The addition operator is also supported for String concatenation:

1. "hello, " + "world" // equals "hello, world"

* **Compound Assignment Operators (複合指派運算子)**

Like C, Swift provides *compound assignment operators* that combine assignment (=) with another operation. One example is the *addition assignment operator* (+=):

1. var a = 1
2. a += 2
3. // a is now equal to 3

The expression a += 2 is shorthand for a = a + 2. Effectively, the addition and the assignment are combined into one operator that performs both tasks at the same time.

* **Comparison Operators (比較運算子)**

Swift supports all standard C comparison operators:

* Equal to (a == b)
* Not equal to (a != b)
* Greater than (a > b)
* Less than (a < b)
* Greater than or equal to (a >= b)
* Less than or equal to (a <= b)

Comparison operators are often used in conditional statements, such as the if statement:

1. let name = "world"
2. if name == "world" {
3. print("hello, world")
4. } else {
5. print("I'm sorry \(name), but I don't recognize you")
6. }
7. // prints "hello, world", because name is indeed equal to "world"

* **Range Operators (區間運算子)**

### Closed Range Operator (閉區間運算子)

The closed range operator (a...b) defines a range that runs from a to b, and includes the values a and b. The value of a must not be greater than b.

The closed range operator is useful when iterating over a range in which you want all of the values to be used, such as with a for-in loop:

1. for index in 1...5 {
2. print("\(index) times 5 is \(index \* 5)")
3. }
4. // 1 times 5 is 5
5. // 2 times 5 is 10
6. // 3 times 5 is 15
7. // 4 times 5 is 20
8. // 5 times 5 is 25

### Half-Open Range Operator

The half-open range operator (a..<b) defines a range that runs from a to b, but does not include b. It is said to be half-open because it contains its first value, but not its final value. As with the closed range operator, the value of a must not be greater than b. If the value of a is equal to b, then the resulting range will be empty.

Half-open ranges are particularly useful when you work with zero-based lists such as arrays, where it is useful to count up to (but not including) the length of the list:

1. let names = ["Anna", "Alex", "Brian", "Jack"]
2. let count = names.count
3. for i in 0..<count {
4. print("Person \(i + 1) is called \(names[i])")
5. }
6. // Person 1 is called Anna
7. // Person 2 is called Alex
8. // Person 3 is called Brian
9. // Person 4 is called Jack

Note that the array contains four items, but 0..<count only counts as far as 3 (the index of the last item in the array), because it is a half-open range

* **Logical Operators (邏輯運算子)**

Logical operators modify or combine the Boolean logic values true and false. Swift supports the three standard logical operators found in C-based languages:

* Logical NOT (!a)
* Logical AND (a && b)
* Logical OR (a || b)

### Combining Logical Operators (組合邏輯)

You can combine multiple logical operators to create longer compound expressions:

1. if enteredDoorCode && passedRetinaScan || hasDoorKey || knowsOverridePassword {
2. print("Welcome!")
3. } else {
4. print("ACCESS DENIED")
5. }
6. // Prints "Welcome!"

This example uses multiple && and || operators to create a longer compound expression. However, the &&and || operators still operate on only two values, so this is actually three smaller expressions chained together. The example can be read as:

If we’ve entered the correct door code and passed the retina scan, or if we have a valid door key, or if we know the emergency override password, then allow access.

Based on the values of enteredDoorCode, passedRetinaScan, and hasDoorKey, the first two subexpressions arefalse. However, the emergency override password is known, so the overall compound expression still evaluates to true.

* [**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) **(字串與字元)**
* **String Literals (字面字面量)**

1. let someString = "Some string literal value"

* **Initializing an Empty String (初始化空字串)**

1. var emptyString = "" // empty string literal
2. var anotherEmptyString = String() // initializer syntax
3. // these two strings are both empty, and are equivalent to each other

Find out whether a String value is empty by checking its Boolean isEmpty property:

1. if emptyString.isEmpty {
2. print("Nothing to see here")
3. }
4. // Prints "Nothing to see here"

* **String Mutability (字串可變性)**

You indicate whether a particular String can be modified (or mutated) by assigning it to a variable (in which case it can be modified), or to a constant (in which case it cannot be modified):

1. var variableString = "Horse"
2. variableString += " and carriage"
3. // variableString is now "Horse and carriage"
4. let constantString = "Highlander"
5. constantString += " and another Highlander"
6. // this reports a compile-time error - a constant string cannot be modified

* **Strings Are Value Types (字串是實值型別)**

Swift’s String type is a value type. If you create a new String value, that String value is copied when it is passed to a function or method, or when it is assigned to a constant or variable. In each case, a new copy of the existing String value is created, and the new copy is passed or assigned, not the original version. Value types are described in [Structures and Enumerations Are Value Types](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ClassesAndStructures.html#//apple_ref/doc/uid/TP40014097-CH13-ID88).

Swift’s copy-by-default String behavior ensures that when a function or method passes you a String value, it is clear that you own that exact String value, regardless of where it came from. You can be confident that the string you are passed will not be modified unless you modify it yourself.

Behind the scenes, Swift’s compiler optimizes string usage so that actual copying takes place only when absolutely necessary. This means you always get great performance when working with strings as value types.

* **Working with Characters (使用子元)**

String values can be constructed by passing an array of Character values as an argument to its initializer:

1. let catCharacters: [Character] = ["C", "a", "t", "!", "🐱"]
2. let catString = String(catCharacters)
3. print(catString)
4. // Prints "Cat!"

* **Concatenating Strings and Characters (連接字串與字元)**

String values can be added together (or concatenated) with the addition operator (+) to create a new Stringvalue

1. let string1 = "hello"
2. let string2 = " there"
3. var welcome = string1 + string2
4. // welcome now equals "hello there"

You can also append a String value to an existing String variable with the addition assignment operator (+=):

1. var instruction = "look over"
2. instruction += string2
3. // instruction now equals "look over there"

You can append a Character value to a String variable with the String type’s append() method:

1. let exclamationMark: Character = "!"
2. welcome.append(exclamationMark)
3. // welcome now equals "hello there!"

* **String Interpolation (字串插值)**

String interpolation is a way to construct a new String value from a mix of constants, variables, literals, and expressions by including their values inside a string literal. Each item that you insert into the string literal is wrapped in a pair of parentheses, prefixed by a backslash:

1. let multiplier = 3
2. let message = "\(multiplier) times 2.5 is \(Double(multiplier) \* 2.5)"
3. // message is "3 times 2.5 is 7.5"

* **Counting Characters (計算字元數量)**

To retrieve a count of the Character values in a string, use the count property of the string’s charactersproperty:

1. let unusualMenagerie = "Koala 🐨, Snail 🐌, Penguin 🐧, Dromedary 🐪"
2. print("unusualMenagerie has \(unusualMenagerie.characters.count) characters")
3. // Prints "unusualMenagerie has 40 characters"

* **Comparing Strings (比較字串)**

Swift provides three ways to compare textual values: string and character equality, prefix equality, and suffix equality.

### String and Character Equality (字串及字元相等)

String and character equality is checked with the “equal to” operator (==) and the “not equal to” operator (!=), as described in [Comparison Operators](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/BasicOperators.html#//apple_ref/doc/uid/TP40014097-CH6-ID70):

1. let quotation = "We're a lot alike, you and I."
2. let sameQuotation = "We're a lot alike, you and I."
3. if quotation == sameQuotation {
4. print("These two strings are considered equal")
5. }
6. // Prints "These two strings are considered equal"

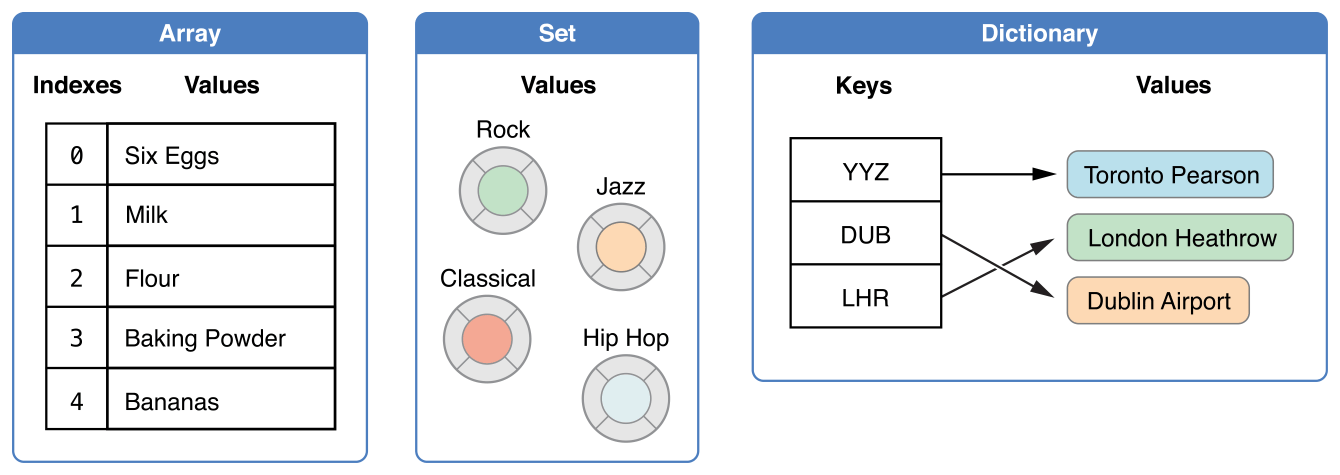
### Prefix and Suffix Equality (前綴及後綴相等)

To check whether a string has a particular string prefix or suffix, call the string’s hasPrefix(\_:) andhasSuffix(\_:) methods, both of which take a single argument of type String and return a Boolean value.

1. let romeoAndJuliet = [
2. "Act 1 Scene 1: Verona, A public place",
3. "Act 1 Scene 2: Capulet's mansion",
4. "Act 1 Scene 3: A street outside Capulet's mansion",
5. "Act 2 Scene 1: Outside Capulet's mansion",
6. "Act 2 Scene 2: Capulet's orchard",
7. "Act 2 Scene 3: Capulet's mansion",
8. ]
9. var act1SceneCount = 0
10. for scene in romeoAndJuliet {
11. if scene.hasPrefix("Act 1 ") {
12. act1SceneCount += 1
13. }
14. }
15. print("There are \(act1SceneCount) scenes in Act 1")
16. // Prints "There are 3 scenes in Act 1"

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

Swift provides three primary collection types, known as arrays, sets, and dictionaries, for storing collections of values. Arrays are ordered collections of values. Sets are unordered collections of unique values. Dictionaries are unordered collections of key-value associations.



* **Mutability of Collections**

If you create an array, a set, or a dictionary, and assign it to a variable, the collection that is created will be mutable. This means that you can change (or mutate) the collection after it is created by adding, removing, or changing items in the collection. If you assign an array, a set, or a dictionary to a constant, that collection is immutable, and its size and contents cannot be changed.

* **Arrays (陣列)**

An array stores values of the same type in an ordered list. The same value can appear in an array multiple times at different positions.

### Array Type Shorthand Syntax (陣列的簡單語法)

The type of a Swift array is written in full as Array<Element>, where Element is the type of values the array is allowed to store. You can also write the type of an array in shorthand form as [Element]. Although the two forms are functionally identical, the shorthand form is preferred and is used throughout this guide when referring to the type of an array.

### Creating an Empty Array (建立空的陣列)

You can create an empty array of a certain type using initializer syntax:

1. var someInts = [Int]()
2. print("someInts is of type [Int] with \(someInts.count) items.")
3. // Prints "someInts is of type [Int] with 0 items."

Alternatively, if the context already provides type information, such as a function argument or an already typed variable or constant, you can create an empty array with an empty array literal, which is written as [](an empty pair of square brackets):

1. someInts.append(3)
2. // someInts now contains 1 value of type Int
3. someInts = []
4. // someInts is now an empty array, but is still of type [Int]

### Creating an Array with a Default Value (建立有預設值的陣列)

Swift’s Array type also provides an initializer for creating an array of a certain size with all of its values set to the same default value. You pass this initializer the number of items to be added to the new array (calledcount) and a default value of the appropriate type (called repeatedValue):

1. var threeDoubles = [Double](count: 3, repeatedValue: 0.0)
2. // threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]

### Creating an Array by Adding Two Arrays Together

You can create a new array by adding together two existing arrays with compatible types with the addition operator (+). The new array’s type is inferred from the type of the two arrays you add together:

1. var anotherThreeDoubles = [Double](count: 3, repeatedValue: 2.5)
2. // anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]
3. var sixDoubles = threeDoubles + anotherThreeDoubles
4. // sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]

### Creating an Array with an Array Literal (建立有數值的陣列)

You can also initialize an array with an array literal, which is a shorthand way to write one or more values as an array collection. An array literal is written as a list of values, separated by commas, surrounded by a pair of square brackets:

* [value 1, value 2, value 3]

The example below creates an array called shoppingList to store String values:

1. var shoppingList: [String] = ["Eggs", "Milk"]
2. // shoppingList has been initialized with two initial items

### Accessing and Modifying an Array (存取和俢改陣列)

You access and modify an array through its methods and properties, or by using subscript syntax.

To find out the number of items in an array, check its read-only count property:

1. print("The shopping list contains \(shoppingList.count) items.")
2. // Prints "The shopping list contains 2 items."

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

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

You can add a new item to the end of an array by calling the array’s append(\_:) method:

1. shoppingList.append("Flour")
2. // shoppingList now contains 3 items, and someone is making pancakes

Alternatively, append an array of one or more compatible items with the addition assignment operator (+=):

1. shoppingList += ["Baking Powder"]
2. // shoppingList now contains 4 items
3. shoppingList += ["Chocolate Spread", "Cheese", "Butter"]
4. // shoppingList now contains 7 items

Retrieve a value from the array by using subscript syntax, passing the index of the value you want to retrieve within square brackets immediately after the name of the array:

1. var firstItem = shoppingList[0]
2. // firstItem is equal to "Eggs"

You can use subscript syntax to change an existing value at a given index:

1. shoppingList[0] = "Six eggs"
2. // the first item in the list is now equal to "Six eggs" rather than "Eggs"

You can also use subscript syntax to change a range of values at once, even if the replacement set of values has a different length than the range you are replacing. The following example replaces "Chocolate Spread", "Cheese", and "Butter" with "Bananas" and "Apples":

1. shoppingList[4...6] = ["Bananas", "Apples"]
2. // shoppingList now contains 6 items

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

1. shoppingList.insert("Maple Syrup", atIndex: 0)
2. // shoppingList now contains 7 items
3. // "Maple Syrup" is now the first item in the list

This call to the insert(\_:atIndex:) method inserts a new item with a value of "Maple Syrup" at the very beginning of the shopping list, indicated by an index of 0.

Similarly, you remove an item from the array with the removeAtIndex(\_:) method. This method removes the item at the specified index and returns the removed item (although you can ignore the returned value if you do not need it):

1. let mapleSyrup = shoppingList.removeAtIndex(0)
2. // the item that was at index 0 has just been removed
3. // shoppingList now contains 6 items, and no Maple Syrup
4. // the mapleSyrup constant is now equal to the removed "Maple Syrup" string

Any gaps in an array are closed when an item is removed, and so the value at index 0 is once again equal to "Six eggs":

1. firstItem = shoppingList[0]
2. // firstItem is now equal to "Six eggs"

If you want to remove the final item from an array, use the removeLast() method rather than theremoveAtIndex(\_:) method to avoid the need to query the array’s count property. Like the removeAtIndex(\_:)method, removeLast() returns the removed item:

1. let apples = shoppingList.removeLast()
2. // the last item in the array has just been removed
3. // shoppingList now contains 5 items, and no apples
4. // the apples constant is now equal to the removed "Apples" string

### Iterating Over an Array

You can iterate over the entire set of values in an array with the for-in loop:

1. for item in shoppingList {
2. print(item)
3. }
4. // Six eggs
5. // Milk
6. // Flour
7. // Baking Powder
8. // Bananas

If you need the integer index of each item as well as its value, use the enumerate() method to iterate over the array instead. For each item in the array, the enumerate() method returns a tuple composed of the index and the value for that item. You can decompose the tuple into temporary constants or variables as part of the iteration:

1. for (index, value) in shoppingList.enumerate() {
2. print("Item \(index + 1): \(value)")
3. }
4. // Item 1: Six eggs
5. // Item 2: Milk
6. // Item 3: Flour
7. // Item 4: Baking Powder
8. // Item 5: Bananas

* [**Control Flow**](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/ControlFlow.html#//apple_ref/doc/uid/TP40014097-CH9-ID120)
* **For Loops**

### For-In Loops

You use the for-in loop to iterate over a sequence, such as ranges of numbers, items in an array, or characters in a string.

Use a for-in loop with an array to iterate over its items.

1. let names = ["Anna", "Alex", "Brian", "Jack"]
2. for name in names {
3. print("Hello, \(name)!")
4. }
5. // Hello, Anna!
6. // Hello, Alex!
7. // Hello, Brian!
8. // Hello, Jack!

You can also iterate over a dictionary to access its key-value pairs. Each item in the dictionary is returned as a (key, value) tuple when the dictionary is iterated, and you can decompose the (key, value) tuple’s members as explicitly named constants for use within the body of the for-in loop. Here, the dictionary’s keys are decomposed into a constant called animalName, and the dictionary’s values are decomposed into a constant called legCount.

1. let numberOfLegs = ["spider": 8, "ant": 6, "cat": 4]
2. for (animalName, legCount) in numberOfLegs {
3. print("\(animalName)s have \(legCount) legs")
4. }
5. // ants have 6 legs
6. // cats have 4 legs
7. // spiders have 8 legs

Items in a Dictionary may not necessarily be iterated in the same order in which they were inserted. The contents of a Dictionary are inherently unordered, and iterating over them does not guarantee the order in which they will be retrieved. For more on arrays and dictionaries, see [Collection Types](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/CollectionTypes.html#//apple_ref/doc/uid/TP40014097-CH8-ID105).

* **While Loops**
  + while evaluates its condition at the start of each pass through the loop.

while condition {

statements

}

* + repeat-while evaluates its condition at the end of each pass through the loop.

repeat {

statements

} while condition

* **Conditional Statements**
  + **If**
  + **Switch**

A switch statement considers a value and compares it against several possible matching patterns. It then executes an appropriate block of code, based on the first pattern that matches successfully. A switchstatement provides an alternative to the if statement for responding to multiple potential states.

In its simplest form, a switch statement compares a value against one or more values of the same type.

switch some value to consider {

case value 1:

respond to value 1

case value 2,

value 3:

respond to value 2 or 3

default:

otherwise, do something else

}

This example uses a switch statement to consider a single lowercase character called someCharacter:

1. let someCharacter: Character = "e"
2. switch someCharacter {
3. case "a", "e", "i", "o", "u":
4. print("\(someCharacter) is a vowel")
5. case "b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
6. "n", "p", "q", "r", "s", "t", "v", "w", "x", "y", "z":
7. print("\(someCharacter) is a consonant")
8. default:
9. print("\(someCharacter) is not a vowel or a consonant")
10. }
11. // Prints "e is a vowel"

* **Control Transfer Statements**
  + **Continue**

The continue statement tells a loop to stop what it is doing and start again at the beginning of the next iteration through the loop. It says “I am done with the current loop iteration” without leaving the loop altogether.

* + **Break**

The break statement ends execution of an entire control flow statement immediately. The break statement can be used inside a switch statement or loop statement when you want to terminate the execution of theswitch or loop statement earlier than would otherwise be the case.

* + **Fallthrough**

Switch statements in Swift don’t fall through the bottom of each case and into the next one. Instead, the entire switch statement completes its execution as soon as the first matching case is completed. By contrast, C requires you to insert an explicit break statement at the end of every switch case to prevent fallthrough. Avoiding default fallthrough means that Swift switch statements are much more concise and predictable than their counterparts in C, and thus they avoid executing multiple switch cases by mistake.

If you need C-style fallthrough behavior, you can opt in to this behavior on a case-by-case basis with thefallthrough keyword. The example below uses fallthrough to create a textual description of a number.

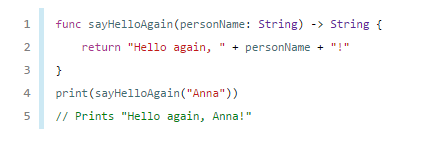
1. let integerToDescribe = 5
2. var description = "The number \(integerToDescribe) is"
3. switch integerToDescribe {
4. case 2, 3, 5, 7, 11, 13, 17, 19:
5. description += " a prime number, and also"
6. fallthrough
7. default:
8. description += " an integer."
9. }
10. print(description)
11. // Prints "The number 5 is a prime number, and also an integer."
    * **return**
    * **throw**

* [**Functions**](https://developer.apple.com/library/prerelease/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Functions.html#//apple_ref/doc/uid/TP40014097-CH10-ID158)
* **Defining and Calling Functions**

The function in the example below is called sayHello(\_:), because that’s what it does—it takes a person’s name as input and returns a greeting for that person. To accomplish this, you define one input parameter—a String value called personName—and a return type of String, which will contain a greeting for that person:

1. func sayHello(personName: String) -> String {
2. let greeting = "Hello, " + personName + "!"
3. return greeting
4. }

All of this information is rolled up into the function’s *definition*, which is prefixed with the func keyword. You indicate the function’s return type with the *return arrow* -> (a hyphen followed by a right angle bracket), which is followed by the name of the type to return.

The definition describes what the function does, what it expects to receive, and what it returns when it is done. The definition makes it easy for the function to be called unambiguously from elsewhere in your code:

1. print(sayHello("Anna"))
2. // Prints "Hello, Anna!"
3. print(sayHello("Brian"))
4. // Prints "Hello, Brian!"

* **Function Parameters and Return Values**

### Functions Without Parameters

Functions are not required to define input parameters. Here’s a function with no input parameters, which always returns the same String message whenever it is called:

1. func sayHelloWorld() -> String {
2. return "hello, world"
3. }
4. print(sayHelloWorld())
5. // Prints "hello, world"

### Functions With Multiple Parameters

Functions can have multiple input parameters, which are written within the function’s parentheses, separated by commas.

This function takes a person’s name and whether they have already been greeted as input, and returns an appropriate greeting for that person:

1. func sayHello(personName: String, alreadyGreeted: Bool) -> String {
2. if alreadyGreeted {
3. return sayHelloAgain(personName)
4. } else {
5. return sayHello(personName)
6. }
7. }
8. print(sayHello("Tim", alreadyGreeted: true))
9. // Prints "Hello again, Tim!"

### Functions with Multiple Return Values

You can use a tuple type as the return type for a function to return multiple values as part of one compound return value.

The example below defines a function called minMax(\_:), which finds the smallest and largest numbers in an array of Int values:

1. func minMax(array: [Int]) -> (min: Int, max: Int) {
2. var currentMin = array[0]
3. var currentMax = array[0]
4. for value in array[1..<array.count] {
5. if value < currentMin {
6. currentMin = value
7. } else if value > currentMax {
8. currentMax = value
9. }
10. }
11. return (currentMin, currentMax)
12. }

* **Function Types**

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

1. func addTwoInts(a: Int, \_ b: Int) -> Int {
2. return a + b
3. }
4. func multiplyTwoInts(a: Int, \_ b: Int) -> Int {
5. return a \* b
6. }

This example defines two simple mathematical functions called addTwoInts and multiplyTwoInts. These functions each take two Int values, and return an Int value, which is the result of performing an appropriate mathematical operation.

The type of both of these functions is (Int, Int) -> Int. This can be read as:

“A function type that has two parameters, both of type Int, and that returns a value of type Int.”

### Using Function Types

You use function types just like any other types in Swift. For example, you can define a constant or variable to be of a function type and assign an appropriate function to that variable:

1. var mathFunction: (Int, Int) -> Int = addTwoInts

“Define a variable called mathFunction, which has a type of ‘a function that takes two Int values, and returns an Int value.’ Set this new variable to refer to the function called addTwoInts.”

The addTwoInts(\_:\_:) function has the same type as the mathFunction variable, and so this assignment is allowed by Swift’s type-checker.

### Function Types as Parameter Types

You can use a function type such as (Int, Int) -> Int as a parameter type for another function. This enables you to leave some aspects of a function’s implementation for the function’s caller to provide when the function is called.

1. func printMathResult(mathFunction: (Int, Int) -> Int, \_ a: Int, \_ b: Int) {
2. print("Result: \(mathFunction(a, b))")
3. }
4. printMathResult(addTwoInts, 3, 5)
5. // Prints "Result: 8"

### Function Types as Return Types

You can use a function type as the return type of another function. You do this by writing a complete function type immediately after the return arrow (->) of the returning function.

The next example defines two simple functions called stepForward(\_:) and stepBackward(\_:). ThestepForward(\_:) function returns a value one more than its input value, and the stepBackward(\_:) function returns a value one less than its input value. Both functions have a type of (Int) -> Int:

1. func stepForward(input: Int) -> Int {
2. return input + 1
3. }
4. func stepBackward(input: Int) -> Int {
5. return input - 1
6. }

* [**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)
* **Comparing Classes and Structures**

**Classes and structures in Swift have many things in common. Both can:**

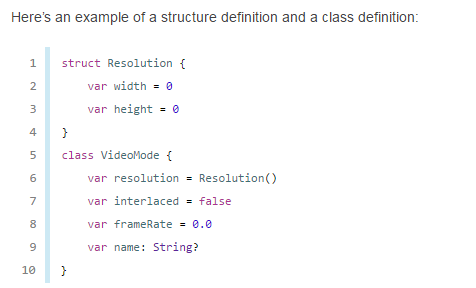
* + Define properties to store values
  + Define methods to provide functionality
  + Define subscripts to provide access to their values using subscript syntax
  + Define initializers to set up their initial state
  + Be extended to expand their functionality beyond a default implementation
  + Conform to protocols to provide standard functionality of a certain kind

**Classes have additional capabilities that structures do not:**

* + Inheritance enables one class to inherit the characteristics of another.
  + Type casting enables you to check and interpret the type of a class instance at runtime.
  + Deinitializers enable an instance of a class to free up any resources it has assigned.
  + Reference counting allows more than one reference to a class instance.

**Definition Syntax**

Classes and structures have a similar definition syntax. You introduce classes with the class keyword and structures with the struct keyword. Both place their entire definition within a pair of braces:

1. class SomeClass {
2. // class definition goes here
3. }
4. struct SomeStructure {
5. // structure definition goes here
6. }

### Class and Structure Instances

The syntax for creating instances is very similar

for both structures and classes:

1. let someResolution = Resolution()
2. let someVideoMode = VideoMode()

### Accessing Properties

You can access the properties of an instance using dot syntax. In dot syntax, you write the property name immediately after the instance name, separated by a period (.), without any spaces:

1. print("The width of someResolution is \(someResolution.width)")
2. // Prints "The width of someResolution is 0"

* **Classes Are Reference Types**

Unlike value types, *reference types* are *not* copied when they are assigned to a variable or constant, or when they are passed to a function. Rather than a copy, a reference to the same existing instance is used instead.

Here’s an example, using the VideoMode class defined above:

1. let tenEighty = VideoMode()
2. tenEighty.resolution = hd
3. tenEighty.interlaced = true
4. tenEighty.name = "1080i"
5. tenEighty.frameRate = 25.0

This example declares a new constant called tenEighty and sets it to refer to a new instance of the VideoModeclass. The video mode is assigned a copy of the HD resolution of 1920 by 1080 from before. It is set to be interlaced, and is given a name of "1080i". Finally, it is set to a frame rate of 25.0 frames per second.

Next, tenEighty is assigned to a new constant, called alsoTenEighty, and the frame rate of alsoTenEighty is modified:

1. let alsoTenEighty = tenEighty
2. alsoTenEighty.frameRate = 30.0

Because classes are reference types, tenEighty and alsoTenEighty actually both refer to the *same* VideoModeinstance. Effectively, they are just two different names for the same single instance.

Checking the frameRate property of tenEighty shows that it correctly reports the new frame rate of 30.0 from the underlying VideoMode instance:

1. print("The frameRate property of tenEighty is now \(tenEighty.frameRate)")
2. // Prints "The frameRate property of tenEighty is now 30.0"

Note that tenEighty and alsoTenEighty are declared as *constants*, rather than variables. However, you can still change tenEighty.frameRate and alsoTenEighty.frameRate because the values of the tenEighty andalsoTenEighty constants themselves do not actually change. tenEighty and alsoTenEighty themselves do not “store” the VideoMode instance—instead, they both *refer* to a VideoMode instance behind the scenes. It is theframeRate property of the underlying VideoMode that is changed, not the values of the constant references to that VideoMode.

### Identity Operators

Because classes are reference types, it is possible for multiple constants and variables to refer to the same single instance of a class behind the scenes. (The same is not true for structures and enumerations, because they are always copied when they are assigned to a constant or variable, or passed to a function.)

It can sometimes be useful to find out if two constants or variables refer to exactly the same instance of a class. To enable this, Swift provides two identity operators:

* Identical to (===)
* Not identical to (!==)

Use these operators to check whether two constants or variables refer to the same single instance:

1. if tenEighty === alsoTenEighty {
2. print("tenEighty and alsoTenEighty refer to the same VideoMode instance.")
3. }
4. // Prints "tenEighty and alsoTenEighty refer to the same VideoMode instance."

Note that “identical to” (represented by three equals signs, or ===) does not mean the same thing as “equal to” (represented by two equals signs, or ==):

* “Identical to” means that two constants or variables of class type refer to exactly the same class instance.
* “Equal to” means that two instances are considered “equal” or “equivalent” in value, for some appropriate meaning of “equal”, as defined by the type’s designer.

When you define your own custom classes and structures, it is your responsibility to decide what qualifies as two instances being “equal”. The process of defining your own implementations of the “equal to” and “not equal to” operators is described in [Equivalence Operators](https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/AdvancedOperators.html#//apple_ref/doc/uid/TP40014097-CH27-ID45).

### Pointers

If you have experience with C, C++, or Objective-C, you may know that these languages use pointers to refer to addresses in memory. A Swift constant or variable that refers to an instance of some reference type is similar to a pointer in C, but is not a direct pointer to an address in memory, and does not require you to write an asterisk (\*) to indicate that you are creating a reference. Instead, these references are defined like any other constant or variable in Swift.

* **Choosing Between Classes and Structures**
* You can use both classes and structures to define custom data types to use as the building blocks of your program’s code.
* However, structure instances are always passed by value, and class instances are always passed by reference. This means that they are suited to different kinds of tasks. As you consider the data constructs and functionality that you need for a project, decide whether each data construct should be defined as a class or as a structure.

As a general guideline, consider creating a structure when one or more of these conditions apply:

* The structure’s primary purpose is to encapsulate a few relatively simple data values.
* It is reasonable to expect that the encapsulated values will be copied rather than referenced when you assign or pass around an instance of that structure.
* Any properties stored by the structure are themselves value types, which would also be expected to be copied rather than referenced.
* The structure does not need to inherit properties or behavior from another existing type.

Examples of good candidates for structures include:

* The size of a geometric shape, perhaps encapsulating a width property and a height property, both of type Double.
* A way to refer to ranges within a series, perhaps encapsulating a start property and a length property, both of type Int.
* A point in a 3D coordinate system, perhaps encapsulating x, y and z properties, each of type Double.

In all other cases, define a class, and create instances of that class to be managed and passed by reference. In practice, this means that most custom data constructs should be classes, not structures.