Collection Types

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's created will be *mutable*. This means that you can change (or *mutate*) the collection after it's 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 can't be changed.

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.

Creating an Empty Array

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

```
var someInts: [Int] = []

print("someInts is of type [Int] with \(someInts.count)
items.")

// Prints "someInts is of type [Int] with 0 items."
```

```
someInts.append(3)

// someInts now contains 1 value of type Int

someInts = []

// someInts is now an empty array, but is still of type [Int]
```

Creating an Array with a Default Value

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 a default value of the appropriate type (called `repeating`): and the number of times that value is repeated in the new array (called `count`):

```
var threeDoubles = Array(repeating: 0.0, count: 3)

// threeDoubles is of type [Double], and equals [0.0, 0.0, 0.0]
```

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:

```
var anotherThreeDoubles = Array(repeating: 2.5, count: 3)

// anotherThreeDoubles is of type [Double], and equals [2.5, 2.5, 2.5]

var sixDoubles = threeDoubles + anotherThreeDoubles

// sixDoubles is inferred as [Double], and equals [0.0, 0.0, 0.0, 2.5, 2.5, 2.5]
```

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:

```
var shoppingList: [String] = ["Eggs", "Milk"]

// shoppingList has been initialized with two initial items
```

Accessing and Modifying an Array

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

```
print("The shopping list contains \(shoppingList.count) items.")

// 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`:

```
   if shoppingList.isEmpty {

print("The shopping list is empty.")

} else {

print("The shopping list isn't empty.")

}

// Prints "The shopping list isn't empty."

shoppingList.append("Flour")

// shoppingList now contains 3 items, and someone is making pancakes

shoppingList += ["Baking Powder"]

// shoppingList now contains 4 items

shoppingList += ["Chocolate Spread", "Cheese", "Butter"]

// shoppingList now contains 7 items

var firstItem = shoppingList[0]

// firstItem is equal to "Eggs"

shoppingList[0] = "Six eggs"

// the first item in the list is now equal to "Six eggs" rather than "Eggs"
```

```
shoppingList[4...6] = ["Bananas", "Apples"]

// shoppingList now contains 6 items

shoppingList.insert("Maple Syrup", at: 0)

// shoppingList now contains 7 items

// "Maple Syrup" is now the first item in the list

let mapleSyrup = shoppingList.remove(at: 0)

// the item that was at index 0 has just been removed

// shoppingList now contains 6 items, and no Maple Syrup

// the mapleSyrup constant is now equal to the removed "Maple Syrup" string
```

If you want to remove the final item from an array, use the `removeLast()` method rather than the `remove(at:)` method to avoid the need to query the array's `count` property. Like the `remove(at:)` method, `removeLast()` returns the removed item:

```
let apples = shoppingList.removeLast()

// the last item in the array has just been removed

// shoppingList now contains 5 items, and no apples

// 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:

```
for item in shoppingList {
print(item)
}
// Six eggs
// Milk
// Flour
// Baking Powder
// Bananas
```

Sets

A *set* stores distinct values of the same type in a collection with no defined ordering. You can use a set instead of an array when the order of items isn't important, or when you need to ensure that an item only appears once.

Creating and Initializing an Empty Set

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

```
var letters = Set<Character>()

print("letters is of type Set<Character> with
\(letters.count) items.")

// Prints "letters is of type Set<Character> with 0 items."
```

```
letters.insert("a")

// letters now contains 1 value of type Character

letters = []

// letters is now an empty set, but is still of type Set<Character>
```

```
var favoriteGenres: Set = ["Rock", "Classical", "Hip hop"]

print("I have \(favoriteGenres.count) favorite music genres.")

// Prints "I have 3 favorite music genres."

if favoriteGenres.isEmpty {

print("As far as music goes, I'm not picky.")

} else {

print("I have particular music preferences.")

}

// Prints "I have particular music preferences."

if let removedGenre = favoriteGenres.remove("Rock") {

print("\(removedGenre)? I'm over it.")

} else {

print("I never much cared for that.")

}

// Prints "Rock? I'm over it."
```

```
if favoriteGenres.contains("Funk") {
print("I get up on the good foot.")
} else {
print("It's too funky in here.")
}
// Prints "It's too funky in here."
let oddDigits: Set = [1, 3, 5, 7, 9]
let evenDigits: Set = [0, 2, 4, 6, 8]
let singleDigitPrimeNumbers: Set = [2, 3, 5, 7]

oddDigits.union(evenDigits).sorted()
// [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
oddDigits.intersection(evenDigits).sorted()
// []
oddDigits.subtracting(singleDigitPrimeNumbers).sorted()
// [1, 9]
oddDigits.symmetricDifference(singleDigitPrimeNumbers).sorted()
// [1, 2, 9]
```

Dictionaries

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

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

print("The airports dictionary contains \(airports.count) items.")

// Prints "The airports dictionary contains 2 items."

if airports.isEmpty {

print("The airports dictionary is empty.")

} else {

print("The airports dictionary isn't empty.")

}

// Prints "The airports dictionary isn't empty."

airports["LHR"] = "London"

// the airports dictionary now contains 3 items

airports["LHR"] = "London Heathrow"

// the value for "LHR" has been changed to "London Heathrow"

if let oldValue = airports.updateValue("Dublin Airport", forKey: "DUB") {

print("The old value for DUB was \(oldValue).")

}

// Prints "The old value for DUB was Dublin."

airports["APL"] = "Apple International"
```

```
// "Apple International" isn't the real airport for APL, so
delete it

airports["APL"] = nil

// APL has now been removed from the dictionary

if let removedValue = airports.removeValue(forKey: "DUB") {

print("The removed airport's name is \(removedValue).")

} else {

print("The airports dictionary doesn't contain a value for DUB.")

}

// 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:

```
for (airportCode, airportName) in airports {

print("\(airportCode): \(airportName)")

}

// LHR: London Heathrow

// YYZ: Toronto Pearson
```

```
for airportCode in airports.keys {
```

```
print("Airport code: \(airportCode)")

}

// Airport code: LHR

// Airport code: YYZ


for airportName in airports.values {

print("Airport name: \(airportName)")

}

// Airport name: London Heathrow

// Airport name: Toronto Pearson

let airportCodes = [String](airports.keys)

// airportCodes is ["LHR", "YYZ"]


let airportNames = [String](airports.values)

// airportNames is ["London Heathrow", "Toronto Pearson"]
```