11. Dictionaries

A dictionary is an unordered collection that stores multiple values of the same type. Each value from the dictionary is asociated with a unique key. All the keys have the same type.

The type of a dictionary is determined by the type of the keys and the type of the values. A dictionary of type `[String:Int]` has keys of type `String` and values of type `Int`.

Declare Dictionaries

To declare a dictionary you can use the square brackets syntax(`[KeyType:ValueType]`).

```
var dictionary: [String:Int]
```

You can initialize a dictionary with a dictionary literal. A dictionary literal is a list of key-value pairs, separated by commas, surrounded by a pair of square brackets. A key-value pair is a combination of a key and a value separate by a colon(:).

[`key` : `value` , `key` : `value` , `...`]

```
var dictionary: [String:Int] = [
    "one" : 1,
    "two" : 2,
    "three" : 3
]
```

Keep in mind that you can create empty dictionary using the empty dictionary literal (`[:]`).

```
var emptyDictionary: [Int:Int] = [:]
```

Getting values

You can access specific elements from a dictionary using the subscript syntax. To do this pass the key of the value you want to retrieve within square brackets immediately after the name of the dictionary. Because it's possible not to have a value associated with the provided key the subscript will return an optional value of the value type.

To unwrap the value returned by the subscript you can do one of two things: use optional binding or force the value if you know for sure it exists.

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
    "two" : 2,
    "three" : 3,
    "four" : 4,
    "five" : 5,
    "six" : 6,
    "seven" : 7,
    "eight" : 8,
    "nine" : 9
]

stringsAsInts["zero"] // Optional(0)
stringsAsInts["three"] // Optional(3)
stringsAsInts["ten"] // nil

// Unwaraping the optional using optional binding
if let twoAsInt = stringsAsInts["two"] {
    print(twoAsInt) // 2
}

// Unwaraping the optional using the forced value operator (!)
stringsAsInts["one"]! // 1
```

To get all the values from a dictionary you can use the `for-in` syntax. It's similar to the array for in syntax with the exception that instead of getting only the value in each step you also get the key associated with that value inside of a tuple.

```
var userInfo: [String: String] = [
    "first_name" : "Andrei",
    "last_name" : "Puni",
    "job_title" : "Mad scientist"
]

for (key, value) in userInfo {
    print("\(key): \(value)")
}
```

To get the number of elements (key-value pairs) in a dictionary you can use the `count` property.

```
print(userInfo.count) // 3
```

Updating values

The simplest way to add a value to a dictionary is by using the subscript syntax:

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
```

```
    "two" : 2
]

stringsAsInts["three"] = 3
```

Using the subscript syntax you can change a the value associated with a key:

```
stringsAsInts["three"] = 10
```

You can use the `updateValue(forKey:)` method to update the value associated with a key, if there was no value for that key it will be added. The method will return the old value wrapped in an optional or nil if there was no value before.

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
    "two" : 2
]

stringsAsInts.updateValue(3, forKey: "three") // nil
stringsAsInts.updateValue(10, forKey: "three") // Optional(3)
```

To remove a value from the dictionary you can use the subscript syntax to set the value to nil, or the `removeValueForKey()` method.

```
stringsAsInts["three"] = nil

stringsAsInts.removeValueForKey("three")
```

Type Inference

Thanks to Swift's type inference, you don't have to declare the type of a dictionary if you initialize it with something other than an empty dictionary literal(`[:]`).

```
// powersOfTwo will have the type [Int:Int]
var powersOfTwo = [
    1 : 2,
    2 : 4,
    3 : 8,
    4 : 16
]

// userInfo will have the type [String:String]
var userInfo = [
    "first_name" : "Silviu",
    "last_name" : "Pop",
```

```
    "job_title" : "evil genius"
]
```

Copy Behavior

Swift's dictionaries are value types. This means that dictionaries are copied when they are assigned to a new constant or variable, or when they are passed to a function or method.

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
    "two" : 2
]

var justACopy = stringsAsInts

justACopy["zero"] = 100

print(stringsAsInts) // [zero: 0, one: 1, two: 2]
print(justACopy) // [zero: 100, one: 1, two: 2]
```

Keep in mind that this is not true for Objective-C dictionaries (`NSDictionary` and `NSMutableDictionary`).

Mutability

If you create 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. Conversely, if you assign a dictionary to a constant, that array or dictionary is immutable, and its size and contents cannot be changed. In other words if you want to be able to change a dictionary declare it using the `var` keyword, and if you don't want to be able to change it use the `let` keyword.

```
var stringsAsInts: [String:Int] = [
    "zero" : 0,
    "one" : 1,
    "two" : 2
]

stringsAsInts["three"] = 3 // [zero: 0, one: 1, two: 2, three: 3]
stringsAsInts["zero"] = nil // [one: 1, two: 2, three: 3]

let powersOfTwo = [
    1 : 2,
    2 : 4,
    3 : 8,
    4 : 16
]

// this will give a runtime error because powersOfTwo is immutable
powersOfTwo[5] = 32
```

```
powersOfTwo.removeValueForKey(1) // this will give a similar error
```