Chapter 16: Loops

Section 16.1: For-in loop

The **for-in** loop allows you to iterate over any sequence.

Iterating over a range

You can iterate over both half-open and closed ranges:

```
for i in 0..<3 {
    print(i)
}

for i in 0...2 {
    print(i)
}

// Both print:
// 0
// 1
// 2
```

Iterating over an array or set

```
let names = ["James", "Emily", "Miles"]

for name in names {
   print(name)
}

// James
// Emily
// Miles
```

Version = 2.1 Version = 2.2

If you need the index for each element in the array, you can use the `enumerate()` method on `SequenceType`.

```
for (index, name) in names.enumerate() {
   print("The index of \(name) is \(index).")
}

// The index of James is 0.
// The index of Emily is 1.
// The index of Miles is 2.
```

`enumerate()` returns a lazy sequence containing pairs of elements with consecutive `Int`s, starting from 0. Therefore with arrays, these numbers will correspond to the given index of each element – however this may not be the case with other kinds of collections.

Version ≥ 3.0

In Swift 3, `enumerate()` has been renamed to `enumerated()`:

```
for (index, name) in names.enumerated() {
   print("The index of \(name) is \(index).")
}
```

Iterating over a dictionary

```
let ages = ["James": 29, "Emily": 24]

for (name, age) in ages {
    print(name, "is", age, "years old.")
}

// Emily is 24 years old.
// James is 29 years old.
```

Iterating in reverse

Version = 2.1 Version = 2.2

You can use the `reverse()` method on `SequenceType` in order to iterate over any sequence in reverse:

```
for i in (0..<3).reverse() {
    print(i)
}

for i in (0...2).reverse() {
    print(i)
}

// Both print:
// 2
// 1
// 0

let names = ["James", "Emily", "Miles"]

for name in names.reverse() {
    print(name)
}

// Miles
// Emily
// James
```

Version ≥ 3.0

In Swift 3, `reverse()` has been renamed to `reversed()`:

```
for i in (0..<3).reversed() {
    print(i)
}
```

Iterating over ranges with custom stride

Version = 2.1 Version = 2.2

By using the `stride(_:_:)` methods on `Strideable` you can iterate over a range with a custom stride:

```
for i in 4.stride(to: 0, by: -2) {
    print(i)
}

// 4
// 2

for i in 4.stride(through: 0, by: -2) {
    print(i)
}
```

```
// 4
// 2
// 0
```

Version = 1.2 Version ≥ 3.0

In Swift 3, the `stride(_:_:)` methods on `Stridable` have been replaced by the global `stride(_:_:_:)` functions:

```
for i in stride(from: 4, to: 0, by: -2) {
    print(i)
}

for i in stride(from: 4, through: 0, by: -2) {
    print(i)
}
```

Section 16.2: Repeat-while loop

Similar to the while loop, only the control statement is evaluated after the loop. Therefore, the loop will always execute at least once.

```
var i: Int = 0

repeat {
   print(i)
   i += 1
} while i < 3

// 0
// 1
// 2
```

Section 16.3: For-in loop with filtering

1. `where` clause

By adding a `where` clause you can restrict the iterations to ones that satisfy the given condition.

```
for i in 0..<5 where i % 2 == 0 {
    print(i)
}

// 0
// 2
// 4


let names = ["James", "Emily", "Miles"]

for name in names where name.characters.contains("s") {
    print(name)
}

// James
// Miles
```

2. `case` clause

It's useful when you need to iterate only through the values that match some pattern:

```
let points = [(5, 0), (31, 0), (5, 31)]
for case (_, 0) in points {
    print("point on x-axis")
}

//point on x-axis
//point on x-axis
```

Also you can filter optional values and unwrap them if appropriate by adding ? mark after binding constant:

```
let optionalNumbers = [31, 5, nil]
for case let number? in optionalNumbers {
    print(number)
}

//31
//5
```

Section 16.4: Sequence Type forEach block

A type that conforms to the SequenceType protocol can iterate through it's elements within a closure:

```
collection.forEach { print($0) }
```

The same could also be done with a named parameter:

```
collection.forEach { item in
    print(item)
}
```

*Note: Control flow statements (such as break or continue) may not be used in these blocks. A return can be called, and if called, will immediately return the block for the current iteration (much like a continue would). The next iteration will then execute.

```
let arr = [1,2,3,4]

arr.forEach {

    // blocks for 3 and 4 will still be called
    if $0 == 2 {
        return
    }
}
```

Section 16.5: while loop

A `while` loop will execute as long as the condition is true.

```
var count = 1

while count < 10 {
    print("This is the \(count) run of the loop")
    count += 1
}
```

Section 16.6: Breaking a loop

A loop will execute as long as its condition remains true, but you can stop it manually using the `break` keyword. For example:

```
var peopleArray = ["John", "Nicole", "Thomas", "Richard", "Brian", "Novak", "Vick", "Amanda",
"Sonya"]
var positionOfNovak = 0

for person in peopleArray {
    if person == "Novak" { break }
    positionOfNovak += 1
}

print("Novak is the element located on position [\(positionOfNovak)] in peopleArray.")
//prints out: Novak is the element located on position 5 in peopleArray. (which is true)
```