Arrays, Loops, and Sets

Swift Sets

We can use a set to store unique elements of the same data type.

```
var paintingsInMOMA: Set = ["The Dream",
"The Starry Night", "The False Mirror"]
```

Empty Sets

An empty set is a set that contains no values inside of it.

```
var team = Set<String>()

print(team)
// Prints: []
```

Populated Sets

To create a set populated with values, use the `Set` keyword before the assignment operator.
The values of the set must be contained within brackets `[]` and separated with commas `,` .

```
var vowels: Set = ["a", "e", "i", "o",
"u"]
```

.insert()

To insert a single value into a set, append `.insert()` to a set and place the new value inside the parentheses `()` .

```
var cookieJar: Set = ["Chocolate Chip",
"Oatmeal Raisin"]

// Add a new element
cookieJar.insert("Peanut Butter Chip")
```

.remove() and .removeAll() Methods

To remove a single value from a set, append `.remove()` to a set with the value to be removed placed inside the parentheses `()` .
To remove every single value from a set at once, append `.removeAll()` to a set.

```
var oddNumbers: Set = [1, 2, 3, 5]

// Remove an existing element
oddNumbers.remove(2)

// Remove all elements
oddNumbers.removeAll()
```

.contains()

Appending `.contains()` to an existing set with an item in the parentheses `()` will return a `true` or `false` value that states whether the item exists within the set.

```
var names: Set = ["Rosa", "Doug", "Waldo"]

print(names.contains("Lola")) // Prints:
false

if names.contains("Waldo"){
  print("There's Waldo!")
} else {
  print("Where's Waldo?")
}
// Prints: There's Waldo!
```

Iterating Over a Set

A `for` - `in` loop can be used to iterate over each item in a set.

```
var recipe: Set = ["Chocolate chips",
"Eggs", "Flour", "Sugar"]

for ingredient in recipe {
  print ("Include \(ingredient) in the
recipe.")
}
```

.isEmpty Property

Use the built-in property `.isEmpty` to check if a set has no values contained in it.

```
var emptySet = Set<String>()

print(emptySet.isEmpty)  // Prints: true

var populatedSet: Set = [1, 2, 3]

print(populatedSet.isEmpty) // Prints:
false
```

.count Property

The property `.count` returns the number of elements contained within a set.

```
var band: Set = ["Guitar", "Bass",
"Drums", "Vocals"]

print("There are \(band.count) players in
the band.")
// Prints: There are 4 players in the
band.
```

.intersection() Operation

The `.intersection()` operation populates a new set of elements with the overlapping elements of two sets.

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC = setA.intersection(setB)
print(setC)  // Prints: ["D", "C"]
```

.union() Operation

The `.union()` operation populates a new set by taking all the values from two sets and combining them.

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC = setA.union(setB)
print(setC)
// Prints: ["B", "A", "D", "F", "C", "E"]
```

.symmetricDifference() Operation

The `.symmetricDifference()` operation creates a new set with all the non-overlapping values between two sets.

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D", "E", "F"]

var setC = setA.symmetricDifference(setB)
print(setC)
// Prints: ["B", "E", "F", "A"]
```

.subtracting() Operation

The `.subtracting()` operation removes the values of one second set from another set and stores the remaining values in a new set.

```
var setA: Set = ["A", "B", "C", "D"]
var setB: Set = ["C", "D"]

var setC = setA.subtracting(setB)
print(setC)
// Prints: ["B", "A"]
```

Array

An array stores an ordered collection of values of the same data type.

Use the initializer syntax, `[Type]()`, to create an empty array of a certain type.

```
var scores = [Int]()

// The array is empty: []
```

Initialize with Array Literal

An array can be initialized with an array literal, which is a short-hand method of writing one or more values as an array collection.

An array literal is written as a list of values, separated by commas, and surrounded by a pair of square brackets.

```
// Using type inference:
var snowfall = [2.4, 3.6, 3.4, 1.8, 0.0]

// Being explicit with the type:
var temp: [Int] = [33, 31, 30, 38, 44]
```

Index

An index refers to an item's position within an ordered list.

Use the subscript syntax, `array[index]`, to retrieve an individual element from an array.

Note: Swift arrays are zero-indexed, meaning the first element has index 0.

```
var vowels = ["a", "e", "i", "o", "u"]

print(vowels[0])  // Prints: a
print(vowels[1])  // Prints: e
print(vowels[2])  // Prints: i
print(vowels[3])  // Prints: o
print(vowels[4])  // Prints: u
```

.count Property

The `.count` property returns the number of elements in an array.

```
var grocery = ["🥓", "🥞", "🍪", "🥛",
"🍊"]

print(grocery.count)

// Prints: 5
```

`.append()` Method and += Operator

The `.append()` method can be called on an array to add an item to the end of the array.
The `+=` addition assignment operator can be used to add the elements of another array to the existing array.

```
var gymBadges = ["Boulder", "Cascade"]

gymBadges.append("Thunder")
gymBadges += ["Rainbow", "Soul"]

// ["Boulder", "Cascade", "Thunder",
"Rainbow", "Soul"]
```

`.insert()` and `.remove()` Methods

The `.insert()` method can be called on an array to add an element at a specified index. It takes two arguments: `value` and `at: index`.
The `.remove()` method can be called on an array to remove an element at a specified index. It takes one argument: `at: index`.

```
var moon = ["🌕", "🌗", "🌘", "🌑"]

moon.insert("🌕", at: 0)

// ["🌕", "🌕", "🌗", "🌘", "🌑"]

moon.remove(at: 4)

// ["🌕", "🌕", "🌗", "🌘"]
```

Iterating Over an Array

In Swift, a `for`-`in` loop can be used to iterate through the items of an array.
This is a powerful tool for working with and manipulating a large amount of data.

```
var employees = ["Michael", "Dwight",
"Jim", "Pam", "Andy"]

for person in employees {
  print(person)
}

// Prints: Michael
// Prints: Dwight
// Prints: Jim
// Prints: Pam
// Prints: Andy
```

Ranges

Ranges created by the `...` operator will include the numbers from the lower bound to (and includes) the upper bound.

```
let zeroToThree = 0...3

// zeroToThree: 0, 1, 2, 3
```

`stride()` Function

Calling `stride()` with the 3 necessary arguments creates a collection of numbers; the arguments decide the starting number to, the (excluded) ending number, and how to increment/decrement from the start to the end.

```
for oddNum in stride(from: 1, to: 5, by:
2) {
  print(oddNum)
}

// Prints: 1
// Prints: 3
```

`for-in` Loop

The `for`-`in` loop is used to iterate over collections, including strings and ranges.

```
for char in "hehe" {
  print(char)
}

// Prints: h
// Prints: e
// Prints: h
// Prints: e
```

`continue` Keyword

The `continue` keyword will force the loop to move on to the next iteration.

```
for num in 0...5 {
  if num % 2 == 0 {
    continue
  }
  print(num)
}

// Prints: 1
// Prints: 3
// Prints: 5
```

break Keyword

To terminate a loop before its completion, use the `break` keyword.

```
for char in
"supercalifragilisticexpialidocious" {
  if char == "c" {
    break
  }
  print(char)
}

// Prints: s
// Prints: u
// Prints: p
// Prints: e
// Prints: r
```

Using Underscore

Use `_` instead of a placeholder variable if the variable is not referenced in the `for` - `in` loop body.

```
for _ in 1...3 {
  print("Olé")
}

// Prints: Olé
// Prints: Olé
// Prints: Olé
```

while Loop

A `while` loop accepts a condition and continually executes its body's code for as long as the provided condition is `true` .
If the condition is never `false` then the loop continues to run and the program is stuck in an infinite loop.

```
var counter = 1
var stopNum = Int.random(in: 1...10)

while counter < stopNum {
  print(counter)
  counter += 1
}

// The loop prints until the stop
condition is met
```