A.STEFANCHUK IT STEP

SWIFT PART 2

OPERATORS

OPERATORS

- Unary (such as -a)
- Binary (such as 2 + 3)
- Ternary - a ? b : c

ASSIGNMENT OPERATOR

- let name = “Darth“
- let (x, y) = (10, 20)
- Assignment does not return any value

```
var x = 1
var y = 2

if x = y {
    // do sth
}
```

Use of '=' in a boolean context, did you mean '=='?

ARITHMETIC OPERATORS

- + (Addition)
- - (Subtraction)
- * (Multiplication)
- / (Division)
- % (Remainder)
- Compound assignment - += -= *=

There is no ++ operator

COMPARISON OPERATORS

- == (Equals)
- != (Not equal)
- There is also === / !=== to compare class references
- >
- <
- >=
- <=

LOGICAL OPERATORS

- ! Logical not
- && Logical and
- || Logical or

```
if (x == y && x > 10) || (x < 100 && y > 50) {
    // do sth
}
```

ADVANCED LOGICAL OPERATORS

- Bitwise not ~
- Bitwise AND &
- Bitwise OR |
- Bitwise XOR ^
- Bitwise shifts << >>

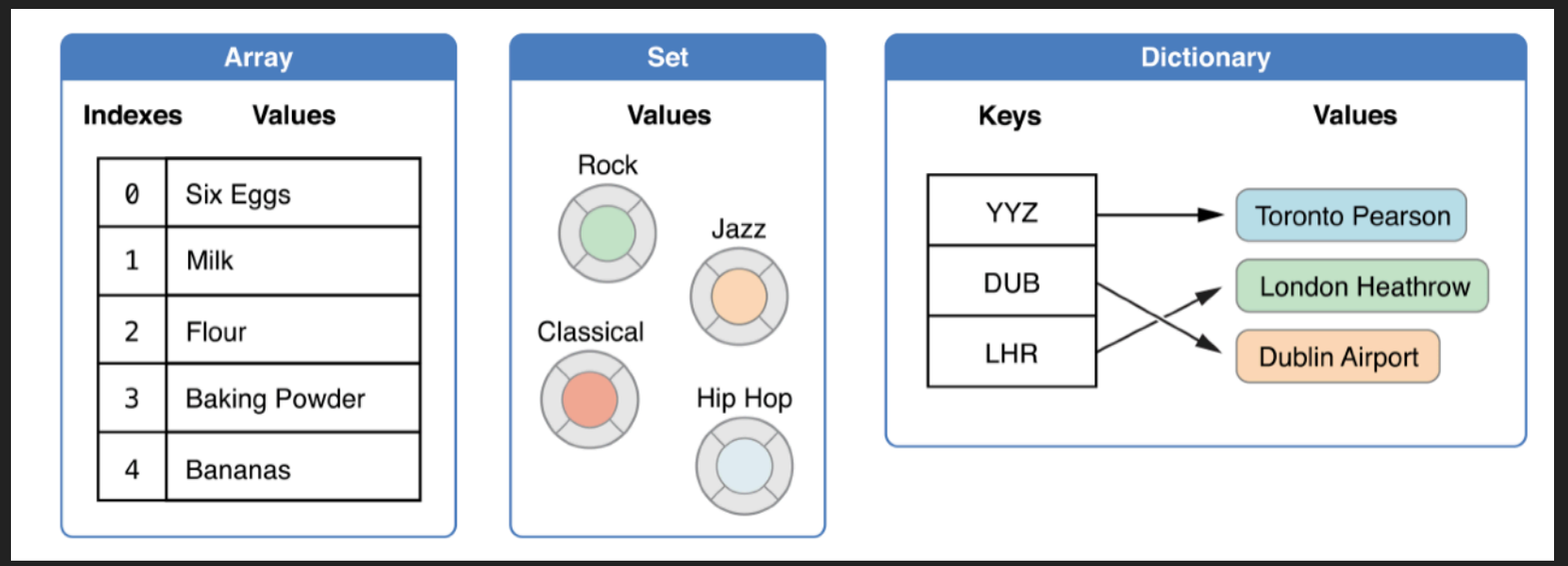
Useful when working with some Bluetooth devices

You can create own custom operators

COLLECTIONS

COLLECTION TYPES

- Arrays - ordered collection of values
- Dictionaries - unordered collection of key-value pairs
- Sets - unordered collection of unique values
- Those are generic collections and structs



ARRAYS

- Ordered list of values and used the most

```
let series = ["Breaking bad", "Money Heist", "Dr. House", "Dark"]
var myFavorites = [String]()
var blackList: Array<String> = []
```

```
var marks = Array(repeating: 5, count: 10)
```

- You can create array just adding two arrays

```
var marks = Array(repeating: 5, count: 10)
var marks2 = [19, 102, 20]

let allMarks = marks + marks2
```

ARRAY OPERATIONS

- Elements are accessed by index

```swift
var series = ["Breaking bad", "Money Heist", "Dr. House", "Dark"]
series.append("How I Met Your Mother")
series.removeFirst()
series.remove(at: 2)
series.insert("Ozark", at: 1)
series[1]
series.isEmpty
```

```swift
for serial in series {
    print(serial)
}
```

Accessing out of bounds element will cause a crash

SETS

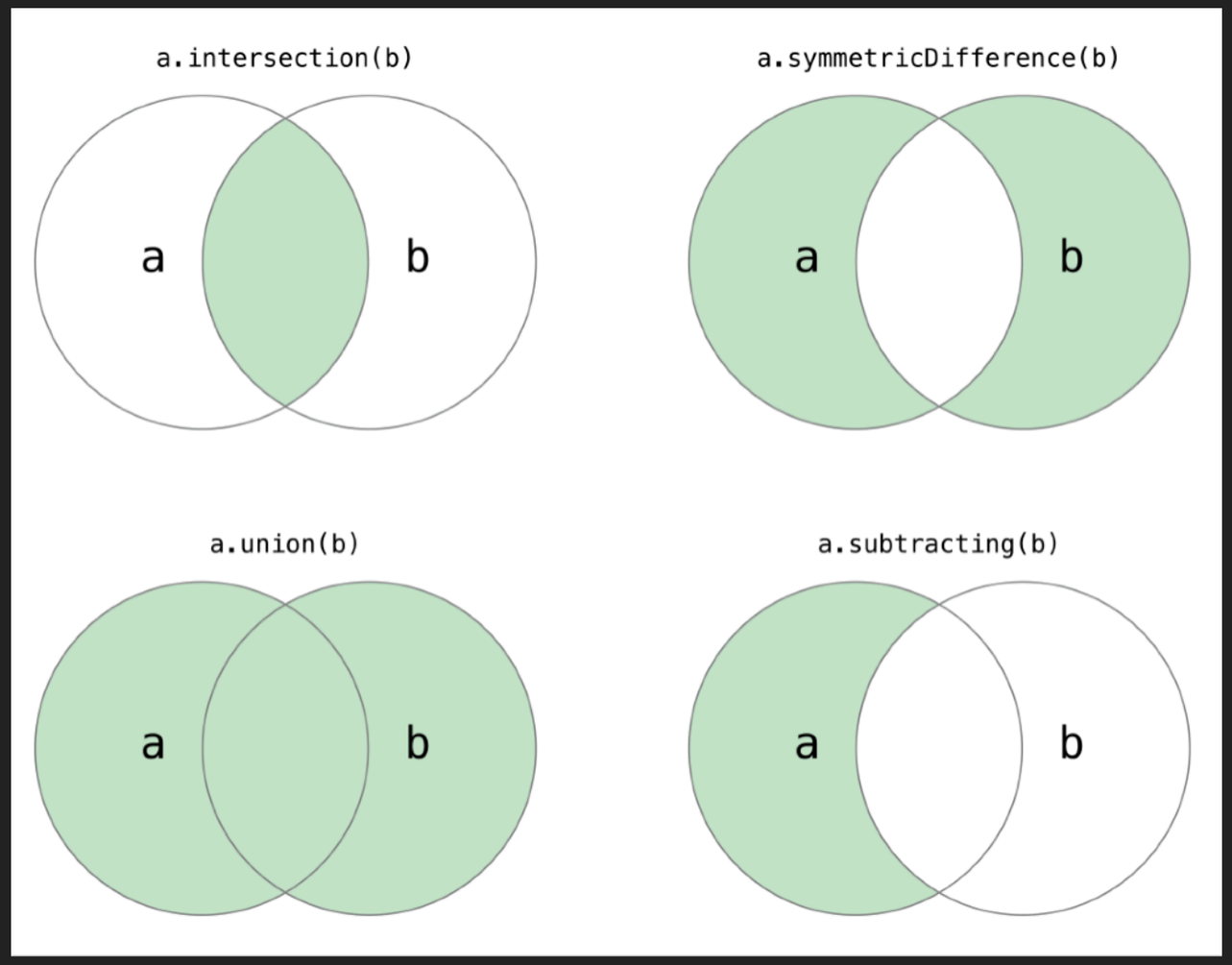
- Unordered collection of unique values
- Stores only hashable value

```swift
var genres: Set<String> = ["Rock", "Rap", "Funk"]
var marks = Set<Int>()
var favorites: Set = ["Apple", "Pear"]
```

```swift
genres.insert("Jazz")
genres.contains("Rock")
genres.remove("Rap") // if finds
```

Used pretty rarely

SETS




```swift
let houseAnimals: Set = ["🐶", "🐱"]
let farmAnimals: Set = ["🐷", "🐔", "🐑", "🐶", "🐱"]
let cityAnimals: Set = ["🐳", "🐭"]

houseAnimals.isSubset(of: farmAnimals)
// true
farmAnimals.isSuperset(of: houseAnimals)
// true
farmAnimals.isDisjoint(with: cityAnimals)
// true
```

DICTIONARIES

- Associations between keys and values

```swift
var namesOfIntegers: [Int: String] = [:]
var namesOfIntegers2 = [Int: String]()
var namesOfIntegers3: Dictionary<Int, String> = [16: "Sixteen", 5: "five"]
```

```swift
let dictValue = namesOfIntegers[10]
namesOfIntegers[20] = "twenty"
namesOfIntegers.removeValue(forKey: 22)

for (keys, values) in namesOfIntegers {
    // do sth with that
}
```

CONTROL FLOW

CONTROL FLOW

- Loops - for, while
- if, guard
- switch
- break, continue

FOR-IN LOOP

- Iterates over sequence of values

```
let names = ["John", "Martha", "Dylan", "Derek"]

for name in names {
    print("Hello \(name)")
}
```

```
for rating in 1...10 {
    // do sth
}
```

```
for _ in 20...100 {
    // do iterations
}
```

```
for hour in stride(from: 10, to: 22, by: 2) {
    print(hour)
}
```

```
10
12
14
16
18
20
```

WHILE AND REPEAT-WHILE

- Only difference is that repeat-while is executed one time for sure

```
while condition {
    statements
}
```

```
repeat {
    statements
} while condition
```

CONDITIONAL STATEMENTS

- if
- switch
- guard - for early exit

```
if someCondition {

} else if anotherCondition {

} else {

}
```

```
guard someCondition else {
    return
}

guard anotherCondition else {
    throw MyError.overflow
}
```

Takes only bool

SWITCH IS A REAL BEAST!

```
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
}
```

Doesn't need break for each case

If you need such behaviour use fallthrough

- Interval matching

```
switch currentCountValue {
case 1...12:
    return .small
case 13...22:
    return .medium
case 23...:
    return .big
default:
    return .medium
}
```

SWITCH IS A REAL BEAST!

- Can be used with tuples

```
let somePoint = (1, 1)
switch somePoint {
case (0, 0):
    print("\(somePoint) is at the origin")
case (_, 0):
    print("\(somePoint) is on the x-axis")
case (0, _):
    print("\(somePoint) is on the y-axis")
case (-2...2, -2...2):
    print("\(somePoint) is inside the box")
default:
    print("\(somePoint) is outside of the box")
}
```

- Value binding

```
let anotherPoint = (2, 0)
switch anotherPoint {
case (let x, 0):
    print("on the x-axis with an x value of \(x)")
case (0, let y):
    print("on the y-axis with a y value of \(y)")
case let (x, y):
    print("somewhere else at (\(x), \(y))")
}
```

- + where

- Very good match with enums

CONTROL TRANSFER STATEMENTS

- continue - skips one iteration of the loop
- break - stops iteration at all
- return - exits current scope of code (func)
- throw
- fallthrough

```
for i in 1...100 {
    if i % 5 != 3 {
        continue
    }

    print(i)
}
```

```
for i in 1...100 {
    if i % 5 != 3 {
        break
    }

    print(i)
}
```

You can mark loop with labels and exit specified loop

LABELED STATEMENT

- You can mark loop with labels and exit specified loop

```
myLoop: for i in 1...1000 {
    for j in 100...200 {
        if i == 300 && j == 150 {
            break myLoop
        }
    }
}
```

I’ve even used it once!

FUNCTIONS

FUNCTIONS

- Self-contained piece of code that perform specific task

```
func makeBurger(with ingradients: [Ingradient], of size: Size) -> Burger {
    // Function body
}
```

- func nameOfFunction(parameters list) -> Return type {}

```
func makeBurger() -> Burger {
    // Function body
}
```

```
func makeBurger() {
    // Function body
}
```

```
func makeBurger(name: String) {}
```

FUNCTIONS

- To call function simply pass params and go

```
makeBurger()
makeBurger(name: "Trump")
```

- You can return multiple values from function with tuples

```
func greet(name: String) -> (male: String, female: String) {
    let maleGreetings = "Greetings, Mr \(name)!"
    let femaleGreetings = "What a nice day, Mrs \(name) :)"
    return (maleGreetings, femaleGreetings)
}
```

- Each function has its own type which is constructed from params and return type

FUNCTIONS RETURN

```
func anotherGreeting(for person: String) -> String {
    return "Hello, " + person + "!"
}

func greeting(for person: String) -> String {
    "Hello, " + person + "!"
}
```

```
func thirdGreeting(for person: String) -> String? {
    if person.isEmpty {
        return nil
    }

    return "Hello, " + person + "!"
}
```

FUNCTIONS PARAMETERS NAMING

- Each parameter can have additional label for readability

```
func function(first: Int, second: Int) {}
function(first: 24, second: 36)
```

```
func function(chairs count: Int, tables number: Int) {
    print("chairs - \(count)")
    print("tables - \(number)")
}
function(chairs: 11, tables: 12)
```

```
func function(_ omitted1: Int, _ ommited2: Int, third param: Int) {}
function(12, 34, third: 55)
```

_ to omit name when calling

FUNCTIONS

- Default parameters

```
func function(first: Int = 10, second: Int = 20) {}
```

- Variadic parameters

```
func arithmeticMean(_ numbers: Double...) -> Double {
    var total: Double = 0
    for number in numbers {
        total += number
    }
    return total / Double(numbers.count)
}
arithmeticMean(1, 2, 3, 4, 5)
```

Same as Array param

IN-OUT PARAMS

- All parameters are constants by default
- To persist changes inside function you should mark such param inout

```
func swapTwoInts(_ a: inout Int, _ b: inout Int) {
    let temporaryA = a
    a = b
    b = temporaryA
}
```

```
var someInt = 3
var anotherInt = 107
swapTwoInts(&someInt, &anotherInt)
```

Never used it :)

FUNCTIONS AS PARAMS

- Back to functions types - you can pass them as params or return types

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {
    return a + b
}
```

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

```
print("Result: \(mathFunction(2, 3))")
// Prints "Result: 5"
```

```
func stepForward(_ input: Int) -> Int {
    return input + 1
}
func stepBackward(_ input: Int) -> Int {
    return input - 1
}
```

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {
    return backward ? stepBackward : stepForward
}
```

- You can nest functions inside of each other

SUMMARY

- There are **arithmetic, comparison** and **logical operators**
- There is **Array**, **Set** and **Dictionary collection** types
- There is **for-each** and **while loops**
- **If, guard, switch** are used for conditions
- **Functions** are independent pieces of code
- Functions can have **parameters**, **return value**, **name for parameters** and be parameters themselves