Swift 4.2

iOS Programming - Spring 2019
Nathan Hull

Apple's New Programming Language for iOS and Mac OS
Swift 4.2
Genesis of Swift

Chris Lattner (Apple’s Director of the Developer Tools department) started work on Swift in 2010 as a side project.
In July 2013, it became a major focus of the Apple Developer Tools group.
Swift is based upon...

Objective-C
Rust
Haskell
Ruby
Python
C#
CLU

and “far too many others to list”
Similarities to Objective-C

- Basic numeric types (Int, UInt, Float, Double)
- Most C operators are carried over to Swift, but there are some new operators
- Curly braces are used to group statements.
Similarities to Objective-C

• Variables are assigned using an equals sign, but compared using two consecutive equals signs. A new identity operator, ===, is provided to check if two data elements refer to the same object.
Similarities to Objective-C

- Square brackets are used with arrays, both to declare them and to get a value at a given index in one of them.

- Control statements, for, while, if, switch are similar, but have extended functionality, e.g. a “for in” that iterates over any collection type, a “switch” that takes non-integer cases, etc.
Similarities to Objective-C

- Class methods are inherited, just like instance methods; `self` in class methods is the class the method was called on.
Differences from Objective-C

• Statements do not need to end with a semicolon (;), though they must be used to allow more than one statement on a line
• Header files are not required
• Strong typing
• Type inference
• ++ and - - operators have been eliminated
Differences from Objective-C

• Functions are first-class objects.

• Enumeration cases can have associated data (algebraic data types).

• Operators can be redefined for classes (operator overloading), and new operators can be created.

• Strings fully support Unicode. Most Unicode characters can be used in either identifiers or operators.
Objective-C Errors Eliminated

• Pointers are not exposed by default. There is no need for the programmer to keep track of and mark names for referencing or de-referencing.
Objective-C Errors Eliminated

• Assignments do not return a value. Thus the common mistake:

```c
if (i=0) when if (i==0) is meant will cause a compile-time error.
```
Objective-C Errors Eliminated

- No need to use `break` statements in `switch` blocks. Individual cases do not fall through to the next case unless the `fallthrough` statement is used.
Objective-C Errors Eliminated

- Variables and constants are always initialized and array bounds are always checked.
Quick Examples in Swift

// this is a single line comment using two slashes.

/* this is also a comment,
but written over multiple lines */

/* multiline comments
/* can be nested! */
so that you can block out code containing multiline comments
*/
Quick Examples in Swift

// Swift variables are declared with "var"
// followed by a name, a type, and a value

var iAm_Double: Double = 70
Quick Examples in Swift

// if the type is omitted, Swift will
// infer it from the variable's initial value

var implicitInteger = 70
var implicitDouble = 70.0
var 你好 = "你好世界"
var beast = "🐶🐷"
Quick Examples in Swift

// Swift constants are declared with "let"
// followed by a name, a type, and a value

let numberOfBananas: Int = 10

let π = 3.1415926535
Quick Examples in Swift

// if the type is omitted,
// Swift will infer it from the constant's value

let numberOfApples = 3
let numberOfOranges = 5
let msg = "Hello, Nurse!"
Quick Examples in Swift

// values of variables and constants can be
// interpolated in strings:

let applePrint = "I have \(numberOfApples) apples."
let fruitPrint = "I have \(numberOfApples + numberOfOranges) pieces of fruit."
Quick Examples in Swift

Inferring a Swift String from two Characters

```swift
let dog: Character = "🐶"
let cow: Character = "牪"
let dogCow = dog + cow
// dogCow is "🐶牪"
```
Quick Examples in Swift

// define an array

var operas = ["Pinafore", "Bohème", "Don Giovanni"]
Quick Examples in Swift

// example of if statement; .isEmpty, .count

if operas.isEmpty {
    print("No operas to be heard.")
} else {
    print("There are \
      \(operas.count) operas in my array")
}
// define a dictionary with four items, each with a person's name and age

let people = 
    [
        "Barack": 57,
        "Miley": 25,
        "Santa": 435,
        "Francis": 81
    ]
Quick Examples in Swift

// With that dictionary you can enumerate and
// extract both values in a single loop

for (name, age) in people {
    print("\(name) is \(age) years old.")
}
Quick Examples in Swift

// functions and methods are declared with
// the "func" syntax
// the return type is specified with ->

func sayHello(personName: String) -> String {
    let greeting = "Hello, " + personName + "!"
    return greeting
}
More Details on Swift
Collections

var names = ["Wacko", "Yakko", "Dot"]

var languages = ["Fortran": 1957,
                 "COBOL": 1959,
                 "C": 1972,
                 "Objective-C": 1983,
                 "Swift": 2014]
collections

names = ["Wacko", "Yakko", "Dot"]

Should these always be strings??
You probably don’t want:

names = ["Wacko", "Yakko", “Dot”, true, 14] as [any]
Typed Collections

var names: [String] = ["Wacko", "Yakko", "Dot"]

or simply inferred from:

names = ["Wacko", "Yakko", "Dot"]
Loops

for character in "煊煊煊煊煊".characters {
    print(character)
}

for character in "煊煊煊煊煊".characters {
    print(character)
}
for number in 1...5 {
    print ("\(number) times 4 is \(number * 4)"")
}
Half Closed Range Loop

for number in 0..<5 {
    print ("\(number) times 4 is \(number * 4)")
}

0 times 4 is 0
1 times 4 is 4
2 times 4 is 8
3 times 4 is 12
4 times 4 is 16
Modifying an Array

```javascript
var colors = ["red", "blue"]
```

["red", "blue"]
Modifying an Array

```javascript
var colors = ["red", "blue"]

print(colors[1])

"blue"
```
Modifying an Array

```javascript
var colors = ["red", "blue"]
colors += ["green", "pink"]

["red", "blue", "green", "pink"]
```
Modifying an Array

```javascript
var colors = ["red", "blue"]
colors += ["green", "pink"]
colors[0] = "purple"
print(colors)
```

"[purple, blue, green, pink]"
Modifying an Array

```javascript
var colors = ["red", "blue"]
colors += ["green", "pink"]
colors[0] = "purple"
colors[1...2] = ["yellow", "white"]
print(colors)

"[purple, yellow, white, pink]"
```
Modifying a Dictionary

```javascript
var languages = [
    "Fortran": 1957,
    "COBOL": 1959,
    "C": 1972
]
```

"Fortran": 1957, "COBOL": 1959, "C": 1972]
Modifying a Dictionary

```java
var languages = [
    "Fortran": 1957,
    "COBOL": 1959,
    "C": 1972
];

languages["Objective-C"] = 2555,
print(languages)
```

"[Fortran: 1957, C: 1972, COBOL: 1959, Objective-C: 2555]"
Modifying a Dictionary

```javascript
var languages = [
  "Fortran": 1957,
  "COBOL": 1959,
  "C": 1972
];
languages["Objective-C"] = 2555;
languages["Objective-C"] = 1983;
print(languages);

"[Fortran: 1957, C: 1972,
  COBOL: 1959, Objective-C: 1983]
"
Retrieving from a Dictionary

```javascript
var languages = [
  "Fortran": 1957,
  "COBOL": 1959,
  "C": 1972
]
```

Question: What happens if we ask for:

```javascript
languages = ["Ruby"]
```
Retrieving from a Dictionary

```swift
var languages = [
"Fortran": 1957,
"COBOL": 1959, 
"C": 1972
]

let yearMade = languages["Ruby"]
print(yearMade)

"nil"
```
```swift
var languages = [
    "Fortran": 1957,
    "COBOL": 1959,
    "C": 1972
]

let yearMade: Int? = languages["COBOL"]
if yearMade == nil {
    print("Language not found")
} else {
    let theYear = yearMade!
    print(theYear)
}

"1959"
```
Retrieving from a Dictionary

Querying an Optional

```
var languages = [
    "Fortran": 1957,
    "COBOL": 1959,
    "C": 1972
]

let yearMade: Int? = languages["Ruby"]
if yearMade == nil{
    print("Language not found")
} else {
    let theYear = yearMade!
    print(theYear)
}

"Language not found"
```
Retrieving from a Dictionary

Unwrapping an Optional

```swift
var languages = [
    "Fortran": 1957,
    "COBOL": 1959,
    "C": 1972
]

let yearMade: Int? = languages["COBOL"]
if yearMade == nil{
    print("Language not found")
} else {
    if let theYear = yearMade {
        print(theYear) } } }

"1959"
```
Switch Statements

```javascript
var numbah = 2
switch numbah {
    case 0: print("neither prime nor composite");
    case 1: print("neither prime nor composite");
    case 2: print("prime number");
}
```

Error: Switch must be exhaustive, consider adding default
Switch Statements

var numbah = 2

switch numbah {
    case 0: print("neither prime nor composite");
    case 1: print("neither prime nor composite");
    case 2: print("prime number");
    default: print("an untested number");
}
var numbah = 10
switch numbah {
  case 0: print("neither prime nor composite")
  case 1: print("neither prime nor composite")
  case 2: print("prime number")
  default: print("an untested number")
}
Switch Statements

var numbah = 10
switch numbah {
    case 0: print("neither prime nor composite")
    case 1: print("neither prime nor composite")
    case 2: print("prime number")
    case 3...10: print("num 10 or less")
    default: print("an untested number")
}

"num 10 or less"
Functions
Simple Function

```swift
func howdy()
{
    print("Howdy!")
}

howdy()
```

“Howdy!”
Function with Parameter

```swift
func howdy_name (person: String){
    print("Howdy, \(person)!")
}

howdy_name("partner")
```

“Howdy, partner!”
Function with Default Parameter

```swift
func howdy_name2 (person: String = "partner") {
    print("Howdy, \(person)!")
}
howdy_name2()
```

“Howdy, partner!”
Function with Default Parameter

```swift
func howdy_name2 (person: String = "partner") {
    print("Howdy, \(person)!")
}
howdy_name2()
howdy_name2(person: "Miss Kitty")

"Howdy, partner!"
"Howdy, Miss Kitty!"
```
And now, some background on...
Some background on…

Tuples
A tuple type is a comma-separated list of zero or more types, enclosed in parentheses.
Tuples

A tuple type is a comma-separated list of zero or more types, enclosed in parentheses.

(Int, String)
Tuples

You can use a tuple type as the return type of a function to enable the function to return a single tuple containing multiple values.
Tuples

You can use a tuple type as the return type of a function to enable the function to return a single tuple containing multiple values.

func howdy1() -> (Int, String)
Function returning a Tuple

```swift
func howdy1() -> (Int, String) {
    let times = 3
    var greeting = "Hello!"
    return (times, greeting)
}

print(howdy1().0)
print(howdy1().1)
```

3
Hello!
Tuples

You can also name the elements of a tuple type and use those names to refer to the values of the individual elements. An element name consists of an identifier followed immediately by a colon (:).
You can also name the elements of a tuple type and use those names to refer to the values of the individual elements. An element name consists of an identifier followed immediately by a colon (:

```swift
func howdy() -> (countOfPeople: Int, completeGreeting: String)
```
func howdy() -> 
    (countOfPeople: Int, completeGreeting: String)
{
    let times=3
    var greeting="Hello!"
    return (times, greeting)  
}

var instructions = howdy()
print(instructions.countOfPeople)
print(instructions.completeGreeting)
Function with Array Input

```swift
func Addem(numbers: Int...) -> Int {
    var tot = 0
    for num in numbers {
        tot += num
    }
    return tot
}
Addem() 0
```
func Addem(numbers: Int...) -> Int {
    var tot = 0
    for num in numbers {
        tot += num
    }
    return tot
}

Addem()  // 0
Addem(64)  // 64
func Addem(numbers: Int...) -> Int {
    var tot = 0
    for num in numbers {
        tot += num
    }
    return tot
}

Addem()
Addem(64)
Addem(1,3,7,15,31)
Functions are a first-class type. This means that a function can return another function as its value.
func anyNumFound(list: [Int], condition: Int -> Bool) -> Bool
{
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}

func divBy2(number: Int) -> Bool {
    return number % 2 == 0
}

var numbers = [17, 19, 7, 1]

anyNumFound(numbers, divBy2)
func anyNumFound(list: [Int], condition: Int -> Bool) -> Bool
{
    for item in list {
        if condition(item) {
            return true
        }
    }
    return false
}

func divBy2(number: Int) -> Bool {
    return number % 2 == 0
}

var numbers = [17, 19, 4, 1]

anyNumFound(numbers, divBy2)
true
Closures

A closure is a block of code that can be called later. Functions are actually just a special case of closures which are named.
Closures

The code in a closure has access to things like variables and functions that were available in the scope where the closure was created, even if the closure is in a different scope when it is executed.
You can write a closure without a name by surrounding code with braces ({}). Use `in` to separate the arguments and return type from the body.
var num2 = [16, 5, 7, 1]
var x = num2.map({
    (number: Int) -> Int in
    let result = number * number
    return result
})
print(x)
Closures can be written in much simplified, reduced form. If a Closure’s type is already known, you can omit the type of the parameters and/or return type.
Single statement closures implicitly return the value of their only statement.
var num2 = [16, 5, 7, 1]
let mappedNums = num2.map({ num in num * num * num })
print(mappedNums)  // Output: [4096, 125, 343, 1]
Closures

You can refer to parameters by number instead of by name—often used in very short closures.

A closure passed as the last argument to a function can appear immediately after the parentheses.
var num3 = [1, 5, 16, 7]
let descendNumbers = sorted(num3) { $0 > $1 }

print(descendNumbers)  // [16, 7, 5, 1]
Use the keyword “class” followed by the class’s name to create a class.
A property declaration in a class is written the same way as a constant or variable declaration.
Classes and Objects

Likewise, method and function declarations are written in the same way.
Classes and Objects

Note that we do NOT have to import an .h (header) file because the implementation IS the public interface as well!
Classes and Objects

And, unlike Objective-C, you don’t necessarily have to have a base class - Although you may use one (like NSObject) if you need to.
class monster {
    // Properties
    // Methods
    // Initializers
}

class monster{
    var num_heads = 1
}

Classes - Stored Property
class monster{
    var num_heads = 1
    var description: String{
        get { return "\(num_heads) Num of heads "}
    }
}
class monster{
    var num_heads = 1
    var description: String{
        return "\(num_heads) Num of heads"
    }
}
class monster{
    var num_heads = 1
    var description: String{
        return "\(num_heads) Num of heads "
    }
}
let grendel = monster()
class monster{
    var num_heads = 1
    var description: String{
        return "\(num_heads) Num of heads "}
}

let grendel = monster()
grendel.num_heads

grendel.description
class monster {
    var num_heads = 1
    var description: String {
        return "\(num_heads) Num of heads "
    }
}

let grendel = monster()
grendel.num_heads = 2
grendel.description

2
2 Num of heads
class two_headed: monster {
    override init() {
        super.init()
        num_heads = 2
    }
}

Classes - Sub Class
class two_headed: monster {
    override init() {
        super.init()
        num_heads = 2
    }
}

let orthros = two_headed()
orthros.description

2
2 Num of heads
class dragon: monster{
    var num_wings = 2
    override init() {
        super.init()
        num_heads = 2
    }
    override var description: String{
        return super.description + "& \(num_wings) Num of wings "}
}
class dragon: monster{
    var num_wings = 2
    override init() {
        super.init()
        num_heads = 2
    }
    override var description: String{
        return super.description + "& \(num_wings) Num of wings "}
}

let ladon = dragon()
ladon.description

2
2 Num of heads and 2 Num of wings
class capturedMonster: monster{
    override var num_heads: Int {
        willSet{
            didSet{
            }}}}
class capturedMonster: monster{
    override var num_heads: Int {
        didSet{
            if newValue > 4 {
                print("Run!")
            }
        }
    }
}
class capturedMonster: monster{
    override var num_heads: Int {
        willSet{
            if newValue > 4 {
                print("Run!")
            }
        }
    }
}

let hydra = capturedMonster()
hydra.description
hydra.num_heads = 99

1 Num of heads
99 Run!
class superMonster: monster {
    var victims = 0
    func devour() {
        victims += 1
    }
}

Classes - Methods
class superMonster: monster {
    var victims = 0
    func devour() {
        victims+=1
    }
}

let godzilla = superMonster()
godzilla.devour()
godzilla.victims
class superMonster: monster {
    var victims = 0
    func resetVictims(victims: Int) {
        self.victims = victims
    }
}

let godzilla = superMonster()

godzilla.resetVictims(8)
godzilla.victims
 Enums

Support all the functionality of C enumerations, but add:
  • Dot Notation
  • Enumerated Member Values
  • Associated Values
enum Rank: Int {
    case Ace = 1
    case Two, Three, Four, Five, Six, Seven, Eight, Nine, Ten
    case Jack, Queen, King
}

(cont)
func simpleDescription() -> String {
    switch self {
    case .Ace: return "ace"
    case .Jack: return "jack"
    case .Queen: return "queen"
    case .King: return "king"
    default: return String(self.rawValue)
    }
}
let card = Rank.Ace
let aceRawValue = card.rawValue
card.simpleDescription()