

Introduction to iOS development with Swift

Lesson 2



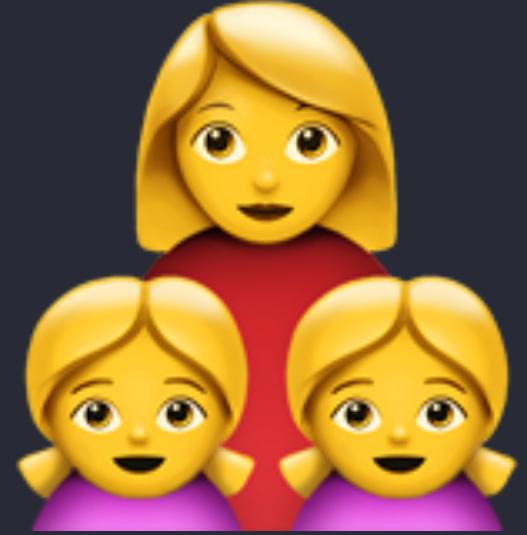
Adrien Humilière
Dashlane

adhumidant@gmail.com



- Classes and inheritance
- Collections
- Loops
- Optionals
- Type Casting and Inspection

Classes and inheritance



```
class Person {  
    let name: String  
  
    init(name: String) {  
        self.name = name  
    }  
  
    func sayHello() {  
        print("Hello there!")  
    }  
}  
  
let person = Person(name: "Jasmine")  
print(person.name)  
person.sayHello()
```

Inheritance

- Base class: Vehicle
- Subclass: Tandem
- Superclass: Bicycle

Inheritance

```
class Vehicle {  
    var currentSpeed = 0.0  
  
    var description: String {  
        return "traveling at \(currentSpeed) km per hour"  
    }  
  
    func makeNoise() {  
        // do nothing - a vehicle doesn't necessarily make noise  
    }  
}
```

Subclass

```
class SomeSubclass: SomeSuperclass {  
    // subclass definition goes here  
}
```

```
class Bicycle: Vehicle {  
    var hasBasket = false  
}
```

Subclass

```
class Tandem: Bicycle {  
    var currentNumberOfPassengers = 0  
}
```

Override methods

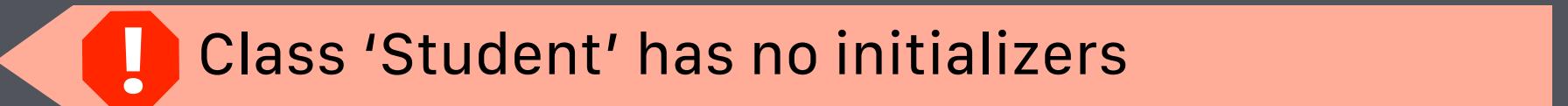
```
class Train: Vehicle {  
    override func makeNoise() {  
        print("Choo Choo!")  
    }  
}
```

Override computed properties

```
class Car: Vehicle {  
    var gear = 1  
    override var description: String {  
        return super.description + " in gear \\" + gear + ")"  
    }  
}
```

Override init

```
class Person {  
    let name: String  
  
    init(name: String) {  
        self.name = name  
    }  
}  
  
class Student: Person {  
    var favoriteSubject: String  
}
```



```
class Person {  
    let name: String  
  
    init(name: String) {  
        self.name = name  
    }  
}  
  
class Student: Person {  
    var favoriteSubject: String  
  
    init(name: String, favoriteSubject: String) {  
        self.favoriteSubject = favoriteSubject  
        super.init(name: name)  
    }  
}
```

References

- When you create an instance of a class:
 - Swift returns the address of that instance
 - The returned address is assigned to the variable
- When you assign the address of an instance to multiple variables:
 - Each variable contains the same address
 - Update one instance, and all variables refer to the updated instance

```
class Person {  
    let name: String  
    var age: Int  
  
    init(name: String, age: Int) {  
        self.name = name  
        self.age = age  
    }  
}  
  
var jack = Person(name: "Jack", age: 24)  
var myFriend = jack  
  
jack.age += 1  
  
print(jack.age) // 25  
print(myFriend.age) // 25
```

```
struct Person {  
    let name: String  
    var age: Int  
}  
  
var jack = Person(name: "Jack", age: 24)  
var myFriend = jack  
  
jack.age += 1  
  
print(jack.age) // 25  
print(myFriend.age) // 24
```

Memberwise initializers

- Swift does not create memberwise initializers for classes
- Common practice is for developers to create their own for their defined classes

Class or structure?

- Start new types as structures
- Use a class:
 - When you're working with a framework that uses classes
 - When you want to refer to the same instance of a type in multiple places
 - When you want to model inheritance

Collections



Collection types



Array

Dictionary

Arrays

```
[value1, value2, value3]
```

```
var names: [String] = ["Anne", "Gary", "Keith"]
```

Arrays

```
[value1, value2, value3]
```

```
var names = ["Anne", "Gary", "Keith"]
```

```
var numbers = [1, -3, 50, 72, -95, 115]
```

Arrays

```
[value1, value2, value3]
```

```
var names = ["Anne", "Gary", "Keith"]
```

```
var numbers: [Double] = [1, -3, 50, 72, -95, 115]
```

Arrays – contains

```
let numbers = [4, 5, 6]
if numbers.contains(5) {
    print("There is a 5")
}
```

Arrays types

```
var myArray: [Int] = []
var myArray: Array<Int> = []
var myArray = [Int]()
```

Working with arrays

```
var myArray = [Int](repeating: 0, count: 100)
let count = myArray.count
if myArray.isEmpty { }
```

Working with arrays

```
var names = ["Anne", "Gary", "Keith"]
let firstName = names[0]
print(firstName) // Anne
```

```
names[1] = "Paul"
print(names) // ["Anne", "Paul", "Keith"]
```

Working with arrays

```
var names = ["Amy"]
names.append("Joe")
names += ["Keith", "Jane"]
print(names) // ["Amy", "Joe", "Keith", "Jane"]
```

Working with arrays

```
var names = ["Amy", "Brad", "Chelsea", "Dan"]
names.insert("Bob", at: 0)
print(names) // ["Bob", "Amy", "Brad", "Chelsea", "Dan"]
```

Working with arrays

```
var names = ["Amy", "Brad", "Chelsea", "Dan"]
let chelsea = names.remove(at:2)
let dan = names.removeLast()
print(names) // ["Amy", "Brad"]
```

```
names.removeAll()
print(names) // []
```

Working with arrays

```
var myNewArray = firstArray + secondArray
```

Dictionaries

```
[key1: value1, key2: value2, key3: value3]
```

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]
```

Dictionaries

```
var myDictionary = [String: Int]()
var myDictionary = Dictionary<String, Int>()
var myDictionary: [String: Int] = [:]
```

Add/remove/modify a dictionary

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]  
  
scores["Oli"] = 399  
  
let oldValue = scores.updateValue(100, forKey: "Richard")
```

Add/remove/modify a dictionary

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

scores["Oli"] = 399

if let oldValue = scores.updateValue(100, forKey: "Richard") {
    print("Richard's old value was \(oldValue)")
}
```

Add/remove/modify a dictionary

```
var scores = ["Richard": 100, "Luke": 400, "Cheryl": 800]
scores["Richard"] = nil
print(scores) // ["Cheryl": 800, "Luke": 400]

if let oldValue = scores.removeValue(forKey: "Luke") {
    print("Luke's score was \(oldValue) before he stopped playing")
}
print(scores) // ["Cheryl": 800]
```

Accessing a dictionary

```
var scores = {"Richard": 500, "Luke": 400, "Cheryl": 800}

let players = Array(scores.keys) // ["Richard", "Luke", "Cheryl"]
let points = Array(scores.values) // [500, 400, 800]

print(myScore)
if let myScore = scores["Luke"] {
  print(myScore)
}
```

Accessing a dictionary

```
var scores = ["Richard": 500, "Luke": 400, "Cheryl": 800]

let players = Array(scores.keys) // ["Richard", "Luke", "Cheryl"]
let points = Array(scores.values) // [500, 400, 800]

print(scores["Luke"]) // Optional(400)
if let myScore = scores["Luke"] {
  print(myScore) // 400
}
```

Loops



Loops

for

while

for loops

```
for index in 1...5 {  
    print("This is number \$(index)")  
}
```

```
for _ in 1...5 {  
    print("Hello!")  
}
```

for loops

```
let names = ["Joseph", "Cathy", "Winston"]
for name in names {
    print("Hello \(name)")
}
```

```
for letter in "ABCDEFG".characters {
    print("The letter is \(letter)")
}
```

for loops

```
for (index, letter) in "ABCDEFG".characters.enumerated() {  
    print("\(index): \(letter)")  
}
```

for loops

```
let vehicles = ["unicycle" : 1, "bicycle" : 2, "tricycle" : 3]
for (vehicleName, wheelCount) in vehicles {
    print("A \vehicleName has \wheelCount wheels")
}
```

while loops

```
var numberOfLives = 3

while numberOfLives > 0 {
    playMove()
    updateLivesCount()
}
```

while loops

```
var numberOfLives = 3
var stillAlive = true

while stillAlive {
    print("I still have \$(numberOfLives) lives.")
    numberOfLives -= 1
    if numberOfLives == 0 {
        stillAlive = false
    }
}
```


Optionals

!?

nil

```
struct Book {  
    let name: String  
    let publicationYear: Int  
}  
  
let firstHarryPotter = Book(name: "Harry Potter and the  
Sorcerer's Stone", publicationYear: 1997)  
let secondHarryPotter = Book(name: "Harry Potter and the  
Chamber of Secrets", publicationYear: 1998)  
  
let books = [firstHarryPotter, secondHarryPotter]
```

nil

```
let unannouncedBook = Book(name: "Harry Potter 8",  
                           publicationYear: ???)
```

nil

```
let unannouncedBook = Book(name: "Harry Potter 8",  
                           publicationYear: 0)
```

nil

```
let unannouncedBook = Book(name: "Harry Potter 8",  
                           publicationYear: 2019)
```

nil

```
let unannouncedBook = Book(name: "Harry Potter 8",  
                           publicationYear: nil)
```



Nil is not compatible with expected argument type

```
struct Book {  
    let name: String  
    let publicationYear: Int?  
}  
  
let firstHarryPotter = Book(name: "Harry Potter and the  
Sorcerer's Stone", publicationYear: 1997)  
let secondHarryPotter = Book(name: "Harry Potter and the  
Chamber of Secrets", publicationYear: 1998)  
  
let books = [firstHarryPotter, secondHarryPotter]  
  
let unannouncedBook = Book(name: "Rebels and Lions",  
publicationYear: nil)
```

Specifying the type of an optional

```
var serverResponseCode: Int = 404
```

```
var serverResponseCode: Int = nil
```



'nil' requires a contextual type

```
var serverResponseCode: Int? = 404
```

```
var serverResponseCode: Int? = nil
```

Working with optional values

```
if publicationYear != nil {  
    let actualYear = publicationYear!  
    print(actualYear)  
}
```

```
let unwrappedYear = publicationYear!
```



error: Execution was interrupted

Working with optional values

```
if let constantName = someOptional {  
    //constantName has been safely unwrapped for use within {}  
}
```

```
if let unwrappedPublicationYear = book.publicationYear {  
    print("The book was published in \(unwrappedPublicationYear)")  
} else {  
    print("The book does not have an official publication date.")  
}
```

Functions and optionals

```
let string = "123"  
let possibleNumber = Int(string)
```

```
let string = "Cynthia"  
let possibleNumber = Int(string)
```

Functions and optionals

```
func printFullName(firstName: String, middleName: String?,  
lastName: String)
```

```
func textFromURL(url: URL) -> String?
```

Failable initializers

```
struct Toddler {  
    var birthName: String  
    var monthsOld: Int  
}
```

Failable initializers

```
init?(birthName: String, monthsOld: Int) {  
    if monthsOld < 12 || monthsOld > 36 {  
        return nil  
    } else {  
        self.birthName = birthName  
        self.monthsOld = monthsOld  
    }  
}
```

Failable initializers

```
let possibleToddler = Toddler(birthName: "Joanna", monthsOld: 14)
if let toddler = possibleToddler {
    print("\(toddler.birthName) is \(toddler.monthsOld) months old")
} else {
    print("The age you specified for the toddler is not between 1
and 3 yrs of age")
}
```

Optional chaining

```
class Person {  
    var age: Int  
    var residence: Residence?  
}
```

```
class Residence {  
    var address: Address?  
}
```

```
class Address {  
    var buildingNumber: String?  
    var streetName: String?  
    var apartmentNumber: String?  
}
```

Optional chaining

```
if let theResidence = person.residence {  
    if let theAddress = theResidence.address {  
        if let theApartmentNumber = theAddress.apartmentNumber {  
            print("He/she lives in apartment number  
                  \\\(theApartmentNumber). »)  
        }  
    }  
}  
  
if let apartmentNumber =  
    person.residence?.address?.apartmentNumber
```

Implicitly Unwrapped Optionals

```
class ViewController: UIViewController {  
    @IBOutlet weak var label: UILabel!  
}
```

Unwraps automatically

Should only be used when need to initialize an object without supplying the value and you'll be giving the object a value soon afterwards

Type Casting and Inspection



```
func getClientPet() -> Animal {  
    //returns the pet  
}  
  
let pet = getClientPet() //`pet` is of type `Animal`
```

```
if pet is Dog {  
    print("The client's pet is a dog")  
} else if pet is Cat {  
    print("The client's pet is a cat")  
} else if pet is Bird {  
    print("The client's pet is a bird")  
} else {  
    print("The client has a very exotic pet")  
}
```

```
let pets = allPets() //`pets` is of type `[Animal]`  
var dogCount = 0, catCount = 0, birdCount = 0  
for pet in pets {  
    if pet is Dog {  
        dogCount += 1  
    } else if pet is Cat {  
        catCount += 1  
    } else if pet is Bird {  
        birdCount += 1  
    }  
}  
print("Brad looks after \$(dogCount) dogs, \$(catCount) cats,  
and \$(birdCount) birds.")
```

Type casting

```
func walk(dog: Dog) {  
    print("Walking \(dog.name)")  
}  
  
func cleanLitterBox(cat: Cat) {...}  
  
func cleanCage(bird: Bird) {...}  
  
for pet in pets {  
    if pet is Dog {  
        walk(dog: pet) // Compiler error  
    }  
    ...  
}
```

Type casting

```
for pet in pets {  
    if let dog = pet as? Dog {  
        walk(dog: dog)  
    } else if let cat = pet as? Cat {  
        cleanLitterBox(cat: cat)  
    } else if let bird = pet as? Bird {  
        cleanCage(bird: bird)  
    }  
}
```

Any

```
var items: [Any] = [5, "Bill", 6.7, Dog()]
```

Any

```
var items: [Any] = [5, "Bill", 6.7, Dog()]
let firstItem = items[0]

if firstItem is Int {
    print("The first element is an integer")
} else if firstItem is String {
    print("The first element is a string")
} else {
    print("The first element is neither an integer nor a string")
}
```

Any

```
var items: [Any] = [5, "Bill", 6.7, Dog() ]  
  
if let firstItem = items[0] as? Int {  
    print(firstItem + 4)  
}
```