Swift

- Optionals
- Type casting and inspection
- Guard
- Enumerations
Variables with nil

```swift
struct Book {
    var name: String
    var 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 unannouncedBook = Book(name: "Rebels and Lions", publicationYear: 0)

• Zero isn’t accurate, because that would mean the book is over 2,000 years old.

let unannouncedBook = Book(name: "Rebels and Lions", publicationYear: nil)
```
Optionals

• Normal variable in Swift cannot be nil

```swift
var string = nil // error!!
```

• Optionals contain either an instance of the expected type or nothing at all (nil).

```swift
var string: String? = nil // this works
var string: String? = "string" // this works as well
```

```swift
struct Book {
    var name: String
    var publicationYear: Int?
}
```
Working with Optionals

- Optionals can be unwrapped using the **force-unwrap** operator `!`:

```swift
let unwrappedYear = publicationYear! // runtime error
```

- Before unwrapping an optional we need to make sure the value is not `nil`:

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

- Shorter version:

```swift
if let actualYear = publicationYear {
    print(actualYear)
} else { }
```
Working with Optionals

• Unwrapping multiple optionals:

```swift
if let actualYear = publicationYear {
    if let bookEdition = edition {
        print(actualYear, bookEdition)
    }
}
```

• Optionals in functions:

```swift
func textFromURL(url: URL?) -> String?
{
    return nil
}
```

• Failable initializers:

```swift
init?() {
    return nil
}
```
Optional Chaining

- Unwrapping nested optionals:

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

class Residence {
    var address: Address?
}

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

- Shorter version:

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

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

- Force cast:

```swift
let cars = allVehiclesFrom(manufacturer: "Porsche") as! [Car]
```

- Use `as!` only when you are certain that the specific type is correct.

- If not your app will crash

```swift
class Vehicle {}
class Car : Vehicle {}
class Motorcycle : Vehicle {}

func allVehicles() -> [Vehicle] {
    //returns the all vehicles
}

let vehicles = allVehicles()
for vehicle in vehicles {
    if let car = vehicle as? Car {
        //..
    } else if let motorcycle = vehicle as? Motorcycle {
        //..
    }
}
The Any Type

• The Any type can represent an instance of any type: String, Double, func, struct, class …

```swift
var items: [Any] = [5, "Tom", 6.7, Car()]
if let firstItem = items[0] as? Int {
    print(firstItem+4)  // 9
}
```

• The AnyObject type can represent any class within Swift, but not a structure.
func singHappyBirthday() {
    if birthdayIsToday {
        if invitedGuests > 0 {
            if cakeCandlesLit {
                print("Happy Birthday to you!")
            } else {
                print("The cake's candles haven't been lit.")
            }
        } else {
            print("It's just a family party.")
        }
    } else {
        print("No one has a birthday today.")
    }
}

guard condition else {
    //false: execute some code
}
//true: execute some code

guard condition else {
    //false: execute some code
}
//true: execute some code

guard condition else {
    //false: execute some code
}
//true: execute some code
Guard

• If statements only allow access to the constant within the braces.

```swift
if let eggs = goose.eggs {
    print("The goose laid \(eggs.count) eggs.")
}
//`eggs` is not accessible here
```

• Guard statements allow access to the constant throughout the rest of the function

```swift
guard let eggs = goose.eggs else {
    return
}
//`eggs` is accessible hereafter
print("The goose laid \(eggs.count) eggs.")
```

• Unwrapping multiple optionals:

```swift
func processBook(title: String?, price: Double?, pages: Int?) {
    guard let theTitle = title, let thePrice = price, let thePages = pages else {
        return
    }
    print("\(theTitle) costs \(price) and has \(pages) pages.")
}
```
Enumerations

- Define a enumeration:

```swift
enum CompassPoint {
    case north
    case east
    case south
    case west
}
```

- Using enumerations:

```swift
var compassHeading: CompassPoint = .west
var compassHeading = CompassPoint.west

// The compiler assigns `compassHeading` as a `CompassPoint`
compassHeading = .north
```
Enumerations

- Type safety benefits:

```swift
struct Movie {
    var name: String
    var releaseYear: Int
    var genre: String
}

let movie = Movie(name: "Finding Dory", releaseYear: 2016, genre: "Animated")

enum Genre {
    case animated, action, romance, documentary, biography, thriller
}

struct Movie {
    var name: String
    var releaseYear: Int
    var genre: Genre
}

let movie = Movie(name: "Finding Dory", releaseYear: 2016, genre: .animated)
```

```swift
let movie = Movie(name: "Finding Dory", releaseYear: 2016, genre: "Tom")
```
Segues and Navigation Controllers
Segue
Segue Demo
Navigation Controllers
Navigation Controllers Demo
Playgrounds

• Start learning Swift by going through the 10 Playgrounds in L2P

• Control Flow
• Strings
• Functions
• Structures
• Classes
• Loops
• Optionals
• Type Casting
• Guard
• Enumerations
Access to Macs

- 8 Mac’s in the RBI
- Ahornstr. 55, E1 basement
- Open:
  - Mo. - Th. 9:00 - 20:00
  - Fr. 9:00 - 17:00
- Account: guest, no pw
- Logout deletes all of your data