import UIKit

//swift memory size

var x = 10

print(type(of:x))

let s = MemoryLayout.size(ofValue: x)//8 : 변수의 바이트 확인

let t = MemoryLayout<Int>.size // 자료형의 바이트 확인

print(s, t)

// Int

// 8 8

// 타입 추론

//"Implicitly"는 Swift 컴파일러가 변수 또는 상수의 타입을 자동으로 추론하는 것을 의미합니다. 예를 들어, 다음과 같이 변수를 선언할 때: let x = 10

//반면에, "explicit"는 타입을 명시적으로 선언하는 것을 의미합니다. 예를 들어, 다음과 같이 변수를 선언할 때:

//let x: Double = 3.14

//타입 선언은 코드의 명확성을 높이는 데 도움이 되지만, 타입 추론은 코드를 간결하게 유지하는 데 도움이 됩니다.

var userCount : Int = 10 // : Int가 type annotation

var signalStrength = 2.231 // type inference

let meaningOfLife = 42 // meaningOfLife Int형으로 타입 추론됨

let pi = 3.14159 // pi는 Double형으로 추론됨

let anotherPi = 3 + 0.14159 // anotherPi는 Double형으로 추론됨

var myChar3 : Character = "X"

//주의 여기서 : Character 생략불가, 생략하면 String형으로 인식

func logMessage(\_ s: String) { // {앞에 ->Void나 ->() 생략

print("Message: \(s)")

}

// 일급객체로써 변수에 넣을때는 생략 불가 즉 함수의 자료형을 명시하때는 생략 불가

let logger: (String) -> Void = logMessage //여기서 ->Void는 생략 불가

logger("Hello")

// 튜플

let myTuple = (10, 12.1, "Hi")

var myString = myTuple.2

print(myString) //출력되는 값은 : Hi

let (myInt, myFloat, myString2) = myTuple

// 튜플의 가장 강력한 점은 함수에서 여러 값들을 한 번에 반환하는 것

let myTuple2 = (count: 10, length: 12.1, message: "Hi") //과제 : myTuple의 자료

print(type(of:myTuple2)) // (count: Int, length: Double, message: String)

var y : Int? = 10 //y?

var z : Int! //z?

print(y,z)

//Optional(10) nil

// 옵셔널 변수는 초기값이 없으면 자동으로 닐이 들어감

// 초기 값이 있으면 Optional() 안에 들어감

print(Int("100"))

// 100이 아닌 Optional(100)을 리턴함,print(Int("100")), Int형 initializer

print(Int("Hi"))

// Optional(100)

// nil

// 정수 값을 반환할 수 없음, 아무런 값도 반환할 수 없다는 의미로 nil을 반환

//Swift에서 기본 자료형(Int, Double, String 등)은 nil값을 저장할 수 없음

//nil도 저장하려면 옵셔널 타입으로 선언해야 함

//forced unwrapping

var x2 : Int? //옵셔널 정수형 변수 x 선언

var y2 : Int = 0

x2 = 10 // 주석처리하면? : 오류남 nil을 풀어버려서 crash 발생

print(x2) // Optional(10)

print(x2!) // forced unwrapping해서 10이 나옴

print(y2) // 0

//x = x+2 //가능? x 옵셔널 형을 풀고 연산해야함

//y = x //가능? X 옵셔널 값을 변수에 넣을때는 옵셔널 변수로 선언되어 있어야함 즉 일반 변수에 넣을때는 !붙여서 풀어 넣줘야함

var x3 : Int?

x3 = 10

if x3 != nil {

print(x3!)

}

else {

print("nil")

}

// 주의 :

// if x!=nil

// 이라고 쓰면 안됨 옵셔널 x를 풀고 그안에 다시 nil을 넣을수 없다는 뚯

// optional binding

var x4 : Int?

x4 = 10

if let xx = x4 { //옵셔널 변수 x가 값(10)이 있으므로 언래핑해서 일반 상수 xx에 대입하고 if문 실행

print(x4,xx)

}

// if let x = x {

// print(x)

// } // 이 두가지 방법도 가능

// if let x {

// print(x)

// }

else {

print("nil")

}

var x5 : Int?

if let xx = x5 { //옵셔널 변수 x1이 값이 없어서 if문의 조건이 거짓이 되어 if문 실행하지 않고 else로 감

print(xx)

}

else {

print("nil")

}

//Optional(10) 10

//nil

var pet1: String?

var pet2: String?

pet1 = "cat"

pet2 = "dog"

if let firstPet = pet1, let secondPet = pet2 {

print(firstPet, secondPet)

} else {

print("nil")

}

// short form of if-let to the Optional Binding

if let pet1, let pet2 {

print(pet1, pet2)

} else {

print("nil")

}

//cat dog

//cat dog

//두 가지 Optional 형 : Int? vs Int!

let x : Int? = 1

let y : Int = x! // 강제 언래핑

let z = x // 옵셔널 인트형을 넣어줌

print(x,y,z) //Optional(1) 1 Optional(1)

print(type(of:x),type(of:y),type(of:z))

//Optional<Int> Int Optional<Int>

let a : Int! = 1 //Implicitly Unwrapped Optional

let b : Int = a //Optional로 사용되지 않으면 자동으로 unwrap함

// Int! 컴파일러가 엇 nil이잘 없는 옵셔널형(명시적으로 알려주는 것임 )이군아 자동으로 풀라는 말이네 하고 풀어버림

let c : Int = a!

let d = a //Optional로 사용될 수 있으므로 Optional형임 : d변수에 타입이 확실히 없어서 옵셔널 저장

let e = a + 1 // 컴파일러가 옵셔널 + 1 안되는데 Int! 라고 명시적으로 Nll 잘 없는 경우네 자동으로 풀자 하고 인트형 연산해줌

print(a,b,c,d,e) //Optional(1) 1 1 Optional(1) 2

print(type(of:a),type(of:b),type(of:c),type(of:d), type(of:e))

//Optional<Int> Int Int Optional<Int> Int

//즉 옵셔널이 항상 유효한 값을 가질 경우 옵셔널이 암묵적인 언래핑(implicitly unwrapped)이 되도록 선언하는 것임 : 컴파일러에게 명시적으로 알려줌 항상 유효한 값을 가진다고

var y : Int? = 10

print(y as Any) // 애니타입으로 변환시 경고 메세지 사라짐

// 프린트 함수에는 Any 즉 일반타입이 들어갈수 있는데

var x: Any = "Hi"

print(x, type(of:x))

x = 10

print(x, type(of:x))

x = 3.5

print(x, type(of:x))

//type을 검사해서 사용

// Hi String

// 10 Int

// 3.5 Double

//Nil-Coalescing Operator (Nil합병연산자)

// 옵셔널을 푸는 3가지 방법중 하나

let defaultAge = 1

var age : Int?

age = 3

print(age) //과제:값은?

var myAge = age ?? defaultAge

print(myAge) //과제: 값은?

// Optional(3)

// 3

let defaultColor = "black"

var userDefinedColor: String? // defaults to nil

var myColor = userDefinedColor ?? defaultColor

//nil이므로 defaultColor인 black으로 할당됨

print(myColor) //black

userDefinedColor = "red"

myColor = userDefinedColor ?? defaultColor

//nil이 아니므로 원래 값인 red가 할당됨

print(myColor) //red, 주의 optional(red)가 아님

// black

// red

// 삼항 연산자 따로 존재함

var x = 1

var y = 2

print("Largest number is \(x > y ? x : y)")

//Largest number is 2

// 클래스로부터 만들어진 객체를 인스턴스라 한다.

// 형 변환(as로 upcasting)

// 상속 관계가 있는 클래스들끼리만 타입 캐스팅 가능

// 자식(부모로부터 상속받아 더 많은 것을 가지고 있음)인스턴스를 부모로 캐스팅은 문제가 없음

// 형 변환(as! as?로 downcasting)

//다운캐스팅은 부모 인스턴스를 자식 클래스로 변환하는 데 사용

//성공 확신이 있으면 as! 키워드를 사용하며 강제 변환(forced conversion) : 변환이 안되면 crash

//성공 확신이 없으면 as?를 사용하여 안전하게 변환 ν 변환이 안되면 nil을 리턴하므로 옵셔널 타입으로 반환함

var x : Any = "Hi"

print(x, type(of:x))

x = 10

// 부모 as!,? 자식 : 다운 케스팅

var y : Int = x as! Int

var z : Int? = x as? Int

print(x, type(of:x))

print(y, type(of:y))

print(z, type(of:z))

// Hi String

// 10 Int

// 10 Int

// Optional(10) Optional<Int>

//Is 연산자

// 타입검사 : 이게 이런형이니?

let x = 1

if x is Int {

print("Int!")

}

// 객체가 MyClass라는 이름의 클래스의 인스턴스인지 검사

// 인스턴스가 해당 클래스인가?

// 인스턴스 is 클래스

// if myobject is MyClass {

// myobject는 MyClass의 인스턴스이다

// }

class A {}

let a = A()

if a is A{

print("Yes")

}

// Int!

// Yes

텍스트이(가) 표시된 사진

자동 생성된 설명

func sayHello(){

print("Hello")

print(#function) // 함수 이름 출력

}

sayHello()

print(type(of:sayHello))

//Hello

//sayHello()

//() -> ()

func sayHello2() -> Void {

print("Hello")

}

func sayHello3() -> () { // ()는 빈 튜플임 이것의 별명이 Void

print("Hello")

}

sayHello()

sayHello2()

sayHello3()

// 함수의 타입 (자료형,자료형,...) -> 리턴형 (Int, Int) -> Int

// 리턴형이 Void형이면 ()

func add(x: Int, y: Int) -> Int {

print(#function) // 함수 이름 출력

return(x+y)

}

print(add(x:10, y:20))

print(type(of:add))

func add1(first x: Int, second y: Int) -> Int {

print(#function) //외부 내부:자료형,외부 내부:자료형 -> 리턴형

return(x+y) //함수 정의할 때는 내부 매개변수명을 사용

}

print(add1(first:10, second:20))

print(type(of:add1))

func add2(\_ x: Int, \_ y: Int) -> Int {

print(#function)

return(x+y)

}

print(add2(10, 20))

print(type(of:add2))

// swift에서는 보통 첫번째 매개변수만 생략하고 나머지는 외부 매개변수명이 있음 : 공식 문서 보면 이름이 하나의 문장이 됨

func add3(\_ x: Int, with y: Int) -> Int {

print(#function)

return(x+y)

}

print(add3(10, with:20))

print(type(of:add3))

// add(x:y:)

// 30

// (Int, Int) -> Int

// add1(first:second:)

// 30

// (Int, Int) -> Int

// add2(\_:\_:)

// 30

// (Int, Int) -> Int

// add3(\_:with:)

// 30

// (Int, Int) -> Int

// 자료형은 다 같음, 함수이름은 다 다름 !!!!!!!!!!!!!!!!

// 중간고사

func tableView(\_ tableView: UITableView,numberOfRowsInSection section: Int) -> Int {

return items.count

}

// 자료형 (UITableView, Int) -> Int

// 함수 이름 tableView(\_: numberOfRowsInSection:)

//디폴트 매개변수(아규먼트) 정의하기

func sayHello(count: Int = 1, name: String = "길동") -> String {

return ("\(name), 너의 번호는 \(count)")

}

var message = sayHello()

print(message)

//길동, 너의 번호는 1

//1급객체 swift에 함수는 일급객체임

func up(num: Int) -> Int {

return num + 1

}

func down(num: Int) -> Int {

return num - 1

}

let toUp = up

print(up(num:10)) // 11

print(toUp(10)) // 11 toUp(num:10) 매개변수명을 사용하지 않음 extraneous 불필요하다고 함

// 함수 변수 대입시 불 필요함

let toDown = down

print(down(num:10)) // 9

print(toDown(10)) // 9

// 2 함수안에 매개변수 타입으로 함수 타입을 사용할 수 있음

func upDown(Fun: (Int) -> Int, value: Int) {// (Int) -> Int 타입인 함수 다 들어갈 수 있음

let result = Fun(value)

print("결과 = \(result)")

}

upDown(Fun:toUp, value: 10) //toUp(10)

upDown(Fun:toDown, value: 10) //toDown(10)

// 결과 = 11

// 결과 = 9

// 3) 함수도 리턴값으로 사용할 수 있다.

func decideFun(x: Bool) -> (Int) -> Int {

//매개변수형 리턴형이 함수형

if x {

return toUp

} else {

return toDown

}

}

let r = decideFun(x:true) // let r = toUp

print(type(of:r)) //(Int) -> Int

print(r(10)) // toUp(10)

let j = decideFun(x:false) // let j = toDown

print(type(of:j)) //(Int) -> Int

print(j(10)) // toDown(10)

// (Int) -> Int

// 11

// (Int) -> Int

// 9

// Swift의 함수는 1급 객체이다.

// 1급 객체(first class object) 또는 1급 시민(first class citizen)

// 다음 조건을 충족하는 객체를 1급 객체라고 한다.

// 1) 변수에 저장할 수 있다.

// 2) 매개변수로 전달할 수 있다.

// 3) 리턴값으로 사용할 수 있다.

//시험

//클로저 : 이름 없는 함수, 익명함수

func add(x: Int, y: Int) -> Int {

return(x+y)

}

print(add(x:10, y:20))

let add1 = { (x: Int, y: Int) -> Int in

return(x+y)

}

// 변수에 담아 사용시 매개변수명 사용하지 않음

print(add1(10, 20))

// 30

// 30

//프로퍼티는

//1. 초기값이 있거나

//2. init을 이용해서 초기화하거나

//3. 옵셔널 변수(상수)로 선언(자동으로 nil로 초기화)

class Man{

var age : Int = 1 // 프로퍼티

var weight : Double = 3.0

} //오류 나는 이유? == 초기값이 없어서 그럼, 초기값을 주거나, 생성자를 만들면 오류가 나지 않음

// 클래스에 저장속성은 반드시 초기 값이 있어야함

//

class Man2{

var age : Int? //옵셔널 변수는 nil로 자동 초기화

var weight : Double!

}

class Man3{

var age : Int

var weight : Double

init(){ //initializer로 초기화

age = 1

weight = 3.5

}

}

//인스턴스 메서드와 클래스 메서드, 인스턴스 변수 선언 방법

class Man{

var age : Int = 1

var weight : Double = 3.5

func display(){ //인스턴스 메서드

print("나이=\(age), 몸무게=\(weight)")

}

class func cM(){ // class키워드로 만든 클래스 메서드는 자식 클래스에서 override가능 함

print("cM은 클래스 메서드입니다.")

}

static func scM(){

print("scM은 클래스 메서드(static)")

}

}

var kim : Man = Man() // 일반 자료형의변수를 만들때는 var age : Int 만하면 만둘수 있지만

// var kim : Man 클래스의 인스턴스를 담는 변수는 이렇게만 하면 만들 수 없음

// 생성자까지 호출해줘야함

kim.display() //인스턴스 메서드는 인스턴스가 호출

print(kim.age)

Man.cM() //클래스 메서드는 클래스가 호출

Man.scM() //클래스 메서드는 클래스가 호출

// 나이=1, 몸무게=3.5

// 1

// cM은 클래스 메서드입��다.

// scM은 클래스 메서드(static)

프로퍼티, 메서드, 생성자 만들기

//프로퍼티는

//1. 초기값이 있거나

//2. init을 이용해서 초기화하거나

//3. 옵셔널 변수(상수)로 선언(자동으로 nil로 초기화)

class Man{

var age : Int = 1 // 프로퍼티

var weight : Double = 3.0

} //오류 나는 이유? == 초기값이 없어서 그럼, 초기값을 주거나, 생성자를 만들면 오류가 나지 않음

// 클래스에 저장속성은 반드시 초기 값이 있어야함

//

class Man2{

var age : Int? //옵셔널 변수는 nil로 자동 초기화

var weight : Double!

}

class Man3{

var age : Int

var weight : Double

init(){ //initializer로 초기화

age = 1

weight = 3.5

}

init(yourAge: Int, yourWeight : Double){

age = yourAge

weight = yourWeight

}

//designated initializer 로 초기화 가능함

// 모든 프로퍼티(age, weight)를 다 초기화시키는 생성자

// init()을 하나라도 직접 만들면 기본적으로 만들어지는 눈에 안보이는 default initializer는 사라짐

// 변수이름이 같을때

// 현재 클래스 내 메서드나 프로퍼티를 가리킬 때 메서드나 프로퍼티 앞에 self.을 붙임

init(age: Int, weight : Double){

self.age = age

self.weight = weight

}

}

var kim : Man3 = Man3(yourAge:10, yourWeight:20.5) //일반적인 init()호출 방법

//인스턴스 메서드와 클래스 메서드, 인스턴스 변수 선언 방법

class Man{

var age : Int = 1

var weight : Double = 3.5

func display(){ //인스턴스 메서드

print("나이=\(age), 몸무게=\(weight)")

}

class func cM(){ // class키워드로 만든 클래스 메서드는 자식 클래스에서 override가능 함

print("cM은 클래스 메서드입니다.")

}

static func scM(){

print("scM은 클래스 메서드(static)")

}

}

var kim : Man = Man() // 일반 자료형의변수를 만들때는 var age : Int 만하면 만둘수 있지만

// var kim : Man 클래스의 인스턴스를 담는 변수는 이렇게만 하면 만들 수 없음

// 생성자까지 호출해줘야함

kim.display() //인스턴스 메서드는 인스턴스가 호출

print(kim.age)

Man.cM() //클래스 메서드는 클래스가 호출

Man.scM() //클래스 메서드는 클래스가 호출

// 나이=1, 몸무게=3.5

// 1

// cM은 클래스 메서드입��다.

// scM은 클래스 메서드(static)

//failable initialize가 있는 클래스의 인스턴스 생성

//init다음에 "?"나 "!"를 하며 옵셔널 값이 리턴됨

// 성공시 인스턴스 생성

class Man{

var age : Int = 1

var weight : Double = 3.5

func display(){

print("나이=\(age), 몸무게=\(weight)")

}

init?(age: Int, weight : Double){ // 옵셔널 형으로 실패시 반환함

if age <= 0 {

return nil

}

else {

self.age = age

self.weight = weight

}

} // failable initialize

}

var kim : Man? = Man(age:10, weight:20.5)

kim!.display() // nil 나올경우 크래시 날 수 있움

//1-1.옵셔널 형으로 선언

if let kim1 = kim { //1-2.옵셔널 바인딩

kim1.display()

}

//2.인스턴스 생성과 동시에 옵셔널 바인딩

if let kim2 = Man(age:2, weight:5.5) {

kim2.display() // 옵셔널 형 어짜피 안쓸거라 바로 방인딩 해서 사용

}

//3.인스턴스 생성하면서 바로 강제 언래핑

var kim3 : Man = Man(age:3, weight:7.5)!

kim3.display()

//4.옵셔널 인스턴스를 사용시 강제 언래핑

var kim4 : Man? = Man(age:4, weight:10.5)

kim4!.display()

//가장 좋은 방법은 1, 2번째

//상속 : 상속은 클래스만 가능

// 부모 클래스는 하나만 가능하며 여러 개라면 나머지는 프로토콜

class Man{

var age : Int

var weight : Double

func display(){

print("나이=\(age), 몸무게=\(weight)")

}

init(age: Int, weight : Double){

self.age = age

self.weight = weight

}

}

class Student : Man {

//비어있지만 Man의 모든 것을 가지고 있음

}

var kim : Man = Man(age:10, weight:20.5)

kim.display()

var lee : Student = Student(age:20,weight:65.2)

lee.display()

print(lee.age)

// 나이=10, 몸무게=20.5

// 나이=20, 몸무게=65.2

// 20

class Man{

var age : Int

var weight : Double

func display(){

print("나이=\(age), 몸무게=\(weight)")

}

init(age: Int, weight : Double){

self.age = age

self.weight = weight

}

}

class Student : Man {

var name : String

func displayS() {

print("이름=\(name), 나이=\(age), 몸무게=\(weight)")

}

init(age: Int, weight : Double, name : String){

self.name = name

super.init(age:age, weight:weight) //이 줄을 안쓰면?

}//error:'super.init' isn't called on all paths before returning from initializer

} //자식 클래스에서 designated initializer를 만들면 부모 init()상속 안됨

/\*

자식쪽에서 부모쪽으로 접근할때는 super키워드를 사용하고

현재 자기자신을 접근할때는 self를 사용한다.

\*/

var lee : Student = Student(age:20,weight:65.2,name:"홍길동")

lee.displayS()

lee.display()

// 이름=홍길동, 나이=20, 몸무게=65.2

// 나이=20, 몸무�=65.2

class Man{

var age : Int = 1

var weight : Double = 3.5

func display(){

print("나이=\(age), 몸무게=\(weight)")

}

init(age: Int, weight : Double){

self.age = age

self.weight = weight

}

}

class Student : Man {

var name : String = "김소프"

override func display() { //같은 이름의 메서드가 부모에도 있음

print("이름=\(name), 나이=\(age), 몸무게=\(weight)")

}

init(age: Int, weight : Double, name : String){

self.name = name

super.init(age:age, weight:weight)

}

}

var lee : Student = Student(age:20,weight:65.2,name:"홍길동")

lee.display()

// override : 부모와 자식에 같은 메서드가 있으면 자식 우선

// 부모와 자식에 display()라는 메서드가 있어서 Student클래스는 display() 메서드가 두 개임

// • Student클래스의 인스턴스 lee가 display()를 호출할 때, 자식클래스가 새로 만든 display()

// 메서드가 우선적으로 호출되려면 func 앞에 override키워드 씀

//이름=홍길동, 나이=20, 몸무게=65.2

// 프로토콜

protocol Runnable { //대리하고 싶은 함수 목록 작성

var x : Int {get set} //읽기와 쓰기 가능 프로퍼티,{get}은 읽기 전용

//property in protocol must have explicit { get } or { get set } specifier

func run() //메서드는 선언만 있음

}

class Man : Runnable { //채택, adopt

var x : Int = 1 //준수, conform

func run(){print("달린다~")} //준수, conform

}

// γ class Man에 x, run()정의 없다면

// ν type 'Man' does not conform to protocol 'Runnable‘

let han = Man()

print(han.x)

han.run()

// 1

// 달린다~

// 옵셔널 체이닝

//실행문에서 x? 이런식으로 쓰는게 옵셔널체이닝임

//보통은 문장이 길어질때 ! 가 있어도 강제 언레핑이라고하며 ?만 옵셔널체이닝이라고 함 .

//변형하여 실습 결과 자세히 설명

//너무 중요한 실습입니다!!

var x : String? = "Hi"//Hi지우고도 실습 print(x, x!)

print(x, x!) // Optional("Hi") Hi

if let a = x{ //옵셔널 바인딩 if let x 이렇게만 써도됨 (시험)

print(a) // Hi

}

let b = x!.count // x가 옵셔널 타입이라 항상 풀어서 사용해야함 강제 언래핑 방법

print(type(of:b),b) // Int 2

let b1 = x?.count // 옵셔널 체이닝 : 문장중에 ?가 하나라도 있으면 결과가 최종적으로 옵셔널 값으로 나옴

print(type(of:b1),b1, b1!)

//Optional<Int> Optional(2) 2

let c = x ?? "" // 시험 ?? 가뭔지 + "" 는 null 이라 화면에 출력 보이지 않음

print(c) // Hi 출력됨

// Swift에서 ?? 연산자는 nil-coalescing operator(결합 연산자)라고 불리며, 옵셔널 값이 nil인 경우에 대체 값을 제공하는 연산자입니다.

class Person {

var name: String

var age: Int

init(name: String, age: Int) {

self.name = name

self.age = age

}

}

let kim: Person = Person(name: "Kim", age: 20)

print(kim.age)

let han: Person? = Person(name: "Han", age: 25)

//print(han.age) 옵셔널 형 그냥 사용해서 에러남

print(han!.age) //풀어서 사용하기 때문에 값이 없으면 에러날수 있음

print(han?.age) //Optional(25), 옵셔널 체이닝 // print((han?.age)!)

if let hanAge = han?.age { // han?.age 결과 값이 옵셔널이라 그것을 풀어사용

print(hanAge)

} else {

print("nil")

}

// 20

// 25

// Optional(25)

// 25

// 실패가능 생성자랑 같이 사용할때

class Person {

var name: String?

init?(name: String) {

if name.isEmpty {

return nil

}

self.name = name

}

}

// 옵셔널 체이닝을 사용한 코드

let person = Person(name: "John")

let personName = person?.name?.count

if let name = personName {

print(name) // 4

} else {

print("Name is nil")

}

텍스트이(가) 표시된 사진

자동 생성된 설명

테이블뷰 10단계

도표이(가) 표시된 사진

자동 생성된 설명

1

View에 Table View를 드래그하여 추가

storyboard에서 Library추가 단축키 shift+cmd+L

2

Pin Tool로 Add New Constraints : Table View를 화면 전체를 채움

3 뷰컨에 연결되어 있는지 확인

Assistant editor로 storyboard와 소스 연결

ctrl+alt+command+enter

Table View outlet 설정 : table (접근하기 위해)

4

UITableViewDelegate와 UITableViewDataSource 채택하고 필수 메서드 준수

// 셀 개수, 셀을 어떻게 보여줄지, 섹션의 개수 자동으로 구현안하면 1 (수정시 NumberofSections 메서드 사용)

// 기본셀을 직접 생성해서 사용해서 연결해줄 필요 없음

func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

return 10

}

func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

// 클래스 인스턴스를 만듬

let cell = UITableViewCell(style: .subtitle, reuseIdentifier: "myCell")// .init 생략 가능

cell.textLabel?.text = "\(indexPath.row)"

// 옵셔널 체이닝 사용 textLabel 이 UILabel? 옵셔널 타입임 문자열만 넣을수 있어서 문자열 보관법 사용

cell.detailTextLabel?.text = indexPath.description

// description 가 스트링 타입임

// [0, 0] 앞은 섹션의 수, 뒤는 셀의 개수

cell.imageView?.image = UIImage(named: "apple-logo")

print(indexPath.description) // cellForRowAt 외부매개변수 자체를 다루는 일은 잘 없음 프레임워크 내부에서 처리되는 이름임

// description 이라는거 안에 섹션과 row를 묶은 배열을 가지고 있는 것임 문자열로

return cell

}

func numberOfSections(in tableView: UITableView) -> Int {

return 3

}

// 57p

5

내가 만든 table의 delegate와 dataSource는 이 ViewController 클래스에 구현.

이 클래스가 table의 delegate이야

table.dataSource = self

table.delegate = self

6

Table View Cell을 직접 디자인 : Table View Cell 추가

cell의 identifier추가: myCell

7

Cell을 관리할 swift파일 만들기 : [Subclass of:] 부분에 부모 클래스로 UITableViewCell 먼저 지정

myCell과 관리할 클래스(myTableViewCell) 연결

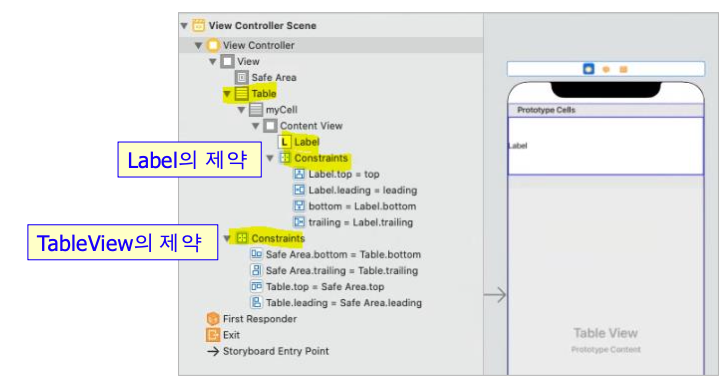
Identity inspector의 class부분 콤보 박스 아래쪽 방향 화살표 누려면 MyTableViewCell 선택 가능

8

myCell의 content view에 Label추가하고 Constraints설정

+Constraint to margins 체크하면 모두 0으로 하더라도 마진 만큼 안쪽에 위치

+Constraint to margins 체크 해제하면 마진 없이 완전히 꽉 참



9

ViewController.swift파일 수정

tableView(\_: cellForRowAt:) 테이블뷰의특정위치에삽입할셀에대한데이터소스를요청

테이블뷰의 재사용 메커니즘 : reusable table-view cell를 리턴하는 함수

cell identifier와 indexPath 지정

let cell = tableView.dequeueReusableCell(withIdentifier: "myCell", for: indexPath) as!MyTableViewCell

// 반환값이 부모 타입이라 자식 타입으로 다운 케스팅을해서 myLabel접근 가능 기본은 UITableViewCell형으로 리턴함

10

cell이 선택되었을 때 반응하기 : tableView(\_:didSelectRowAt:)

row가 선택되면 delegate에게 알림 indexPath를 알수있음

func tableView(\_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

print(indexPath.description) // 이쪽에 보이는컨 손솔창에서 보이는 것임 화면에 보이려면

// cellForRowAt 함수 이용

}

import UIKit

let food = ["1 아이스 아메리카노", "2 치킨", "3 스테이크", "4 피자"]

let name = ["1 우기 커피", "2 우기닭", "3 우테이크", "4 우자"]

class ViewController: UIViewController, UITableViewDelegate, UITableViewDataSource {

@IBOutlet weak var table: UITableView!

override func viewDidLoad() {

super.viewDidLoad()

// Do any additional setup after loading the view.

table.dataSource = self

table.delegate = self

}

func numberOfSections(in tableView: UITableView) -> Int {

return 3 // 3개의 섹션을 구현

}

func tableView(\_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {

return food.count // 로우(셀) 개수 10개

}

func tableView(\_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {

let cell = tableView.dequeueReusableCell(withIdentifier: "myCell", for: indexPath) as!MyTableViewCell // 반환값이 부모 타입이라 자식 타입으로 다운 케스팅을해서 myLabel접근 가능

//cell.myLabel.text = indexPath.description // String 타입이라 description으로 넣어줌

//cell.myLabel.text = "\(indexPath.row)"// indexPath.row 는 int 형 중요

//print(indexPath.description, terminator: " ")

// [0, 0] [0, 1] [0, 2] [0, 3] [0, 4] [0, 5] [0, 6] [0, 7] 처음 화면 보여지는 놈들 만큼

// 함수가 반복됨 엄청 바쁜 함수임

cell.myLabel.text = food[indexPath.row]

cell.name.text = name[indexPath.row]

return cell

}

func tableView(\_ tableView: UITableView, didSelectRowAt indexPath: IndexPath) {

print(indexPath.description, terminator: " ") // 셀 선택시 매번 호출하여 해당 인덱스 패스를 확인 할 수 있음

}

}

MyTableViewCell 소스

//

// MyTableViewCell.swift

// Table047

//

// Created by comsoft on 2023/04/20.

//

import UIKit

class MyTableViewCell: UITableViewCell {

@IBOutlet weak var name: UILabel!

@IBOutlet weak var myLabel: UILabel!

override func awakeFromNib() {

super.awakeFromNib()

// Initialization code

}

override func setSelected(\_ selected: Bool, animated: Bool) {

super.setSelected(selected, animated: animated)

// Configure the view for the selected state

}

}