

# Swift 기초

- XCode와 Swift 버전
- XCode 7.2 : Swift 2.1
- XCode 7.3 : Swift 2.2
- Xcode 8 : Swift 3

- 닫힌(Closed) 범위
  - $1...10$  : 1에서 10까지, 10 포함
- 반 닫힌(Half Closed) 범위
  - $1..<10$  : 9까지

# 데이터 다루기

- 변수 : 변경 가능
  - var 로 선언
- 상수 : 변경 불가능
  - let 으로 선언
- 타입 정보 필요(선언 생략 가능)

- 변수 : 변하는 값 다루기
  - var로 선언
  - 타입 선언 : 콜론(:) 이후 선언
  - 타입에 맞는 값만 대입 가능

- 변수

```
var i = 1 // 타입 선언 생략  
var f : Float = 1.1 // 타입 선언
```

- 문자열

```
var str = "Hello, Swift"  
str = "Hello, iOS"
```

- 상수
  - 변경 불가능
  - let 으로 선언

- 상수 실습

```
let constant = 123  
constant = 456
```

❗ Cannot assign to 'let' value 'constant'



- 초기화
  - 변수나 상수 - 사용 전에 초기화 필수
  - 자동 초기화 안됨.
  - 초기화 전에 사용하면 에러.
  - 선언과 초기화 분리 가능. 타입 선언 생략 불가

- 부울 : Bool
- 정수 : Int, UInt
- 실수 : Float, Double
- 문자, 문자열 : Character, String

- 문자 : Character

```
let char : Chracter = "a"
```

- 유니코드 문자 다루기

```
let char2 : Character = "\u{63}" // c
let char3 : Character = "\u{2665}" // Black Heart - ♥
```

- 문자열

```
var str = "Hello, Swift Language"
```

- 타입 변환

```
var strFromInt = String(100) // "100"  
var strFromBool = String(true) // "true"
```

- String Interpolation

```
let str2 = "Swift"  
let str3 = "Hello, \(str2)"  
let str4 = "1 + 2 = \(1+2)"
```

- 길이

```
let str = "Hello Swift"  
str.characters.count // 11
```

```
// 유니코드 문자를 포함하는 문자열  
let str2 = "I \u{2665} Swift" // I ♥ Swift  
str2.characters.count // 9
```

- 문자열 붙이기

- + 연산자

```
var str = "Hello"  
str = str + " Swift"
```

- append(\_:), append(\_:) 함수

```
var str2 = "Hello"  
// String 덧붙이기  
str2.append(" Swift")
```

- 문자열 비교 : ==

- 문자열 시작, 끝

```
str.hasPrefix("Hello")  
str.hasSuffix("ground")
```

- 튜플(tuple)

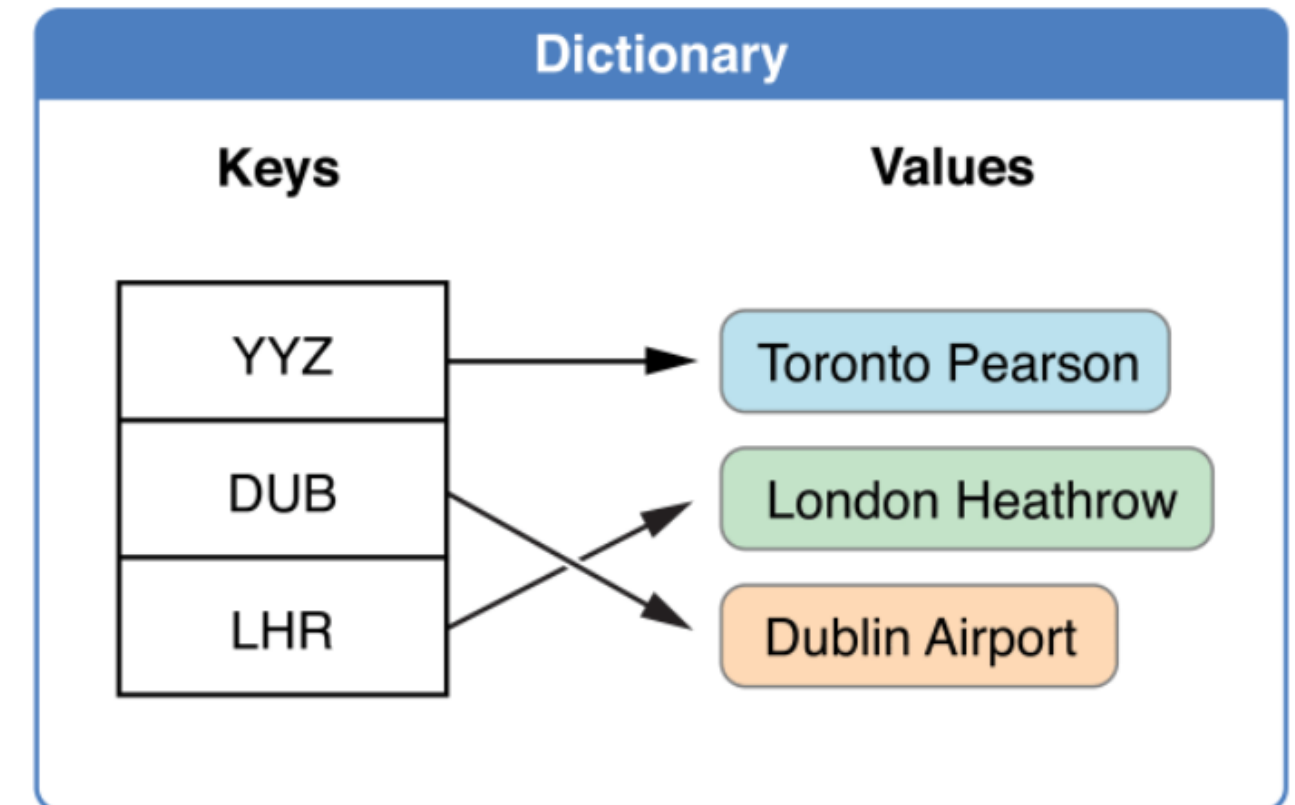
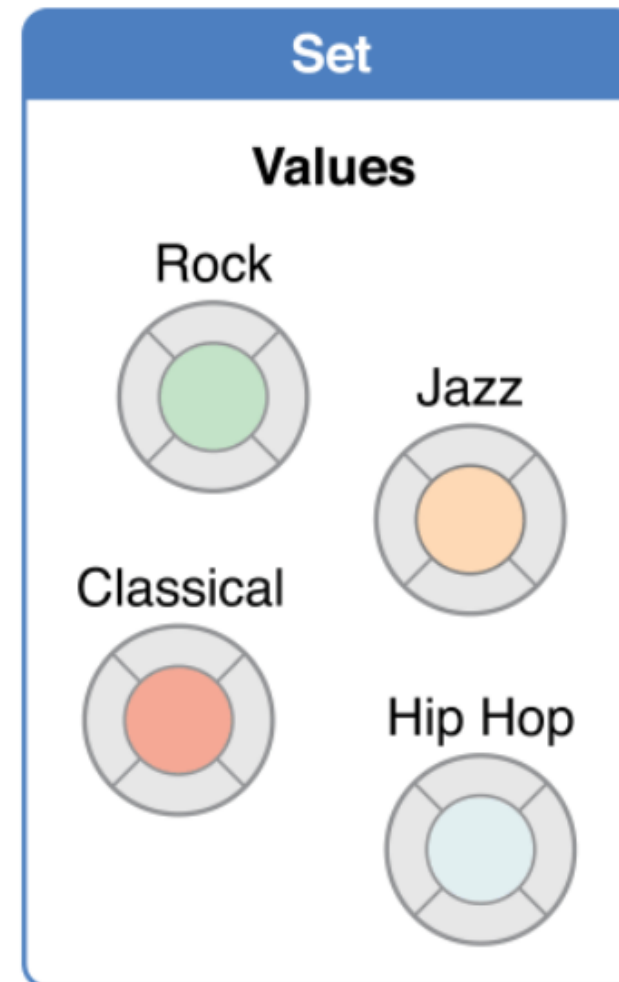
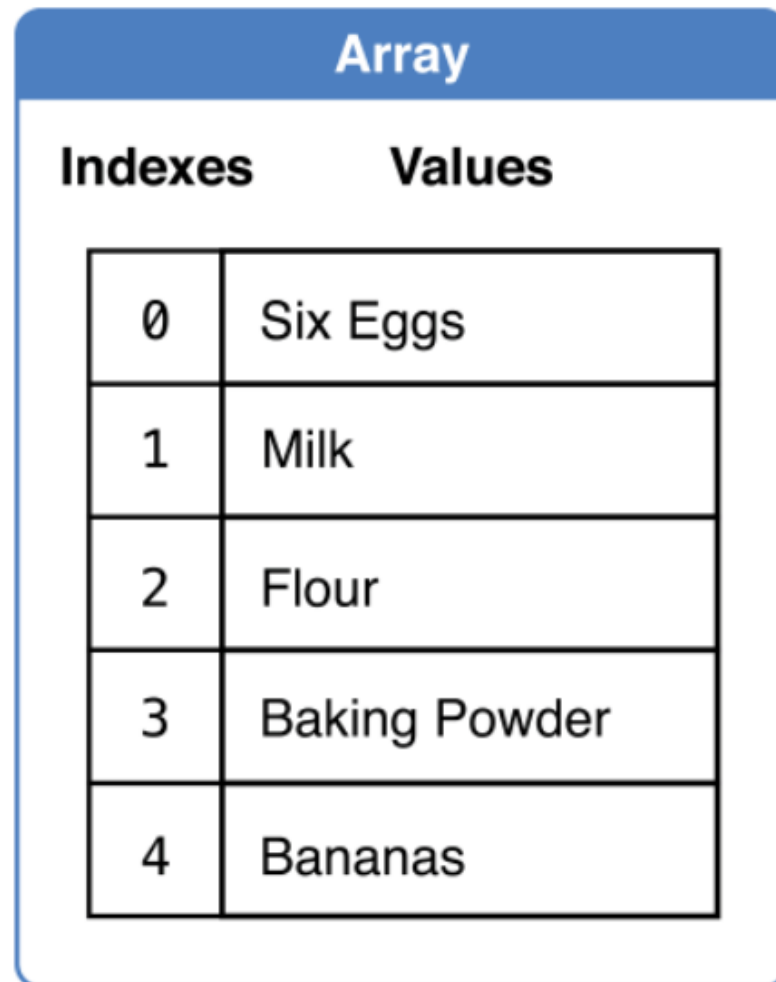
(1, "one", "일")

- 변수로 다루기

```
var one = (1, "one", "일")
```



- 컬렉션



- 배열
  - 인덱스 기반, 다수의 데이터 다루기  
`Array<Element>, []`
- 배열 변수, 상수
  - var로 배열 생성 - 배열 내용 수정 가능
  - let으로 배열 생성 - 배열 내용 수정 불가(immutable)

- 배열 생성

```
var intArray : [Int] = [1, 2, 3, 4, 5]
```

```
let strArray = ["A", "B", "C"]
```

```
let floatArray = Array<Float>([1.1, 2.2, 3.3])
```

- 공백 배열, (타입 선언 필요)

```
var emptyArray = [Int]()
```

```
var emptyArray2 = Array<Double>()
```

```
var emptyArray3 : [String] = []
```

- 원소 개수와 공백 확인

```
floatArray.count  
emptyArray.isEmpty
```

- 항목 접근. 첨자 표기(subscript)

```
let element1 = intArray[0]  
let element2 = floatArray[1]
```

- 항목 추가

```
mutating func append(_ newElement: Element)  
mutating func insert(_ newElement: Element, at i: Int)  
+ 연산자
```

- 배열 항목 삭제

```
intArray.removeAtIndex(3)  
intArray.removeLast()
```

- 딕셔너리(Dictionary)
  - 키 - 값 방식으로 다수의 값 다루기
  - 딕셔너리 내 키는 유일
  - 키 : 해쉬값을 제공할 수 있는 Hashable

- 객체 생성

```
var dic = ["1월":"January", "2월":"February", "3월":"March"]  
var dic2 : [String: Int] = ["일" : 1, "이" : 2, "삼" : 3]  
var dic3 : Dictionary<Int, String> =  
    [1 : "One", 2 : "Two", 3 : "Three"]
```

- 빈 딕셔너리 객체 생성

```
var emptyDic = [Int: Int]()
```

- 원소 갯수
  - count 프로퍼티
- 접근(subscript)

```
dic["2월"]  
dic["NotExistKey"] // nil
```

- 원소 추가 - 새로운 키로 값 설정

```
dic1["4월"] = "april"  
dic1["5월"] = "May"
```

- 변경 - 있는 키

- updateValue(\_: forKey:)

```
dic1["4월"] = "잔인한 달"  
dic1.updateValue("계절의 여왕", forKey: "5월")
```

- 원소 삭제

- removeValue(forKey:)

```
dic.removeValue(forKey: "2월")  
dic["3월"] = nil
```



# 제어문

- for 문 – C 스타일

- 삭제( Swift 3 )

```
for ( var i = 0 ; i < 10 ; i++) {  
    print("For 반복문 \ \(i)")  
}
```

- for-in( 범위 ) : C 스타일의 for 대신

```
for item in 0..  
10 {  
    print(item)  
}
```

- for-in( 컬렉션 )

```
for item in [1, 2, 3, 4, 5] {  
    print(item)  
}
```

- while

```
while i < 10 {  
    print("i = \(i++)")  
}
```

- repeat-while

```
repeat {  
    if j > 10 { break }  
} while true
```

- 부울 형태의 조건
- 조건부의 괄호 생략 가능
- 바디의 중괄호 생략 불가(심지어 1줄도)

```
if 3 > 2 {  
    print("3이 2보다 크다!")  
}
```

- 조건절 : 결과가 Bool

```
if obj {}  
if obj != nil {}
```

- if 조건문과 비슷
  - 바디에는 exit 명령 필수
  - 조건 작성 방식이 다르다.
- 바인딩 스코프

```
func example() {  
    guard let val = someFunc() else {  
    }  
    print("val은 유효한 값 : \(val)")  
}
```

- Early Exit 구현

if	guard
<pre>for item in array {     if item &lt; 0 {         break     }     print(item) }</pre>	<pre>for item in array {     guard item &gt; 0 else {         break     }     print(item) }</pre>

- 바인딩
  - 유효한 값 판단(nil 여부)

```
if let val = someFunc() {  
    print("\(val)")  
}
```

- 다중 바인딩

```
if let val1 = someFunc(), let val2 = anotherFunc() {  
    print("val1, val2는 모두 유효한 값")  
}
```

- switch 조건문
- 조건에 맞는 case 하나만 실행
  - break 불필요
  - fallthrough : 다음 case 실행
- 모든 조건 다루기
  - 모든 상황에 대한 case 작성
  - default

```
case 2:  
    print("2")  
}
```

❗ Switch must be exhaustive, consider adding a default clause



- 기본

```
switch someValue {  
  case 1:  
    print("1")  
  case 2:  
    print("2")  
  default:  
    print("Other")  
}
```

nil, 옵셔널

- 옵셔널 - nil이 될 수 있는 변수
- 타입 뒤에 물음표(?)로 선언
- 타입 선언 필수

```
var optionalVar : Int?
```

- 예러 보기

```
var optionalStr : String? = "ABC"
```

```
// OptionalType 직접 사용은 불가!
```

```
optionalStr.lowercaseString
```

- if-let 바인딩

```
if let realStr = optionalStr {  
    print("문자열의 값 \ (realStr)")  
}  
else {  
    print("문자열이 nil이다.");  
}
```

- 옵셔널에서 값 얻어오기(Unwrapping)

- 대문자 만들기 : `uppercaseString`

- 직접 접근

~~`var str = optionalStr.uppercaseString`~~

- ?를 이용한 접근

`var str = optionalStr?.uppercaseString`

- !를 이용한 접근

`var str = optionalStr!.uppercaseString`

# 함수

- 파라미터, 리턴값이 없는 함수

```
func greeting() {  
    print("Hello Swift")  
}
```

- 사용

```
greeting()
```



- -> 로 결과 타입 작성
- 반환타입이 없으면(void) - 생략 가능

```
func areYouOK() -> Bool {  
    return true  
}  
let ok = areYouOK()
```

```
func favoriteDrink() -> String {  
    return "Coffee"  
}  
let drink = favoriteDrink()
```

- 파라미터가 1개 있는 함수 정의

```
func greeting(person : String) {  
    print("Hello " + person)  
}
```

- 사용

```
greeting(person : "Friend")
```

# 파라미터 이름

- 파라미터 이름 : 내부 파라미터 이름, 외부 파라미터 이름

- 내부 파라미터 이름

- 함수 내부에서 접근, 사용

```
func greeting(person : String) {  
    print("Hello " + person)  
}
```

- 외부 파라미터 이름

- 함수 외부(함수 호출)에서 사용

- 자동으로 내부 파라미터 이름으로 사용

```
greeting(person: "My Friend")  
greeting(person:"My Friend", emotion: "Smile")
```

- 외부 파라미터 이름
- 자동으로 내부 파라미터 이름 사용
- 수동으로 설정하기

```
func greeting(who person : String) {  
    print("Hello " + person)  
}  
greeting(who: "Swift")
```

```
func greeting(person : String, with emotion : String) {  
    print("Hello " + person + " with " + emotion)  
}  
greeting(person: "My Friend", with: "Hug")
```

- 외부 파라미터 이름 사용 생략 : \_ 기호 사용

```
func greeting(_ person : String) {  
    print("Hello " + person)  
}
```

```
greeting("Friend")
```

```
func greeting(person : String, _ emotion : String) {  
    print("Hello " + person + " with " + emotion)  
}
```

```
greeting(person: "Swift", "Passion")
```

- 동작 결과로 옵셔널 반환

```
func nilReturnFunction() -> Int? {  
    return nil  
}
```

```
// 옵셔널 타입.
```

```
let ret = nilReturnFunction()
```

# 클래스

- 정의하기

```
class Greeting {  
}
```

- 객체 생성

```
var rect = Rectangle()
```



- 사각형 클래스(Rectangle)
- 저장 프로퍼티
  - 가로 길이
  - 세로 길이
- 계산 프로퍼티
  - 크기
  - 정사각형인가?

- 저장 프로퍼티
- 초기값

```
class MyClass {  
    // 초기값을 설정한 프로퍼티  
    var intProperty = 0  
  
    // 초기값을 설정하지 않은 옵셔널 프로퍼티  
    var floatProperty : Float?  
  
    // 에러 - 초기화되지 않는 프로퍼티  
var strProperty : String  
}
```

- 정의

```
var [프로퍼티 이름] : [타입] {  
    get {  
        return RETURN_VALUE  
    }  
    set( newValue ) {  
  
    }  
}
```

- 저장 프로퍼티와 계산 프로퍼티

```
class Person {  
    // 상수  
    let thisYear = 2016  
    var birthYear : Int = 0  
  
    // 계산 프로퍼티  
    var age : Int {  
        get {  
            return thisYear - birthYear  
        }  
        set {  
            birthYear = thisYear - newValue  
        }  
    }  
}
```

# 초기화

# 메소드 사용하기

- 메소드 정의

```
class Counter {  
    var count = 0  
  
    func increment() {  
        count++  
    }  
    func increment(amount : Int) {  
        count += amount  
    }  
    func increment(amount: Int, times : Int) {  
        count += amount * times  
    }  
}
```

- 객체 생성 후 사용

```
let counter = Counter()  
  
counter.increment()          // 1  
counter.increment(amount: 5) // 6  
counter.increment(amount:3, times: 5)
```

- 모든 객체는 사용하기 전에 초기화
- 프로퍼티 초기화
  - 초기값과 함께 선언된 프로퍼티
  - 옵셔널 타입의 프로퍼티
  - 초기값이 없고, 옵셔널 타입이 아닌 프로퍼티

- 클래스 선언

```
class Rectangle {  
    // 객체 생성시 0으로 초기화  
    var width = 0  
    // 초기화 되지 않음 - Initializer 필요  
    var height  
    // 옵셔널 타입. nil로 자동 초기화  
    var name : String?  
}
```

- 객체 생성

```
var obj = Rectangle()
```



- Convenience init
  - 단독으로 객체 초기화 불가
  - 초기화가 필요한 모든 프로퍼티를 초기화하지 않음
  - 다른 초기화 메소드에 의존. (Initializer Delegation)
- Initializer Delegation
  - 다양한 객체 생성 방법 제공 -> init 메소드 다수
  - 초기화 코드의 중복 방지. 재사용 높이기
  - 다른 init 메소드 호출하기

- Designated Initializer들

```
class Rectangle {  
    var width : Int  
    var height : Int  
    init() {  
        width = 0  
        height = 0  
    }  
    init(width : Int, height : Int) {  
        self.width = width  
        self.height = height  
    }  
}
```

모든 프로퍼티  
초기화

모든 프로퍼티  
초기화

- Convenience\_INITIALIZER
  - 초기화 위임 이후에 다른 초기화 동작 작성

```
convenience_init([파라미터]) {  
    // 초기화 위임  
    // 초기화 코드  
}
```
- Initializer Delegate



- Initializer

```
class MyClass {  
    var a, b : Int  
    init() {  
        a = 0  
        b = 0  
    }  
    init(a:Int, b:Int) {  
        self.a = a  
        self.b = b  
    }  
    convenience init(b:Int) {  
        self.init() // Initializer delegation  
        self.b = b  
    }  
}
```

designated initializer

designated initializer

initializer 위임

- birthYear 조건 체크

```
init?(birthYear : Int) {  
    if birthYear <= 1900 {  
        return nil  
    }  
    else {  
        self.birthYear = birthYear  
    }  
}
```

# 타입 메소드

- 타입 메소드, 인스턴스 메소드

```
class MyClass {  
    var property = 0  
  
    // 타입 메소드  
    static func typeMethod() {  
        property = 2 // 에러. 타입 메소드에서 프로퍼티 접근 불가  
        print("Type method works")  
    }  
  
    func instanceMethod() {  
        property = 1 // 인스턴스 메소드에서 프로퍼티 접근 가능  
        print("Instance method works")  
    }  
}
```

ARC



- 메모리 관리
  - 필요한 객체 유지
  - 필요없는 객체 해제
- 사용 중인 객체를 유지하려면?
- 사용 중이라는 표시 - 소유하기(own)
- 객체 소유하는 방법 : 강한 참조(Strong pointer)

- 객체 생성 - 객체 소유

```
var ptr : MyClass! = MyClass()
```

- 소유 해제 : 옵셔널로 선언

```
ptr = nil
```

- 객체를 소유하는 포인터 - 강한 참조
- 소유권 해제가 객체의 해제는 아니다.

```
print("객체 생성 - 소유")
```

```
var obj : MyClass! = MyClass()
```

```
print("다른 포인터로 소유하기")
```

```
var anotherPointer = obj
```

```
print("소유권 해제")
```

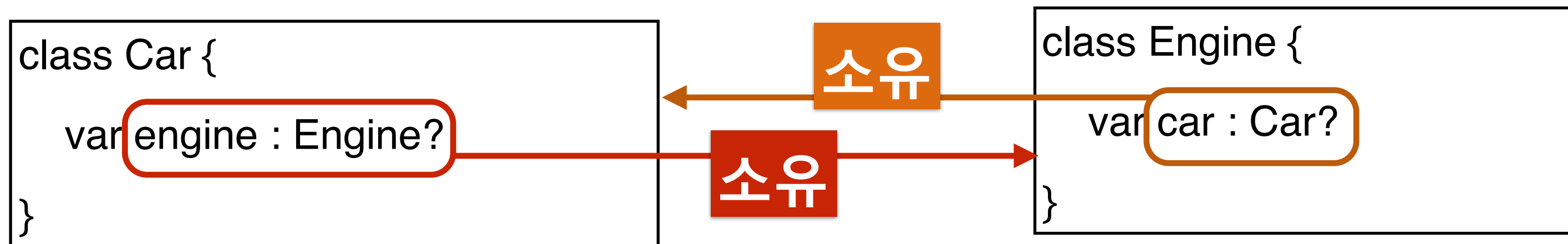
```
obj = nil
```

- 지역 변수
- 소유권 해제

```
class MyApplication {  
    func sayHello() {  
        var obj : MyClass!  
        obj = MyClass()  
        // other job...  
        print("Hello World!")  
    }  
}
```

## 강한 참조의 문제

- 두 클래스에서 상호 참조 ( 두 개 이상의 관계에서도 가능 )
- 서로 소유하므로 해제되지 않음 ( 메모리 누수 )
- 수동으로 해제되도록 작성해야 함



- weak
  - 참조하던 객체가 해제되면 자동 nil
  - nil이 되므로 옵셔널
- 권장 사용
  - 상호 독립적으로 사용 가능
  - 사용자와 스마트폰
  - 운전자와 자동차

- 클래스 선언

```
class Person {  
    var phone : Phone!  
    deinit { print("Person 객체 해제") }  
}  
  
class Phone {  
    weak var owner : Person!  
    deinit { print("Phone 객체 해제") }  
}
```

- 사용

```
var owner : Person! = Person()  
var iphone : Phone! = Phone()  
iphone.owner = owner  
owner.phone = iphone  
// 그리고 nil 대입하면?
```

- 컬렉션에 객체 저장
  - 컬렉션이 객체 소유
- 컬렉션에서 객체 삭제
  - 소유권 해제
- 컬렉션 객체 해제
  - 소유권 해제



상속

- 클래스 상속

```
class Rectangle {  
    var width = 0, height = 0  
    func size() -> Int { return width * height }  
}  
class Square : Rectangle { }
```

- 부모 클래스의 프로퍼티, 메소드 사용

```
var square = Square()  
square.width = 80  
square.height = 80  
square.size()
```

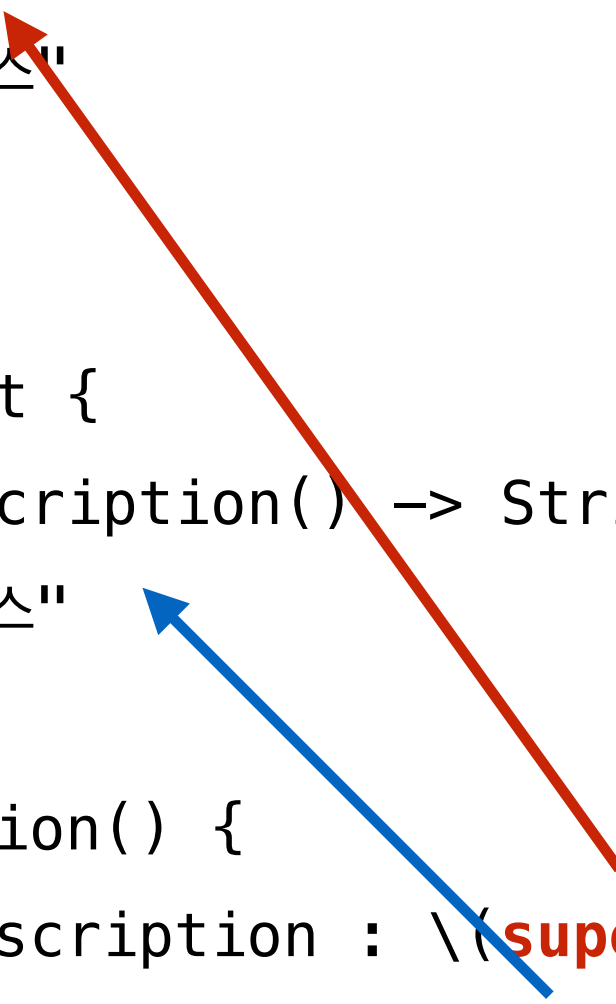
- 메소드 재정의

```
class Rectangle {  
    var width = 0  
    var height = 0  
    func size() -> Int {  
        return width * height  
    }  
}  
  
class Square : Rectangle {  
    override func size() -> Int {  
        return width * width  
    }  
}
```

- super : 부모 클래스를 참조하는 포인터
- 재사용사 부모 클래스를 참조하려면? - super사용
- 초기화 메소드에서도 super 사용

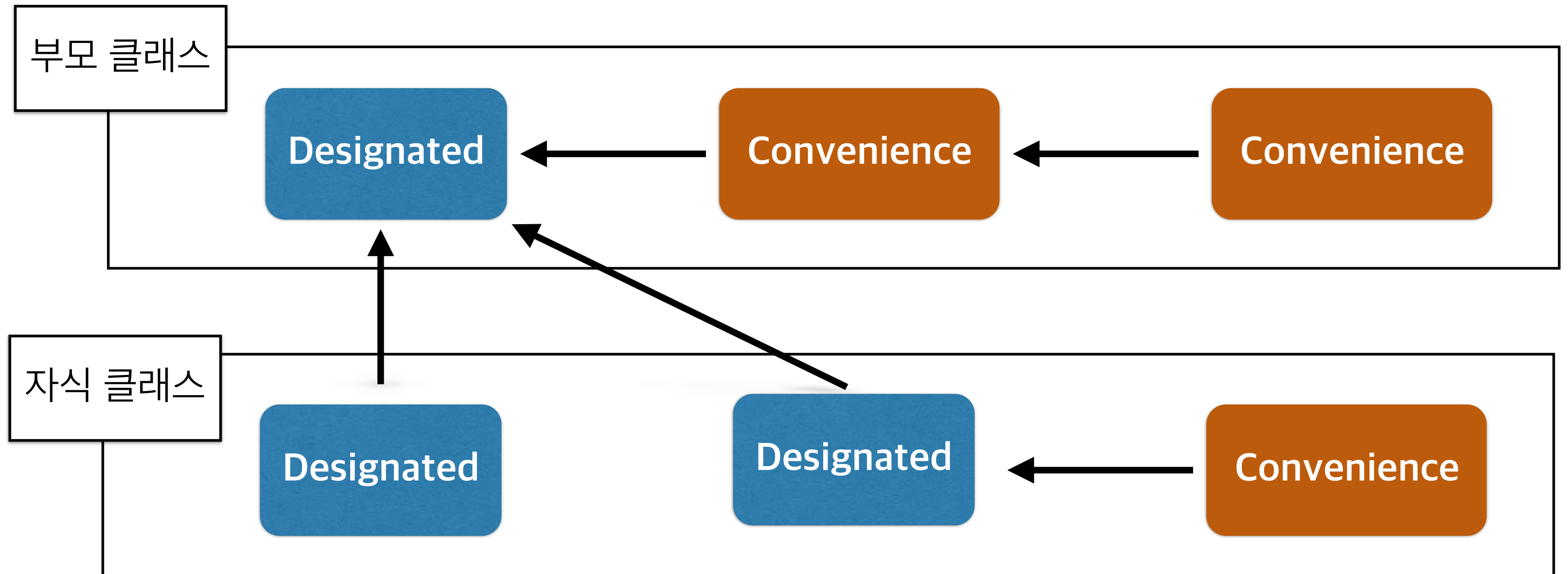
## 재사용과 super

```
class Parent {  
    func description() -> String {  
        return "부모 클래스"  
    }  
}  
  
class Child : Parent {  
    override func description() -> String {  
        return "자식 클래스"  
    }  
}  
  
func printDescription() {  
    print("super.description : \(super.description())")  
    print("self.description : \(self.description())")  
}  
}
```



# 상속과 초기화 메소드

# 초기화 메소드의 델리게이션



- 자식 클래스의 Designated Initializer

```
class Parent {  
    var a : Int  
    init(a : Int) { self.a = 0 }  
}  
  
class Child : Parent {  
    var b : Int  
    init(a : Int, b : Int) {  
        self.b = b  
        super.init(a: a)  
    }  
}
```

- 객체 생성

```
var obj = Child(a: 10, b: 20)  
var obj2 = Child(a : 10)
```



# 자식 클래스의 Convenience Initializer

- 자식 클래스의 Convenience Initializer

```
class Child : Parent {  
    var b : Int  
    init(a : Int, b : Int) {  
        self.b = b  
        super.init(a: a)  
    }  
    convenience init(b : Int) {  
        self.init(a: 10, b : b)  
        // 나머지 초기화 동작  
    }  
}
```

같은 클래스의 Designated Initializer로 위임

- 객체 생성

```
var obj = Child(a: 10, b: 20)  
var obj2 = Child(b: 30)
```

- 초기화 메소드 재정의

```
class Parent {  
    var a : Int  
    init(a : Int) {  
        self.a = a  
    }  
}  
  
class Child : Parent {  
    var b = 0  
    override init(b : Int) {  
        self.b = b  
        self.init(a : 0)  
    }  
}
```

- Required Initializer
- 반드시 작성(재정의)해야 하는 초기화 메소드

```
required init() {  
}
```

# 구조체, Enum

- 구조체 프로퍼티, 함수 정의

```
struct Point {  
    var x = 0  
    var y = 0  
  
    func description() -> String {  
        return "Point : \(x), \(y)"  
    }  
}
```

- 구조체 객체 생성

```
var p1 = Point()  
var p2 = Point(x: 3, y: 5)
```

- Enum 정의

```
enum Day {  
    case AM  
    case PM  
}
```

- Enum 사용

```
var now : Day  
now = Day.AM  
now = Day.PM  
now = Day.Morning // Error
```

- Enum 타입
  - 값 설정 생략 가능. 0부터 시작

```
enum Pet : Int {  
    case Cat = 0  
    case Dog  
    case Other  
}
```

- 값 설정 생략 불가

```
enum Device : String {  
    case Phone = "휴대폰"  
    case Pad = "패드"  
}
```

- 타입이 있는 Enum
  - case에 할당된 값 : `rawValue`
  - `rawValue`에서 Enum 생성 ( 옵셔널 )  
`var ael = Pet(rawValue: 0)`
  - Enum에서 `rawValue` 얻기  
`ael.rawValue`



# 프로토콜

- 클래스 : 인터페이스(interface) + 구현(implementation)
- 프로토콜 : 인터페이스 Only
  - 구현이 없어서 단독 사용 불가
  - 클래스, 구조체와 함께 사용
  - 메소드, 프로퍼티 구현 약속

- 프로토콜 정의

```
protocol Singing {  
    func sing()  
}
```

- 클래스 채택 - **override** 아님

```
class Bird : Singing {  
    func sing() {  
        print("짹짹")  
    }  
}
```

- 객체 생성, 메소드 호출

```
var sparrow = Bird()  
sparrow.sing()
```

- 다중 프로토콜 채택

```
protocol Dancing {  
    func dance()  
}  
class Human : Dancing, Singing {  
    func sing() {  
        print("랄라라~")  
    }  
    func dance() {  
        print("춤추기")  
    }  
}
```

- 프로토콜을 타입으로 사용
  - 프로토콜에 작성한 메소드, 프로퍼티만 사용

```
var singingAnimal : Singing = Human()  
singingAnimal.sing()  
singingAnimal.stepping()
```

- 타입 오류

```
var dancingBird : Dancing = Bird()
```

- 다수의 프로토콜(Protocol Composition)

PROTOCOL1 & PROTOCOL2

- 예

```
func entertain(who : Singing & Dancing) {  
}
```

- 프로토콜 간 상속

```
protocol Entertaining : Singing, Dancing {  
}
```

# 클로저



- 함수 정의

```
func sayHello() {  
    print("Hello")  
}
```

- 함수 호출

```
sayHello()
```

- 함수 참조

```
var hello = sayHello  
hello()
```

- 함수 참조 - 파라미터로 사용하기

```
var hello = sayHello  
greet(hello)
```

- 함수 파라미터 정의?

```
func greet(arg : ????) {  
}
```

- `() -> ()`

```
func sayHello() {}
```

- `(String) -> ()`

```
func sayByeBye(who : String) { }
```

- `(String, String) -> ()`

```
func say(who : String, what : String) { }
```

- `(Int, Int) -> Int`

```
func add(i : Int, j : Int) -> Int {  
    return i + j  
}
```

- 같은 함수 타입 :  $(\text{Int}, \text{Int}) \rightarrow \text{Int}$

```
func add(i : Int, j : Int) -> Int {  
    return i + j
```

```
}
```

```
func multiply(i : Int, j : Int) -> Int {  
    return i * j
```

```
}
```

- 클로저를 사용하는 API
- Array의 sort 메소드

`func sorted(by order: (Element, Element) -> Bool) -> [Element]`

- 파라미터 타입
  - Array<Int>의 경우 : (Int, Int) -> Bool

- 함수 타입에 맞는 함수 작성 : `(Int, Int) -> Bool`

```
func sortFunc(a : Int, b : Int) -> Bool {  
    return a < b  
}
```

- `sorted(by:)`의 파라미터로 입력

```
let sorted = array.sorted(by: sortFunc)
```

- 함수 정의 후 사용하기
- 클로저 표현식(Inline 방식)

```
{ ( PARAMETERS ) -> RETURN_TYPE in  
    // 코드  
}
```

- 예

```
array.sorted(by: { (a:Int, b:Int) -> Bool in  
    return a < b  
})
```

- 타입 선언 생략

```
array.sorted(by: { a, b -> Bool in return a < b } )
```

- 1줄 return 인 경우 -> return 생략

```
array.sorted(by: { a, b -> Bool in a < b } )
```

- 반환 타입 선언 생략

```
array.sorted(by: { a, b in a < b })
```

- 파라미터 선언 생략

```
array.sorted(by : { $0 < $1 })
```



- 반환값으로 정의하기

```
func greeting() -> (() -> ()) {  
    return { () -> () in  
        print("How are you?")  
    }  
}
```

- 축약하기

```
func greeting() -> () -> () {  
    return {  
        print("How are you?")  
    }  
}
```

- 호출하기

```
greeting()()
```

- 파라미터에 사용하기

```
func add(i : Int, _ j : Int, _ handler: (Int) -> Void ) {  
    let sum = i + j  
    // 파라미터로 전달된 클로저 실행.  
    handler(sum)  
}
```

- 사용하기

```
add(3, 4, { (result : Int) -> Void in  
    print("3 + 4 = \(result)")  
})
```

# 에러 처리

- 에러 발생 안하는 함수

```
func cannotThrowErrors() {}
```

- 에러 발생 가능한 함수

```
func canThrowError() throws {}
```

- 에러가 발생할 수 있는(throws) 함수 호출

```
try canThrowError()
```

- 발생한 에러 정보 얻기

```
do {  
    try canThrowError()  
} catch let error {  
    print("에러 : \(error)")  
}
```

- 반환값이 있는 경우
- try?
  - 에러 발생 - nil 반환

```
let result = try? dangerousFunction()
```
- try!
  - 에러 발생 - 크래쉬

```
let result = try! dangerousFunction()
```

**extension**

- 클래스 정의

```
class Dog {  
    func eat() { print("사료 먹기") }  
}
```

- 클래스 확장

```
extension Dog {  
    func bike() { print("멍멍멍") }  
}
```

- 사용 코드

```
var myDog = Dog()  
myDog.eat()  
myDog.bike()
```



# 타입 체킹, 변환

- 타입 체크 : is
- 타입 변환 : as

- 타입 체크 : is

```
let raz = Cat("라즈")  
raz is Cat           // true  
raz is Animal        // true  
raz is Dog            // false
```

- 프로토콜 채택 체크 : is

- 타입 변환 - as를 사용
  - as
  - as? : 타입 변환 불가시 nil. 옵셔널
  - as! : 강제. 에러 발생 가능

- 타입 변환과 바인딩

```
if let dog = raz as? Fish {  
    print("물고기")  
}  
else {  
    print("물고기 아님")  
}
```

- Any : 모든 타입

```
var anyVar : Any = 2  
anyVar = "b"  
anyVar = MyClass()  
var anyArray : [Any] = [1, "2", 3.0]
```

- AnyObject : 모든 레퍼런스 타입

- 타입에 대한 정보가 없음
  - 타입별 메소드, 프로퍼티 사용 불가
  - 타입 체크, 타입 캐스팅 필요

# 파운데이션 프레임워크



- 파운데이션 프레임워크

`import Foundation`

- iOS, OS X용 애플리케이션 개발의 기본 제공

- 데이터 다루기
- 파일 시스템 다루기
- 네트워크 다루기
- 등등등

# NSObject

- NSObject
- 파운데이션 프레임워크에서 최상위 부모 클래스
  - 메모리관리
  - 객체 비교
  - 객체 서술자
  - 셀렉터
  - Objective C 런타임과 동작

- NSObject의 자식 클래스

- 객체 비교 메소드 작성

```
class Rectangle : NSObject {  
    var width = 0;  
    var height = 0  
    override func isEqual(object: AnyObject?) -> Bool {  
        // 타입 비교  
        return false  
    }  
}
```

# 셀렉터

- 셀렉터 : 메소드 식별 정보
  - 사용자 이벤트에 반응하는 메소드
  - 타이머의 시간 간격에 호출하는 메소드
  - 알림이 발생하면 동작하는 메소드
  - 자주 사용

```
func addTarget(_ target: Any?,  
               action: Selector,  
               forControlEvents: UIControlEvents)
```

- 셀렉터
  - 메소드 식별 정보 : 메소드 이름, 파라미터 정보
  - #selector(TYPE.METHOD\_NAME)
  - 컴파일 단계에서 체크

- 파라미터가 없는 메소드

```
class MyClass : NSObject {  
    func greeting() {  
        print("Hello")  
    }  
}
```

- 메소드 식별 문자열

`greeting`

- 셀렉터

`#selector(MyClass.greeting)`



- 같은 타입 내 셀렉터.
- 타입 이름 선언 생략 가능

```
class MyClass : NSObject {  
    func greeting() {  
        print("Hello")  
    }  
  
    func justDoIt() {  
        // 같은 타입 내에서 셀렉터 접근  
        let sel = #selector(greeting)  
    }  
}
```

- 파라미터가 1개인 메소드

```
extension MyClass {  
    func greeting(person : String) {}  
}
```

- 셀렉터

```
#selector(MyClass.greeting(person:))
```

- 외부 파라미터 이름

```
extension MyClass {  
    func greeting(who person : String) {}  
}
```

- 셀렉터

```
#selector(MyClass.greeting(who:))
```

- 외부 파라미터 이름 생략

```
extension MyClass {  
    func greeting(_ person : String) {}  
}
```

- 셀렉터

```
#selector(MyClass.greeting(_:))
```

# 프로토콜

- Objective C 런타임에서 동작하는 프로토콜
  - NSObjectProtocol 상속
  - @objc 키워드로 프로토콜 선언
- 선택 항목(optional) 작성 가능

- 프로토콜 선언

```
@objc protocol Baking {  
    func baking()  
    optional func makeDough()  
}
```

- 클래스 작성, 구현, 예러내보기

```
class Bakery : NSObject, Baking {  
    // 해보기!  
}
```

# 날짜와 시간 다루기



- 날짜와 시간 다루기 : Date, DateFormatter
- 캘렌더 : Calendar
- 데이트 컴포넌트 : DateComponent

- 날짜 : Date
- 시간 간격(time interval)을 이용한 시간 데이터
- 기준시 : 1970년 1월 1일, 0시

- 날짜 객체 생성

```
init(timeIntervalSinceReferenceDate ti: TimeInterval)
```

```
init(timeIntervalSinceNow: TimeInterval)
```

```
func addingTimeInterval(_ timeInterval: TimeInterval) -> Date
```

- 날짜 객체 생성

```
let now = Date()
```

```
let yesterday = Date(timeIntervalSinceNow: (-60*60*24))
```

```
let tomorrow = now.addingTimeInterval(60 * 60 * 24)
```

- DateFormatter : 포맷에 맞는 날짜 표시

- 날짜와 시간 포맷

```
var dateStyle: DateFormatter.Style
```

```
var timeStyle: DateFormatter.Style
```

- 커스텀 날짜 포맷

```
var dateFormat: String!
```

- 연/월/일 : y, M, d

- 시/분/초 : h, m, s

- 날짜 데이터와 문자열 변환

```
func string(from date: Date) -> String
```

```
func date(from string: String) -> Date?
```

- Date, DateFormatter 예제

```
let formatter = DateFormatter()
```

```
formatter.timeStyle = .short
```

```
let timeStr = formatter.string(from:now)
```

```
let customFormatter = DateFormatter()
```

```
customFormatter.dateFormat = "yyyy/MM/dd"
```

```
let dateStr = customFormatter.string(from:tomorrow)
```

```
let date = customFormatter.date(from:"2016/12/25")!
```

- 캘린더 : Calendar

```
let calendar = Calendar.current
```

- 날짜 컴포넌트 : DateComponent

- 날짜에서 달력 시스템에 기반한 정보 얻기

```
func component(_ component: Calendar.Component, from date: Date) -> Int  
func dateComponents(_ components: Set<Calendar.Component>,  
                    from start: DateComponents,  
                    to end: DateComponents) -> DateComponents
```

- 필요한 항목 : Calendar.Component
  - year, month, day
  - weekday
  - weekOfMonth, weekOfYear



- 오늘 날짜 정보(연 기준 주, 월 기준 주)

```
let calendar = Calendar.current
```

```
// 연과 월 기준 주차
```

```
let weekOfYear = calendar.component(.weekOfYear, from: now)
```

```
let weekOfMonth = calendar.component(.weekOfMonth, from: now)
```

- 요일

```
let weekday = calendar.component(.weekday, from: now)
```

```
let monthStr = calendar.standaloneMonthSymbols[month-1]
```

```
// 요일은 1부터 시작
```

```
let weekdayStr = calendar.standaloneWeekdaySymbols[(weekday-1)]
```

- 날짜를 구성하는 복합 데이터 : DateComponent

```
var era: Int
```

```
var year, month, day : Int    // 년, 월, 일
```

```
var hour, minute, second : Int // 시, 분, 초
```

```
var yearForWeekOfYear: Int
```

```
var weekOfYear: Int
```

```
var weekOfMonth: Int
```

- NSDate -> NSDateComponent

```
func dateComponents(_ components: Set<Calendar.Component>, from date: Date) -> DateComponents
```

- NSDateComponent -> NSDate

```
func date(from components: DateComponents) -> Date?
```

- 캘린더와 날짜 컴포넌트

```
let component = DateComponents()  
  
component.year = 2016           // 2016년  
component.weekOfYear = 30       // 30번째 주  
component.weekday = 1           // 일요일  
  
if let date = calendar.date(from: component) {  
    // 데이터 포매터  
    let formatter = DateFormatter()  
    formatter.dateFormat = "yyyy/MM/dd"  
  
    formatter.string(from:date)   // 2016/07/17  
}
```

# 파일 다루기

- FileManager : 파일 시스템 다루기
  - 폴더 내 파일 목록
  - 파일 존재 확인
  - 파일 복사/이동/삭제
  - 등등

```
let fm = FileManager.default
```

- 폴더 내 파일 목록

```
func contentsOfDirectory(atPath path: String) throws -> [String]
```

- 특정 폴더의 파일 목록 출력

```
let fm = FileManager.default
```

```
let contentsOfDir = try? fm.contentsOfDirectory(atPath: dirPath)
```

- 파일 존재 확인

```
func fileExists(atPath path: String) -> Bool
```

- 복사/이동/삭제

```
func copyItem(atPath srcPath: String, toPath dstPath: String) throws
```

```
func moveItem(atPath srcPath: String, toPath dstPath: String) throws
```

```
func removeItem(atPath path: String) throws
```



# 직렬화

- 직렬화 : NSKeyedArchiver

- Any -> Data, File로 저장

```
class func archivedData(withRootObject rootObject: Any) -> Data
class func archiveRootObject(_ rootObject: Any,
                             toFile path: String) -> Bool
```

- 역직렬화 : NSKeyedUnarchiver

- Data, File -> Any

```
class func unarchiveObject(withData data: Data) -> Any?
class func unarchiveObject(withFile path: String) -> Any?
```

- 직렬화

```
let data = NSKeyedArchiver.archivedData(withRootObject: str)
```

- 읽어오기

```
let data = NSKeyedUnarchiver.unarchiveObject(withData: data)
```

```
let str = data as? String
```

- 다수의 데이터 직렬화

- 키-밸류 방식
- NSMutableData 사용
- 인코딩

```
func encode(_ boolv: Bool, forKey key: String)
```

```
func encode(_ intv: Int, forKey key: String)
```

```
func encode(_ objv: Any?, forKey key: String)
```

- 디코딩

```
func decodeBool(forKey key: String) -> Bool
```

```
func decodeInteger(forKey key: String) -> Int
```

```
func decodeObject(forKey key: String) -> Any?
```

# 키-밸류 방식으로 직렬화

```
// 바이너리 데이터를 저장할 NSData 객체
var mdata = NSMutableData()

// 데이터를 인코딩하는 아카이버
var archiver = NSKeyedArchiver(forWritingWith: mdata)

archiver.encode(true, forKey: "BoolData")
archiver.encode(77, forKey: "IntData")
archiver.encode("StringValue", forKey: "StrData")
archiver.finishEncoding()

// NSData에서 복원
var unarchiver = NSKeyedUnarchiver(forReadingWith: mdata as Data)
let boolData = unarchiver.decodeBool(forKey: "BoolData")
let intData = unarchiver.decodeInteger(forKey: "IntData")
let strData = unarchiver.decodeObject(forKey: "StrData") as! String
```

- 커스텀 타입 직렬화 : NSCoder 프로토콜
  - 인코딩/디코딩 함수 정의

```
func encode(withCoder aCoder: NSCoder)
init?(coder aDecoder: NSCoder)
```
- 클래스만 가능. 구조체 불가

## 커스텀 클래스의 직렬화

```
class Person : NSObject, NSCoding {  
    var name : String  
    var birthYear : Int  
  
    func encode(with aCoder: NSCoder) {  
        aCoder.encode(name, forKey: "Name")  
        aCoder.encode(birthYear, forKey: "Year")  
    }  
  
    required init(coder aDecoder: NSCoder) {  
        name = aDecoder.decodeObject(forKey: "Name") as! String  
        birthYear = aDecoder.decodeInteger(forKey: "Year")  
    }  
}
```

- 커스텀 클래스의 객체 직렬화

```
var obj = Person(name: "태연", birthYear: 1989)
```

```
let filePath = "파일 경로"
```

```
let ret = NSKeyedArchiver.archiveRootObject(obj, toFile: filePath)
```

```
// 복원
```

```
let obj2 = NSKeyedUnarchiver.unarchiveObject(withFile:filePath) as! Person
```

```
print("name : \(obj2.name) - birthYear : \(obj2.birthYear)")
```



# 타이머

- 타이머 : Timer

- 타이머 시작

```
class func scheduledTimer(timeInterval ti: TimeInterval,  
                           target aTarget: Any,  
                           selector aSelector: Selector,  
                           userInfo: Any?,  
                           repeats yesOrNo: Bool) -> Timer
```

// 클로저를 이용한 타이머

```
class func scheduledTimer(withTimeInterval interval: TimeInterval,  
                           repeats: Bool,  
                           block: @escaping (Timer) -> Swift.Void) -> Timer
```

- 타이머 중지

```
func invalidate()
```

- 타이머 예제

```
class Alarm : NSObject {  
    func ring(timer : Timer) {  
        print("Wake UP!")  
    }  
}
```

```
let obj = Alarm()  
var timer = Timer.scheduledTimer(timeInterval: 0.5,  
                                  target: obj,  
                                  selector: #selector(Alarm.ring(timer:)),  
                                  userInfo: nil, repeats: true)
```

알림

- 알림 : Notification
- 알림 센터(NotificationCenter)를 이용, 알림 발송(notifier)
- 알림 센터에 알림 청취(receive) - 알림 감시자 등록

- 알림 센터

```
let notiCenter = NotificationCenter.default
```

- 알림 발송

```
func post(_ notification: Notification)
```

```
func post(name aName: NSNotification.Name, object anObject: Any?)
```

- 알림 청취 : 옵저버 등록/해제

```
func addObserver(_ selector:name:object)
```

```
func removeObserver(_ observer: Any)
```

- 알림정의

```
let NotiName = NSNotification.Name("CustomNotification")
```

- 알림 발생 감시

```
let obj = MyClass()
```

```
notiCenter.addObserver(obj,
```

```
    selector: #selector(MyClass.handleNoti(noti:)),
```

```
    name:NotiName, object: nil)
```

- 알림 발생

```
notiCenter.post(name: NotiName, object: nil)
```

# JSON 다루기



- JSON 다루기 : JSONSerialization

- JSON 파싱

```
class func jsonObject(with data: Data,  
                      options opt: JSONSerialization.ReadingOptions = []) throws -> Any
```

- JSON 생성

```
class func data(withJSONObject obj: Any,  
               options opt: JSONSerialization.WritingOptions = []) throws -> Data
```

- JSON 파싱
- JSON의 노드의 타입 정보 필요
- 타입 체크와 타입 변환, 바인딩 필요

- JSON 파싱

```
let result = try JSONSerialization.jsonObject(with: data, options: [])
if let root = result as? [String: Any],
    let devices = root["device"] as? [Any],
    let iPhone7 = devices[0] as? [String:Any] {
    print("iPhone7 : ", iPhone7)
    if let spec = iPhone7["spec"] as? [String:Any],
        let cpu = spec["cpu"] as? String {
        print("iPhone7's cpu is", cpu)
    }
}
```