

Object Oriented Programming

- ❖ Object-Oriented Programming(OOP, 객체 지향 프로그래밍)
 - ✓ 컴퓨터 프로그래밍의 패러다임 중 하나
 - ✓ 객체 지향 프로그래밍은 컴퓨터 프로그램을 명령어의 목록으로 보는 시각에서 벗어나 여러 개의 독립된 단위, 즉 "객체"들의 모임으로 파악하고자 하는 것이며 각각의 객체는 메시지를 주고받고, 데이터를 처리할 수 있음
 - ✓ 객체 지향 프로그래밍은 프로그램을 유연하고 변경이 용이하게 만들기 때문에 대규모 소프트웨어 개발에 많이 사용
 - ✓ 또한 프로그래밍을 더 배우기 쉽게 하고 소프트웨어 개발과 보수를 간편하게 하며, 보다 직관적인 코드 분석을 가능하게 하는 장점이 있음
 - ✓ 지나친 프로그램의 객체화 경향은 실제 세계의 모습을 그대로 반영하지 못한다는 비판을 받기도 함
 - ✓ 3대 특징
 - ❑ Encapsulation(캡슐화): 객체의 속성(data fields)과 행위(메서드, methods)를 하나로 묶고 실제 구현 내용 일부를 외부에 감추어 은닉하는 것
 - ❑ Inheritance(상속성): 하위 클래스가 상위 클래스의 모든 것을 물려받는 것
 - ❑ Polymorphism(다형성): 동일한 메시지에 대하여 다르게 반응하는 성질
 - ✓ 용어
 - ❑ Property: 속성(Attribute), 변수(Variable), Field, Data
 - ❑ Method: Function, Operation, Behavior

클래스

Class

❖ Class

- ✓ 프로그램 단위의 종류로 아파트의 모델 하우스처럼 동일한 목적으로 사용되는 객체들의 모형으로 사용자가 만드는 자료형
- ✓ 동일한 목적을 달성하기 위해 사용되는 변수와 함수를 묶어서 관리하기 위한 목적으로 생성
- ✓ Kotlin 에서는 변수와 함수로만 프로그래밍할 수 있지만 객체 지향 프로그래밍을 지원
- ✓ 객체지향 프로그래밍에서는 클래스를 선언하고 클래스 내에 여러 구성 요소를 담은 후 객체를 생성해서 사용
- ✓ 클래스의 구성 요소는 constructor, property, method
- ✓ 클래스는 Class header 와 Class body로 구성
 - Class header는 몸체를 나타내는 { } 앞 부분으로 클래스 이름과 타입 파라미터(상위 클래스, 구현하는 인터페이스)를 정의하는 부분
 - Class body는 { } 안의 내용으로 프로퍼티와 메소드 등 클래스가 포함하는 구성 요소를 정의하는 부분으로 body가 없다면 { }는 생략 가능
 - 클래스 내부에서 property 나 method는 이름 만으로 호출 가능
 - property 와 method는 외부에서는 instance를 이용해서 호출
- ✓ 코틀린 소스는 확장자가 kt인 파일로 만들어지는데 그 파일 안에 클래스를 다양한 형태로 정의해서 사용
- ✓ 일반적으로는 파일 안에 파일 이름과 같은 이름의 클래스를 정의해서 사용
- ✓ 하나의 파일에 클래스가 없을 수도 있고 1개 이상의 클래스를 선언해서 사용하는 것이 가능하며 각 클래스는 번역이 될 때 별도의 .class 파일을 생성

Class

❖ Class

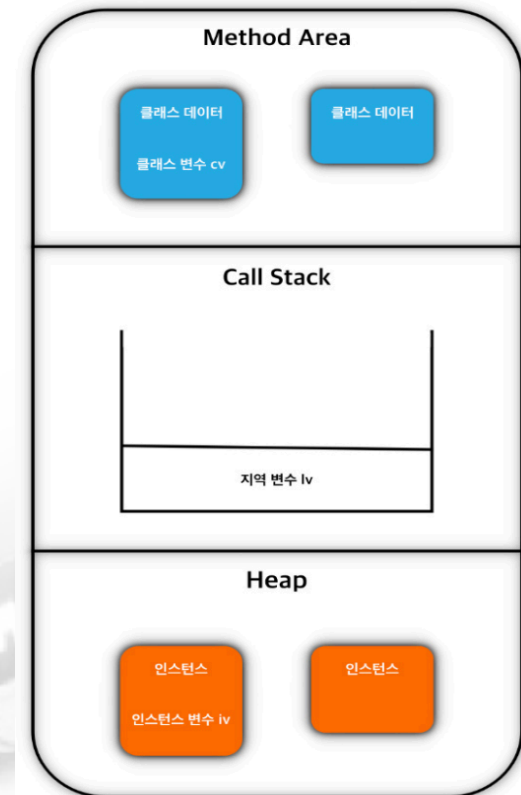
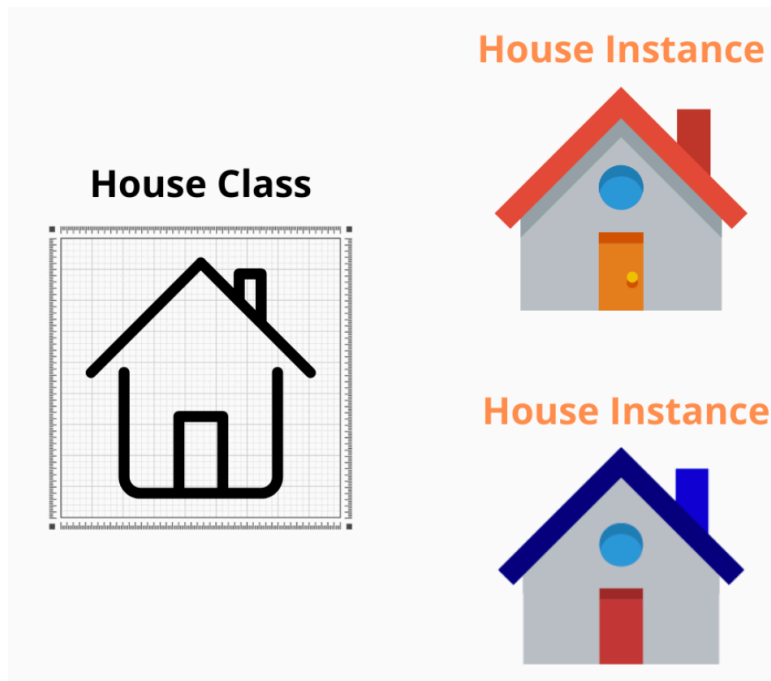


클래스 다이어그램 표기법

Class

❖ Instance

- ✓ 클래스를 이용해서 메모리를 할당받은 객체
- ✓ 생성자를 호출해서 생성 – 생성자는 클래스 이름과 동일한 이름의 특별한 메소드
- ✓ 프로퍼티 나 메소드를 .을 이용해서 호출



실습 – 클래스 선언과 인스턴스 생성

❖ MyClass 생성 – MyClass.kt

```
class MyClass {  
    val myVariable=10  
    fun myFunction(){  
        println("메소드")  
    }  
}
```

❖ 실행 파일 – main1.kt

```
fun main(args: Array<String>) {  
    val obj1 = MyClass()  
    println(obj1)  
    println(obj1.myVariable)  
    obj1.myFunction()  
  
    val obj2 = MyClass()  
    println(obj2)  
}
```

Constructor(생성자)

❖ Constructor(생성자)

- ✓ Instance를 생성할 때 호출하는 특별한 함수로 Heap 영역에 메모리 할당을 하고 그 참조를 리턴하는 함수
- ✓ 생성자를 호출하면 자신의 프로퍼티를 저장할 수 있는 공간과 자신의 클래스 참조 만을 가지고 생성
- ✓ 호출은 함수처럼 하는데 Kotlin에서는 new를 붙이지 않고 호출
- ✓ 생성자는 클래스 이름과 동일
- ✓ 코틀린에서는 주 생성자(Primary Constructor)와 보조 생성자로 구분
- ✓ 클래스 안에 별도의 생성자를 만들지 않으면 매개변수가 없는 주 생성자가 자동으로 생성됨

Constructor(생성자)

❖ Primary Constructor: 메모리 할당

- ✓ 클래스 선언 부분에 작성
- ✓ 하나의 클래스에 하나만 선언 가능
- ✓ 반드시 만들 필요는 없음
- ✓ 매개변수를 갖는 형태로 만들 수 있으며 매개변수에 기본값을 설정할 수 있음
- ✓ Primary Constructor는 메모리 할당 만 수행하기 때문에 실행 코드 영역을 가질 수 없음
- ✓ 메모리 할당이 끝나고 프로퍼티를 초기화 하고자 할 때는 프로퍼티를 생성할 때 매개변수로 받은 데이터를 대입하는 것이 가능
- ✓ 실행하는 코드를 작성하고자 하는 경우에는 클래스 몸체 안에 `init { }` 을 만들어서 작성하는데 `init`에서 변수를 사용하면 주 생성자의 매개변수 부터 찾아옴
- ✓ Primary Constructor를 만들면 기본적으로 제공되는 생성자는 만들어지지 않음
- ✓ Primary Constructor를 만들 때 매개변수 앞에 `val` 이나 `var`를 붙이면 클래스의 프로퍼티로 생성됨
- ✓ `constructor` 라는 예약어를 사용해도 되고 생략하는 것도 가능

실습 – Primary Constructor

❖ 클래스 생성을 위한 파일 – User.kt

```
class User1{} //기본 주 생성자 제공
```

```
class User2 constructor(){} //매개변수가 없는 주 생성자 작성
```

```
class User3 constructor(num:Int, name:String){} //매개변수가 있는 주 생성자 작성
```

```
//매개변수가 있고 초기화 코드가 있는 주 생성자 작성
```

```
class User4 constructor(num:Int, name:String){
```

```
    val num = num
```

```
    val name = name
```

```
    init{
```

```
        println("초기화 블록")
```

```
    }
```

```
}
```

실습 – Primary Constructor

❖ 클래스 생성을 위한 파일 – User.kt

//매개변수가 있고 초기화 코드가 있는 주 생성자 작성

```
class User5 constructor(num:Int, name:String){  
    val num = num  
    val name = "noname"  
  
    init{  
        println("초기화 블록에서는 생성자의 매개변수를 사용")  
        println("name is $name")  
    }  
}
```

실습 – Primary Constructor

❖ 실행 파일 – main2.kt

```
fun main(args: Array<String>) {  
    val obj1 = User1()  
    val obj2 = User2()  
    val obj3 = User3(1, "itstudy")  
    val obj4 = User4(2, "adam")  
    val obj5 = User5(3, "itggangpae")  
}
```

Constructor(생성자)

❖ Secondary Constructor(보조 생성자)

- ✓ 프로퍼티의 메모리 할당이 끝나고 호출 – primary constructor -> init{} -> secondary
- ✓ constructor 라는 예약어와 함께 클래스 내부에 생성
- ✓ Secondary Constructor는 메모리 할당이 끝나고 호출되기 때문에 매개변수에 val 이나 var를 추가해서 프로퍼티 생성이 불가능
- ✓ Primary Constructor 와 함께 생성이 가능하며 독자적으로 생성하는 것도 가능
- ✓ Overloading 가능 – 하나의 클래스에 매개변수의 개수나 자료형을 다르게 해서 여러 개 생성 가능
- ✓ Primary Constructor 와 Secondary Constructor를 같이 만드는 경우에는 Secondary Constructor에서 반드시 Primary Constructor를 호출해야 하는데 이 때는 매개변수 뒤에 :을 추가하고 this(매개변수)를 이용해서 호출
- ✓ 생성자가 1개인 경우는 Primary Constructor 만 생성해서 사용하는 것이 일반적이며 여러 개의 생성자를 만드는 경우는 Secondary Constructor를 이용

실습 – Secondary Constructor

❖ Person.kt

```
class Person(var name : String){  
    init{  
        println("초기화 블록")  
    }  
  
    constructor(num : Int, name : String):this(name){  
        println("Secondary Constructor: $num ... $name")  
    }  
}  
  
fun main(args: Array<String>) {  
    val person1 = Person("adam")  
    val person2 = Person(1, "itggangpae")  
}
```

Property

❖ Property

- ✓ 클래스 나 Top Level(클래스 외부)에서 val 이나 var를 이용해서 선언하는 변수 와 Primary Constructor에서 val 이나 var을 추가한 매개변수
- ✓ 클래스 내부에서 사용할 때는 프로퍼티 이름으로 사용하거나 this.프로퍼티이름 으로 사용
- ✓ this가 붙으면 함수 내에서 선언된 변수에서는 찾지 않고 클래스에서 부터 찾음
- ✓ 클래스 외부에서 사용할 때는 인스턴스를 가지고 .을 이용해서 사용
- ✓ Property라고 부르는 이유는 함수 내에 선언한 변수와 다르게 getter(값을 리턴하는 함수) 와 setter(값을 설정하는 함수) 같은 접근자 메소드를 내장하고 있기 때문
- ✓ val로 선언하면 getter 만 생성
- ✓ var로 선언하면 getter 와 setter를 생성
- ✓ Top Level에서 const val 로 선언하면 getter 가 생성되는데 수정할 수 없음
- ✓ getter는 값을 리턴하도록 setter는 값을 설정하도록 만들어져 있는데 접근자 메소드를 직접 호출하는 것이 아니고 프로퍼티를 호출하면 =(대입 연산자) 의 존재 여부에 따라 getter 와 setter를 알아서 호출

Property

❖ Property

- ✓ getter 와 setter를 직접 생성하는 것도 가능 – 이름은 get 과 set으로 고정
- ✓ 형식

```
var <propertyName>[:<Property Type>] = <property _initializer>
```

```
[<getter>]
```

```
[<setter>]
```

- var 대신에 val 사용 가능
 - Type은 초기값이 있어서 추론이 가능한 경우 생략 가능
 - 나머지는 전부 생략 가능
- ✓ getter 와 setter 안에는 field 라는 변수가 존재하는데 이 필드가 프로퍼티의 값을 저장하는 변수이며 이를 backing field 라고 함

Property

❖ Property

- ✓ 기본적으로 제공되는 getter 와 setter는 backing field의 값을 리턴하거나 매개변수를 받아서 backing field 의 값을 변경하는 역할만 수행
- ✓ 작업을 수행하고자 하면 get 과 set을 직접 정의하면 됨
- ✓ setter에서 많이 수행하는 작업은 로그 기록이나 값의 변경을 감시해서 특수한 작업(유효성 검사)을 수행
- ✓ getter에서는 로그 기록을 수행하거나 데이터를 변경해서 리턴하는 경우가 많음
- ✓ getter 와 setter를 만들 때 규칙
 - get 과 set 함수 안에서는 field라는 이름을 이용해서 프로퍼티에 접근
 - var로 선언하면 get 과 set 모두 정의할 수 있지만 val로 선언하면 get 만 정의 가능
 - val로 선언한 프로퍼티는 get 함수를 만들면 초기값을 명시하지 않아도 됨
 - var로 선언한 프로퍼티는 get 함수를 만들더라도 초기값을 명시해야 함
 - Top Level 에서는 const val 이 가능한데 이 경우는 get 과 set을 만들 수 없음

실습 – Property

❖ Property1.kt

```
class User {  
    val num: Int = 1  
    get() = field  
  
    var name: String = "cyberadam"  
    get() = field  
    set(value) {field=value}  
}  
  
fun main(args: Array<String>) {  
    val user=User()  
    println("num : ${user.num}")  
    println("name : ${user.name}")  
}
```

실습 – Property

❖ Property2.kt

```
class Student {  
    var num: Int = 1  
        get() = field  
        set(value) {field=value}  
  
    var name: String = "cyberadam"  
        get(){  
            println("name 속성에 접근하고자 함")  
            return field  
        }  
        set(value) {field=value}  
  
    var score: Int = 0  
        get() = field  
        set(value) {  
            if(value >= 0){  
                field = value  
            }else{  
                println("점수는 0보다 작을 수 없음")  
                field = 0  
            }  
        }  
}
```

실습 – Property

❖ Property2.kt

```
fun main(args: Array<String>) {  
    val student = Student()  
    student.num = 10  
    student.name = "cyberadam"  
    println("name : ${student.name}")  
    student.score = -100  
}
```

Property

❖ Property

- ✓ 함수 내에 선언한 변수는 프로퍼티가 아니며 함수 외부에 선언한 변수(최상위 레벨, 클래스 내부에 선언한 변수)만 프로퍼티
- ✓ Primary Constructor에 선언된 매개변수 앞에 val 이나 var를 붙이면 프로퍼티가 되지만 이 프로퍼티 들에는 get 과 set 함수를 작성할 수 없음
- ✓ Primary Constructor에서 만들어진 프로퍼티에 get 과 set을 만들고자 하는 경우에는 val 이나 var를 제거하고 클래스 안에 동일한 이름의 프로퍼티를 생성한 후 생성자에 매개변수로 넘겨받은 데이터를 클래스 안의 프로퍼티에 대입하도록 작성하고 클래스 내부의 프로퍼티에 get 과 set을 만드는 방식을 이용

실습 – Property

❖ Property3.kt

```
class Customer(num:Int, name:String, isFever:Boolean) {  
    var num = num  
    var name = name  
    var isFever = isFever  
    get(){  
        println("발열 여부를 가져옴")  
        return field  
    }  
    set(value){  
        print("발열 여부를 변경하려고 함")  
        field = value  
    }  
}  
  
fun main(args: Array<String>) {  
    var customer = Customer(1, "jessica", false)  
    println("발열 여부 : ${customer.isFever}")  
    customer.isFever = true  
}
```

Property

❖ Property 초기화

- ✓ 선언할 때 초기화 – Primary Constructor에서 기본값 설정 포함
- ✓ 초기화 블록에서 초기화 – init
- ✓ lateinit을 앞에 붙여서 나중에 초기화 할 수 있도록 선언 – 기본 자료형은 안됨
- ✓ 자료형 뒤에 by lazy를 설정해서 나중에 초기화
 - 처음 호출 할 때 초기화
 - val 로 선언한 프로퍼티에만 사용 가능
 - 클래스 내부의 프로퍼티 뿐 아니라 클래스 외부에서 선언한 프로퍼티에도 사용 가능
- ✓ lateinit 이나 by lazy를 사용하는 경우는 필수적으로 사용하지 않는 프로퍼티에 주로 이용

❖ Property 감시

- ✓ 자료형 뒤에 by Delegates.observable을 추가하고 (초기값, {(props, old, new)를 매개변수로 갖는 메소드})를 설정하면 프로퍼티의 값이 변경될 때 마다 함수가 호출됨
- ✓ props는 변경된 프로퍼티 정보가 old는 이전 값이고 new는 새로운 값

실습 – Property

❖ Property4.kt

```
val upperData: String by lazy {  
    println("최상위 레벨의 lazy init")  
    "지연 생성"  
}  
  
class UserInitializer {  
    var num : Int  
  
    val name: String by lazy {  
        println("클래스 내부의 프로퍼티 lazy init")  
        "adam"  
    }  
  
    //지연 초기화  
    lateinit var address:String  
  
    init {  
        println("초기화 블록")  
        num = 1  
    }  
}
```

실습 – Property

❖ Property4.kt

```
fun main(args: Array<String>) {  
  
    val user = UserInitializer()  
    println("name use before...")  
    println("name : ${user.name}")  
    println("name use after....")  
    user.address = "서울시 양천구 목동"  
}
```


실습 – Property

❖ Property5.kt

```
import kotlin.properties.Delegates
```

```
class Observer {  
    var name: String by Delegates.observable("noname", { props, old, new ->  
        println("Properties : $props")  
        println("oldValue : $old ... newValue : $new")  
    })  
}
```

```
fun main(args: Array<String>) {  
    val observer=Observer()  
    println(observer.name)  
    observer.name="adam"  
    observer.name="itggangpae"  
}
```

Method

❖ Method

- ✓ 전달받은 인수를 처리하여 결과를 돌려주는 작은 프로그램
- ✓ 함수(function)라고도 하는데 함수는 전역 공간에 생성되고 메소드는 클래스 안에 만들어진 것(소속이 있음)으로 구분
- ✓ 클래스 외부에서는 인스턴스를 이용해서 호출하고 클래스 내부에서는 메소드 이름만으로 호출 할 수 있고 this.메소드이름 으로 호출 가능
- ✓ 메소드에는 this라는 숨겨진 포인터가 존재하는데 이 this를 이용하면 함수 내에서는 데이터를 찾지 않고 클래스에서부터 찾음

실습 – Method

❖ Method.kt

```
class MethodClass {  
    fun myFunction(){  
        println("메소드")  
    }  
  
    fun innerCall(){  
        //내부에서 이름만으로 호출  
        myFunction()  
        //this를 이용한 호출  
        this.myFunction()  
    }  
}  
  
fun main(args: Array<String>) {  
    val methodClass= MethodClass()  
    //클래스 외부에서는 인스턴스를 이용해서 호출  
    methodClass.myFunction()  
    methodClass.innerCall()  
}
```

Inheritance

상속

❖ 상속

- ✓ 하위 클래스가 상위 클래스의 모든 멤버를 물려 받는 것
- ✓ 상속은 객체지향언어의 주요한 특징 중 하나인 재 사용성(CBD로 많이 이전)과 코드의 간결성(중복된 코드를 하나의 클래스에 표현)을 제공
- ✓ 상속의 장점
 - 코드를 재활용함으로써 간소화된 클래스 구조 이용
 - 클래스의 기능 테스트에 대한 생산성 및 정확성이 증가
 - 클래스의 수정/추가에 대한 유연성 및 확장성이 증가
- ✓ 자신이 만든 클래스로부터 상속(코드의 중복 제거나 다형성 구현이 목적) 받을 수 있고 프레임워크(ex JDK, Spring..)가 제공해주는 클래스로부터 상속(기능 확장이 목적)을 받을 수 있음
- ✓ Kotlin의 모든 클래스는 상속을 받아야 되는데 상위 클래스를 기재하지 않는다면 상위 클래스는 Any
- ✓ Kotlin은 단일 상속만 지원 - 하나의 클래스로부터 상속

❖ Any 클래스

- ✓ Kotlin의 최상위 클래스로 equals() 와 toString() 만을 소유

상속 – Any Class 상속

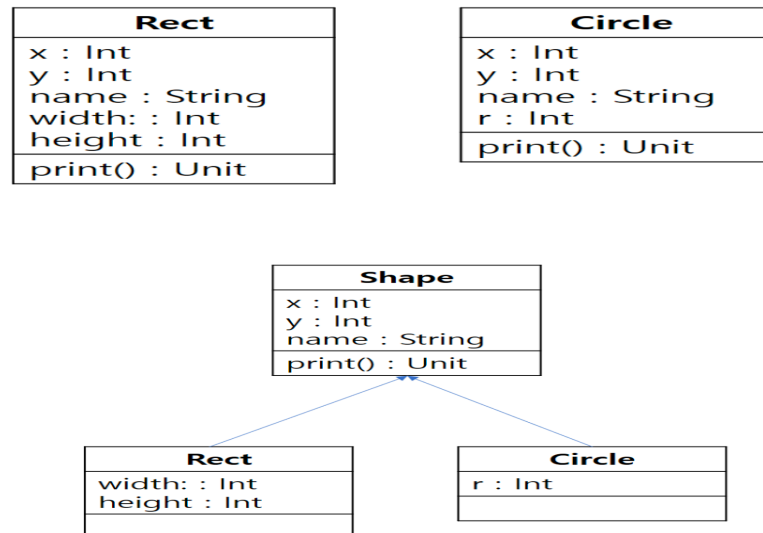
❖ AnyClass.kt

```
class AnyShape {  
    var x: Int = 0  
    var y: Int = 0  
    var name: String = "Rect"  
  
    fun draw() {  
        println("draw $name : location : $x, $y")  
    }  
}  
  
fun main(args: Array<String>) {  
    val obj1: AnyShape = AnyShape()  
    val obj2: AnyShape = AnyShape()  
    val obj3 = obj1  
    //Shape에 정의하지 않은 equals 메소드 호출 - Any 로 부터 상속을 받아서 사용 가능  
    println("obj1.equals(obj2) is ${obj1.equals(obj2)}")  
    println("obj1.equals(obj3) is ${obj1.equals(obj3)}")  
}
```

상속

❖ 상속을 이용한 클래스 정의

- ✓ 여러 클래스를 만들다 보면 클래스들에 공통적인 내용이 존재할 수 있는데 이 경우는 상위 클래스에 공통적인 내용을 정의하고 상속받는 방법을 이용하는 것이 가능 - 코드의 중복 제거
- ✓ 상속을 받을 때는 클래스 이름 뒤에 : **상위클래스이름** 를 추가
- ✓ 상속하는 클래스를 Base, Super 클래스라고 함
- ✓ 상속받는 클래스는 Derivation, Sub 클래스라고 함
- ✓ 상위 클래스를 만들 때는 class 앞에 open 을 추가하고 상속을 받지 않을 때는 final을 추가



상속 - 공통된 부분 추출

❖ ShapeClass.kt

//상속가능한 ShapeSuper 클래스

```
open class ShapeSuper {  
    var x: Int = 0  
        set(value) {  
            if(value < 0) field = 0  
            else field = value  
        }  
  
    var y: Int = 0  
        set(value) {  
            if(value < 0) field = 0  
            else field = value  
        }  
  
    lateinit var name: String  
  
    fun print() {  
        println("$name : location : $x, $y")  
    }  
}
```


상속 - 공통된 부분 추출

❖ ShapeClass.kt

//ShapeSuper 클래스를 상속받는 Rect 클래스

```
class Rect: ShapeSuper() {  
    var width: Int = 0  
        set(value) {  
            if(value < 0) field = 0  
            else field = value  
        }  
    var height: Int = 0  
        set(value) {  
            if(value < 0) field = 0  
            else field = value  
        }  
}
```

//ShapeSuper 클래스를 상속받는 Circle 클래스

```
class Circle: ShapeSuper() {  
    var r: Int = 0  
        set(value) {  
            if(value < 0) field = 0  
            else field = value  
        }  
}
```

상속 - 공통된 부분 추출

❖ ShapeClass.kt

```
fun main(args: Array<String>) {  
    val rect=Rect()  
    rect.name="Rect"  
    rect.x=10  
    rect.y=10  
    rect.width=20  
    rect.height=20  
    rect.print()  
  
    val circle=Circle()  
    circle.name="Circle"  
    circle.x=30  
    circle.y=30  
    circle.r=5  
    circle.print()  
}
```

상속

❖ 상위 클래스의 생성자 호출

- ✓ 하위 클래스에서는 반드시 상위 클래스의 생성자를 호출해야 함
- ✓ 하위 클래스에 별도의 생성자를 정의하지 않고 상위 클래스의 디폴트 생성자를 호출하는 경우에는 **하위 클래스이름 : 상위 클래스이름()**

```
open class Super {  
  
}  
  
class Sub: Super() {  
  
}  
//.....  
val sub=Sub()
```



```
open class Super constructor(){  
  
}  
  
class Sub constructor(): Super() {  
  
}  
//.....  
val sub=Sub()
```

상속

❖ 상위 클래스의 생성자 호출

- ✓ 상위 클래스에 매개변수가 있는 Primary Constructor를 만든 경우
open class Super(name:String){}

class Sub:Super(){} //에러

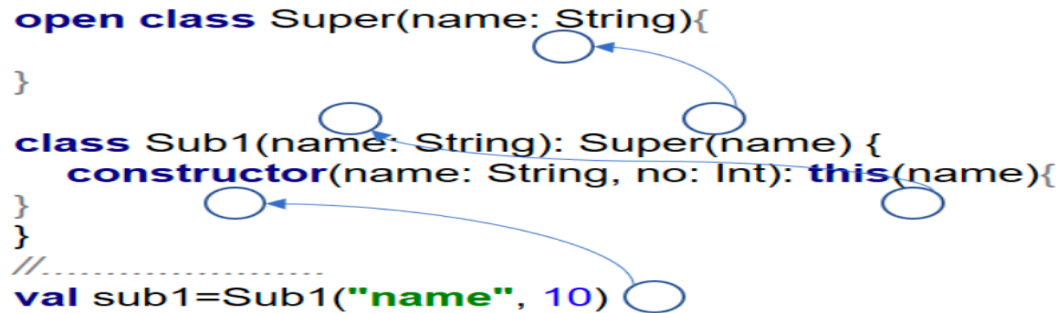
class Sub:Super("cyberadam"){} //에러 아님

상속

❖ 상위 클래스의 생성자 호출

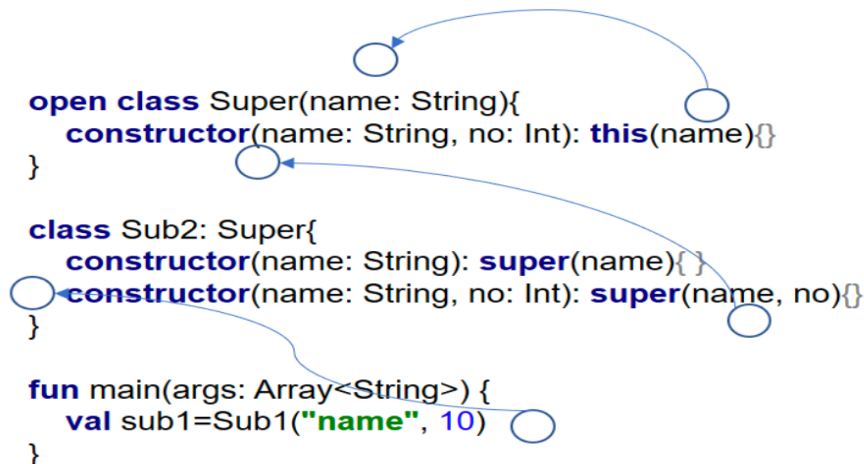
- ✓ 하위 클래스에 주 생성자가 있는 경우

```
open class Super(name: String){  
}  
  
class Sub1(name: String): Super(name) {  
    constructor(name: String, no: Int): this(name){  
    }  
}  
// .....  
val sub1=Sub1("name", 10)
```



- ✓ 하위 클래스에 보조 생성자가 있는 경우 - super를 이용해서 상위 클래스의 생성자를 호출

```
open class Super(name: String){  
    constructor(name: String, no: Int): this(name){  
    }  
  
class Sub2: Super{  
    constructor(name: String): super(name){}  
    constructor(name: String, no: Int): super(name, no){  
    }  
  
fun main(args: Array<String>) {  
    val sub1=Sub1("name", 10)  
}
```



상속 – 상위 클래스의 생성자 호출

❖ SuperConstructor.kt

```
open class SuperCon(name: String){  
    constructor(name: String, no: Int): this(name){}  
}
```

```
class Sub1Con(name: String): SuperCon(name) {  
    constructor(name: String, no: Int): this(name){ }  
}
```

```
class Sub2Con: SuperCon{  
    constructor(name: String): super(name){ }  
    constructor(name: String, no: Int): super(name, no){}  
}
```

```
fun main(args: Array<String>) {  
    val sub1=Sub1Con("name", 10)  
    val sub2=Sub2Con("name", 10)  
}
```

상속

- ❖ 상속 관계에서의 생성자 호출 순서
 - ✓ 상위 클래스의 Primary Constructor
 - ✓ 상위 클래스의 init 블록
 - ✓ 상위 클래스의 Secondary Constructor
 - ✓ 하위 클래스의 Primary Constructor
 - ✓ 하위 클래스의 init 블록
 - ✓ 하위 클래스의 Secondary Constructor

상속 – 생성자 수행 흐름

❖ ConstructorFlow.kt

```
open class SuperFlow {
    constructor(name: String, no: Int){
        println("Super ... constructor(name, no)")
    }
    init {
        println("Super ... init call...")
    }
}
class SubFlow(name: String): SuperFlow(name, 10){
    constructor(name: String, no: Int): this(name){
        println("Sub ... constructor(name, no) call")
    }
    init {
        println("Sub ... init call...")
    }
}

fun main(args: Array<String>) {

    SubFlow("cyberadam")
    println(".....")
    SubFlow("itggangpae",10)
}
```


상속

❖ Method Overriding

- ✓ 하위 클래스에 상위 클래스에 존재하는 메소드와 원형이 동일한 메소드를 다시 정의하는것
- ✓ 오버라이딩 된 메소드는 오브젝트가 메소드를 호출하면 객체의 타입(메모리 할당 시 호출한 생성자)에 따라 메소드를 호출
- ✓ 자신이 만든 클래스에서 오버라이딩을 하는 경우는 대부분 다형성을 구현하기 위해서이고 API가 제공하는 클래스의 메소드를 오버라이딩을 하는 경우는 기능 확장을 위해서(상위 클래스의 메소드를 호출하고 추가할 코드를 작성)
- ✓ 상위 클래스에서 오버라이딩이 가능한 메소드를 만들 때는 메소드 앞에 open을 추가해야 하며 오버라이딩이 불가능하도록 할 때는 final을 추가하며 기본은 final
- ✓ 하위 클래스에 메소드를 재정의할 때는 override를 추가해야 하며 override를 추가하면 자동으로 open이 추가된 것과 동일한 효과

상속

❖ super

- ✓ 하위 클래스의 메소드나 프로퍼티 안에서 상위 클래스의 멤버에 접근할 때 사용하는 숨겨진 포인터가 super
- ✓ super를 이용하면 메소드 내부나 클래스에서 찾지 않고 상위 클래스에서부터 찾음
- ✓ 하위 클래스에서 메소드나 프로퍼티를 오버라이딩 하는 경우 상위 클래스의 멤버에 접근해야 하는 경우가 발생하는데 이 경우에 super를 사용
- ✓ SDK에서 제공하는 클래스를 상속받아서 사용하는 경우 오버라이딩을 하면 이 경우에는 대부분 상위 클래스의 메소드를 호출해야 함
- ✓ 안드로이드에서는 강제하는 경우도 있음

상속 – MethodOverriding

❖ Overriding.kt

```
open class ShapeOverride {  
    open fun disp() {  
        println("최상위 메소드")  
    }  
}  
  
open class RectOverride: ShapeOverride() {  
    override fun disp() {  
        super.disp()  
        println("처음 재정의한 메소드")  
    }  
}  
  
class RoundRectOverride: RectOverride() {  
    override fun disp() {  
        super.disp()  
        println("마지막으로 재정의한 메소드")  
    }  
}
```

상속 – MethodOverriding

❖ Overriding.kt

```
fun main(args: Array<String>) {  
    val roundRectOverride = RoundRectOverride()  
    roundRectOverride.disp()  
}
```



상속

❖ Property Overriding

- ✓ 프로퍼티도 오버라이딩 가능
- ✓ 메소드와 동일한 방식으로 수행
- ✓ 규칙
 - 상위 클래스의 프로퍼티와 이름 및 타입이 동일해야 함
 - 상위에 val 로 선언된 프로퍼티는 하위에서 val, var 로 재정의 가능
 - 상위에서 var로 선언된 프로퍼티는 하위에서 var로 재정의 가능, val은 불가
 - 상위에서 Nullable로 선언된 경우 하위에서 Non-Null 로 선언 가능
 - 상위에서 Non-Null 로 선언된 경우 하위에서는 Nullable로 재정의 불가

상속 – PropertyOverriding

❖ PropertyOverriding.kt

```
open class Super {  
    open val name: String = "Park"  
    open var age: Int = 10  
    open val email: String?=null  
    open val address: String="seoul"  
}
```

```
open class AdamSoft: Super() {  
    final override var name: String = "cyberadam"//가능  
    //override val age: Int = 20//에러  
    override val email: String = "ggangpae1@adamsoft.com"//가능  
    //override val address: String? = null//에러  
}
```

```
class TriglowPictures: AdamSoft() {  
    //override var name: String = "itggangpae"//에러  
}
```

```
fun main(args: Array<String>) {  
    val adam = AdamSoft()  
    println("${adam.name}")  
}
```

상속

❖ 상속 관계에서의 Casting

✓ 자동 형 변환(Smart Casting)

- 코드에서 명시적으로 캐스팅하지 않아도 형 변환이 수행되는 경우
- Any 타입의 인스턴스를 is 연산자를 이용해서 데이터의 자료형을 확인한 경우
- 하위 클래스의 인스턴스를 상위 클래스 타입의 변수에 대입하는 경우

✓ as를 이용한 형 변환 - 상속 관계에서만 가능

- 하위 클래스 타입의 변수에 상위 클래스 타입의 인스턴스를 대입할 때는 as를 이용해서 형 변환을 수행해서 대입해야 함
- 모든 인스턴스가 형 변환되는 것은 아니며 원래 자료형이 하위 클래스 타입이어야 가능
- null 허용 인스턴스의 경우 null 인 경우에는 예러 발생

✓ null을 허용하는 인스턴스의 형 변환

- as 대신에 as?를 이용해서 형 변환
- null이 대입되면 null을 반환하고 null이 아닌 데이터를 대입하면 정상적으로 형 변환을 수행

상속 - 자동 형 변환

❖ SmartCasting.kt

```
fun smartCast(data: Any): Int{  
    //val result = data * data //이 문장은 에러
```

```
    //data 가 Int 로 형 변환 됨  
    if(data is Int) return data * data  
    else return 0
```

```
}
```

```
class MyClass1 {  
    fun fun1(){  
        println("fun1()...")  
    }  
}
```

```
class MyClass2 {  
    fun fun2(){  
        println("fun2()...")  
    }  
}
```

```
}
```

```
fun smartCast2(obj: Any){  
    if(obj is MyClass1) obj.fun1()  
    else if(obj is MyClass2) obj.fun2()  
}
```


상속 - 자동 형 변환

❖ SmartCasting.kt

```
open class SuperSmart
```

```
class SubSmart: SuperSmart(){  
}
```



상속 - 자동 형 변환

❖ SmartCasting.kt

```
fun main(args: Array<String>) {  
  
    //기본형 데이터 사이의 형 변환  
    val data1: Int = 10  
    val data2: Double = data1.toDouble()  
  
    println("result : ${smartCast(10)}")  
    println("result : ${smartCast(10.0)}")  
  
    smartCast2(MyClass1())  
    smartCast2(MyClass2())  
  
    //상위 클래스의 참조형 변수에 하위 클래스의 인스턴스 대입 가능  
    val obj1: SuperSmart = SubSmart()  
    //val obj2: SubSmart = SuperSmart() //에러  
  
    //상위 클래스 타입으로 만들어진 데이터를 하위 클래스 타입의 변수에 대입할 때는 강제 형  
    변환을 수행해야 함  
    val obj2: SubSmart = obj1 as SubSmart  
    //val obj3 = SuperSmart() as SubSmart //에러는 아니지만 예외 발생  
}
```

상속 - 기타 형 변환

❖ AsOptionalCasting.kt

```
class NullCasting{  
    fun superFun(){  
        println("superFun()...")  
    }  
}
```

```
fun main(args: Array<String>) {
```

```
    val obj: NullCasting? = null  
    //val objcopy: NullCasting = obj as NullCasting //런타임 에러  
    val objcopy: NullCasting? = obj as? NullCasting  
    //objcopy.superFun() //에러  
    if(objcopy != null){  
        objcopy.superFun()  
    }else{  
        println("objcopy is null")  
    }  
}
```

상속

❖ Polymorphism(다형성)

- ✓ 동일한 메시지에 대하여 다르게 반응하는 성질
- ✓ 동일한 코드가 호출하는 변수에 대입된 인스턴스에 따라 다른 메소드를 호출하는 것
- ✓ 클래스 타입의 변수는 변수를 선언할 때 설정한 데이터 타입을 가지고 자신의 멤버를 호출하지만 오버라이딩 된 메소드 만은 대입된 인스턴스를 가지고 호출

상속 - 다형성

❖ Polymorphism.kt

```
open class Starcraft{  
    open fun attack(){}  
}
```

```
final class Protoss : Starcraft(){  
    override fun attack() {  
        super.attack()  
        println("프로토스의 공격")  
    }  
}
```

```
final class Zerg : Starcraft(){  
    override fun attack() {  
        super.attack()  
        println("저그의 공격")  
    }  
}
```

```
final class Terran : Starcraft(){  
    override fun attack() {  
        super.attack()  
        println("테란의 공격")  
    }  
}
```

상속 - 다형성

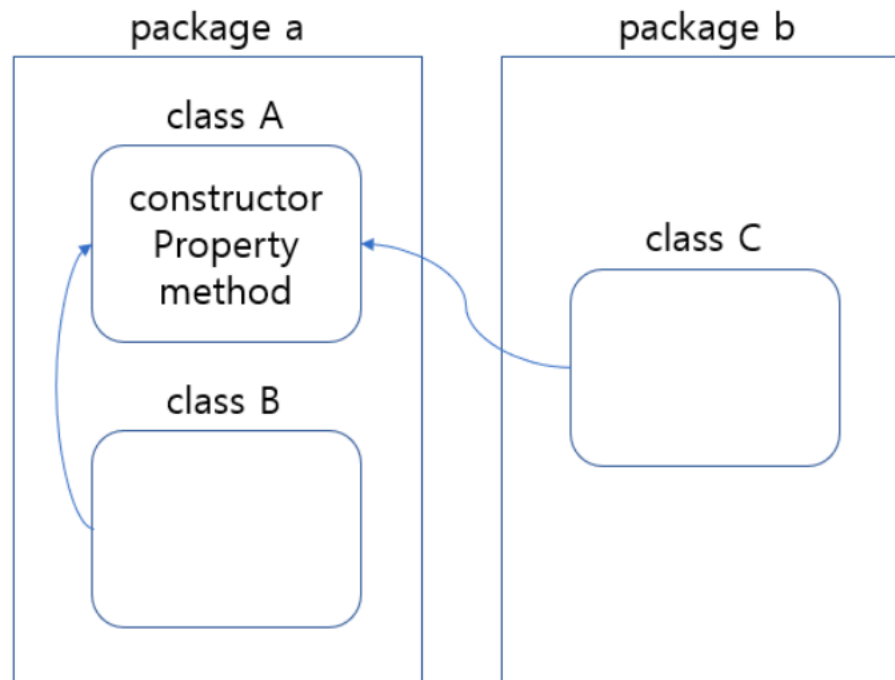
❖ Polymorphism.kt

```
fun main(args: Array<String>) {  
    var star : Starcraft = Starcraft()  
  
    star = Protoss()  
    star.attack()  
  
    star = Zerg();  
    star.attack()  
  
    star = Terran();  
    star.attack()  
}
```

Visibility Modifier(접근 제한자)

❖ 접근 제한자

- ✓ 외부에서 클래스, 생성자, 프로퍼티, 메소드(함수) 등을 이용할 때 사용 범위를 설정하기 위한 것
- ✓ public, internal, protected, private



Visibility Modifier(접근 제한자)

- ❖ 최상위 레벨의 접근 제한자(클래스 외부 선언)
 - ✓ public: (Default) 만약 접근제한자가 작성되지 않는다면 자동으로 public이 적용되며 public은 접근 제한이 없다는 의미로 어디에서든 import만 되면 접근이 가능
 - ✓ private: 동일 file 내에서만 접근이 가능
 - ✓ internal: 같은 module내에 어디서나 접근이 가능
 - ✓ protected: top-level에서는 사용 불가능

상속 – 최상위 레벨의 접근 제한

❖ TopModifier.kt

```
//top-level시 접근제한자  
val myData1: Int = 10
```

```
private val myData2: String = "hello"
```

```
internal fun myFun1() {  
    println("myFun() call..")  
}
```

```
private fun myFun2(){  
    println("myFun() call..")  
}
```

```
fun main(args: Array<String>) {  
    //top-level시 접근제한자  
    println("$myData1 .. ")  
    println("$myData2 .. ")  
    myFun1()  
    myFun2()  
}
```

상속 – 최상위 레벨의 접근 제한

❖ TopModifierExtern.kt

```
fun main(args: Array<String>) {  
    //top-level시 접근 제한자  
    println("$myData1 .. ")  
    //println("$myData2 .. ")//error  
    myFun1()  
    //myFun2()//error  
}
```

Visibility Modifier(접근 제한자)

- ❖ 클래스 내부의 접근 제한(클래스 내부의 멤버에 선언)
 - ✓ public: 클래스 내부 와 외부에서 인스턴스를 이용해서 접근 가능
 - ✓ private: 동일 클래스내에서만 접근 가능
 - ✓ protected: private + 서브 클래스에서 사용 가능
 - ✓ internal: 같은 모듈에 선언된 클래스에서 사용 가능
- ❖ property 의 접근 제한
 - ✓ get() 의 경우 프로퍼티의 접근제한자와 항상 동일한 접근제한자가 적용
 - ✓ set() 의 경우 프로퍼티의 접근제한자와 다른 접근제한자 설정이 가능하지만 범위를 넓혀서 설정할 수 없음
 - ✓ public > protected/internal > private
- ❖ 생성자의 접근 제한
 - ✓ 외부에서 인스턴스 생성의 범위를 제한할 목적으로 사용
- ❖ 상속 관계에서의 접근 제한
 - ✓ open 과 private은 같이 사용 못함
 - ✓ 하위 클래스에서 상위 클래스의 멤버를 오버라이딩 할 때 접근 범위를 줄일 수 없음

상속 - 클래스 내부의 접근 제한

❖ ClassModifier.kt

```
class Singleton private constructor() {  
  
    companion object {  
        @Volatile private var instance: Singleton? = null  
  
        @JvmStatic fun getInstance(): Singleton =  
            instance ?: synchronized(this) {  
                instance ?: Singleton().also {  
                    instance = it  
                }  
            }  
    }  
}  
  
private var data: Int = 10  
  
fun getData(): Int {  
    return data  
}  
  
fun setData(data : Int) {  
    this.data = data  
}  
}
```

상속 - 클래스 내부의 접근 제한

❖ ClassModifier.kt

```
fun main(args: Array<String>) {  
    //val singleton : Singleton = Singleton() //에러  
  
    val singleton1 = Singleton.getInstance()  
    val singleton2 = Singleton.getInstance()  
    println("$singleton1")  
    println("$singleton2")  
  
    //singleton1.data = 10; //에러  
    singleton1.setData(100)  
    println("${singleton1.getData()}")  
}
```

Abstract

Abstract Class

- ❖ Abstract Method(추상 메소드)
 - ✓ 내용이 없는 함수
 - ✓ 선언 부분만 존재하는 함수
 - ✓ 추상 메소드는 앞에 `abstract` 라는 예약어를 추가하고 코드 블록(`{ }`)을 작성하지 않으면 됨
 - ✓ 클래스 안의 메소드만 가능
- ❖ Abstract Property
 - ✓ `abstract`가 붙은 프로퍼티
- ❖ Abstract Class
 - ✓ `class` 앞에 `abstract`를 붙인 클래스
 - ✓ 인스턴스를 생성할 수 없는 클래스
 - ✓ `open` 예약어를 붙이지 않아도 상속할 수 있음
 - ✓ 추상 메소드나 추상 프로퍼티가 있으면 반드시 추상 클래스이어야 함
 - ✓ 추상 클래스를 상속받으면 하위 클래스에서는 추상 메소드와 추상 프로퍼티를 반드시 재정의 해야 함

Abstract Class – 추상 클래스 생성

❖ Abstract.kt

```
//추상 클래스 만들고 하위에서 오버라이드에 의해 재정의
abstract class AbstractSuper {
    //일반 프로퍼티로 하위 클래스에서 오버라이딩 할 필요가 없음
    val data1: Int = 10
    //추상 프로퍼티로 하위 클래스에서 반드시 오버라이딩 해야
    abstract val data2: Int

    //일반 메소드로 하위 클래스에서 오버라이딩 하지 않아도 됨
    fun myFun1() {
    }
    //추상 메소드 – 하위 클래스에서 반드시 재정의 해야 함
    abstract fun myFun2()
}

class AbstractSub: AbstractSuper() {
    //추상 프로퍼티와 추상 메소드 오버라이딩
    override val data2: Int = 10
    override fun myFun2() {

    }
}
```


Abstract Class – 추상 클래스 생성

❖ Abstract.kt

```
fun main(args: Array<String>) {  
    //val obj1=AbstractSuper()//error  
    val obj2=AbstractSub()  
}
```



Abstract Class – 추상 클래스 활용

❖ AbstractStarcraft.kt

```
abstract class AbstractStarcraft{  
    abstract fun attack()  
}
```

```
final class AbstractProtoss : AbstractStarcraft(){  
    override fun attack() {  
        println("프로토스의 공격")  
    }  
}
```

```
final class AbstractZerg : AbstractStarcraft(){  
    override fun attack() {  
        println("저그의 공격")  
    }  
}
```

```
final class AbstractTerran : AbstractStarcraft(){  
    override fun attack() {  
        println("테란의 공격")  
    }  
}
```

Abstract Class – 추상 클래스 활용

❖ Starcraft.kt

```
fun main(args: Array<String>) {  
    //var star : AbstractStarcraft = AbstractStarcraft()  
    var star : AbstractStarcraft? = null  
  
    star = AbstractProtoss()  
    star.attack()  
  
    star = AbstractZerg();  
    star.attack()  
  
    star = AbstractTerran();  
    star.attack()  
}
```

Interface

❖ Interface

- ✓ 추상 메소드를 선언하기 위한 목적의 개체
- ✓ 인스턴스 생성이 불가능
- ✓ 데이터를 저장하는 목적의 프로퍼티를 생성할 수 없기 때문에 프로퍼티를 만들 때 초기값을 대입할 수 없음
- ✓ 일반 메소드(Default Method)와 추상 메소드를 생성할 수 있는데 추상 메소드를 만들 때 `abstract`를 추가하지 않아도 추상 메소드가 됨
- ✓ 인터페이스는 클래스에서 구현하는데 생성자를 호출하는 형태가 아닌 인터페이스 이름만 : 뒤에 추가해서 사용
- ✓ 인터페이스끼리도 상속 가능
- ✓ 인터페이스는 여러 개의 인터페이스를 상속받을 수 있음
- ✓ 하나의 클래스에 여러 개의 인터페이스를 구현하는 것도 가능
- ✓ 클래스와 인터페이스를 동시에 상속받고 구현하는 것도 가능한데 이 때 클래스는 하나만 상속받아야 하며 클래스를 먼저 기재하는 것이 일반적

Interface 활용

❖ Interface.kt

```
interface StarcraftInt{  
    fun attack()  
}
```

```
final class ProtossImpl : StarcraftInt{  
    override fun attack() {  
        println("프로토스의 공격")  
    }  
}
```

```
final class ZergImpl : StarcraftInt{  
    override fun attack() {  
        println("저그의 공격")  
    }  
}
```

```
final class TerranImpl : StarcraftInt{  
    override fun attack() {  
        println("테란의 공격")  
    }  
}
```

Interface 활용

❖ Interface.kt

```
fun main(args: Array<String>) {  
    var star : StarcraftInt? = null  
  
    star = ProtossImpl()  
    star.attack()  
  
    star = ZergImpl();  
    star.attack()  
  
    star = TerranImpl();  
    star.attack()  
  
}
```

Interface

❖ Interface 와 Property

- ✓ 인터페이스에 프로퍼티 선언 가능
- ✓ 추상형으로 선언되어 있거나 `get()`, `set()` 를 정의해 주어야 함
- ✓ 추상 프로퍼티가 아니라면 `val` 의 경우는 `get()`이 꼭 선언되어 있어야 함
- ✓ 추상 프로퍼티가 아니라면 `var` 의 경우는 `get()`, `set()` 이 꼭 선언되어 있어야 함
- ✓ 인터페이스의 프로퍼티를 위한 `get()`, `set()`에서는 `field`를 사용할 수 없음
- ✓ 프로퍼티의 `get`, `set` 메소드에 로직을 추가할 수 는 있음

Interface

❖ InterfaceProperty.kt

```
interface MyInterface {  
  
    var prop1: Int // abstract  
  
    // val prop2: String = "adam"//error  
  
    // val prop3: String//error  
    //     get() = field  
  
    // var prop4: String//error  
    //     get() = "adam"  
  
    val prop5: String  
        get() = "adam"  
  
    var prop6: String  
        get() = "itggangpae"  
        set(value) {  
  
        }  
}
```


Interface

❖ InterfaceProperty.kt

```
interface PropertyInterface {  
  
    var data1: Int  
  
    var data2: Int  
        get() = 0  
        set(value){  
            if(value > 0)  
                calData(value)  
        }  
  
    val data3: Boolean  
        get(){  
            if(data1 > 0) return true  
            else return false  
        }  
  
    private fun calData(arg: Int) {  
        data1 = arg * arg  
    }  
}
```

Interface

❖ InterfaceProperty.kt

```
class MyClass: PropertyInterface {  
    override var data1: Int = 0  
}
```

```
fun main(args: Array<String>) {  
    val obj = MyClass()  
    println("data1 : ${obj.data1}, data2 : ${obj.data2}, data3 : ${obj.data3}")  
    obj.data2=5  
    println("data1 : ${obj.data1}, data2 : ${obj.data2}, data3 : ${obj.data3}")  
}
```

함수 식별

❖ 함수 식별

- ✓ 동일한 원형의 추상 메소드가 여러 개 있는 경우에는 한 번만 구현하면 됨
- ✓ 추상 메소드와 구현된 함수가 있는 경우에는 구현된 함수를 호출하고 재정의할 수 있음
- ✓ 구현된 함수가 여러 개 있는 경우에는 `super<클래스나 인터페이스 이름>.메소드이름`으로 명시적으로 호출할 수 있음

함수 식별

❖ MultiAbstractMethod.kt

```
interface Inter1 {  
    fun funA()  
}
```

```
abstract class Super1 {  
    abstract fun funA()  
}
```

```
open class Sub1: Super1(), Inter1 {  
    override fun funA() {  
        println("한 번만 오버라이딩")  
    }  
}
```

```
fun main(args: Array<String>) {  
    val obj = Sub1()  
    obj.funA()  
}
```

함수 식별

❖ MultilplementMethod.kt

```
interface Interface2 {  
    fun funA() {  
        println("Interface2 funA...")  
    }  
}  
  
interface Interface3 {  
    fun funA() {  
        println("Interface3 funA...")  
    }  
}  
  
class Sub2: Interface2, Interface3 {  
    override fun funA() {  
        super<Interface2>.funA()  
        super<Interface3>.funA()  
    }  
    fun some(){  
        super<Interface2>.funA()  
    }  
}
```

함수 식별

❖ MultiImplementMethod.kt

```
fun main(args: Array<String>) {  
    val obj = Sub2()  
    obj.funA()  
    obj.some()  
}
```



다양한 코틀린 클래스

DTO(Data Transfer Class)

❖ DTO

- ✓ VO(Value-Object) 클래스라고도 함
- ✓ 연관된 여러 개의 데이터를 하나의 클래스로 묶어서 사용하기 위한 목적으로 생성
- ✓ data 라는 예약어와 함께 선언
- ✓ 제약조건
 - 주 생성자가 선언되어야 하며 주 생성자의 매개변수는 최소 하나 이상 선언되어 있어야 함
 - 모든 주 생성자의 매개변수는 var 혹은 val 로 선언되어야 함
 - 데이터 클래스는 abstract, open, sealed, inner 등의 예약어를 추가할 수 없음

```
data class Data(val name: String, val age: Int)
```

```
//data class Data1()//error
```

```
//data class Data2(name: String)//error
```

```
//data abstract class Data3(val name: String)//error
```

```
//data class Data4(val name: String, no: Int)//error
```


DTO(Data Transfer Class)

❖ DTO

- ✓ 5개의 메소드를 자동 생성
- ✓ hashCode(): 해시 코드를 리턴해주는 메소드
- ✓ equals(): 인스턴스의 데이터가 같은지를 비교해주는 메소드 - 주 생성자의 프로퍼티 값 만을 가지고 비교
- ✓ toString(): 인스턴스의 데이터를 문자열로 리턴해주는 메소드
- ✓ component(): 인스턴스의 프로퍼티 값을 가져와야 할 때 사용 - component1(), component2() 등으로 생성됨
- ✓ copy(): 인스턴스를 복제해서 다른 인스턴스를 생성하는데 데이터의 일부분만 수정해서 복제하고자 할 때 이용

DTO – toString 과 componentN

❖ PropertyDisplay.kt

```
data class Property(val name: String, val age: Int){  
    var email:String = "ggangpae1@gmail.com"  
}
```

```
fun main(args: Array<String>) {  
    val property = Property("박문석", 50)  
    println(property.toString())  
    println(property)  
    println("이름:${property.component1()}")  
    println("나이:${property.component2()}")  
}
```

DTO – copy

❖ Copy.kt

```
fun main(args: Array<String>) {  
    val property = Property("박문석", 50)  
    val weakcopy = property  
    val deepcopy = property.copy("아담")  
    println("해시코드:${property.hashCode()} 데이터:${property.toString()}")  
    println("해시코드:${weakcopy.hashCode()} 데이터:${weakcopy.toString()}")  
    println("해시코드:${deepcopy.hashCode()} 데이터:${deepcopy.toString()}")  
}
```

DTO – equals

❖ Equals.kt

```
class Product(val name: String, val price: Int)
```

```
data class Factory(val name: String, val phone: String){  
    var email: String = "ggangpae1@gmail.com"  
}
```

DTO – equals

❖ Equals.kt

```
fun main(args: Array<String>) {  
  
    //값은 동일하지만 equals를 재정의하지 않았으므로 서로 다름  
    var product1=Product("prod1",100)  
    var product2=Product("prod1",100)  
    println(product1.equals(product2))  
  
    //내용이 다르기 때문에 서로 다름  
    var factory1=Factory("adam","010")  
    var factory2=Factory("adam","011")  
    println(factory1.equals(factory2))  
  
    //내용이 같고 equals가 재정의 되어 있으므로 2개는 일치  
    var factory3=Factory("adam","012")  
    var factory4=Factory("adam","012")  
    println(factory3.equals(factory4))  
  
    factory3.email = "ggangpae1@gmail.com"  
    factory4.email = "itstudy@kakao.com"  
    println(factory3.equals(factory4))  
}
```

Enum

❖ Enum

- ✓ 열거형 상수
- ✓ 상수 여러 개를 대입해서 선언하고 이 값 중의 하나를 지정해서 사용할 수 있도록 해주는 개체
- ✓ 옵션을 설정할 때 주로 이용
- ✓ 생성

```
enum class 열거형 상수 이름{  
    상수 이름 나열  
}
```

- ✓ 사용: 열거형 상수 이름.상수이름
- ✓ name 속성을 이용하면 상수 이름을 리턴
- ✓ ordinal 속성을 호출하면 정수 값을 리턴
- ✓ 상수 이름을 나열할 때 생성자를 이용해서 데이터를 설정하는 것이 가능

Enum

❖ Enum.kt

```
enum class Direction {  
    NORTH, SOUTH, WEST, EAST  
}
```

```
enum class Priority(val no : Int) {  
    MAX(10), NORMAL(5), MIN(1)  
}
```

```
fun main(args: Array<String>) {  
    val direction: Direction = Direction.NORTH  
    println("${direction.name} ... ${direction.ordinal}")  
  
    val direction1=Direction.valueOf("WEST")  
    println("${direction1.name} .. ${direction1.ordinal}")  
  
    val priority: Priority = Priority.NORMAL  
    println("${priority.name} .. ${priority.no}")  
}
```

Enum

❖ Enum에 익명 클래스 이용

- ✓ 열거형 상수는 Enum 클래스를 상속받는 클래스의 서브 클래스의 인스턴스
- ✓ 열거형 상수 안에 {}를 추가하고 프로퍼티나 메소드를 선언하는 것이 가능



Enum – Anonymous Class

❖ EnumAnonymous.kt

```
enum class AnonyDirection {  
    NORTH {  
        override val data1: Int = 10  
        override fun myFun(){  
            println("north myFun....")  
        }  
    },  
    SOUTH {  
        override val data1: Int = 20  
        override fun myFun(){  
            println("south myFun....")  
        }  
    };  
  
    abstract val data1: Int  
    abstract fun myFun()  
}  
  
fun main(args: Array<String>) {  
    val direction: AnonyDirection = AnonyDirection.NORTH  
    println(direction.data1)  
    direction.myFun()  
}
```

Nested Class

❖ Nested Class

- ✓ 클래스 내부에 만들어 진 클래스
- ✓ 클래스 이름은 외부클래스이름.내부클래스이름
- ✓ 내부 클래스에서는 외부 클래스의 멤버에 접근할 수 없음
- ✓ 내부 클래스에서 외부 클래스의 멤버에 접근하기 위해서는 inner 라는 예약어를 이용해서 클래스를 생성
- ✓ inner 가 추가된 내부 클래스는 외부에서 인스턴스 생성을 할 수 없고 외부 클래스의 인스턴스를 생성하고 내부 클래스의 인스턴스 생성을 해야 함

Nested – nested Class

❖ Nested.kt

```
class Outer {  
    var score = 80  
    class Nested {  
        val name: String = "Adam"  
        fun myFun(){  
            println("Nested.. myFun...")  
            //score = 90 //에러 외부 클래스의 멤버에 직접 접근이 안됨  
        }  
    }  
}  
  
fun main(args: Array<String>) {  
    val obj: Outer.Nested = Outer.Nested()  
    println("${obj.name}")  
    obj.myFun()  
}
```

Nested – inner Class

❖ Inner.kt

```
class OuterInner {  
    private var no: Int = 10  
    fun outerFun() {  
        println("outerFun()...")  
    }  
    inner class Nested {  
        val name: String = "kkang"  
        fun myFun(){  
            println("Nested.. myFun...")  
            no=20  
            outerFun()  
        }  
    }  
}  
  
fun createNested(): Nested {  
    return Nested()  
}
```

Nested – inner Class

❖ Inner.kt

```
fun main(args: Array<String>) {  
    //val obj: OuterInner.Nested = OuterInner.Nested() //에러 외부 클래스의 인스턴스를 생성  
    하고 내부 클래스의 인스턴스를 생성해야 함  
  
    val obj1: OuterInner.Nested = OuterInner().Nested()  
    obj1.myFun()  
    val obj2: OuterInner.Nested = OuterInner().createNested()  
    obj2.myFun()  
}
```

Sealed Class

❖ Sealed Class

- ✓ sealed 라는 예약어로 선언하는 클래스
- ✓ 열거형 상수와 유사한 용도로 사용
- ✓ sealed class 안에 sealed class 로 부터 상속받는 클래스를 생성할 수 있고 외부에 상속받는 클래스를 만들 수 있음
- ✓ 다른 파일에 상속받는 클래스를 만들 수 없음
- ✓ abstract를 내장하고 있어서 인스턴스를 생성할 수 없음
- ✓ 생성자가 private 으로 강제되어 있음
- ✓ 변수를 선언하면 하위 클래스의 인스턴스만 대입할 수 있음
- ✓ 추상 클래스나 인터페이스는 확장이 목적이지만 sealed class는 일반적으로 확장이 목적이 아니고 제한이 목적

Sealed – sealed Class

❖ Sealed.kt

```
sealed class Shape(){  
    //내부 클래스로 하위 클래스를 생성  
    class Circle(val radius: Double) : Shape()  
    class Rect(val width: Int, val height: Int) : Shape()  
}  
//외부에 하위 클래스를 생성  
class Triangle(val bottom: Int, val height: Int): Shape()  
  
fun main(args: Array<String>) {  
    //val shape : Shape = Shape() //에러 abstract가 없지만 추상 클래스이므로 인스턴스 생성 안됨  
    val shape1: Shape = Shape.Circle(10.0)  
    val shape2: Shape = Triangle(10, 10)  
}
```

Object - Anonymous Class

❖ Anonymous Class

- ✓ 이름없는 클래스
- ✓ 클래스의 인스턴스를 여러 개 만들 필요가 없는 경우 클래스를 미리 만들어 두는 것은 자원의 낭비이므로 필요할 때 만들어서 사용하는 것이 효율적
- ✓ 별도로 클래스를 만들지 않고 바로 인스턴스를 만들어서 사용하는 것
- ✓ object 라는 예약어와 함께 생성

```
val 인스턴스이름 = object{  
    프로퍼티  
    메소드  
}
```
- ✓ 사용법은 일반 인스턴스와 동일하게 사용
- ✓ 외부 클래스의 멤버에는 접근 가능
- ✓ 외부 클래스에서 내부 Anonymous의 멤버에는 바로 접근할 수 없고 접근할 수 있도록 할려면 Anonymous를 생성할 때 private 추가

Object – Anonymous Class

❖ Anonymous.kt

```
class AnonymousOuter {  
  
    private var no: Int = 0  
  
    //외부 클래스에서 접근할 수 있도록 private을 추가해서 생성  
    private val anonymousInner = object {  
        val name: String = "adam"  
        fun innerFun(){  
            println("innerFun...")  
            //외부 클래스의 멤버에는 접근 가능  
            no++  
        }  
    }  
  
    fun outerFun(){  
        anonymousInner.name  
        anonymousInner.innerFun()  
    }  
}
```

object – Anonymous Class

❖ Anonymous.kt

```
fun main(args: Array<String>) {  
    val obj=AnonymousOuter()  
    //obj.anonymousInner.name //myInner가 private 이므로 외부에서 인스턴스를 이용해서 접근할 수 없음  
}
```

Object - Anonymous Class

- ❖ 다른 클래스를 상속받거나 인터페이스를 구현한 Anonymous Class
 - ✓ object 다음에 다른 클래스 이름이나 생성자 또는 인터페이스를 기재해서 다른 클래스를 상속받거나 인터페이스를 구현한 Anonymous Class 도 생성 가능
- ❖ object 다음에 클래스 이름을 명시
 - ✓ object 다음에 클래스 이름을 명시하면 싱글톤 패턴의 클래스 생성이 가능
 - ✓ 생성자를 명시할 수 없기 때문에 하나의 인스턴스만 만들어서 사용
 - ✓ 인스턴스 생성을 할 수 없기 때문에 클래스 이름만으로 사용
- ❖ companion
 - ✓ 이름있는 object 클래스를 만들 때 클래스 이름만으로 멤버에 접근하기 위한 예약어
 - ✓ java의 static 과 유사한 효과

object – Named Anonymous Class

❖ NamedAnonymous.kt

```
interface SomeInterface {  
    fun interfaceFun()  
}  
  
open class SomeClass {  
    fun someClassFun(){  
        println("someClassFun....")  
    }  
}  
  
class SomeOuter {  
    val someInner: SomeClass = object : SomeClass(), SomeInterface {  
        override fun interfaceFun() {  
            println("interfaceFun....")  
        }  
    }  
}
```

object – Named Anonymous Class

❖ NamedAnonymous.kt

```
object ObjectClass {  
    fun myFun() {  
        println("Singleton Pattern....")  
    }  
}  
  
class CompanionOuter {  
    companion object NestedClass {  
        val no: Int = 0  
        fun myFun() {println("no:$no")}  
    }  
  
    fun myFun(){  
        myFun()  
    }  
}
```

object – Named Anonymous Class

❖ NamedAnonymous.kt

```
fun main(args: Array<String>) {  
    val obj = SomeOuter()  
    obj.someInner.someClassFun()  
  
    //val obj: ObjectClass = ObjectClass()//error  
    ObjectClass.myFun()  
  
    CompanionOuter.NestedClass.no  
    CompanionOuter.NestedClass.myFun()  
  
    CompanionOuter.myFun()  
}
```

클래스 사이의 관계

클래스 사이의 관계

❖ Association(연관 관계)

- ✓ 서로 다른 2개의 클래스의 인스턴스가 단방향 또는 양방향으로 연결을 가지는 경우
- ✓ 2개의 인스턴스의 수명 주기는 다름
- ✓ 서로 다른 인스턴스를 소유하지는 않음

❖ Dependency(의존 관계)

- ✓ 서로 다른 2개의 클래스의 인스턴스들에서 하나의 인스턴스가 존재해야만 다른 하나의 인스턴스를 생성할 수 있는 관계로 메소드의 실행동안처럼 짧게 등장

❖ Aggregation

- ✓ Association의 특별한 경우인데 모든 오브젝트가 각자의 라이프사이클을 가지고 있으며 한 오브젝트가 다른 오브젝트를 소유하고 있는 경우로 선생님이 어떤 부서에 Aggregation 되면 소속된 관계이기 때문에, 한 선생님이 여러 부서에 Aggregation 될 수는 없으며 그렇다고 해서 부서가 소멸될 때 선생님도 소멸되는 것은 아는데 이것을 "has-a" 관계라고 함

❖ Composition

- ✓ Aggregation의 특별한 경우인데, 이것을 "죽음의" 연관관계라고 부르기도 하는데 강력하게 연관된 Aggregation이며 자식 오브젝트는 자신의 라이프사이클을 가지지 않고, 부모 오브젝트가 소멸될 경우 자식 오브젝트도 함께 소멸되는 경우로 집과 그 안의 방 사이의 관계라 할 수 있는데 집은 여러 개의 방을 가지고 있고 방은 절대로 독립적인 라이프사이클을 가질 수 없으며 우리가 집을 소멸시키면, 방도 함께 소멸

❖ Generalization: 여러 클래스들의 공통된 내용을 묶어서 상위 클래스를 만드는 것

❖ Realization: 인터페이스의 내용을 구현한 클래스와 인터페이스의 관계

클래스 사이의 관계

❖ Association.kt

```
class Patient(val name: String) {  
    fun doctorList(d: Doctor) { // 인자로 참조  
        println("Patient: $name, Doctor: ${d.name}")  
    }  
}
```

```
class Doctor(val name: String) {  
    fun patientList(p: Patient) { // 인자로 참조  
        println("Doctor: $name, Patient: ${p.name}")  
    }  
}
```

```
fun main() {  
    val doctor = Doctor("KimSabu")  
    val patient = Patient("Adam")  
    doctor.patientList(patient)  
    patient.doctorList(doctor)  
}
```

클래스 사이의 관계

❖ Dependency.kt

```
data class Item(var name:String){}
```

```
class ItemFactory{  
    fun production(){  
        val item1 = Item("망치")  
        val item2 = Item("도끼")  
        println(item1)  
        println(item2)  
    }  
}
```

```
fun main() {  
    val factory = ItemFactory()  
    factory.production()  
}
```

클래스 사이의 관계

❖ Aggregation.kt

```
// 여럿의 오리를 위한 List 매개변수
class Pond(_name: String, _members: MutableList<Duck>) {
    val name: String = _name
    val members: MutableList<Duck> = _members
    constructor(_name: String): this(_name, mutableListOf<Duck>())
}

class Duck(val name: String)

fun main() {
    // 두 개체는 서로 생명주기에 영향을 주지 않는다.
    val pond = Pond("myFavorite")
    val duck1 = Duck("Duck1")
    val duck2 = Duck("Duck2")

    // 연못에 오리를 추가 - 연못에 오리가 집합한다
    pond.members.add(duck1)
    pond.members.add(duck2)
    // 연못에 있는 오리들
    for (duck in pond.members) {
        println(duck.name)
    }
}
```

클래스 사이의 관계

❖ Composite.kt

```
class Car(val name: String, val power: String) {  
  
    private var engine = Engine(power) // Engine클래스 객체는 Car에 의존적이다  
  
    fun startEngine() = engine.start()  
    fun stopEngine() = engine.stop()  
}  
  
class Engine(power: String) {  
    fun start() = println("Engine has been started.")  
    fun stop() = println("Engine has been stopped.")  
}  
  
fun main() {  
    val car = Car("tico", "100hp")  
    car.startEngine()  
    car.stopEngine()  
}
```