05

05-2. 제네릭

다양한 기법

제네릭의 이해

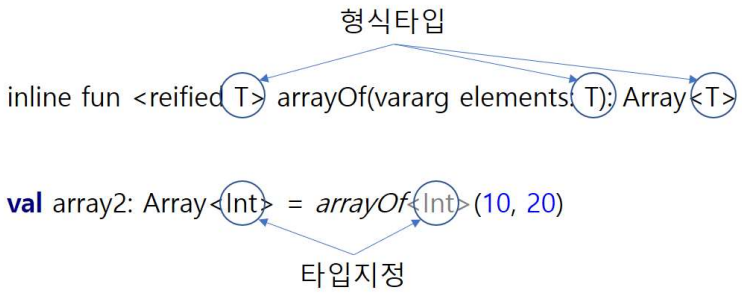
제네릭이란?

- 제네릭이란 형식타입
- 타입을 예측할수 없거나 하나의 타입으로 고정할수 없는 경우
- 제네릭으로 형식 타입을 선언하고 실제 이용시 정확한 타입을 부여

```
val array = arrayOf("kkang", 10, true)
```

- inline fun <reified T> arrayOf(vararg elements: T): Array<T>

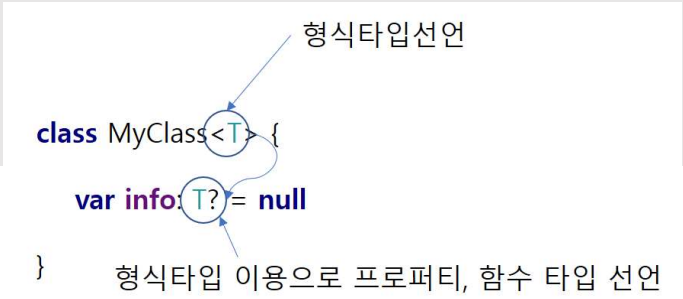
```
val array2: Array<Int> = arrayOf<Int>(10, 20)
```



제네릭의 이해

제네릭 선언 및 이용

```
class MyClass<T> {
    var info: T? = null
}

fun main(args: Array<String>) {
    val obj1=MyClass<String>()
    obj1.info="hello"

    val obj2=MyClass<Int>()
    obj2.info=10
}
```



제네릭의 이해

타입 유추에 의한 이용

```
class MyClass2<T>(no: T){
    var info: T? = null
}

fun main(args: Array<String>) {

    val obj3=MyClass2<Int>(10)
    obj3.info=20

    val obj4=MyClass2("hello")
    obj4.info="world"
}
```

제네릭의 이해

형식타입 여러 개 선언

```
class MyClass<T, A> {
    var info: T? = null
    var data: A? = null
}

fun main(args: Array<String>) {
    val obj: MyClass<String, Int> = MyClass()
    obj.info="hello"
    obj.data=10
}
```

제네릭의 이해

함수와 제네릭

```
class MyClass<T, A> {
    var info: T? = null
    var data: A? = null

    fun myFun(arg: T): A? {
        return data
    }
}
```

```
fun <T> someFun(arg: T): T? {
    return null
}
```

제네릭 제약

타입제약

- 제네릭 제약(Generic Constraint) 란 형식타입을 선언하면서 특정 타입만 대입되도록 제약하는 것

```
class MathUtil<T: Number> {
    fun plus(arg1: T, arg2: T): Double {
        return arg1.toDouble() + arg2.toDouble()
    }
}

fun main(args: Array<String>) {
    val obj = MathUtil<Int>()
    obj.plus(10, 20)

    val obj2 = MathUtil<Double>()

//    val obj3 = MathUtil<String>()//error
}
```

제네릭 제약

여러 개의 타입으로 제약

```
interface MyInterface1
interface MyInterface2

class MyHandler1: MyInterface1, MyInterface2

class MyHandler2: MyInterface1

class  MyClass<T> where T: MyInterface1, T: MyInterface2{
    //.............
}

fun main(args: Array<String>) {
    val obj = MyClass<MyHandler1>()

    val obj2 = MyClass<MyHandler2>()//error
}
```

제네릭 제약

Null 불허 제약

- 제네릭의 형식타입은 기본으로 Nullable로 선언

```
class MyClass<T> {
    fun myFun(arg1: T, arg2: T){
        println(arg1?.equals(arg2))//arg1.equals(arg2) 는 error
    }
}

fun main(args: Array<String>) {
    val obj = MyClass<String>()
    obj.myFun("hello", "hello")

    val obj2 = MyClass<Int?>()
    obj2.myFun(null, 10)
}
```

제네릭 제약

Null 불허 제약

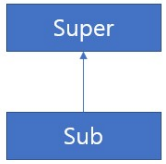
- Non-Nullable 로 선언

```
class MyClass<T: Any> {
    fun myFun(arg1: T, arg2: T){
        println(arg1.equals(arg2))
    }
}

fun main(args: Array<String>) {
    val obj = MyClass<String>()
    obj.myFun("hello", "hello")

    val obj2 = MyClass<Int?>()//error
    obj2.myFun(null, 10)
}
```

Variance

Variance란?

- 제네릭에서 Variance(가변, 공변)란 상하위 관계에서 타입 변형과 관련.




```
open class Super {
    open fun sayHello() {
        println("i am super sayHello...")
    }
}
class Sub: Super(){
    override fun sayHello() {
        println("i am sub sayHello....")
    }
}
fun main(args: Array<String>) {
    val obj: Super = Sub()
    obj.sayHello()

    val obj2: Sub = obj as Sub
    obj2.sayHello()
}
```

Variance

invariance

- 제네릭은 타입이지 클래스가 아니다.

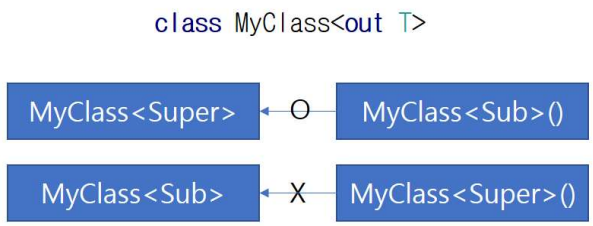

```
open class Super

class Sub: Super()

class MyClass<T>

fun main(args: Array<String>) {
    val obj = MyClass<Sub>()

    val obj2: MyClass<Super> =  obj//error
}
```

Variance

covariance

- out 어노테이션을 이용하여 하위 제네릭 타입으로 선언된 객체를 상위 제네릭 타입에 대입.

```
open class Super

class Sub: Super()

class MyClass<out T>

fun main(args: Array<String>) {
    val obj = MyClass<Sub>()
    val obj2: MyClass<Super> =  obj

    val obj3 = MyClass<Super>()
    val obj4: MyClass<Sub> = obj3//error
}
```



Variance

---

covariance

- out 어노테이션을 사용하는 규칙

- 하위 제네릭 타입이 상위 제네릭 타입에 대입 가능
- 상위 제네릭 타입이 하위 제네릭 타입에 대입 불가능
- 함수의 리턴 타입으로 선언가능
- 함수의 매개변수 타입으로 선언 불가능
- val 프로퍼티에 선언가능
- var 프로퍼티에 선언 불가능

```
open class Super

class Sub: Super()

class MyClass<out T>(val data: T) {
    val myVal: T? = null
    var myVal2: T? = null//error
    fun myFun(): T {
        return data
    }
    fun myFun3(arg: T) { }//error
}
fun main(args: Array<String>) {
    val obj = MyClass<Sub>(Sub())
    val obj2: MyClass<Super> =  obj

    val obj3 = MyClass<Super>(Super())
    val obj4: MyClass<Sub> = obj3//error
}
```

Variance

covariance

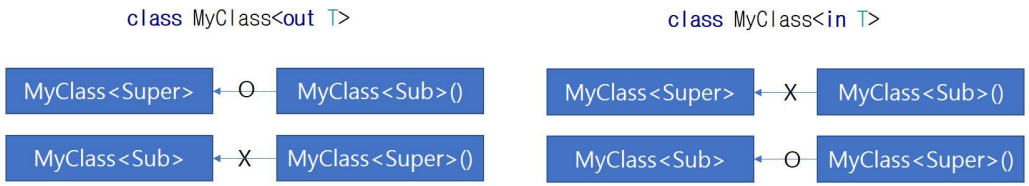
- `public interface MutableList<E> : List<E>, MutableCollection<E> { }`
- `public interface List<out E> : Collection<E> { }`

```
fun main(args: Array<String>) {
    val mutableList: MutableList<Int> = mutableListOf(10, 20)
    val mutableList2: MutableList<Number> = mutableList//error

    val immutableList: List<Int> = listOf(10, 20)
    val immutableList2: List<Number> = immutableList
}
```

Variance

contravariance

- in 어노테이션을 이용해 상위 제네릭 타입이 하위 제네릭 타입에 대입되어 사용.



in 어노테이션 규칙

- 하위 제네릭 타입이 상위 제네릭 타입에 대입 불가능
- 상위 제네릭 타입이 하위 제네릭 타입에 대입 가능
- 함수의 리턴 타입으로 선언 불가능
- 함수의 매개변수 타입으로 선언 가능
- val 프로퍼티에 선언 불가능
- var 프로퍼티에 선언 불가능

Variance

contravariance

```
open class Super

class Sub: Super()

class MyClass<in T>() {

    val myVal: T? = null//error
    var myVal2: T? = null//error

    fun myFun(): T? {//error
        return null
    }
    fun myFun3(arg: T) { }
}

fun main(args: Array<String>) {
    val obj = MyClass<Sub>()
    val obj2: MyClass<Super> =  obj//error

    val obj3 = MyClass<Super>()
    val obj4: MyClass<Sub> = obj3
}
```

Unit, Nothing 타입

Unit 타입

- 자바의 void와 Unit의 차이

```
public class JavaTest {
    public void javaFun(){ }

    public static void main(String[] args){
        JavaTest obj=new JavaTest();
        System.out.println(obj.javaFun());//error
    }
}
```

```
fun myFun1(){ }
fun myFun2(): Unit { }
```

Unit, Nothing 타입

Unit 타입

- void는 함수의 리턴값이 없다는 일종의 예약어이지만 Unit은 타입
- Unit은 kotlin.Unit만 대입되는 특이한 타입

```
fun myFun1(){ }
fun myFun2(): Unit { }

fun myFun3(): Unit {
    return Unit
}

val myVal: Unit = Unit

fun main(args: Array<String>) {
    println(myFun1())
}
```

실행결과

```
kotlin.Unit
```

Unit, Nothing 타입

Unit 타입

- 제네릭에서 Unit의 이용

```
interface MyInterface<T> {
    fun myFun(): T
}

class MyClass: MyInterface<String> {
    override fun myFun(): String {
        return "hello"
    }
}

class MyClass2: MyInterface<Unit> {
    override fun myFun() {

    }
}
```

Unit, Nothing 타입

Nothing

- Nothing 타입으로 선언이 되면 이곳에는 null 만 대입
- Nothing은 결국 값이 없다는 것을 명시적으로 표현하기 위해서 사용

```
fun myFun(arg: Nothing?): Nothing {
    throw Exception()
}
val myVal: Nothing? = null
```

Unit, Nothing 타입

Nothing

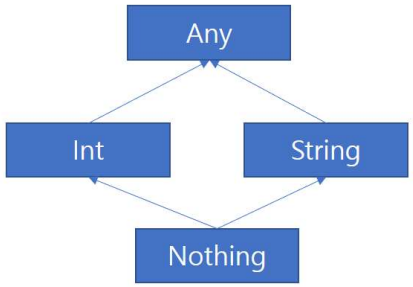
- 함수의 리턴 타입으로 Nothing 사용
- 함수는 리턴이 없다는 것을 명시적으로 선언하고 싶은 경우

```
fun myFun(): Nothing {
    while(true){
        //.....
    }
}
fun myFun2(): Nothing? {
    return null
}
fun myFun3(): Nothing {
    throw Exception()
}
```

Unit, Nothing 타입

Nothing

- 제네릭에서 Nothing의 이용
- Nothing 타입은 다른 어떤 타입의 프로퍼티에도 대입이 가능




```
val myVal1: Nothing? = null

val myVal2: Int? = myVal1
val myVal3: String? = myVal1
```

Unit, Nothing 타입

Nothing

```
class MyClass<T>

fun someFun(arg: MyClass<in Nothing>){ }

fun main(args: Array<String>) {
    someFun(MyClass<Int>())
    someFun(MyClass<String>())
}
```

감사합니다

단단히 마음먹고 떠난 사람은
산꼭대기에 도착할 수 있다.
산은 올라가는 사람에게만 정복된다.

윌리엄 셰익스피어
William Shakespeare