

Java Interop

2021/01/22

Hyojeong Yu



Calling Java code from Kotlin



Calling Kotlin from Java

Calling Java from Kotlin

Getters and Setters

Kotlin

```
class KotlinMember(var age: Int)

fun main() {
    val member: KotlinMember = KotlinMember(28)
    println("member.age = ${member.age}")
    member.age = 29
}
```

Java

```
public class JavaMember {
    private Integer age = -1;

    public JavaMember(Integer age) {
        this.age = age;
    }
}
```

```
fun main() {
    val member : JavaMember = JavaMember(28)
    println("member.age = ${member.age}") // X
    member.age = 29 // X
}
```

Setters and Getters are necessary

Methods returning void

```
public class JavaSample {  
    public static void returnVoidFunction() {  
        // ...  
    }  
}
```

```
fun main() {  
    val void: Unit = JavaSample.returnVoidFunction()  
}
```

If a Java method returns void, it will return Unit when called from Kotlin.

Escaping for Java identifiers that are keywords in Kotlin

```
@Test
fun ItShouldReturnLdapJoinPageWhenUserIsNotSignedInLdap() {
    val username = "lunit"

    // when(IdapService.findByUsername(username)).thenReturn(null)
    `when` (IdapService.findByUsername(username)).thenReturn(null)

    mockMvc.perform(get("/")
        .withUserSession(username)
    )
        .andExpect(status().is3xxRedirection)
        .andExpect(redirectedUrl("/users/join-form"))
}
```

- Some of the Kotlin keywords are valid identifiers in Java: in, object, is, etc.
- you can still call the method escaping it with the **backtick (`) character**

Null-Safety and Platform types

```
public class _3_null_safety {  
    public static ArrayList<String> list;  
  
    static {  
        list = new ArrayList<String>();  
        list.add(null);  
    }  
}  
  
fun main() {  
    val size = _3_null_safety.list.size // non-null (primitive int)  
    val item = _3_null_safety.list[0] // platform type inferred (ordinary Java object)  
  
    item.substring(1) // allowed, may throw an exception if item == null  
  
    val nullable: String? = item // allowed, always works  
    val notNull: String = item // allowed, may fail at runtime  
}
```

- Types of Java declarations are treated specially in Kotlin and called **platform types**.
- Null-checks are relaxed for such types, so that safety guarantees for them are the same as in Java

Notation for Platform Types

<code>T!</code>	<code>T</code> or <code>T?</code>
<code>(Mutable)Collection<T>!</code>	Java collection of <code>T</code> may be mutable or not, may be nullable or not
<code>Array<(out) T>!</code>	Java array of <code>T</code> (or a subtype of <code>T</code>), nullable or not

- **Platform types** cannot be mentioned explicitly in the program
- So there's no syntax for them in the language.
- Nevertheless, the compiler and IDE need to display them sometimes (in error messages, parameter info etc)
- So we have a **mnemonic notation for them**

Nullability annotations

JetBrains	@Nullable, @NotNull
Android	com.android.annotations and android.support.annotations
JSR-305	javax.annotation
FindBugs	edu.umd.cs.findbugs.annotations
Eclipse	org.eclipse.jdt.annotation
Lombok	lombok.NonNull

- Java types which have **nullability annotations** are represented not as platform types, but **as actual nullable or non-null Kotlin types**

Annotating type parameters

With annotation type parameters

```
public class _4_Null_Annotations {  
    @NotNull  
    public static Set<@NotNull String> toSet(@NotNull Collection<@NotNull String> elements) {  
        return new HashSet<String>(elements);  
    }  
}
```

```
fun toSet(elements: (Mutable)Collection<String>) : (Mutable)Set<String> { ... }
```

Without them

```
fun toSet(elements: (Mutable)Collection<String!>) : (Mutable)Set<String!> { ... }
```

Mapped types

Mapped types: 특별하게 다뤄지는 일부 자바 타입

특별하게?

- 해당 자바 타입들은 그대로 로딩되지 않고 대응되는 코틀린 타입으로 매핑됩니다
- 매핑은 컴파일 시점에만 중요하며 런타임때는 해당 표현들이 바뀌지 않고 유지됩니다.

기본 타입(byte, short, int, ...) + 일부 내장 클래스(Object, Enum, ...)

박싱 데이터 (Byte, Short, Integer, ...)

- 자바의 박싱된 기본 타입은 nullable 한 코틀린 타입으로 매핑됩니다.
- 컬렉션의 타입 파라미터로 박싱된 기본 타입을 사용하는 경우는 플랫폼 타입으로 매핑됩니다.

컬렉션 타입, 배열 타입

Mapped types - example

Java type	Kotlin type
byte	kotlin.Byte
short	kotlin.Short
int	kotlin.Int
long	kotlin.Long
char	kotlin.Char
float	kotlin.Float
double	kotlin.Double
boolean	kotlin.Boolean

Java type	Kotlin type
java.lang.Object	kotlin.Any!
java.lang.Cloneable	kotlin.Cloneable!
java.lang.Comparable	kotlin.Comparable!
java.lang.Enum	kotlin.Enum!
java.lang.Annotation	kotlin.Annotation!
java.lang.CharSequence	kotlin.CharSequence!
java.lang.String	kotlin.String!
java.lang.Number	kotlin.Number!
java.lang.Throwable	kotlin.Throwable!

Mapped types - example

Java type	Kotlin read-only type	Kotlin mutable type	Loaded platform type
Iterator<T>	Iterator<T>	MutableIterator<T>	(Mutable)Iterator<T>!
Iterable<T>	Iterable<T>	MutableIterable<T>	(Mutable)Iterable<T>!
Collection<T>	Collection<T>	MutableCollection<T>	(Mutable)Collection<T>!
Set<T>	Set<T>	MutableSet<T>	(Mutable)Set<T>!
List<T>	List<T>	MutableList<T>	(Mutable)List<T>!
ListIterator<T>	ListIterator<T>	MutableListIterator<T>	(Mutable)ListIterator<T>!
Map<K, V>	Map<K, V>	MutableMap<K, V>	(Mutable)Map<K, V>!
Map.Entry<K, V>	Map.Entry<K, V>	MutableMap.MutableEntry<K, V>	(Mutable)Map.(Mutable)Entry<K, V>!

Mapped types - example

Java type	Kotlin type
java.lang.Byte	kotlin.Byte?
java.lang.Short	kotlin.Short?
java.lang.Integer	kotlin.Int?
java.lang.Long	kotlin.Long?
java.lang.Character	kotlin.Char?
java.lang.Float	kotlin.Float?
java.lang.Double	kotlin.Double?
java.lang.Boolean	kotlin.Boolean?

Java type	Kotlin type
int[]	kotlin.IntArray!
String[]	kotlin.Array<(out) String>!

Java generics in Kotlin

- Java's wildcards are converted into type projections,
 - `Foo<? extends Bar>` becomes `Foo<out Bar!>`!
 - `Foo<? super Bar>` becomes `Foo<in Bar!>`!
- Java's raw types are converted into star projections,
 - `List` becomes `List<*>`!, i.e. `List<out Any?>`!
- `ArrayList<Integer>()` is indistinguishable from `ArrayList<Character>()`

```
if (set is List<Int>){} // Error: cannot check if it is really a List of Ints
if (set is List<*>){}  // OK: no guarantees about the contents of the list
```

Java Arrays

- Arrays in Kotlin are invariant (무공변성)
Array<String> - Array<Any> : 아무 연관이 타입
- Passing an array of a subclass as an array of superclass to a Kotlin method is also prohibited, but for Java methods this is allowed

Java Arrays – Covariance and contravariance?

- Integer는 Number를 상속받아 만들어진 객체입니다.
그래서 Integer는 Number의 하위 타입이라고 할 수 있어 아래와 같은 코딩이 가능합니다.

```
public void test() {  
    List<Number> list;  
    list.add(Integer.valueOf(1));  
}
```

- 하지만 List<Integer>는 List<Number>의 하위타입이 될 수 없습니다.
이러한 상황에서 Java나 Kotlin에서는 type parameter에 타입 경계를 명시하여 Sub-Type, Super-Type을 가능하게 해줍니다.
그걸 **변성**이라고 합니다.

경계	bound	kotlin	java
상위 경계	Upper bound	Type<out T>	Type<? extends T>
하위 경계	Lower bound	Type<in T>	Type<? super T>

Java Arrays – Covariance and contravariance?

	의미
공변성(covariant)	T'가 T의 서브타입이면, C<T'>는 C<T>의 서브타입이다.
반공변성(contravariant)	T'가 T의 서브타입이면, C<T>는 C<T'>의 서브타입이다.
무공변성(invariant)	C와 C<T'>는 아무 관계가 없다.

참고자료: <https://velog.io/@lsb156/covariance-contravariance>

Java Arrays – 공변성 (Covariant)

```
interface Cage<T> {  
    fun get(): T  
}  
  
open class Animal  
  
open class Hamster(var name: String) : Animal()  
  
class GoldenHamster(name: String) : Hamster(name)  
  
fun tamingHamster(cage: Cage<out Hamster>) {  
    println("길들이기 : ${cage.get().name}")  
}
```

```
fun main() {  
    val animal = object : Cage<Animal> {  
        override fun get(): Animal {  
            return Animal()  
        }  
    }  
  
    val hamster = object : Cage<Hamster> {  
        override fun get(): Hamster {  
            return Hamster("Hamster")  
        }  
    }  
  
    val goldenHamster = object : Cage<GoldenHamster> {  
        override fun get(): GoldenHamster {  
            return GoldenHamster("Leo")  
        }  
    }  
}  
  
tamingHamster(animal) // compile Error  
tamingHamster(hamster)  
tamingHamster(goldenHamster)  
}
```

- tamingHamster 함수는 Hamster의 서브타입만을 받기때문에 animal 변수는 들어갈 수 없습니다.

Java Arrays – 반공변성 (Covariant)

```
interface Cage<T> {  
    fun get(): T  
}  
  
open class Animal  
  
open class Hamster(var name: String) : Animal()  
  
class GoldenHamster(name: String) : Hamster(name)  
  
fun tamingHamster(cage: Cage<out Hamster>) {  
    println("길들이기 : ${cage.get().name}")  
}
```

```
fun main() {  
    val animal = object : Cage<Animal> {  
        override fun get(): Animal {  
            return Animal()  
        }  
    }  
    val hamster = object : Cage<Hamster> {  
        override fun get(): Hamster {  
            return Hamster("Hamster")  
        }  
    }  
    val goldenHamster = object : Cage<GoldenHamster> {  
        override fun get(): GoldenHamster {  
            return GoldenHamster("Leo")  
        }  
    }  
  
    tamingHamster(animal)  
    tamingHamster(hamster)  
    tamingHamster(goldenHamster) // compile Error  
}
```

- 공변성의 반대 개념으로 자기 자신과 부모 객체만을 허용합니다.
- ancestorOfHamster에서 햄스터의 조상을 찾는 함수를 구현하여 햄스터를 포함한 그 조상들만 허용하도록 제한하였습니다. 하위타입인 Cage<GoldenHamster>는 제한에 걸려있어 compile error가 나는것을 확인 할 수 있습니다.

Java Arrays – 무공변성 (invariant)

```
interface Cage<T> {  
    fun get(): T  
}  
  
open class Animal  
  
open class Hamster(var name: String) : Animal()  
  
class GoldenHamster(name: String) : Hamster(name)  
  
fun tamingHamster(cage: Cage<out Hamster>) {  
    println("길들이기 : ${cage.get().name}")  
}
```

```
fun main() {  
    val animal = object : Cage<Animal> {  
        override fun get(): Animal {  
            return Animal()  
        }  
    }  
    val hamster = object : Cage<Hamster> {  
        override fun get(): Hamster {  
            return Hamster("Hamster")  
        }  
    }  
    val goldenHamster = object : Cage<GoldenHamster> {  
        override fun get(): GoldenHamster {  
            return GoldenHamster("Leo")  
        }  
    }  
  
    tamingHamster(animal) // compile Error  
    tamingHamster(hamster) // compile Error  
    tamingHamster(goldenHamster)  
}
```

- 위 코드의 Cage<Animal>, Cage<Hamster>, Cage<GoldenHamster>는 서로 각각 연관이 없는 객체로서 무공변성의 적절한 예입니다.

Java Arrays

- 박싱 / 언박싱 오퍼레이션 비용을 없애기 위해 자바 플랫폼의 기본 제이터 타입을 가진 배열을 사용한다.
- 이런 경우를 다루기 위해 모든 기본 타입 배열을 위한 특수 클래스 `IntArray`, `DoubleArray`, `CharArray` 를 제공하고 있다.
- 특수 클래스들은 `Array` 클래스와 상관 없으며 최대 성능을 위해 자바의 기본 타입 배열로 컴파일 된다.
 - 오버헤드가 생길일이 적다

```
public class JavaArrayExample {  
    public void removeIndices(int[] indices) {  
        // code here...  
    }  
}
```

```
val javaObj = JavaArrayExample()  
val array = intArrayOf(0, 1, 2, 3)  
javaObj.removeIndices(array) // passes int[] to method
```

Java Varargs

```
public class JavaVarargsExample {  
    public void removeIndicesVarArg(int... indices) {  
        // ...  
    }  
}
```

```
fun main() {  
    val javaObj = JavaVarargsExample()  
    val array = intArrayOf(0, 1, 2, 3)  
    javaObj.removeIndicesVarArg(*array)  
}
```

Use **spread operator (*)** to pass the IntArray

It's currently not possible to pass null to a method that is declared as varargs.

Operators

- 자바는 연산자 구문을 사용하기에 알맞은 메서드 표시 방법이 없이 때문에, 코틀린은 연산자 오버로딩과 다른 규칙(`invoke()` 등)에 맞는 이름과 시그니처를 가진 자바 메서드를 허용한다
- Since Java has no way of marking methods for which it makes sense to use the operator syntax, Kotlin allows using any Java methods with the right name and signature as operator overloads and other conventions (`invoke()` etc.)



Operators

- 코틀린에서 중위 호출 구문을 사용해서 자바 메서드를 호출하는 것은 허용하지 않는다

Operators – 중위 호출 구문

```
val map = mapOf(1 to "one", 2 to "two", 3 to "three")
```

to는 코틀린 키워드가 아니라 중위 호출 이라는 특별한 방식으로 to라는 일반 메서드를 호출한 것이다.

중위 호출 시에는 수신 객체와 유일한 메서드 인자 사이에 메서드 이름을 넣는다
(객체, 메서드 이름, 유일한 인자 사이에는 공백이 들어가야 한다)

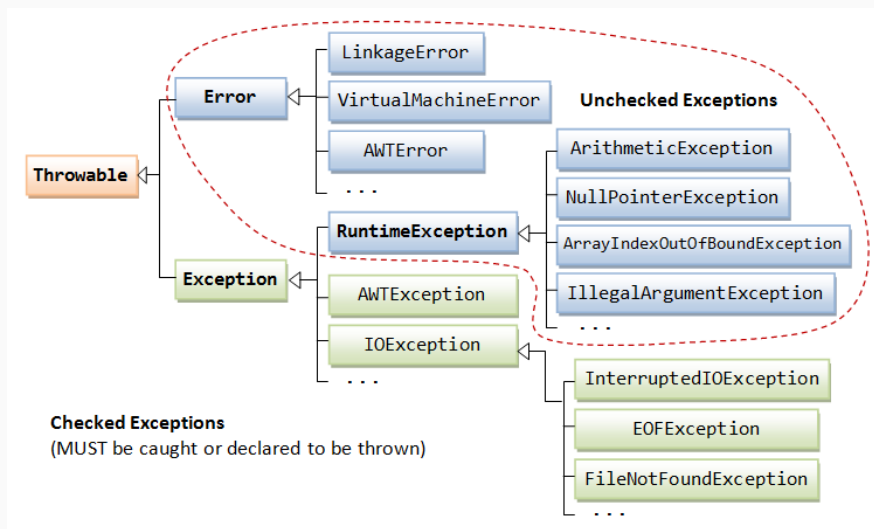
```
public infix fun <A, B> A.to(that: B): Pair<A, B> = Pair(this, that)
```

인자가 하나뿐인 일반 메서드나, 인자가 하나 뿐인 확장 함수에 중위 호출을 사용할 수 있다.
함수를 중위 호출을 사용하게 허용하고 싶으면 infix 변경자를 함수 선언 앞에 추가해야 한다.
다음은 to 함수의 정의를 간략하게 줄인 코드이다.

Checked Exceptions

- 코틀린에서 모든 예외는 unchecked exception 이다.
- 이는 컴파일러가 예외 처리(catch) 를 강제하지 않는다는 의미이다.

Checked Exceptions vs Unchecked Exception (Runtime Exception)



Error

시스템에 비정상적인 상황이 생겼을 때 발생

System Level 에서 발생하는
심각한 수준의 오류

-> System 에서 Error 에 대한 처리를 함

Checked

Unchecked

Exception

- 반드시 예외를 처리 해야 한다
- 컴파일 단계에서 확인 가능
- 반드시 예외를 처리할 필요는 없다
- 컴파일 단계에서 확인 불가능

Checked Exceptions

Compile Error

```
public class Exception {  
    public static void main(String[] args) {  
        FileReader fileReader = new FileReader("test.txt");  
    }  
}  
  
public FileReader(String fileName) {  
    super(new FileInputStream(fileName));  
}
```

Unhandled exception: java.io.FileNotFoundException

[Add exception to method signature](#)

[More actions...](#)

[java.io.FileReader](#)


public **FileReader**(@NotNull String fileName)
throws [java.io.FileNotFoundException](#)

Creates a new `FileReader`, given the name of the file to read, using the platform's [default charset](#).

Params: fileName – the name of the file to read

Throws: [java.io.FileNotFoundException](#) – if the named file does not exist, is a directory rather than a regular file, or for some other reason cannot be opened for reading.

External annotations: [@org.jetbrains.annotations.NotNull](#)

 < 15 >

Checked Exceptions – Exception Handling

Not Recommended

```
public class Exception {  
    public static void main(String[] args) throws FileNotFoundException {  
        FileReader fileReader = new FileReader("test.txt");  
    }  
}
```

Recommended

```
public class Exception {  
    public static void main(String[] args) {  
        try {  
            FileReader fileReader = new FileReader("test.txt");  
        } catch (FileNotFoundException e) {  
            // ...  
        }  
    }  
}
```

Unchecked Exceptions

No Compile Error

```
public class Exception {  
    public static void main(String[] args) {  
        int[] array = new int[3];  
        System.out.println(array[10]);  
    }  
}
```

Runtime

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
Index 10 out of bounds for length 3 at Exception.main(Exception.java:4)

Object Methods

Java.lang.Object -> Any

Java.lang.Object

public open operator fun equals(other: Any?): Boolean
public open fun hashCode(): Int
public open fun toString(): String

Any

public final Class getClass()
public int hashCode()
public boolean equals(Object obj)
protected Object clone() throws CloneNotSupportedException
public String toString()
public final void notify()
public final void notifyAll()
public final void wait(long timeout) throws InterruptedException
public final void wait(long timeout, int nanos) throws InterruptedException
public final void wait() throws InterruptedException
protected void finalize() throws Throwable

Object Methods

wait() notify()

(foo as java.lang.Object).wait()

getClass()

val fooClass = foo::class.java // bound class reference
val fooClass = foo.javaClass // extension property

clone()

to override clone()

class needs to extend kotlin.Cloneable

```
class Example : Cloneable {  
    override fun clone(): Any {  
        return Any()  
    }  
}
```

finalize()

override finalize()

Without using the override keyword

Finalize() must not be private

```
class C {  
    protected fun finalize() {  
        // finalization logic  
    }  
}
```

Inheritance from Java classes

- 코틀린에서 최대 한 개의 자바 클래스를 상속할 수 있다.
 - 클래스는 다중 상속 불가능
- 자바 인터페이스는 필요한 만큼 상속 가능

Accessing static members

- 자바 클래스의 정적 멤버는 그 클래스를 위한 "컴페니언 오브젝트"로 구성한다. 그 "컴페니언 오브젝트"를 값으로 전달할 수 없지만, 멤버에 접근할 수는 있다

```
class Character implements java.io.Serializable, Comparable<Character>, Constable {  
    public static boolean isLetter(char ch) {  
        return isLetter((int)ch);  
    }  
}
```

```
if (Character.isLetter(a)) {  
    // ...  
}
```

Java Reflection

- Java reflection 은 코틀린 클래스에 대해서도 동작하며, 반대로도 동작한다
- 자바 getter/setter 메서드 또는 코틀린 프로퍼티를 위한 지원 필드 구하기, 자바 필드를 위한 KProperty 구하기, KFunction 을 위한 자바 메서드 나 생성자 구하기 또는 그 반대 구하기 등을 지원한다.

SAM Conversions

- SAM 변환은 인터페이스에만 동작한다. 추상 클래스가 단지 1개의 추상 메서드만 갖고 있다 해서 추상 클래스에는 동작하지 않는다.
- SAM 변환은 자바에서 작성한 인터페이스일 때만 동작한다. 코틀린에서는 인터페이스 대신에 함수를 사용하는 것이 좋습니다.

SAM Conversions – example

View class

```
public class View {  
    public interface OnClickListener {  
        public void onClick(View v);  
    }  
}
```

Button class

```
public class Button {  
    public void setOnClickListener(View.OnClickListener listener) {  
        // ...  
    }  
}
```

General

```
val button: Button = Button()  
button.setOnClickListener(object: View.OnClickListener {  
    override fun onClick(v: View?) {  
        // ...  
    }  
})
```

SAM Conversions – example

SAM Conversion

```
button.setOnClickListener({v: View? ->
    // ...
})
```

Lambda parameter

```
button.setOnClickListener(){v: View? ->
    // ...
}
```

Only lambda parameter

```
button.setOnClickListener{v:View? ->
    // ...
}
```

Data type inferred

```
button.setOnClickListener{v ->
    // ...
}
```

Parameter in lambda: it

```
button.setOnClickListener {
    // it.visibility = View.GONE
}
```

Using JNI with Kotlin

```
external fun foo(x: Int): Double
```

```
var myProperty: String  
    external get  
    external set
```

- 네이티브(C나 C++) 코드로 구현한 함수를 선언하려면 external 수식어를 지정해야 한다
- 프로시저의 나머지는 자바와 똑같이 동작한다.

Calling Kotlin from Java

Properties

Kotlin `var firstName: String`

`var isXXX: Boolean`

Java

```
private String firstName;  
public String getFirstName() {  
    return firstName;  
}  
public void setFirstName(String firstName) {  
    this.firstName = firstName;  
}
```

```
private Boolean isXXX;  
public Boolean isXXX() {  
    return isXXX;  
}  
public void setXXX(Boolean isXXX) {  
    this.isXXX = isXXX;  
}
```

val -> getter
var -> getter, setter

Package-level functions

Kotlin

```
// example.kt
package demo
class Foo
fun bar() {
    // ...
}
```

```
// example.kt
@file:JvmName("DemoUtils")
package demo
class Foo
fun bar() {
    // ...
}
```

Java

```
new demo.Foo();
demo.ExampleKt.bar();
```

```
new demo.Foo();
demo.DemoUtils.bar();
```

Top level function 은 Java 클래스의 정적 메서드로 컴파일 됩니다

Package-level functions

Kotlin
@JvmName

```
// example.kt
@file:JvmName("Utils")
@file:JvmMultifileClass
package demo
class Foo
fun foo() {
    // ...
}
```

```
// newutils.kt
@file:JvmName("Utils")
@file:JvmMultifileClass
package demo
fun bar() {
}
```

Java

```
public static void main(String[] args) {
    new demo.Foo();
    demo.Utils.foo();
    demo.Utils.bar();
}
```

Without @JvmMultifileClass
Every kotlin file needs it!

Duplicate JVM class name 'demo/Utils' generated from:
package-fragment demo, package-fragment demo

Instance fields

Kotlin

```
class User(id: String) {  
    val ID = id  
}
```

Java

```
class JavaClient {  
    public String getID(User user) {  
        return user.ID;  
    }  
}
```

'ID' has private access in 'User'

Add 'lateinit var' property 'ID' to 'User'

More actions...

User

public final val ID: String

untitled4.main

Instance fields

Kotlin

```
class User(id: String) {  
    @JvmField val ID = id  
}
```

Java

```
class JavaClient {  
    public String getID(User user) {  
        return user.ID;  
    }  
}
```

@JvmField

- Not open
- Not override
- Not const
- Not a delegated property

Static fields

1. @JvmField
2. lateinit
3. const

```
class Key(val value: Int) {  
    companion object {  
        @JvmField  
        val COMPARATOR: Comparator<Key> = compareBy<Key> { it.value }  
    }  
}
```

```
// 자바  
Key.COMPARATOR.compare(key1, key2);  
// Key 클래스에 public static final 필드
```

Static methods

```
class C {  
    companion object {  
        @JvmStatic fun foo() {}  
        fun bar() {}  
    }  
}
```

C.foo(); // 동작
C.bar(); // 예러: 정적 메서드 아님
C.Companion.foo(); // 인스턴스 메서드 유지
C.Companion.bar(); // 오직 이렇게만 동작

```
object Obj {  
    @JvmStatic fun foo() {}  
    fun bar() {}  
}
```

Obj.foo(); // 동작
Obj.bar(); // 예러
Obj.INSTANCE.bar(); // 동작, 싱글톤 인스턴스를 통해 호출
Obj.INSTANCE.foo(); // 역시 동작

- @JvmStatic 을 컴패니언 오브젝트나 이름 가진 오브젝트에 붙이면 정적 메서드를 생성한다

Visibility

Kotlin	Java
Private 멤버	Private 멤버
Private 최상위 선언	패키지-로컬 선언
protected	protected
internal	public
public	public

```
kotlin.jvm.JvmClassMappingKt.getKotlinClass(MainView.class);
```

- 자바의 Class 에서 코틀린의 KClass 로 변환하는 법?

Overloads generation

```
class Foo @JvmOverloads constructor(x: Int, y: Double = 0.0) {  
    @JvmOverloads fun f(a: String, b: Int = 0, c: String = "abc") {  
        ...  
    }  
}
```

// 생성자:

```
Foo(int x, double y)  
Foo(int x)
```

// 메서드

```
void f(String a, int b, String c) { }  
void f(String a, int b) { }  
void f(String a) { }
```

- 디폴트 파라미터 값을 갖는 코틀린 함수 -> @JvmOverloads 를 사용해야 가능

Checked exceptions

```
// example.kt
package demo
fun foo() {
    throw IOException()
}
```

```
// Java
try {
    demo.Example.foo();
}
catch (IOException e) {
    // 예러: foo()는 throws 목록에 IOException을 선언하지 않았음
    // ...
}
```

코틀린에는 checked exception 이 없다.

```
// example.kt
package demo

@Throws(IOException::class)
fun foo() {
    throw IOException()
}
```

Null-safety

- 자바에서 코틀린 함수를 호출할 때 non-null 파라미터에 null 을 전달하는 것을 막을 수 없다.
- 이것이 코틀린이 non-null을 기대하는 모든 public 함수 에 대해 런타임 검사를 생성하는 이유이다.
- 이 방법은 자바 코드에서 즉시 NullPointerException 을 발생하게 만든다

Translation of type Nothing

- Nothing 타입은 자바에 알맞은 비슷한 것이 없다.
- 사실 java.lang.Void 를 포함한 모든 자바 레퍼런스 타입은 값으로 null 을 가질 수 있는데 Nothing 은 그것조차 안 된다.
- 따라서, 이 타입을 자바에서 정확하게 표현할 수 없다. 이것이 코틀린이 Nothing 타입의 인자를 사용할 때 raw 타입 을 생성하는 이유이다.

```
fun emptyList(): List<Nothing> = listOf()
```

```
// 자바에서는 다음으로 해석
```

```
// List emptyList() { ... }
```

Thank you!