Contenido

[1.- Definición de Herencia 3](#_Toc156919140)

[2.- Clase Any 3](#_Toc156919141)

[3.- Sintaxis De Herencia 3](#_Toc156919142)

[3.1.- Constructor Primario En Herencia 4](#_Toc156919143)

[3.2.- Constructor Secundario En Herencia 4](#_Toc156919144)

[4.- Sobrescribir Métodos (Polimorfismo) 4](#_Toc156919145)

[5.- Sobrescribir Propiedades 5](#_Toc156919146)

[6.- Clases abstractas 5](#_Toc156919147)

[7.- Interfaces 6](#_Toc156919148)

[8.- Para que utilizarlos las clases abstractas y las interfaces 9](#_Toc156919149)

[8.1.- Clases abstractas: 9](#_Toc156919150)

[8.2.- Interfaces: 9](#_Toc156919151)

[9.- Ejemplos 9](#_Toc156919152)

[9.1.- Ejemplo 9](#_Toc156919153)

[9.2.- Ejemplo 9](#_Toc156919154)

[9.3.- Ejemplo 10](#_Toc156919155)

[9.4.- Ejemplo polimórfico. 10](#_Toc156919156)

[9.5.- Ejemplo(Clase abstracta) 11](#_Toc156919157)

[9.6.- Ejemplo (interfaces) 12](#_Toc156919158)

[10.- Manejo de Excepciones 12](#_Toc156919159)

[10.1.- Ejemplo de tipos de excepciones 13](#_Toc156919160)

[10.2.- Ejemplos 13](#_Toc156919161)

[12.- Control de excepciones 14](#_Toc156919162)

[12.1.- Errores en tiempo de compilación y en tiempo de ejecución 14](#_Toc156919163)

[12.2.- Manejo de Excepciones 15](#_Toc156919164)

[12.2.1.- Excepciones No Revisadas 15](#_Toc156919165)

[12.2.2.- Excepciones Revisadas en Java 16](#_Toc156919166)

[13.- Excepciones en Kotlin 18](#_Toc156919167)

[13.1.- Bloque try...catch 19](#_Toc156919168)

[14.- Java Interop 20](#_Toc156919169)

# 1.- Definición de Herencia

La herencia es un mecanismo de la Programación Orientada a Objetos, para diseñar dos o más entidades que son diferentes, pero comparten muchas características en común.

El proceso consiste en definir una clase conocida como: clase base, clase padre, superclase o ancestro; que contenga todas las características en común.

Luego defines las clases que heredan las características, denominadas como subclases, clases hijas, clases derivadas o clases descendientes.

# 2.- Clase Any

La clase Any es la raíz de la jerarquía de clases en Kotlin. Cada clase en el lenguaje derivará de ella si no especificas una superclase.

Creamos la clase ejemplo:

class Ejemplo // Hereda de Any

Si vas a la declaración de Any.kt desde IntelliJ IDEA, verás que tiene 3 métodos:

equals(): Indica si otro objeto es igual al actual

hashCode(): Retorna el código hash asociado al objeto (es un identificador de 32 bits que se almacena en un Hash en la instancia de la clase)

toString(): Retorna la representación en String del objeto

Por ejemplo, ejecuta los métodos anteriores con dos instancias de Ejemplo:

fun main() {

val ejemplo1 = Ejemplo()

val ejemplo2 = Ejemplo()

println(ejemplo1.toString())

println(ejemplo1.hashCode())

println(ejemplo1.equals(ejemplo2))

}

# 3.- Sintaxis De Herencia

Añade el modificador **open** para habilitar su capacidad de herencia

Añade dos puntos para expresar en la cabecera, que hereda de otra clase

Añade los paréntesis al final del nombre de la superclase para especificar la llamada a su constructor.

open class Ancestro

class Descendiente: Ancestro()

## 3.1.- Constructor Primario En Herencia

Si la superclase tiene constructor primario, debes inicializarlo pasando los parámetros en la llamada de la sintaxis de herencia.

open class Ancestro(val propiedad:Boolean)

class Descendiente(propiedad: Boolean) : Ancestro(propiedad)

## 3.2.- Constructor Secundario En Herencia

Si la clase base no tiene constructor primario o deseas realizar la llamada del mismo, desde un constructor secundario de la subclase, entonces usa el constructor junto a la palabra reservada super para la llamada.

open class Weapon(val damage: Int, val speed: Double)

class Bow : Weapon {

constructor(damage: Int, speed: Double) : super(damage, speed)

}

# 4.- Sobrescribir Métodos (Polimorfismo)

Aplicar polimorfismo con la sobrescritura de métodos en Kotlin, requiere habilitar el método con el modificador open. Luego usa el modificador override desde el método que se ha convertido en polifórmico de la subclase.

El polimorfismo es la capacidad de un objeto para ser tratado como un objeto de su tipo padre o de un tipo de interfaz que implementa. Esto permite a los objetos tener diferentes comportamientos basados en su tipo real en tiempo de ejecución.

open class Character(val name: String) {

open fun die() = println("Morir")

}

class Mage(name: String) : Character(name) {

override fun die() = println("Mago muriendo")

}

fun main() {

val jaina = Mage("Jaina")

jaina.die()

}

# 5.- Sobrescribir Propiedades

Similar a la sobreescritura de métodos, sobrescribir una propiedad requiere usar el modificador override sobre la propiedad en la subclase.

Si la propiedad está declarada con val en la clase padre, es posible reescribirla como var en la clase hija. Sin embargo, lo contrario no es posible.

open class BaseItem(val name: String) {

open var quantity = 1

}

class PopularItem(name: String) : BaseItem(name) {

override var quantity = 6

}

fun main() {

val notebook = BaseItem("Cuaderno")

val pencil = PopularItem("Lapicero")

println("${notebook.name} x ${notebook.quantity}")

println("${pencil.name} x ${pencil.quantity}")

}

# 6.- Clases abstractas

Son clases de las cuales no se pueden crear objetos (es una clase que no puede ser instanciada y debe ser extendida por otras clases). Una clase abstracta es incompleta o inútil sin algunas subclases concretas (no abstractas), desde las cuales puede instanciar objetos. Una subclase concreta de una clase abstracta implementa todos los métodos y propiedades definidas en la clase abstracta; de lo contrario, esa subclase también es una clase abstracta.

abstract class Employee (val firstName: String, val lastName: String) {

abstract fun earnings(): Double

}

No todos los miembros tienen que ser abstractos. En otras palabras, podemos tener la implementación predeterminada del método en una clase abstracta.

abstract class Employee (val firstName: String, val lastName: String) {

// ...

fun fullName(): String {

return lastName + " " + firstName;

}

}

Ejemplo, la clase Programmer extiende la clase abstracta Employee. En Kotlin usamos un solo carácter de dos puntos (:).

class Programmer(firstName: String, lastName: String) : Employee(firstName, lastName) {

override fun earnings(): Double {

// calculate earnings

}

}

# 7.- Interfaces

Una interfaz es simplemente una colección de métodos relacionados que normalmente le permiten decirles a los objetos qué hacer y también cómo hacerlo de forma predeterminada. (Los métodos predeterminados en las interfaces son una nueva característica agregada a Java 8). En otras palabras, una interfaz es un contrato que las clases de implementación deben cumplir.

interface StudentRepository {

fun getById(id: Long): Student

fun getResultsById(id: Long): List<Result>

}

En el código anterior, hemos declarado una interfaz de StudentRepository. Esta interfaz contiene dos métodos abstractos: getById() y getResultsById(). Incluir la palabra clave abstract es redundante en un método de interfaz porque ya son implícitamente abstractos.

Una interfaz es inútil sino se implementa clase a partir de él, así que vamos a crear una clase que implementará esta interfaz.

class StudentLocalDataSource : StudentRepository {

override fun getResults(id: Long): List<Result> {

// do implementation

}

override fun getById(id: Long): Student {

// do implementation }}

## Reglas de interfaces en Kotlin:

Una clase puede implementar tantas interfaces como desee, pero solo puede heredar de una sola clase (similar a Java).

El modificador override es obligatorio en Kotlin, a diferencia de Java.

Junto con los métodos, también podemos declarar propiedades en una interfaz Kotlin.

Un método de interfaz Kotlin puede tener una implementación predeterminada (cuando decimos implementar, queremos decir que puede tener código escrito dentro de la función, no solamente el nombre y los argumentos)(similar a Java 8).

Como se indicó, una interfaz Kotlin puede tener propiedades, pero no puede mantener el estado. (Sin embargo, recuerde que las clases abstractas pueden mantener el estado). Por lo tanto, la siguiente definición de interfaz con una declaración de propiedad funcionará.

interface StudentRepository {

val propFoo: Boolean // will work

// ...

}

Pero si intentamos agregar algún estado a la interfaz asignando un valor a la propiedad, no funcionará.

interface StudentRepository {

val propFoo: Boolean = true // Error: Property initializers are not allowed in interfaces

// ..

}

Sin embargo, una propiedad de interfaz en Kotlin puede tener métodos de obtención y establecimiento (aunque solo este último si la propiedad es una variable)..

interface StudentRepository {

var propFoo: Boolean

get() = true

set(value) {

if (value) {

// do something

}

}

// ...

}

También podemos anular una propiedad de la interfaz si lo desea, para redefinirla.

class StudentLocalDataSource : StudentRepository {

// ...

override var propFoo: Boolean

get() = false

set(value) {

if (value) {

}

}

}

Veamos un caso en el que tenemos una clase que implementa múltiples interfaces con el mismo nombre de método. ¿Cómo decide la clase a qué método de interfaz llamar?

interface InterfaceA {

fun funD() {}

}

interface InterfaceB {

fun funD() {}

}

Aquí tenemos dos interfaces que tienen un método con la misma nombre funD(). Vamos a crear una clase que implemente estas dos interfaces y reemplace el método funD().

class classA : InterfaceA, InterfaceB {

override fun funD() {

super.funD() // Error: Many supertypes available, please specify the one you mean in angle brackets, e.g. 'super<Foo>'

}

}

El compilador está confundido acerca de llamar al método super.funD() porque las dos interfaces que implementa la clase tienen la misma firma de método.

Para resolver este problema, envolvemos el nombre de la interfaz para la que queremos llamar al método entre paréntesis angulares <InterfaceName>.

class classA : InterfaceA, InterfaceB {

override fun funD() {

super<InterfaceA>.funD()

}

}

Aquí vamos a llamar al método funD() de InterfaceA.

# 8.- Para que utilizarlos las clases abstractas y las interfaces

## 8.1.- Clases abstractas:

Cuando quieres crear una clase base común que proporcione funcionalidad básica para las clases hijas, pero no queremos instanciar objetos de la clase base.

Cuando quieres obligar a las clases hijas a implementar ciertos métodos o propiedades.

Cuando quieres crear una clase que represente un concepto genérico y luego especializarlo en las clases hijas.

## 8.2.- Interfaces:

Cuando quieres definir un comportamiento que se debe implementar en múltiples clases.

Cuando quieres que una clase implemente más de una interfaz.

Cuando quieres crear un contrato que obligue a las clases a implementar ciertos métodos o propiedades.

La elección entre clases abstractas e interfaces depende de la situación específica y de los requisitos de diseño.

# 9.- Ejemplos

## 9.1.- Ejemplo

open class Animal {

open fun makeSound() {

println("El animal hace un sonido")

}

}

class Dog: Animal() {

override fun makeSound() {

println("El perro ladra")

}

}

final class Cat: Animal() {

override fun makeSound() {

println("El gato maulla")

}

}

La clase "Animal" es marcada como "open" para que se pueda heredar. La clase "Dog" hereda de "Animal" y sobreescribe la función "makeSound". La clase "Cat" también hereda de "Animal" y sobreescribe la función "makeSound", pero se marca como "final" para evitar la herencia adicional.

## 9.2.- Ejemplo

open class Vehicle {

var speed = 0

fun increaseSpeed() {

speed += 10

}

}

class Car: Vehicle() {

var color = "Red"

}

final class Bike: Vehicle() {

var weight = 20

}

## 9.3.- Ejemplo

open class Grandfather {

open var name: String = "Abuelo"

var age: Int = 80

open fun talk() {

println("Hola, soy el abuelo y tengo ${age} años")

}

}

class Father: Grandfather() {

var job: String = "Doctor"

override var name: String = "Padre"

override fun talk() {

println("Hola, soy el padre, trabajo como ${job} y tengo ${age} años")

}

}

class Son: Father() {

var hobby: String = "Jugar videojuegos"

override fun talk() {

println("Hola, soy el hijo, mi hobby es ${hobby} y tengo ${age} años")

}

}

## 9.4.- Ejemplo polimórfico.

open class Animal {

open fun makeSound() {

println("Hace sonido de animal")

}

}

class Dog : Animal() {

override fun makeSound() {

println("Guau")

}

}

class Cat : Animal() {

override fun makeSound() {

println("Miau")

}

}

fun main() {

val animals = arrayOf(Dog(), Cat(), Animal())

for (animal in animals) {

animal.makeSound()

}}

## 9.5.- Ejemplo(Clase abstracta)

abstract class Person {

abstract val name: String

abstract val age: Int

fun sayHello() {

println("Hello, my name is $name and I am $age years old.")

}

}

class Student(override val name: String, override val age: Int, val studentId: String) : Person() {

fun attendClass() {

println("$name is attending class with student ID $studentId.")

}

}

class Teacher(override val name: String, override val age: Int, val subject: String) : Person() {

fun teach() {

println("$name is teaching $subject.")

}

}

## 9.6.- Ejemplo (interfaces)

interface Person {

val name: String

val age: Int

fun sayHello() {

println("Hello, my name is $name and I am $age years old.")

}

}

class Student(override val name: String, override val age: Int, val studentId: String) : Person {

fun attendClass() {

println("$name is attending class with student ID $studentId.")

}

}

class Teacher(override val name: String, override val age: Int, val subject: String) : Person {

fun teach() {

println("$name is teaching $subject.")

}

}

# 10.- Manejo de Excepciones

Todas las clases de excepción en Kotlin son descendientes de la clase Throwable. Cada excepción tiene un mensaje, seguimiento de pila y una causa opcional.

Para lanzar un objeto excepción, se usa la expresión throw:

fun main() {

throw Exception("Hola")

}

Para detectar una excepción, use la expresión try:

try {// algún código

} catch (e: SomeException) {// handler

} finally {// bloque finalmente opcional

}

El bloque catch recoge la excepción y la trata y el bloque finally que es opcional se ejecuta después de catch y antes de terminar la función, suele utilizarse para cerrar elementos que podrían quedar vulnerables al producirse la excepción, como pueden ser ficheros, conexiones…

## 10.1.- Ejemplo de tipos de excepciones

|  |  |
| --- | --- |
| **Nombre** | **Descripcion** |
| FileNotFoundException | Lanza una excepción cuando el fichero no se encuentra. |
| ClassNotFoundException | Lanza una excepción cuando no existe la clase. |
| EOFException | Lanza una excepción cuando llega al final del fichero. |
| ArrayIndexOutOfBoundsException | Lanza una excepción cuando se accede a una posición de un array que no exista. |
| NumberFormatException | Lanza una excepción cuando se procesa un numero pero este es un dato alfanumérico. |
| NullPointerException | Lanza una excepción cuando intentando acceder a un miembro de un objeto para el que todavía no hemos reservado memoria. |
| IOException | Generaliza muchas excepciones anteriores. La ventaja es que no necesitamos controlar cada una de las excepciones. |
| Excepcion | Es la clase padre de IOException y de otras clases. Tiene la misma ventaja que IOException. |

## 10.2.- Ejemplos

fun main() {

try {

val result = divide(10, 0)

println("El resultado de la división es $result")

} catch (e: ArithmeticException) {

println("¡No se puede dividir entre cero!")

} finally {

println("Esto siempre se imprime, sin importar si hubo una excepción o no.")

}

}

fun divide(a: Int, b: Int): Int {

return a / b

}

fun main() {

val array = arrayOf(1, 2, 3)

try {

println(array[3])

} catch (e: ArrayIndexOutOfBoundsException) {

println("¡Error! El índice está fuera del rango del array.")

}

}

fun main() {

val numberString = "3.14"

try {

val number = numberString.toInt()

println("El número es $number")

} catch (e: NumberFormatException) {

println("¡Error! La cadena no se puede convertir a un número entero.")

} finally {

println("Esto siempre se imprime, sin importar si hubo una excepción o no.")

}

}

# 12.- Control de excepciones

## 12.1.- Errores en tiempo de compilación y en tiempo de ejecución

Programando en Java o Kotlin se pueden producir diferentes tipos de error. El más común es el que se produce en tiempo de compilación y generalmente ocurre cuando hay alguna sentencia que no está bien escrita, por ejemplo, si falta un punto y coma al final de una línea en Java, cuando hay comillas o paréntesis que se abren, pero no se cierran, al intentar asignar una cadena de caracteres a una variable que es de tipo entero, etc.

Si un programa da errores en tiempo de compilación, no se crea el archivo de bytecode con la extensión .class y, por tanto, hasta que no se arreglen los fallos, no compilará y, en consecuencia, tampoco se podrá ejecutar.

La siguiente línea de código provocará un error en tiempo de compilación

System.out.println("¡Hola mundo!); (Java)

Println ("¡Hola mundo!) (Kotlin)

Tenemos un típico ejemplo de error en tiempo de compilación.

Existen otros errores que se producen en tiempo de ejecución a los que llamaremos excepciones. El programa compila y se puede ejecutar, pero, por algún motivo, se produce un fallo y el programa se “ROMPE”. Debemos identificar estas situaciones y se deben controlar en el programa.

import java.util.Scanner

fun main() {

val scanner = Scanner(System.`in`)

println("Este programa calcula la media de dos números")

print("Introduzca el primer número: ")

val numero1 = scanner.nextLine().toDouble()

print("Introduzca el segundo número: ")

val numero2 = scanner.nextLine().toDouble()

println("La media es ${calcularMedia(numero1, numero2)}")

}

fun calcularMedia(numero1: Double, numero2: Double): Double {

return (numero1 + numero2) / 2

}

Un programa que pide dos números y realiza la división, esta utilizando la clase Scanner de Java, en lugar de las clases de Kotlin para leer por teclado. Este programa funciona y compila, pero si en lugar de introducir un número, se escribe texto el programa se “ROMPE” y produce una salida no deseada, ni controlada.

## 12.2.- Manejo de Excepciones

Las excepciones son usadas para indicar un problema en nuestro código durante una ejecución del programa. Es la capacidad de abordar (o manejar) la excepción que podría ocurrir. Si no manejamos cualquier excepción que ocurra, nuestro programa detendrá abruptamente la ejecución, cerrando nuestra aplicación inmediatamente.

El manejo de excepciones permite a nuestro programa continuar la ejecución incluso si hubo una excepción.

Para entender las excepciones en Kotlin debemos entenderlas en Java, tenemos dos clases de excepciones: revisadas y no revisadas.

### 12.2.1.- Excepciones No Revisadas

Estas son excepciones que son creadas debido a fallas en el código. Son una subclase directa o indirecta de la superclase **RuntimeException**.

Los ejemplos de excepciones no revisadas incluyen:

• ArithmeticException: lanzada cuando divides entre cero.

• ArrayIndexOutOfBoundExceptions: arrojada cuando un arreglo ha sido accedido con un índice ilegal.

• SecurityException: Arrojada por el administrador de seguridad para indicar una violación de seguridad.

• NullPointerException: arrojada cuando se invoca un método o propiedad de un objeto nulo.

Un método que podría crear una excepción no revisada.

fun divideByZero(numerator: Int, denominator: Int): Int {

try {

return numerator / denominator

} catch (e: ArithmeticException) {

println("Error: ${e.message}")

return 0 // O cualquier valor predeterminado que desees devolver en caso de división por cero

}

}

fun main() {

val result = divideByZero(7, 0)

println("El resultado es: $result")

}

Estos tipos de excepción pueden ser corregidas, codificando correctamente. En el código de arriba, deberíamos haber revisado si el denominador era cero antes de realizar la operación. Para estas excepciones, el desarrollador no necesita atrapar la excepción usando el bloque try...catch. En otras palabras, no estamos forzados por el compilador a encerrar el código que podría disparar la excepción en un bloque try...catch. En su lugar, solo deberíamos asegurar que las excepciones nunca ocurran.

### 12.2.2.- Excepciones Revisadas en Java

Un método que podría devolver una excepción revisada necesita declararlo en su firma de método usando la palabra clave throws. Si llamas a un método que crea una excepción revisada, necesitarás lanzarla de nuevo desde tu función o atraparla y manejarla usando un bloque try...catch.

Las excepciones revisadas son excepciones que son revisadas en tiempo de compilación. Este tipo de excepciones heredan de la clase Exception. Como ejemplo de este tipo de excepción está IOException. Esto puede ocurrir cuando intentas acceder a un archivo que no puede ser abierto porque no existe. (FileNotFoundException es una subclase de IOException).

public void editFile(File file, String text) {

try {

file.getParentFile().mkdirs();

FileOutputStream fileOutputStream = new FileOutputStream(file);

Writer writer = new BufferedWriter(new OutputStreamWriter(fileOutputStream));

try {

writer.write(text);

writer.flush();

fileOutputStream.getFD().sync();

} finally {

writer.close();

}

} catch (IOException e) {

// Log the exception

e.printStackTrace();

}

}

En el código anterior, usamos un bloque try...catch para manejar la IOException dentro del método editFile(). Ahora podemos llamar al método editFile() de manera normal, y el compilador no se generará un error.

editFile(new File(""), "my text");

Se ha refactorizado (Volver a escribir) el método para usar la palabre clave throws en la creación del método. Esto indica a la función que llama que necesitan manejar la excepción IOException que podría ser creada cuando se llama al método editFile().

public void editFile(File file, String text) **throws** IOException {

file.getParentFile().mkdirs();

FileOutputStream fileOutputStream = new FileOutputStream(file);

Writer writer = new BufferedWriter(new OutputStreamWriter(fileOutputStream));

try {

writer.write(text);

writer.flush();

fileOutputStream.getFD().sync();

} finally {

writer.close();

}

}

Para llamar al método de arriba, necesitamos encerrarlo en un bloque try...catch para manejar la excepción.

try {

editFile(new File(""), "my text");

} catch (IOException e) {

e.printStackTrace();

}

# 13.- Excepciones en Kotlin

La principal diferencia entre los mecanismos de excepciones de Kotlin y Java es que **todas las excepciones son no revisadas en Kotlin**. En otras palabras, no están declaradas de manera explícita en las firmas de función, como lo están en Java.

fun editFile(file: File, text: String) {

file.parentFile.mkdirs()

val fileOutputStream = FileOutputStream(file)

val writer = BufferedWriter(OutputStreamWriter(fileOutputStream))

try {

writer.write(text)

writer.flush()

fileOutputStream.fd.sync()

} finally {

writer.close()

}

}

Hemos convertido el método editFile() a una función Kotlin. Puedes ver que la función no tiene la declaración throws IOException en su declaración de función, throws no es siquiera una palabra clave en Kotlin.

También, podemos llamar a esta función sin implementar el bloque try...catch y el compilador no se genera un error. Todas las excepciones son no revisadas.

Si creemos que esta excepción podría producirse, deberíamos implementar el método con un bloque try...catch, pero esto no es obligatorio para el compilador Kotlin.

try {

editFile(File(""), "text 123")

} catch (e: IOException) {

e.printStackTrace()

}

Si la excepción se produce dentro de la función editFile() se creará una instancia de la clase IOException, nuestro bloque catch será ejecutado, y simplemente imprimiremos la pila para la depuración del código.

## 13.1.- Bloque try...catch

El constructor try con cláusulas catch y finally en Kotlin es similar al de Java.

fun foo() {

try {

throw Exception("Exception message")

} catch (e: Exception) {

println("Exception handled")

} finally {

println("inside finally block")

}

}

Estamos creando un objeto Exception dentro del bloque try. Manejamos todas las subclases de tipo Exception en el bloque catch. El bloque opcional finally siempre se ejecuta, es aquí donde típicamente cerramos cualquier recurso o conexiones que fueron abiertas previamente para prevenir fugas de recurso. Por ejemplo, si abres un archivo o creas una base de datos o conexión de red en un bloque try, deberías cerrarlo o liberarlo en un bloque finally.

En Kotlin el constructor throw es una expresión y puede combinarse con otras expresiones.

val letter = 'c'

val result =

if (letter in 'a'..'z')

letter

else

throw IllegalArgumentException("A letter must be between a to z")

También, el constructor try puede ser usado como una expresión.

fun foo(number: Int) {

val result = try {

if (number != 1) {

throw IllegalArgumentException()

}

true

} catch (e: IllegalArgumentException) {

false

}

println(result)

}

foo(2) // false

Asignamos el valor devuelto desde el bloque try...catch a la variable result. Si el número no es 1, este arroja IllegalArgumentException y el bloque catch es ejecutado. La expresión false en el valor del bloque catch será asignado a la variable result. Si el número es 1 en su lugar, entonces el valor de expresión true será asignado a la variable result.

# 14.- Java Interop

Las excepciones en Kotlin se comportan de manera normal en Java, existe una notación útil llamada @Throws en Kotlin. Debido a que todas las excepciones en Kotlin son no revisadas, los desarrolladores que utilizan tu código Kotlin desde Java podrían no ser conscientes de que tus funciones lanzan excepciones. Se puede alertar con la anotación @Throw que necesitan manejar la excepción.

Veamos un ejemplo práctico de esta anotación.

fun addNumberToTwo(a: Any): Int {

if (a !is Int) {

throw IllegalArgumentException("Number must be an integer")

}

return 2 + a

}

Definimos una función Kotlin que puede crear una excepción IllegalArgumentException solo si el tipo pasado a la función no es de tipo Int.

Nosotros llamamos a esta función de nivel superior addNumberToTwo() directamente desde Java de la siguiente manera:

public void myJavaMethod() {

Integer result = FunctionsKt.addNumberToTwo(5);

System.out.println(result); // 7

}

Esto funciona bien; el compilador no detecta la posible excepción. Sin embargo, si queremos comunicar con en Java que la función de nivel superior addNumberToTwo() crea una excepción, simplemente agregamos la anotación @Throws a la creación de la función.

**@Throws(IllegalArgumentException::class)**

fun addNumberToTwo(a: Any): Int {

if(a !is Int) {

throw IllegalArgumentException("Number must be an integer")

}

return 2 + a

}

Esta anotación @Trows puede aceptar una lista separada por comas de argumentos de excepción de clases. En el código de arriba, solo incluimos una clase de excepción (IllegalArgumentException).

Ahora tenemos que actualizar nuestro código Java para manejar la excepción.

public void myJavaMethod() throws IllegalArgumentException {

Integer result = FunctionsKt.addNumberToTwo(5);

System.out.println(result);

}