

Apuntes de [Kotlin]

Compilar el código: *\$ kotlinc name.kt -include-runtime -d name.jar*

Ejecutar el programa: *\$ java -jar name.jar*

Ejecutar la consola 'REPL': *\$ kotlinc-jvm*

Usar la línea de comandos para ejecutar scripts (.kts): *\$ kotlinc -script name.kts [params]*

Compilar una biblioteca sin la 'runtime' para ser usada en otros programas: *\$ kotlinc name.kt -d name.jar*

Ejecutar binarios producidos por el compilador de Kotlin: *\$ kotlin -classpath name.jar HelloKt*
(*HelloKt is the main class name inside the file named name.kt*)

Sintaxis básica



El punto de entrada en un programa escrito en Kotlin (y en Java) es la función **'main'**. Esta función recibe un array que contiene los argumentos de la línea de comandos. Las funciones y variables en Kotlin pueden declararse en un "nivel superior", es decir, directamente dentro de un paquete.

```
fun main(args: Array<String>) {
    println("Hello World!")
}
```

Si un archivo Kotlin contiene una sola clase (potencialmente con declaraciones de nivel superior relacionadas), su nombre debe ser el mismo que el nombre de la clase, con la extensión '.kt'. Si un archivo contiene varias clases, o solo declaraciones de nivel superior, el nombre debe describir lo que contiene el archivo en formato *'UpperCamelCase'* (e.g. **ProcessDeclarations.kt**)

Kotlin sigue las convenciones de nomenclatura de Java. Los nombres de los paquetes se escriben siempre en minúsculas y sin guiones bajos (e.g. **org.example.myproject**)

Los nombres de las clases y los objetos se escriben en *'UpperCamelCase'*:

```
open class DeclarationProcessor { ... }

object EmptyDeclarationProcessor : DeclarationProcessor() { ... }
```

Los nombres de funciones, propiedades y variables locales en *'lowerCamelCase'*:

```
fun processDeclarations() { ... }

var declarationCount = ...
```

Los nombres de las constantes (propiedades marcadas con *const*) deben usar nombres en mayúsculas y separados por un guión bajo:

```
const val MAX_COUNT = 8
val USER_NAME_FIELD = "UserName"
```

Variables y tipos básicos

En Kotlin, **todo es un objeto** en el sentido de que podemos llamar funciones y propiedades de miembro en cualquier variable. Algunos de los tipos como los números, los caracteres o los booleanos pueden tener una representación interna especial que se representa como valores primitivos en tiempo de ejecución, pero para el usuario se comportan como clases ordinarias.

La declaración de valores se realiza utilizando **var** o **val**. Las variables declaradas como **val** son inmutables o 'read-only', es decir, no se pueden reasignar, mientras que las **var** son mutables y por tanto se le pueden asignar un nuevo valor pero únicamente del mismo tipo declarado. La recomendación es crear variables inmutables, que son más seguras en entornos '*multithreading*' ya que no se pueden modificar y utilizar las variables mutables cuando sea necesario.

```
val fooVal = 10 // val es inmutable y no se podrá ser reutilizada
val otherVal
otherVal = "My Value" // Podemos declarar la variable 'val' en una línea y asignarle
                        // valor posteriormente. Sigue siendo una sola asignación.
var fooVar = 10
fooVar = 20 // 'fooVar' se puede le puede asignar un nuevo valor pero únicamente
            // del mismo tipo.
```

En la mayoría de los casos, Kotlin puede determinar o inferir cuál es el tipo de una variable, por lo que no tenemos que especificarla explícitamente. Cuando la variable no se inicialice deberemos indicar explícitamente el tipo de la variable.

```
val foo: Int = 7
val bar = 10 // Kotlin infiere automáticamente el tipo
val hello = "Hello"
```

Kotlin proporciona los siguientes tipos que representan números:

```
val double: Double = 64.0
val float: Float = 32.0F // or 32f

val long: Long = 64L
val int: Int = 32
val short: Short = 16
```

```
val byte: Byte = 8
val hexadecimal: Int = 0x16
val binary: Int = 0b101
val char: Char = 'a'
```

A diferencia de Java, en Kotlin **todos los tipos son objetos** y por tanto no hay *'wrappers'* u objetos envoltorio tipo *Integer*, *Double*, etc...

Los caracteres no son números en Kotlin, a diferencia de Java. Los literales de cadena se escriben con comillas simples *'1'*. Los caracteres especiales se escapan con la barra invertida *'\'*

Los guiones bajos se pueden utilizar para hacer que los números grandes sean más legibles:

```
val million = 1_000_000
```

La conversión debe ser invocada explícitamente. Hay conversiones desde un tipo al resto de tipos:

```
val otherLong = int.toLong()
val direct = 25.toLong()
```

Cadenas

Las cadenas son secuencias de caracteres **inmutables** y se representan con el tipo *String* de manera similar a Java. Las cadenas se crean usando las comillas dobles. El escapado de caracteres se hace con una barra invertida *'\'*.

```
val fooString = "My String Is Here!"
val barString = "Printing on a new line?\nNo Problem!"
val bazString = "Do you want to add a tab?\tNo Problem!"
println(fooString)
println(barString)
println(bazString)
println("John Doe"[2]) // => h
println("John Doe".startsWith("J")) // => true
```

Se puede acceder a los elementos de una cadena como si fuera un array (e.g. *s[i]*) e iterar con bucle tipo *for*:

```
for (c in str) {
    println(c)
}
```

Se puede utilizar el operador `+` para concatenar cadenas entre sí y con valores de otro tipo siempre y cuando uno de los elementos de la expresión sea una cadena:

```
val s = "abc" + 1
println(s + "def")
```

Una cadena sin formato está delimitada por una comilla triple ("""). Las cadenas sin formato pueden contener nuevas líneas y cualquier otro carácter:

```
val fooRawString = """
fun helloWorld(val name : String) {
    println("Hello, world!")
}
"""
```

Las cadenas pueden contener expresiones de plantilla o *'template expressions'*. Una expresión de plantilla comienza con un signo de dólar (`$`). Estas *template expressions* son una forma simple y efectiva de incrustar valores, variables o incluso expresiones dentro de una cadena que serán evaluadas y cuyo resultado se concatenará a la cadena resultante.

```
val name = "John Doe"
println("$name has ${name.length} characters") // => John Doe has 8 characters

val age = 40
println("You are ${if (age > 60) "old" else "young"}") // => You are young
```

Packages

La palabra clave *package* funciona de la misma manera que en Java. El nombre del paquete se usa para construir el "Fully Qualified Name" (FQN) de una clase, objeto, interfaz o función.

Los nombres de los *'packages'* se escriben en minúscula y sin guiones bajos.

```
package com.example.kotlin

class MyClass

fun saySomething(): String {
    return "How far?"
}
```

El FQN de la clase será '*com.example.kotlin.MyClass*'. Dado que podemos tener 'top-level functions' como en el ejemplo, el FQN de la función será '*com.example.kotlin.saySomething*'.

Si no se especifica un paquete, el contenido del fichero fuente pertenece al paquete '**default**'.

Imports

En Kotlin, usamos la declaración de importación para permitir que el compilador localice las clases e interfaces, propiedades, enumeraciones, funciones y objetos que se importarán.

En Java, por otro lado, solo está permitido importar clases o interfaces.

```
// 'Bar' esta disponible en el código
import foo.Bar

// Si existe cierta ambigüedad podemos usar la palabra clave 'as'
import foo.Bar
import bar.Bar as bBar

// Todo el contenido de 'foo' está disponible
import foo.*
```

Por defecto, al igual que en Java, el compilador importa de forma implícita una serie de paquetes y por tanto están disponibles de forma automática.

Comentarios

```
// Single-line comments start with //

/*
Multi-line comments look like this.
*/
```

Control de flujo y bucles

Un bucle **for** puede usarse con cualquier elemento que proporcione un iterador como rangos, colecciones, etc...:

```
for (c in "hello") {  
    println(c)  
}  
  
for (i in 1..3) {  
    println(i)  
}  
  
for (i in 6 downTo 0 step 2) {  
    println(i)  
}
```

Los bucles **while** y **do-while** funcionan de la misma manera que en otros lenguajes:

```
while (x > 0) {  
    x--  
}  
  
do {  
    val y = retrieveData()  
} while (y != null) // y is visible here!
```

La instrucción **if** y **if..else** funciona igual que en Java. Además, en Kotlin los bloques **if** se pueden utilizar como una expresión que devuelve un valor. Por este motivo el operador ternario '*condition ? then: else*' no es necesario en Kotlin:

```
// Traditional usage  
var max = a  
if (a < b) max = b  
  
// With else  
var max: Int  
if (a > b) {  
    max = a  
} else {  
    max = b  
}
```

```
// As expression
val max = if (a > b) a else b
```

Los bloques *when* se pueden usar como una alternativa a las cadenas *if-else-if*:

```
when {
  x.isOdd() -> print("x is odd")
  x.isEven() -> print("x is even")
  else -> print("x is funny")
}
```

La instrucción *when* se puede usar con un argumento. Si ninguna de las opciones coincide con el argumento, se ejecuta la opción del bloque *else*:

```
when (i) {
  0, 21 -> println("0 or 21")
  in 1..20 -> println("in the range 1 to 20")
  !in 22..25 -> println("not in the range 22 to 25")
  else -> println("none of the above")
}
```

La instrucción *when* se puede utilizar como una expresión que devuelve un valor. En este caso el bloque *else* es obligatorio.

```
val result = when (i) {
  0, 21 -> "0 or 21"
  in 1..20 -> "in the range 1 to 20"
  else -> "none of the above"
}
println(result)
```

Se pueden utilizar expresiones arbitrarias, y no solo constantes, como condiciones en los bloques:

```
when (x) {
  parseInt(s) -> print("s encodes x")
  else -> print("s does not encode x")
}
```


Arrays

Una matriz está representada por la clase **Array** y es **invariante**, por lo que, por ejemplo, no se puede asignar un **Array<String>** a un tipo de variable **Array<Any>**.

En Kotlin, podemos crear una matriz de elementos del mismo tipo o distinto utilizando la función de biblioteca **arrayOf()**:

```
val cardNames = arrayOf("Jack", "Queen", "King", 3, false)
println(cardNames[1]) // => Queen
```

Podemos forzar la creación de arrays del mismo tipo. De esta forma el compilador comprobará y evitará que se añadan elementos de otro tipo.

```
val myArray = arrayOf<Int>(1, 2, 3, 4)
println(myArray.contentToString()) // => [1, 2, 3, 4]
```

La biblioteca estándar de Kotlin provee funciones para crear arrays de tipos primitivos como **'intArrayOf()'**, **'LongArrayOf()'**, **'charArrayOf()'**, **'doubleArrayOf()'**, etc... Cada una de estas funciones devuelven una instancia de su equivalente en Kotlin como **IntArray**, **LongArray**, **CharArray**, **DoubleArray**, etc....:

```
val cards = intArrayOf(10, 11, 12) // IntArray
println("${cards[1]}") // => 11
```

Para mejorar la eficiencia y rendimiento del código, cuando se utilicen tipos primitivos hay que utilizar las funciones **'intArrayOf()'**, **'LongArrayOf()'**, etc.. en vez de **'arrayOf()'** para así evitar el coste asociado a las operaciones de 'boxing'/unboxing'.

Alternativamente, podemos crear una matriz a partir de un tamaño inicial y una función, que se utiliza para generar cada elemento usando el constructor **'Array()'**:

```
val allCards = Array(12, { i -> i + 1 })
println("${allCards.first()} - ${allCards.last()}") // => 1 - 12
```

Iterando sobre la matriz con **indices**:

```
for (index in cardNames.indices) {  
    println("Element $index is ${cardNames[index]}")  
}
```

Otra forma posible de iterar es usando '*withIndex()*':

```
for ((index, value) in cardNames.withIndex()) {  
    println("$index - $value")  
}
```

Funciones



Las funciones se declaran usando la palabra clave *'fun'*. Los nombres de las funciones empiezan con minúscula. Los parámetros de la función se especifican entre paréntesis después del nombre de la función y tienen la forma *'name: type'*. El tipo de cada parámetro debe especificarse explícitamente.

```
fun powerOf(number: Int, exponent: Int) { ... }
```

Los parámetros de la función pueden tener opcionalmente un valor por defecto, que se utilizará en caso de se omita el argumento al invocar la función. El tipo de retorno de la función, si es necesario, se especifica después de los parámetros:

```
fun hello(name: String = "world"): String { // valor por defecto
    return "Hello, $name!"
}

hello("foo") // => Hello, foo!
hello(name = "bar") // => Hello, bar!
hello() // => Hello, world!

fun bye(bye: String = "Bye", name: String): String {
    return "$bye, $name!!"
}

bye(name = "John", bye = "Good bye") // => Good bye, John!!
bye(name = "John") // => Bye, John!!
```

En la sobrescritura de métodos con valores por defecto siempre se utilizan los mismos valores de parámetros por defecto que el método base. Cuando se sobrescribe un método, los valores por defecto deben omitirse de la firma:

```
open class A {
    open fun foo(i: Int = 10) { ... }
}
```

```
class B : A() {
    override fun foo(i: Int) { ... } // no default value allowed
}
```

Si un parámetro por defecto precede a un parámetro sin valor predeterminado, el valor por defecto solo se puede usar llamando a la función con argumentos con nombre:

```
fun foo(bar: Int = 0, baz: Int) { ... }

foo(baz = 1) // The default value bar = 0 is used
```

Dado que Java no admite valores de parámetros por defecto en los métodos, deberá especificar todos los valores de parámetros explícitamente cuando llame a una función de Kotlin desde Java. Kotlin nos proporciona la funcionalidad para facilitar las llamadas de Java al anotar la función Kotlin con `'@JvmOverloads'`. Esta anotación le indicará al compilador de Kotlin que genere las funciones sobrecargadas de Java para nosotros.

```
@JvmOverloads
fun calcCircumference(radius: Double, pi: Double = Math.PI): Double = (2 * pi) *
radius

// En Java
double calcCircumference(double radius, double pi);
double calcCircumference(double radius);
```

Cuando una función no devuelve ningún valor significativo, su tipo de devolución es `Unit` por defecto. En ese caso es opcional indicar el tipo de retorno. El tipo `Unit` es un objeto en Kotlin que es similar a los tipos `void` en Java y C.

```
fun hello(name: String): Unit {
    print("Hello $name")
}

fun sayHello(name: String) { // compila ya que el compilador infiere el tipo
    'Unit'
    print("Hello $name")
}
```

Los parámetros con nombre permiten código más legible al nombrar los parámetros que se pasan a una función cuando se invoca. Una vez que se utiliza un nombre en un parámetro, el resto de parámetros también deben asignarse con nombre:

```

fun area(width: Int, height: Int): Int {
    return width * height
}

area(10, 12)
area(width = 10, height = 12) // código más legible
area(height = 12, width = 10) // podemos cambiar el orden
area(10, height = 12) // argumento por posición y argumentos con nombre
area(width = 10, 12) // ¡incorrecto! no se permiten argumentos con nombre antes de
argumentos por posición

fun bar(k: Int, m: Long = 1L, j: Boolean = true) = println("$k - $m - $j")

// Una vez que un parámetro ha sido nombrado, todos los siguientes parámetros
deben ser nombrados
bar(10) // => Se omiten los parámetros por defecto
bar(15, 30L)
bar(20, 2L, true)
bar(m = 30L, j = false, k = 10)
bar(k = 10, m = 20L, j = true)
bar(5, m = 2L, j = true)
bar(6, 1L, j = true)

```

Cuando se invoca una función con argumentos posicionales y con nombre, todos los argumentos posicionales deben colocarse antes del primero argumento con nombre. Por ejemplo, la llamada `f(1, y = 2)` está permitida, pero `f(x = 1, 2)` no está permitida.

Para pasar un número variable de argumentos a una función podemos usar la palabra clave `'vararg'` delante del nombre de una variable. Por tanto la función aceptará una lista de parámetros separados por comas que el compilador envolverá en una array. Por tanto, dentro de la función accederemos a los parámetros mediante la notación de array.

Este tipo de parámetros se puede combinar con otros parámetros. Normalmente el parámetro `'vararg'` será el último de la lista. Si hay otros parámetros después de `'vararg'`, deberán usarse parámetros con nombre:

```

fun varargExample(vararg names: Int) {
    println("Argument has ${names.size} elements")
}

varargExample() // => Argument has 0 elements
varargExample(1) // => Argument has 1 elements
varargExample(1, 2, 3) // => Argument has 3 elements

fun car(vararg model: String, year: Int) {}

car("Audi", "A6", year = 2005) // parámetros con nombre después de 'vararg'

```

Para utilizar un array para suministrar un número variable de argumentos se utiliza el operador `'*'` también llamado *'spread operator'* delante del nombre de la variable del array:

```
val intArray = intArrayOf(1, 2, 3, 4)
val array = Array(5, { i -> i + 1 })
varargExample(*intArray) // => Argument has 4 elements
varargExample(*array.toIntArray()) // => Argument has 5 elements
```

Cuando una función consiste en una sola expresión, se pueden omitir los paréntesis. El cuerpo se especifica después de un símbolo `'='`:

```
fun odd(x: Int): Boolean = x % 2 == 1
```

Declarar explícitamente el tipo de retorno de una función cuando es una expresión es opcional cuando puede ser inferido por el compilador o cuando el tipo de retorno es *'Unit'*. Cuando el cuerpo de una función es un bloque hay que especificar el tipo de retorno ya que el compilador no puede inferirlo:

```
fun even(x: Int) = x % 2 == 0 // Optional

fun printHello(name: String?) { // 'Unit'
    if (name != null)
        println("Hello ${name}")
    else
        println("Hi there!")
    // `return Unit` or `return` is optional
}
```

A veces queremos devolver múltiples valores desde una función. Una forma es usar el tipo *'Pair'* de Kotlin. Esta estructura incluye dos valores a los que luego se puede acceder. Este tipo de Kotlin puede aceptar cualquier tipo que suministre a su constructor. Y, lo que es más, los dos tipos ni siquiera necesitan ser iguales. Kotlin también provee el tipo *'Triple'* que retorna tres valores:

```
fun getNumbers(num: Int): Pair<Int?, Int?> {
    require(num > 0, { "Error: num is less than 0" })
    return Pair(num, num * 2)
}

val(num, num2) = getNumbers(10) // destructuring
```

En Kotlin, podemos hacer que la creación de una instancia *'Pair'* sea más compacta y legible utilizando la función *'to'*, que es una función *'infix'* en lugar del constructor de *'Pair'*.

```
val nigeriaCallingCodePair = 234 to "Nigeria"
val nigeriaCallingCodePair2 = Pair(234, "Nigeria") // Same as above
```

Extension functions

Las **funciones de extensión** o *'extension functions'* son una forma de agregar nuevas funcionalidades a una clase sin tener que heredar de dicha clase. Esto es similar a los métodos de extensión de C#. Una función de extensión se declara fuera de la clase que quiere extender. En otras palabras, también es una función de nivel superior o *'top-level function'*. Junto con las funciones de extensión, Kotlin también admite propiedades de extensión.

Para crear una *'extension function'*, debe prefijar el nombre de la clase que está extendiendo antes del nombre de la función. El nombre de la clase o el tipo en el que se define la extensión se denomina **tipo de receptor**, y el **objeto receptor** es la instancia de clase o el valor concreto sobre el que se llama a la función de extensión.

```
fun String.remove(c: Char): String { // 'String' es el tipo receptor
    return this.filter { it != c }    // 'this' corresponde al objeto receptor
}

println("Hello, world!".remove('l')) // => Heo, world! // "Hello World" es el
objeto receptor
```

En caso de que una *'extension function'* tenga la misma firma (mismo nombre y misma lista de parámetros) que una función miembro, es decir, una función de la clase, el compilador invocará antes la función miembro que la función de extensión aunque no se generará ningún error de compilación:

```
class C {
    fun foo() { println("member") }
}

fun C.foo() {
    println("extension")
}

fun C.foo(i: Int) {
    println("extension & overridden")
}
```

```

}

C().foo() // => member
C().foo(5) // => extension & overridden

```

Top-level functions

Las funciones de nivel superior son funciones que se definen fuera de cualquier clase, objeto o interfaz. Esto significa que son funciones a las que llama directamente, sin la necesidad de crear ningún objeto o llamar a ninguna clase. Dado que Java no soporta este tipo de funciones el compilador de Kotlin genera una clase con métodos estáticos. Estas tipo de funciones son especialmente útiles para crear funciones de utilidad o de ayuda.

```

// Code defined inside a file called UserUtils.kt
@file:JvmName("UserUtils") // Definir el nombre del fichero
package com.chikekotlin.projectx.utils

fun checkUserStatus(): String {
    return "online"
}

```

Functions types & Lambdas

Un tipo función es un tipo que consta de una firma de función, es decir, dos paréntesis que contiene la lista de parámetros (que son opcionales) y un tipo de retorno de función. Ambas partes están separados por el operador `'->'`.

```

fun executor(action:() -> Unit) {
    action()
}

// 'action' es el nombre del parámetro y su tipo es '() -> Unit' que es una
función.
// Por tanto el tipo de 'action' es un tipo función.

```

- Ejemplo de un tipo función que no toma parámetros y devuelve 'Unit': `() -> Unit`
- Ejemplo de un tipo función que no toma parámetros y devuelve un String: `() -> String`
- Ejemplo de un tipo función que toma un String y no devuelve nada: `(String) -> Unit`

- Ejemplo de un tipo función que toma dos parámetros y no devuelve nada: `(String, Float) -> Unit`

Debido a que un tipo función es solo un tipo, significa que puede asignar una función a una variable, puede pasarla como un argumento a otra función o puede devolverla desde una función tal y como suceden en las *'high-order functions'*:

```
val morning: (String) -> Unit = { x -> println(x) }
morning("good morning") // => good morning
```

Función *'lambda'* con dos parámetros:

```
val sum: (Int, Int) -> Int = { x, y -> x + y }
sum(10, 20) // => 30
```

Si una función *'lambda'* tiene solo un parámetro, su declaración puede omitirse (junto con `->`). El nombre del único parámetro será *'it'*.

```
val isNegative: (Int) -> Boolean = { it < 0 } // este literal es del tipo '(it: Int) -> Boolean'
isNegative(-5) // => true
```

Para parámetros no utilizados se utiliza el operador `'_'`:

```
val unusedSecondParam: (String, Int) -> Boolean = { s, _ ->
    s.length > 10
}
unusedSecondParam("Hello World", 0) // 0 is unused
```

Anonymous functions

Una función anónima se parece mucho a una declaración de función normal, excepto que se omite su nombre. Su cuerpo puede ser una expresión o un bloque:

```
// Función anónima cuyo cuerpo es una expresión
fun(x: Int, y: Int): Int = x + y

// Función anónima con bloque
```

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

El tipo de los parámetros de una función anónima pueden omitirse si se pueden inferir por el contexto:

```
ints.filter(fun(item) = item > 0)
```

La inferencia de tipo de retorno para funciones anónimas funciona igual que para las funciones normales: el tipo de retorno se deduce automáticamente para funciones anónimas con un cuerpo de expresión y debe especificarse explícitamente (o se supone que es *'Unit'*) para funciones anónimas con un cuerpo de bloque.

High-Order Functions

Una *'high-order function'* o **función de orden superior** es una función que puede tomar funciones como parámetros y/o devolver una función como tipo de retorno.

```
// Función con dos parámetros, uno de ellos es una función
fun foo(str: String, fn: (String) -> String): Unit { // 2nd param is a function
    val applied = fn(str)
    println(applied)
}
foo("Hello", { it.reversed() }) // => olleH

// Esta función de orden superior devuelve una función
fun isPositive(n: Int): (Int) -> Boolean {
    return { n > 0 } // return a function. In other words, instead 'return value'
we have 'return { function }'
}

// Esta función de orden superior devuelve una función de forma más compacta
fun modulo(k: Int): (Int) -> Boolean = { it % k == 0 }

val evens = listOf(1, 2, 3, 4, 5, 6).filter(modulo(2)) // => [2, 4, 6]

// Asignar la función a una variable
val isEven: (Int) -> Boolean = modulo(2)

listOf(1, 2, 3, 4).filter(isEven) // => [2, 4]
listOf(5, 6, 7, 8).filter(isEven) // => [6, 8]
```

El siguiente ejemplo de función de orden superior acepta una función lambda `{ (String) -> Boolean }` como parámetro. Se expresa como "acepta una función 'from String to Boolean'":

```
// El parámetro 'email' podemos usarlo como una función que acepta una cadena y
// devuelve un booleano.
fun isAnEmail(email: (String) -> Boolean) {
    email("myemail@example.com")
}
isAnEmail({ s: String -> s.contains("@") }) // forma completa
isAnEmail { s: String -> s.contains("@") } // Los paréntesis son opcionales
isAnEmail { it.contains("@") } // Uso de 'it'
```

Para pasar una función como parámetro de otra función especificamos su nombre con el operador `:::` y sin utilizar los paréntesis:

```
fun businessEmail(s: String): Boolean {
    return s.contains("@") && s.contains("business.com")
}
isAnEmail(::businessEmail) // Invocar una 'high-order function' pasándole otra
// función por su nombre
```

En Kotlin, hay una convención de que si el último parámetro de una función acepta una función, una expresión 'lambda' que se pasa como el argumento correspondiente se puede colocar fuera de los paréntesis:

```
// lambda expression inside parentheses
val upperCaseLetters = "Hello World".filter({ it.isUpperCase() })

// lambda outside parentheses
val lowerCaseLetters = "Hello World".filter { it.isLowerCase() }

println("$upperCaseLetters - $lowerCaseLetters") // => HW - elloorld
```

Closures

Un *'closure'* es una función que tiene acceso a variables y parámetros que se definen en un ámbito externo. A diferencia de Java, las variables 'capturadas' pueden ser modificadas.

```
fun printFilteredNamesByLength(length: Int) {
    val names = arrayListOf("Adam", "Andrew", "Chike", "Kechi")
```

```

    val filterResult = names.filter {
        it.length == length    // 'length' se define fuera del ámbito de la
    lambda
    }
    println(filterResult)
}

```

Local or Nested Functions

Para llevar más lejos la modularización de programas, Kotlin nos proporciona funciones locales, también conocidas como funciones anidadas o *'nested functions'*. **Una función local es una función que se declara dentro de otra función.**

Podemos hacer que nuestras funciones locales sean más concisas al no pasarles parámetros explícitamente. Esto es posible porque las funciones locales tienen acceso a todos los parámetros y variables de la función de cierre.

```

fun printCircumferenceAndArea(radius: Double): Unit {

    fun calcCircumference(radius: Double): Double = (2 * Math.PI) * radius
    val circumference = "%.2f".format(calcCircumference(radius))

    fun calcArea(radius: Double): Double = (Math.PI) * Math.pow(radius, 2.0)
    val area = "%.2f".format(calcArea(radius))

    print("The circle circumference of $radius radius is $circumference and area
is $area")
}

```

Infix Functions

Las funciones marcadas con la palabra clave *'infix'* se pueden llamar usando la notación *'infix'* (omitiendo el punto y los paréntesis para la llamada). Estas funciones deben cumplir los siguientes requisitos:

- Tienen que ser miembros de una clase o funciones de extensión
- Deben tener un solo parámetro
- Este parámetro no será *'vararg'* ni tener valor por defecto

Para invocar una función *'infix'* en Kotlin no necesitamos usar la notación de puntos ni los paréntesis. Hay que tener en cuenta que las funciones *'infix'* siempre requieren que se

especifiquen tanto el receptor como el parámetro. Cuando se invoca un método en el receptor actual, como por ejemplo dentro de la clase, se necesita usar explícitamente la notación *'this'*. A diferencia de las llamadas a métodos regulares, no se puede omitir.

```
class Student {
    var kotlinScore = 0.0

    infix fun addKotlinScore(score: Double): Unit {
        this.kotlinScore = kotlinScore + score
    }

    fun build() {
        this addKotlinScore 95.0    // Correcto
        addKotlinScore(95.0)       // Correcto
        addKotlinScore 95.0        // Incorrecto: hay que especificar el receptor
        ('this')
    }
}

val student = Student()
student addKotlinScore 95.00 // Invocando la función usando la notación 'infix'
student.addKotlinScore(95)  // Invocando la función con notación normal
```

Inline functions

El compilador de Kotlin crea una clase anónima en versiones anteriores de Java cuando creamos o utilizamos expresiones lambda. Esto genera una sobrecarga, además de la carga de memoria que se genera cuando en una función lambda hace uso de variables de fuera de su entorno como en las *'closures'*.

Para evitar esta sobrecarga tenemos el modificador *'inline'* para las funciones. Una *'High-Order function'* con el modificador *'inline'* se integrará durante la compilación del código. En otras palabras, el compilador copiará la 'lambda' (o función literal) y también el cuerpo de la función de orden superior y los pegará en el sitio de la llamada.

Con este mecanismo, nuestro código se ha optimizado significativamente, no más creación de clases anónimas o asignaciones de memoria extra. Por otro lado el uso de *'inline'* hace que el compilador genere archivos bytecode más grandes. Por esta razón, se recomienda encarecidamente que solo se incluyan funciones de orden superior más pequeñas que acepten lambda como parámetros.

Clases y objetos



Clases

Las clases son los bloques de construcción principales de cualquier lenguaje de programación orientado a objetos. Para definir una clase se usa la palabra clave **'class'**.

```
class Invoice { ... }
```

La declaración de clase consiste en el nombre de la clase, el encabezado de la clase (especificando sus parámetros de tipo, el constructor primario, etc.) y el cuerpo de clase, rodeado de llaves. Tanto el encabezado como el cuerpo son opcionales. Si la clase no tiene cuerpo se pueden omitir las llaves. Si no se especifica visibilidad, la visibilidad por defecto es **public** y por tanto cualquiera puede crear instancias de dicha clase.

```
class Empty
```

En comparación con Java, puede definir varias clases dentro del mismo archivo fuente.

Las clases pueden contener:

- Constructores y bloques **'init'**
- Funciones
- Propiedades
- Clases anidadas e internas
- Declaraciones de tipo **'object'**

Constructores

Una clase en Kotlin puede tener un **constructor primario** y uno o más **constructores secundarios**.

El constructor primario es parte del encabezado de la clase. Este constructor va después del nombre de la clase (y los parámetros de tipo que son opcionales). Por defecto, todos los

constructores son públicos, lo que equivale efectivamente a que sean visible en todas partes donde la clase sea visible.

```
class Person constructor(firstName: String) { ... }
```

Si el constructor principal no tiene anotaciones o modificadores de visibilidad, la palabra clave **'constructor'** se puede omitir:

```
// Podemos omitir la palabra clave 'constructor'
class Person(firstName: String) { ... }

// Las anotaciones o modificadores de visibilidad requieren la palabra clave
'constructor'
class Customer public @Inject constructor(name: String) { ... }
```

Si una clase no-abstracta no declara ningún constructor (primario o secundario), tendrá un constructor primario sin argumentos generado automáticamente. La visibilidad del constructor será **pública** por defecto. Si no desea que su clase tenga un constructor público, es necesario declarar un constructor vacío con una visibilidad que no sea la predeterminada:

```
// Clase con un constructor privado
class DontCreateMe private constructor () { ... }
```

Para crear una instancia de una clase, se invoca al constructor como si de una función regular se tratase. En Kotlin no existe la palabra clave *'new'*:

```
class Person(val name: String) {
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}

val person = Person("John")
```

Constructor primario

El **constructor primario** no puede contener ningún código. El código de inicialización se puede colocar en bloques de inicialización, que se definen con la palabra clave **'init'**.

Durante una inicialización de la instancia, los bloques de inicialización se ejecutan en el mismo orden en que aparecen en el cuerpo de la clase, intercalados con los inicializadores de

propiedades:

```
class InitOrderDemo(name: String) {
    val firstProperty = "First property: $name"

    init {
        println("First initializer block that prints ${name}")
    }

    val secondProperty = "Second property: ${name.length}"

    init {
        println("Second initializer block that prints ${name.length}")
    }
}
```

Los bloques **'init'** pueden usarse para validar los parámetros mediante la palabra clave **'require'**:

```
class Person (val firstName: String, val lastName: String, val age: Int?) {
    init{
        require(firstName.trim().length > 0) { "Invalid firstName argument." }
        require(lastName.trim().length > 0) { "Invalid lastName argument." }

        if (age != null) {
            require(age >= 0 && age < 150) { "Invalid age argument." }
        }
    }
}
```

Tenga en cuenta que los parámetros del constructor primario se pueden usar en los bloques de inicialización. También pueden ser utilizados en los inicializadores de las propiedades en el cuerpo de la clase:

```
class Customer(name: String) {
    // Uso del parámetro 'name' para inicializar la propiedad 'customerKey'
    val customerKey = name.toUpperCase()
}
```

De hecho, para declarar propiedades e inicializarlas desde el constructor principal, Kotlin tiene una sintaxis concisa:

```
class Person(val firstName: String, val lastName: String, var age: Int) { ... }
```


De la misma forma que las propiedades definidas en el cuerpo de la clase, las propiedades declaradas en el constructor primario pueden ser mutables (`'var'`) o de solo lectura (`'val'`).

Cuando se usa el prefijo `'val'` Kotlin genera automáticamente el método `'getter()'` y cuando se usa el prefijo `'var'` Kotlin genera el `'getter()'` y `'setter()'`. Si no necesitamos los accesores se puede definir el constructor sin los prefijos. De esta forma podemos definir nuestros propios métodos accesores.

Constructor secundario

La clase también puede declarar uno o varios **constructores secundarios**, que se definen con la palabra clave `'constructor'`:

```
class Person {
    // Constructor secundario
    constructor(parent: Person) {
        parent.children.add(this)
    }
}
```

Si la clase tiene un constructor primario, cada **constructor secundario debe delegar en el constructor primario**, ya sea directamente o indirectamente a través de otro/s constructor/es secundario/s. La delegación en otro constructor de la misma clase se hace usando la palabra clave `'this'`:

```
class Person(val name: String) { // Constructor primario

    // Constructor secundario
    // Usamos 'this' para invocar al constructor primario
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Hay que tener en cuenta que el código en los bloques de inicialización se convierte efectivamente en parte del constructor primario. La delegación en el constructor primario ocurre como la primera instrucción en el constructor secundario, por lo que el código en todos los bloques de inicialización se ejecuta antes que el constructor secundario. Incluso si la clase no tiene un constructor primario, la delegación todavía ocurre implícitamente y los bloques de inicialización aún se ejecutan antes:

```
class Constructors {
    init {
        println("Init block") // Se ejecuta antes que el constructor secundario
    }
    constructor(i: Int) {
        println("Constructor")
    }
}
```

Propiedades

En Kotlin no se utiliza el concepto de 'campo' cuando hablamos de variables de instancia sino que se emplea el concepto de **propiedades**.

Las propiedades de una clase pueden declararse como mutables (**var**), o de inmutables o de sólo lectura (**val**):

```
class Address {
    var name: String = ...
    var street: String = ...
    var city: String = ...
    var state: String? = ...
    var zip: String = ...
}
```

Para acceder a las propiedades de una clase usamos el operador punto **'.'** ya que a diferencia de Java no hay que utilizar **getters()** ni **setters()** si hemos definido la propiedad con **'val'** o **'var'**. Para usar la propiedad, simplemente nos referimos a ella por su nombre, como si fuera un campo en Java:

```
fun copyAddress(address: Address): Address {
    val result = Address() // there's no 'new' keyword in Kotlin
    result.name = address.name // accessors are called
    result.street = address.street
    // ...
    return result
}
```

'Getters()' and 'Setters()'

La sintaxis completa de definición de una propiedad en Kotlin:

```
{var|val} <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

El inicializador y las funciones '*getter()*' (y '*setter()*' si es una propiedad mutable) son opcionales. El tipo de la propiedad es opcional si puede inferirse desde el inicializador o desde el tipo de retorno del '*getter()*'.

```
var allByDefault: Int? // error: se requiere un inicializador explícito.
var initialized = 1 // propiedad de tipo Int, getter y setter por defecto

val simple: Int? // propiedad de tipo Int, getter por defecto, debe ser
inicializada por el constructor
val inferredType = 1 // propiedad de tipo Int y getter por defecto
```

Si las funciones '*getter()*' (y '*setter()*' en propiedades mutables) por defecto no son suficientes se puede codificar funciones '*getter()*' o '*setter()*' propias como cualquier otra función. Estas funciones están dentro de la propiedad y por tanto tienen que ser indentadas correctamente

```
val isEmpty: Boolean
    get() = this.size == 0

var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // parses the string and assigns values to other
properties
    }
```

Nótese que por convención, el nombre del parámetro de la función '*setter()*' es '*value*' pero no es obligatorio y puede escogerse otro nombre.

Las propiedades pueden ser '*private*', '*protected*', o '*public*' (visibilidad por defecto).

Backing Fields

El campo de respaldo o '*backing field*' es un campo generado automáticamente para cualquier propiedad que solo puede usarse dentro de los accesorios (getter o setter).

Estará presente solo si utiliza la implementación predeterminada de al menos uno de los accesorios, o si un descriptor de acceso personalizado lo hace referencia a través del identificador

'*field*'. Este campo de respaldo se usa para evitar la llamada recursiva y por tanto evitar un '*StackOverflowError*'.

Kotlin proporciona automáticamente este campo de respaldo. Se puede hacer referencia a este campo en los accesores utilizando el identificador '*field*':

```
var counter = 0 // Note: the initializer assigns the backing field directly
set(value) {
    if (value >= 0) field = value
}
```

Este campo es necesario ya que el siguiente código genera un '*StackOverflowError*'. Cuando Kotlin encuentra la propiedad 'selectedColor' llama al '*getter()*' correspondiente. Si usamos 'selectedColor' dentro de la definición del propio '*getter()*' es cuando se producen llamadas recursivas que acaban generando un desbordamiento de la pila. Kotlin provee del '*backing field*' para evitarlo.

```
var selectedColor: Int = someDefaultValue
get() = selectedColor
set(value) {
    this.selectedColor = value
    doSomething()
}

// Código correcto
var selectedColor: Int = someDefaultValue
get() = field
set(value) {
    field = value
    doSomething()
}
```

Constantes en tiempo de compilación

Las propiedades cuyo valor se conoce en el momento de la compilación se pueden marcar como constantes de tiempo de compilación utilizando el modificador '*const*'. Tales propiedades necesitan cumplir los siguientes requisitos:

- Top-level o miembros de un '*objet*'
- Inicializado con un valor de tipo String o un tipo primitivo
- No tener un '*getter()*' propio

Estas propiedades pueden ser utilizadas en anotaciones:

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is deprecated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

Late-Initialized Properties and Variables

Normalmente, las propiedades declaradas con un tipo no nulo deben inicializarse en el constructor. Sin embargo, bastante a menudo esto no es conveniente. Por ejemplo, las propiedades se pueden inicializar mediante la inyección de dependencias, o en el método de configuración de una prueba de unidad. En este caso, no puede proporcionar un inicializador que no sea nulo en el constructor, pero aún así desea evitar las comprobaciones nulas al hacer referencia a la propiedad dentro del cuerpo de una clase.

Para manejar este caso, puede marcar la propiedad con el modificador *'lateinit'*:

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method() // dereference directly
    }
}
```

Para usar este modificador hay que cumplir ciertos requisitos:

- Se puede usar únicamente en las propiedades *'var'* declaradas dentro del cuerpo de una clase. Por tanto no se puede usar en propiedades declaradas en el constructor principal.
- La propiedad no tiene un *'getter()'* o *'setter()'* personalizado.

Acceder a una propiedad antes de que haya sido inicializada lanzará una *'UninitializedPropertyAccessException'*.

Member Functions

Una función miembro es una función que se define dentro de una clase, objeto o interfaz. Las funciones miembro se invocan con el operador *'.'*:

```
class Sample() {
    fun foo() {
```

```

        print("Foo")
    }
}

Sample().foo() // crea una instancia de 'Sample' e invoca el método 'foo'

```

Herencia

La herencia es fundamental para la programación orientada a objetos. Nos permite crear nuevas clases que reutilizan, amplían y/o modifican el comportamiento de los preexistentes. La clase preexistente se llama la **superclase** (o clase base o padre), y la clase nueva que estamos creando se llama la **clase derivada**. Una clase derivada obtendrá implícitamente todos los campos, propiedades y métodos de la superclase (y de la superclase de la superclase si es el caso).

Hay una restricción en cuanto a cuántas clases podemos heredar; en una JVM, solo puede tener una clase base. Pero se puede heredar de múltiples interfaces.

La herencia es transitiva. Si la clase C se deriva de la clase B y esa clase B se deriva de una clase A dada, entonces la clase C es una clase derivada de A.

Todas las clases en Kotlin tienen una superclase común **'Any'**, que es la superclase predeterminada para una clase sin supertipos declarados:

```
class Example // Hereda de 'Any' implícitamente
```

Para declarar que una clase hereda de una clase base, colocamos el tipo de la clase base después de dos puntos en el encabezado de la clase derivada. Por defecto en Kotlin las clases están cerradas a la herencia, es decir, son **'final'**. Para permitir que una clase sea heredada, hay que utilizar la palabra clave **'open'**.

```

open class Base(p: Int)

class DerivedWithConstructor(p: Int) : Base(p)

```

Si la clase derivada tiene un constructor primario, la clase base puede (y debe) inicializarse allí mismo, utilizando los parámetros del constructor primario.

Si la clase no tiene un constructor primario, entonces cada constructor secundario tiene que inicializar el tipo base usando la palabra clave **'super'**, o delegar a otro constructor que haga eso. Tenga en cuenta que en este caso, diferentes constructores secundarios pueden llamar a diferentes constructores de la clase base:

```

open class Base(p: Int) {
    constructor(p: Int, q: Int): this(p)
}

class DerivedWithoutConstructor : Base {
    constructor(p: Int) : super(p)
    // constructor(p: Int, q: Int) : super(p, q)
}

```

Sobreescritura de métodos

Kotlin requiere anotaciones explícitas para la sobreescritura de funciones miembro.

Para que una función pueda ser sobreescrita se utiliza la palabra clave **'open'** delante del nombre de la función. Dado que las clases son **finales** en Kotlin, sólo podemos utilizar la palabra clave **'open'** en funciones miembro de clases que también hayan sido definidas como **'open'**.

Para indicar que una función en la clase derivada sobreescribe una función de la clase padre se utiliza la palabra clave **'override'** delante del nombre de la función. De esta forma le indicamos al compilador que esta función sobreescribe una función de la clase padre y puede realizar las comprobaciones en tiempo de compilación.

Una función con la palabra clave **'override'** también es **'open'** por definición y puede ser sobreescrita por las subclases sucesivas. Es posible marcar una función **'override'** con la palabra clave **'final'** para evitar que sea sobreescrita.

```

open class Base {
    open fun v() { ... }
    open fun x(p: Int) { ... }
    fun nv() { ... }
}

class Derived: Base() {
    override fun v() { ... }

    final override fun x(p: Int) { ... } // Restringir la sobreescritura
}

```

En Kotlin, la herencia está regulada por la siguiente regla: si una clase hereda varias implementaciones del mismo miembro de sus superclases inmediatas, debe invalidar este miembro y proporcionar su propia implementación. Para denotar el supertipo del cual se toma la implementación heredada, usamos la palabra clave **'super'** calificado por el nombre de supertipo entre paréntesis angulares, por ejemplo, **super<Base>**:

```

open class A {
    open fun f() { print("A") }
    fun a() { print("a") }
}

interface B {
    fun f() { print("B") } // interface members are 'open' by default
    fun b() { print("b") }
}

class C() : A(), B {
    // El compilador requiere que 'f()' sea sobrescrito para eliminar la
    ambigüedad
    override fun f() {
        super<A>.f() // call to A.f()
        super<B>.f() // call to B.f()
    }
}

```

Sobrescritura de propiedades

La sobrescritura de propiedades funciona de manera similar a la sobrescritura de métodos.

Las propiedades declaradas en una superclase que luego se vuelven a declarar en una clase derivada deben ir precedidas por la palabra clave **'override'** y deben tener un tipo compatible. También se puede usar la palabra clave **'override'** como parte de la declaración de una propiedad en un constructor primario.

Cada propiedad declarada puede ser sobrescrita por una propiedad con un inicializador o por una propiedad con un método **'getter()'**

```

open class Foo {
    open val x: Int get() { ... }
}

class Bar : Foo() {
    override val x: Int = ...
}

interface Foo1 {
    val count: Int
}

class Bar1(override val count: Int) : Foo1

```


Orden de inicialización

Durante la construcción de una nueva instancia de una clase derivada, **la inicialización de la clase base se realiza como primer paso** (precedida solo por la evaluación de los argumentos para el constructor de la clase base) y, por lo tanto, ocurre antes de que se ejecute la lógica de inicialización de la clase derivada.

Por lo tanto, durante la inicialización de las propiedades de la clase base las propiedades de la clase derivada aún no se han inicializado. Si alguna de esas propiedades se utilizan (de forma directa o indirecta) en la inicialización de la clase base se pueden producir comportamientos extraños o errores en tiempo de ejecución.

```
open class Base(val name: String) {
    init {
        println("Initializing Base")
    }

    open val size: Int =
        name.length.also { println("Initializing size in Base: $it") }
}

class Derived(name: String, val lastName: String) : Base(name.capitalize().also {
    println("Argument for Base: $it") }) {
    init {
        println("Initializing Derived")
    }

    override val size: Int =
        (super.size + lastName.length).also { println("Initializing size in
Derived: $it") }
}

// Argument for Base: Hello
// Initializing Base
// Initializing size in Base: 5
// Initializing Derived
// Initializing size in Derived: 10
```

Invocar la implementación de la superclase

El código en una clase derivada puede llamar a funciones en la superclase e implementaciones de accesores de propiedades usando la palabra clave **'super'**:

```
open class Foo {
    open fun f() { println("Foo.f()") }
    open val x: Int get() = 1
}
```

```

}

class Bar : Foo() {
    override fun f() {
        super.f() // Calling the super function
        println("Bar.f()")
    }
    override val x: Int get() = super.x + 1
}

```

Clases abstractas

Kotlin admite **clases abstractas** al igual que Java. Una clase abstracta es una clase con métodos marcados como abstractos y que por tanto no puede ser instanciada. Si una clase tiene uno o varios métodos abstractos es una clase abstracta y se indica con la palabra clave **'abstract'**.

La subclase concreta de una clase abstracta deberá implementar todos los métodos y propiedades definidos en la clase abstracta; de lo contrario, también será considerada como una clase abstracta.

```

open class Person {
    open fun fullName(): String { ... }
}

abstract class Employee (val firstName: String, val lastName: String): Person() {
    // Variable de instancia en una clase abstracta
    val propFoo: String = "bla bla"

    abstract fun earnings(): Double

    // Podemos tener métodos con implementación por defecto
    override fun fullName(): String {
        return lastName + " " + firstName;
    }
}

```

Las clases abstractas pueden contener métodos con implementación por defecto como cualquier otra clase. Las subclases de la clase abstracta pueden sobrescribir la implementación predeterminada de un método pero solo si el método tiene el modificador **'open'**. Los métodos marcados como **'abstract'** también son **'open'** por defecto. Las clases abstractas también pueden definir variables de instancia al contrario que pasa con las interfaces.

Interfaces

Las interfaces en Kotlin son muy similares a Java 8. Pueden contener declaraciones de métodos abstractos, así como implementaciones de métodos. Lo que los diferencia de las clases abstractas es que las interfaces no pueden almacenar el estado, es decir, no pueden tener variables de instancia. Pueden tener propiedades, pero estas deben ser abstractas o proporcionar implementaciones de accesores.

Una interfaz se define usando la palabra clave *'interface'*. Un método en una interfaz es abstracto por defecto si no se proporciona una implementación.

```
interface MyInterface {
    fun bar() // abstract by default
    fun foo() {
        // optional body
    }
}
```

Una clase u objeto pueden implementar una o varias interfaces:

```
class Child : MyInterface {
    override fun bar() {
        // body
    }
}
```

En una interfaz se pueden declarar propiedades. Una propiedad declarada en una interfaz puede ser abstracta o puede proporcionar implementaciones para el *'getter()'* o *'setter()'*. Las propiedades declaradas en interfaces no pueden tener *'backing fields'* y, por lo tanto, los accesores declarados en interfaces no pueden hacer referencia a ellos.

```
interface MyInterface {
    val prop: Int // abstract

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(prop)
    }
}

class Child : MyInterface {
```

```

    override val prop: Int = 29
}

```

Una interfaz puede derivar de otras interfaces y, por lo tanto, proporcionar implementaciones para sus miembros y declarar nuevas funciones y propiedades. Naturalmente, las clases que implementen dicha interfaz solo tienen que definir las implementaciones que faltan:

```

interface Named {
    val name: String
}

interface Person : Named {
    val firstName: String
    val lastName: String
    override val name: String get() = "$firstName $lastName"
}

data class Employee(
    // implementing 'name' is not required
    override val firstName: String,
    override val lastName: String,
    val position: Position
) : Person

```

En el caso de clases que hereden de varias interfaces, para evitar ambigüedades la subclase deberá proporcionar implementaciones tanto para métodos que tienen una implementación en una de las interfaces como en métodos que tiene implementaciones en varias interfaces.

```

interface A {
    fun foo() { print("A") }
    fun bar() // abstract
}

interface B {
    fun foo() { print("B") }
    fun bar() { print("bar") }
}

class C : A {
    override fun bar() { print("bar") }
}

// la clase 'D' tienen que implementar tanto foo() como bar()
class D : A, B {
    override fun foo() {
        super<A>.foo()
    }
}

```

```

        super<B>.foo()
    }

    override fun bar() {
        super<B>.bar()
    }
}

```

Visibilidad

Las clases, objetos, interfaces, constructores, funciones, propiedades y sus *'setters'* pueden tener modificadores de visibilidad. (Los *'setters'* siempre tienen la misma visibilidad que la propiedad).

- **Public** - Este es el valor predeterminado, y se puede acceder a cualquier clase, función, propiedad, interfaz u objeto que tenga este modificador desde cualquier lugar.
- **Private** - Se puede acceder a una función, interfaz o clase de nivel superior que se declara como privada solo dentro del mismo archivo.

Cualquier función o propiedad que se declare privada dentro de una clase, objeto o interfaz solo puede ser visible para otros miembros de esa misma clase, objeto o interfaz.

Un constructor privado debe usar la palabra clave *'constructor'*. Si un constructor es marcado como privado no se puede instanciar un objeto con ese constructor.

```

class Car private constructor(val name: String, val plateNo: String) {
    // ....
}

```

- **Protected** - Solo se puede aplicar a propiedades o funciones dentro de una clase, objeto o interfaz, no se puede aplicar a funciones, clases o interfaces de nivel superior. Las propiedades o funciones con este modificador solo son accesibles dentro de la clase que lo define y cualquier subclase.
- **Internal** - En un proyecto que tiene un módulo (módulo Gradle o Maven), una clase, objeto, interfaz o función especificada con este modificador dentro de ese módulo solo es accesible desde ese módulo.

Data classes

Las **Data classes** son una forma concisa de crear clases que solo contienen datos. Estas clases se definen con la palabra clave **'data'**.

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

De forma automática el compilador crea los métodos **hashCode()**, **equals()**, **copy()** y **toString()** a partir de todas las propiedades declaradas en el constructor primario. También se generan las funciones **componentN()** que corresponden a las propiedades declaradas en orden en el constructor primario.

Para evitar comportamientos extraños estas clases deben **cumplir ciertos requisitos**:

- El constructor primario necesita tener al menos un parámetro
- Todos los parámetros del constructor primario estarán marcados como **'val'** o **'var'**
- Una **'data class'** no puede ser **'abstract'**, **'open'**, **'sealed'** o **'inner'**
- (Antes de 1.1) Las **'data classes'** no pueden extender de otras clases (pero pueden implementar interfaces)

El compilador sólo tiene en cuenta las propiedades declaradas en el constructor primario a la hora de generar los métodos de forma automática. Por tanto, para excluir propiedades se deben declarar en el cuerpo de la clase.

```
data class DataClassExample(val x: Int, val y: Int, val z: Int) {
    // Propiedad excluida
    var xx: Int = 0
}

val fooData = DataClassExample(1, 2, 4)
val fooCopy = fooData.copy(y = 100)

// El formato de 'toString()' es el mismo 'ClassName(prop=xx, prop=yy, ....)'
println(fooData) // => DataClassExample(x=1, y=2, z=4)
println(fooCopy) // => DataClassExample(x=1, y=100, z=4)
```

El compilador genera la función **copy()** que permite copiar un objeto y en caso necesario, crear la copia alterando algunas de sus propiedades y manteniendo el resto.

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

// Función 'copy()' generada automáticamente
// fun copy(name: String = this.name, age: Int = this.age) = User(name, age)

val jack = User(name = "Jack", age = 1)
```

```
// Copiamos el objeto pero modificando la propiedad 'age'
val olderJack = jack.copy(age = 2)
```

Las funciones `componentN()` permite desestructurar las propiedades:

```
val jane = User("Jane", 35)
val (name, age) = jane
println("$name, $age years of age") // => Jane, 35 years of age
```

Cada tipo se deriva de `'Any'`, que viene con una declaración de método `'hashCode()'`. Esto es el equivalente de un método `'hashCode()'` de clase `'Object'` de Java. Este método es importante cuando se insertan instancias del objeto en colecciones, como un mapa. Al implementar este método, se debe cumplir con una serie de requisitos:

1. Cuando se invoque en el mismo objeto más de una vez durante el tiempo de ejecución, el método `'hashCode()'` debe devolver constantemente el mismo valor, dado que el objeto no se modificó.
2. Si para dos objetos el método `'equals()'` devuelve true, entonces llamar al método `'hashCode()'` en cada uno de ellos debería devolver el mismo valor entero.
3. Si dos objetos no son iguales, es decir, que el método `'equals()'` devuelve false cuando se comparan, no es un requisito que cada método `'hashCode()'` del objeto devuelva valores distintos. Sin embargo, producir un entero distinto para objetos desiguales podría mejorar el rendimiento de las colecciones basadas en 'hash'.

Las `'data classes'` son una forma compacta y legible de devolver dos o más valores de una función. Otra alternativa, menos legible, es utilizar el tipo `'Pair'` o `'Triple'` proporcionado por Kotlin:

```
data class Result(val result: Int, val status: Boolean)

fun checkStatus() = Result(10, true) // función que retorna un tipo 'Result'

val (result, status) = checkStatus() // usamos la desestructuración de datos para acceder a los datos
```

Sealed classes

En Kotlin una `'sealed class'` es una clase abstracta (no se puede crear instancias) que otras clases pueden extender. Estas subclases se definen dentro del cuerpo de la `'sealed class'`, en el mismo archivo por lo que podemos conocer todas las subclases posibles simplemente viendo el archivo.

Las *'sealed class'* se utilizan para representar jerarquías de clases restringidas, de forma que una clase solo pueda heredar de un conjunto limitado de tipos. Son, en cierto sentido, una extensión de las clases de enumeración.

- Podemos agregar el modificador *'abstract'*, pero esto es redundante porque estas clases son abstractas por defecto.
- No pueden tener el modificador *'open'* ni *'final'*.
- Podemos declarar clases de datos y objetos como subclasses a una *'sealed class'* (aún deben declararse en el mismo archivo).
- No pueden tener constructores públicos ya que sus constructores son privados de forma predeterminada.

```
// shape.kt

sealed class Shape

class Circle : Shape()
class Triangle : Shape()
class Rectangle: Shape()
```

Generics

Covarianza y contravarianza son términos que hacen referencia a la capacidad de usar un tipo más derivado (más específico) o menos derivado (menos específico) que el indicado originalmente. Los parámetros de tipo genérico admiten la covarianza y contravarianza para proporcionar mayor flexibilidad a la hora de asignar y usar tipos genéricos. Cuando se hace referencia a un sistema de tipos, la covarianza, contravarianza e invarianza tienen las siguientes definiciones:

- **Covariance** -> Permite usar un tipo más derivado que el especificado originalmente. Puede asignar una instancia de *Class<Derived>* a una variable de tipo *Class<Base>*.
- **Contravariance** -> Permite usar un tipo más genérico (menos derivado) que el especificado originalmente. Puede asignar una instancia de *Class<Base>* a una variable de tipo *Class<Derived>*.
- **Invariance** -> Significa que solo se puede usar el tipo especificado originalmente. Así, un parámetro de tipo genérico invariable no es covariante ni contravariante. No se puede asignar una instancia de *List<Base>* a una variable de tipo *List<Derived>* o viceversa.

Al igual que en Java, en Kotlin las clases pueden tener tipos con parámetros.


```
class Box<T>(t: T) {
    var value = t
}
```

En general, para crear una instancia de una clase genérica tenemos que proveer el tipo a la clase:

```
val box: Box<Int> = Box<Int>(1)
```

Si los parámetros se pueden inferir, como por ejemplo de los argumentos del constructor o por algún otro medio, se pueden omitir los argumentos de tipo:

```
val box = Box(1) // '1' tiene tipo Int así que el compilador infiere el tipo
                 "Box<Int>"
```

La palabra clave 'out'

Digamos que queremos crear una clase de productor que producirá un resultado de algún tipo 'T'. A veces; queremos asignar ese valor producido a una referencia que es de un supertipo del tipo 'T'.

Para lograr eso usando Kotlin, necesitamos usar la palabra clave **'out'** en el tipo genérico. Esto significa que podemos asignar esta referencia a cualquiera de sus supertipos. El valor de salida solo puede ser producido por la clase dada pero no consumido:

```
class ParameterizedProducer<out T>(private val value: T) {
    fun get(): T {
        return value
    }
}

val a = ParameterizedProducer("string") // ParameterizedProducer<String>
val x: ParameterizedProducer<Any> = a // Correcto

val b = ParameterizedProducer(10) // ParameterizedProducer<Int>
val y: ParameterizedProducer<Number> = b // Correcto
val z: ParameterizedProducer<String> = b // ¡Error de compilación!
```

La palabra clave 'in'

A veces, tenemos una situación opuesta, lo que significa que tenemos una referencia de tipo T y queremos poder asignarla al subtipo de T.

Podemos usar la palabra clave **'in'** en el tipo genérico si queremos asignarlo a la referencia de su subtipo. La palabra clave **'in'** solo se puede utilizar en el tipo de parámetro que se consume, no se produce:

```
class ParameterizedConsumer<in T> {
    fun toString(value: T): String { // 'toString()' will only be consuming a
        value of type T.
        return value.toString()
    }
}

val a = ParameterizedConsumer<Number>()

val b: ParameterizedConsumer<Double> = a // Correcto
val c: ParameterizedConsumer<Int> = a // Correcto
val d: ParameterizedConsumer<String> = a // ¡Error de compilación!
```

Star projections

Hay situaciones en las que no es importante el tipo específico de un valor. Para ello usamos el operador **'*'** o *'star projection'*:

```
fun printArray(array: Array<*>) {
    array.forEach { println(it) }
}

// Podemos pasar una matriz de cualquier tipo al método 'printArray()'
printArray(arrayOf(1,2,3))

printArray(arrayOf("hello", "World!!", 5))
```

Generic functions

Las funciones también pueden ser genéricas en los tipos que utilizan. Esto permite escribir una función que puede funcionar con cualquier tipo, en lugar de solo un tipo específico. Para ello, definimos los parámetros de tipo en la firma de función.

```
fun <T> choose(t1: T, t2: T, t3: T): T {
    return when (Random().nextInt(3)) {
        0 -> t1
```

```

        1 -> t2
        else -> t3
    }
}

// Podemos usar esta función con enteros. Si el compilador puede inferir el tipo
se puede omitir.
val r = choose<Int>(5, 7, 9)
val r = choose(5, 7, 9)

// También es válido usar la función con Strings
val s = choose<String>("BMW", "Audi", "Ford")
val s = choose("BMW", "Audi", "Ford")

```

Generic constraints

El conjunto de todos los tipos posibles que pueden sustituirse por un parámetro de tipo dado puede estar restringido por restricciones genéricas.

El tipo más común de restricción es un límite superior que corresponde a la palabra clave de extensión de Java:

```

fun <T : Comparable<T>> sort(list: List<T>) { ... }

sort(listOf(1, 2, 3)) // OK. Int is a subtype of Comparable<Int>
sort(listOf(HashMap<Int, String>())) // Error: HashMap<Int, String> is not a
subtype of Comparable<HashMap<Int, String>>

```

El límite superior predeterminado (si no se especifica) es *'Any?'*.

Nested classes

Al igual que las funciones, Kotlin permite las clases internas, es decir, clases definidas dentro de otra clase. Son equivalentes a las clases internas estáticas en Java.

```

class OuterClass {
    class NestedClass {
        fun nestedClassFunc() { }
    }
}

val nestedClass = OuterClass.NestedClass().nestedClassFunc()

```

Inner class

Las clases internas, por otro lado, pueden hacer referencia a la clase externa en la que se declaró. Para crear una clase interna, colocamos la palabra clave **'inner'** antes de la palabra clave **'class'**.

```
class OuterClass() {
    val oCPropt: String = "Yo"

    inner class InnerClass {
        fun innerClassFunc() {
            val outerClass = this@OuterClass
            print(outerClass.oCPropt)
        }
    }
}
val demo = OuterClass().InnerClass().innerClassFunc() // => yo
```

Enumeraciones

Las clases de enumeración son similares a los tipos **'enum'** de Java. El uso más básico de las clases de enumeración es la implementación de enumeraciones de tipos seguros. Cada constante de la enumeración es un objeto. Las constantes de la enumeración están separadas por comas.

```
enum class Country {
    Spain, France, Portugal
}
```

Las enumeraciones pueden tener constructor:

```
enum class Direction(val angle: Int) {
    North(90), West(180), South(270), East(0)
}
```

En Kotlin las constantes de la enumeración pueden declarar sus propias clases anónimas con sus métodos correspondientes, así como sobrescribir métodos primarios.

Si la enumeración define algún miembro, debe separar las definiciones de constantes de enumeración de las definiciones de miembros con un punto y coma, al igual que en Java.

```
enum class ProtocolState {
    WAITING {
        override fun signal() = TALKING
    },

    TALKING {
        override fun signal() = WAITING
    };

    abstract fun signal(): ProtocolState
}
```

En Kotlin las enumeraciones disponen de forma predeterminada de los métodos:

- ***EnumClass.valueOf(value: String): EnumClass*** -> Devuelve la constante de enumeración por su nombre. Lanza un *IllegalArgumentException* si no existe la constante.
- ***EnumClass.values(): Array<EnumClass>*** -> Retorna un array con las constantes de enumeración.

Además de los métodos las instancias de enumeración vienen con dos propiedades predefinidas. Uno es **'name'** de tipo 'String' y el segundo es **'ordinal'** de tipo 'Int' para obtener la posición de la constante dentro de la enumeración, teniendo en cuenta que empiezan por 0:

```
enum class Country {
    Spain, France, Portugal
}

println(Country.Spain) // => Spain
println(Country.valueOf("Spain")) // => Spain

println(Country.Portugal.name) // => Portugal
println(Country.France.ordinal) // => 1

fun countries() {
    for (country in Country.values()) {
        println("Country: $country")
    }
}
```

Objects

Los objetos son muy similares a las clases. A veces necesitamos crear un objeto con una ligera modificación de alguna clase, sin declarar explícitamente una nueva subclase para ello. Java maneja este caso con clases internas anónimas. Kotlin generaliza ligeramente este concepto con *'object expressions'* y *'objects declarations'*.

Estas son algunas de las características de los objetos en Kotlin:

- Pueden tener propiedades, métodos y un bloque init.
- Estas propiedades o métodos pueden tener modificadores de visibilidad.
- No pueden tener constructores (primarios o secundarios).
- Pueden extender otras clases o implementar una interfaz.

Hay importantes diferencias semánticas entre un *'object expression'* y un *'object declaration'*

- Los *'object expression'* se ejecutan (y se inicializan) inmediatamente, donde se usan.
- Los *'object declaration'* se inicializan cuando se accede por primera vez.
- Por su parte, un *'companion object'* se inicializa cuando se carga la clase correspondiente.

Objects expressions

Para crear un objeto de una clase anónima que hereda de algún tipo (o tipos), escribimos:

```
fun countClicks(window: JComponent) {
    var clickCount = 0
    var enterCount = 0

    window.addMouseListener(object : MouseAdapter() {
        override fun mouseClicked(e: MouseEvent) {
            clickCount++
        }

        override fun mouseEntered(e: MouseEvent) {
            enterCount++
        }
    })
    // ...
}
```

Objects declarations

Colocamos la palabra clave **'object'** antes del nombre del objeto que queremos crear. De hecho, estamos creando un **SINGLETON** cuando creamos objetos en Kotlin usando esta construcción ya que solo existe una instancia de un objeto.

```
object ObjectExample {
    val baseUrl: String = "http://www.myapi.com/"
    fun hello(): String {
        return "Hello"
    }
}

println(ObjectExample.hello()) // => Hello

fun useObject() {
    ObjectExample.hello() // => Hello
    val someRef: Any = ObjectExample // Usamos el nombre de los objetos tal como
son
}
```

Al igual que una declaración de variable, una declaración de objeto no es una expresión y no se puede utilizar en el lado derecho de una declaración de asignación.

Los objetos en Kotlin pueden utilizarse también para crear constantes.

```
object APIConstants {
    val baseUrl: String = "http://www.myapi.com/"
}
```

Companion objects

Los *'companion objects'* son un tipo de *'object declaration'*. Como Kotlin no admite clases, métodos o propiedades estáticas como las que tenemos en Java, Kotlin provee los *'companion objects'*. Estos objetos son básicamente un objeto que pertenece a una clase que se conoce como la clase complementaria del objeto. Este objeto se indica con la palabra clave *'companion'*.

Similar a los métodos estáticos en Java, un *'companion object'* no está asociado con una instancia de clase, sino con la propia clase.

Se puede llamar a los miembros del *'companion object'* usando simplemente el nombre de la clase como el calificador, como si fuera un método estático.

Un *'companion object'* puede tener nombre que facilitará el ser invocado desde Java aunque es opcional.

```
class Person private constructor(var firstName: String, var lastName: String) {

    // Podemos omitir el nombre del objeto
    companion object {
        var count: Int = 0
    }
}
```

```

    fun create(firstName: String, lastName: String): Person =
        Person(firstName, lastName)

    // Podemos tener bloques 'init' dentro de un 'companion object'
    init {
        println("Person companion object created")
    }
}

val person = Person.create("John", "Doe")

class MyClass {

    fun sayHello() = println("hello")

    // Objeto con el nombre 'Factory' y que utilizaremos como 'Factory Pattern'
    companion object Factory {
        fun create(): MyClass = MyClass()

        fun sayHelloFromCompanion() = MyClass().sayHello() // Podemos acceder a
        miembros de la clase
    }
}

val myClass = MyClass.create()
MyClass().sayHello() // incorrecto
MyClass.Factory.sayHelloFromCompanion() // Invocar un método del 'companion'

```


Other



Destructuring data

Los objetos pueden ser desestructurados en múltiples variables. Esta sintaxis se llama '**declaración de desestructuración**'. Una declaración de desestructuración crea múltiples variables a la vez.

```
val (a, b, c) = fooCopy
println("$a $b $c") // => 1 100 4
```

Desestructurando en un bucle '**for**':

```
for ((a, b, c) in listOf(fooData)) {
    println("$a $b $c") // => 1 100 4
}

val mapData = mapOf("a" to 1, "b" to 2)
// Map.Entry is destructurable as well
for ((key, value) in mapData) {
    println("$key -> $value")
}
```

Colecciones

Kotlin proporciona su API de colecciones como una biblioteca estándar construida sobre la API de colecciones de Java como '*ArrayList*', '*Maps*', etc... Kotlin tiene dos variantes de colecciones: **mutables** e **inmutables**. Una colección mutable nos brinda la capacidad de modificar una colección ya sea agregando, eliminando o reemplazando un elemento. Las colecciones inmutables no se pueden modificar y no tienen estos métodos de ayuda.

Lists - [Immutable]

Una lista es una colección ordenada de elementos. Esta es una colección popular ampliamente utilizada.

Podemos crear una **lista inmutable** usando la función `listOf()`. Los elementos no se pueden agregar ni eliminar.

```
val fooList = listOf("a", "b", "c", 1, false)
val numbers: List<Int> = listOf(1, 2, 3, 4)
val emptyList: List<String> = emptyList<String>() // lista vacía
val nonNullsList: List<String> = listOfNotNull(2, 45, 2, null, 5, null) // lista
de valores no nulos

println(fooList.size) // => 3
println(fooList.first()) // => a
println(fooList.last()) // => c
println(fooList.indexOf("b")) // 1

// Se puede acceder a los elementos de una lista por su índice
println(fooList[1]) // => b
```

Se puede crear una **lista mutable** utilizando la función `mutableListOf()`:

```
val fooMutableList = mutableListOf("a", "b", "c")
fooMutableList.add("d")
println(fooMutableList.last()) // => d
println(fooMutableList.size) // => 4
```

Con la función `'arrayListOf()'` crea una lista mutable y devuelve un tipo `'ArrayList'` de la API de colecciones de Java.

Sets - [Immutable]

Un conjunto o 'set' es una colección desordenada de elementos únicos. En otras palabras, es una colección que no admite duplicados.

Podemos crear un conjunto (o 'set') inmutable utilizando la función `'setOf()'`:

```
val fooSet = setOf("a", "b", "c")
println(fooSet.contains("a")) // => true
println(fooSet.contains("z")) // => false
```

Con la función '*mutableSetOf()*' podemos crear un conjunto mutable:

```
// creates a mutable set of int types only
val intsMutableSet: MutableSet<Int> = mutableSetOf(3, 5, 6, 2, 0)
intsMutableSet.add(8)
intsMutableSet.remove(3)
```

La función '*hashSetOf()*' retorna un '*HashSet*' de la API de colecciones de Java el cual almacena los elementos en una tabla 'hash'. Podemos añadir o quitar elementos de este conjunto porque es **mutable**.

La función '*LinkedSetOf()*' retorna un '*LinkedHashSet*' de la API de colecciones de Java. También es un conjunto mutable.

Maps - [Immutable]

Los mapas asocian una clave a un valor. Las claves deben ser únicas, y por tanto no se permite duplicados. En cambio no hay obligación de que los valores asociados sean únicos. Cada clave sólo podrá asociarse a un solo elemento. De esa manera, cada clave se puede usar para identificar de forma única el valor asociado, ya que el mapa se asegura de que no pueda haber claves duplicadas en la colección. Los mapas implementan una forma eficiente de obtener el valor correspondiente a una determinada clave.

Podemos crear un **mapa ('map') inmutable** usando la función '*mapOf()*':

```
val fooMap = mapOf("a" to 8, "b" to 7, "c" to 9)

// Se puede acceder a los valores en el mapa por su clave
println(fooMap["a"]) // => 8

// iterar por un mapa con un bucle 'for'
for ((key, value) in fooMap) {
    println("Key $key and value $value")
}
```

La función '*LinkedHashMap()*' retorna un '*LinkedHashMap*' de la API de colecciones de Java, que es **mutable**.

La función '*sortedMapOf()*' retorna un '*SortedMap*' de la API de colecciones de Java que también es **mutable**.

Sequences

Las secuencias representan colecciones '*lazily-evaluated*'. Podemos crear una secuencia utilizando la función '*generateSequence()*'. Las secuencias son excelentes cuando el tamaño de la colección es desconocido a priori:

```
val fooSequence = generateSequence(1, { it + 1 })
val x = fooSequence.take(10).toList()
println(x) // => [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// An example of using a sequence to generate Fibonacci numbers:
fun fibonacciSequence(): Sequence<Long> {
    var a = 0L
    var b = 1L
    fun next(): Long {
        val result = a + b
        a = b
        b = result
        return a
    }
    return generateSequence(::next)
}

val y = fibonacciSequence().take(10).toList()
println(y) // => [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Kotlin proporciona '*higher-order functions*' para trabajar con colecciones:

```
val z = (1..9).map { it * 3 }
    .filter { it < 20 }
    .groupBy { it % 2 == 0 }
    .mapKeys { if (it.key) "even" else "odd" }

println(z) // => {odd=[3, 9, 15], even=[6, 12, 18]}
```

Rangos de valores

Un rango se define como un intervalo que tiene un valor de inicio y un valor final. **Los rangos son cerrados**, lo que significa que el valor inicial y final están incluidos en el rango. Los rangos se crean con el operador `..` o con funciones como *rangeTo()* o *downTo()*.

Para crear un intervalo sin incluir el último elemento usamos la función *until*.

```
val oneToNine = 1..9
val oneToFive: IntRange = 1.rangeTo(5)
```

```

val fiveToOne = 5.downTo(1)
print(fiveToOne) // => 5 downTo 1 step 1

val oneToTen = (1..10).step(2).reversed() // => 9, 7, 5, 3, 1
println("${tenToOne.first} - ${tenToOne.last}") // => 10 - 1

val oneToFour = 1.until(5)
print(r) // => 1..4

```

Los tipos *IntRange*, *LongRange*, *CharRange* tienen una característica extra y es que permite iterar sobre los intervalos.

Una vez que se crea un intervalo, se puede usar el operador *in* para probar si un valor dado está incluido en el intervalo o el operador *!in* para comprobar si un valor no está en el intervalo:

```

// Iterar con un bucle 'for'
for (i in 1..10) { // equivalent of 1 <= i && i <= 10
    print(i)
}

// Iterar en sentido inverso
for (i in 4 downTo 1) {
    print(i)
}

// Iterar por un intervalo sin incluir el último elemento
for (i in 1 until 10) {
    // i in [1, 10), 10 is excluded
    println(i)
}

// Pasos arbitrarios
for (i in 1..4 step 2) {
    print(i)
}

for (i in 4 downTo 1 step 2) {
    print(i)
}

```

Smart Casting

Podemos verificar si un objeto es de un tipo en particular usando el operador *is* o si no es de un tipo con el operador *!is*.

Si un objeto pasa una verificación de tipo entonces se puede usar como ese tipo sin realizar la conversión explícitamente:

```
fun smartCastExample(x: Any): Boolean {
    if (x is Boolean) {
        // x is automatically cast to Boolean
        return x
    } else if (x is Int) {
        // x is automatically cast to Int
        return x > 0
    } else if (x is String) {
        // x is automatically cast to String
        return x.isNotEmpty()
    } else {
        return false
    }
}

println(smartCastExample("Hello, world!")) // => true
println(smartCastExample("")) // => false
println(smartCastExample(5)) // => true
println(smartCastExample(0)) // => false
println(smartCastExample(true)) // => true
```

La conversión inteligente ('smart cast') también funciona con bloques *when* o bucles *while*:

```
fun smartCastWhenExample(x: Any) = when (x) {
    is Boolean -> x
    is Int -> x > 0
    is String -> x.isNotEmpty()
    else -> false
}
```

Conversión explícita o 'Explicit Casting'

Podemos usar el operador *as* (o el operador de conversión no segura o '*unsafe cast operator*') para convertir explícitamente una referencia de un tipo a otro tipo en Kotlin.

Si la operación de conversión explícita es ilegal, tenga en cuenta que se lanzará una excepción de tipo '*ClassCastException*'. Para evitar que se lance una excepción al realizar la conversión, podemos usar el operador de conversión seguro *as?*. Este operador intentará la conversión y si no se puede realizar la conversión devolverá 'null' en vez de lanzar la excepción.

```
val circle = shape as Circle

val circle: Circle? = shape as? Circle // Conversión segura
```

Valores nulos ('nullable types')

Para que una variable contenga el valor 'null' debe especificarse explícitamente como 'nullable'. Una variable se puede especificar como 'nullable' agregando un `?` a su tipo.

Podemos acceder a una variable o método 'nullable' utilizando el operador `'?.'` también llamado 'Safe Call Operator'. Un método o variable sólo será invocado si tiene una valor no nulo. En caso de que sea nulo será ignorado evitando un 'NullPointerException'

Kotlin provee el operador `'?:'`, también llamado 'Elvis Operator' para especificar un valor alternativo para usar si una variable es nula. Cuando la expresión de la izquierda del operador `'?:'` no es nulo entonces lo devuelve. En caso de que sea nulo devuelve la expresión de la derecha. La expresión de la derecha sólo será evaluada si la expresión de la izquierda es 'null'.

```
val name: String = null // no compilará ya que no puede contener valores nulos
var fooNullable: String? = "abc"

fooNullable?.length // => 3

// 'Elvis Operator'
fooNullable?.length ?: -1 // => 3

fooNullable = null
val len: Int? = fooNullable?.length // El tipo de retorno de 'fooNullable' puede
ser 'null' y por tanto debemos usar Int?

fooNullable?.length // => null
fooNullable?.length ?: -1 // => -1

// Encadenar 'safe calls'. La cadena retorna 'null' si alguna de ellas es 'null'
fun getCountryNameSafe(person: Person?): String? {
    return person?.address?.city?.country?.name
}

// Dado que 'throw' y 'return' son expresiones en Kotlin se pueden usar en la
parte derecha del operador 'Elvis'
fun foo(node: Node): String? {
    val parent = node.getParent() ?: return null
    val name = node.getName() ?: throw IllegalArgumentException("name expected")
    // ...
}
```

De manera similar, podemos devolver tipos *'nullable'* y no *'nullable'* desde una función.

```
fun getName(): String? = name // Esta función puede o no devolver una referencia nula.

fun getNotNullName(): String = name ?: "John" // Esta función no devolverá una referencia nula

getName() // => null
getNotNullName() // => John
```

Con *'smart cast'*, el compilador rastrea las condiciones dentro de una expresión *'if'*. Si realizamos la verificación de que una variable no es nula, entonces el compilador nos permitirá acceder a la variable como si hubiera sido declarada como un tipo no anulable:

```
var l = if (name != null) name.length else -1
```

El operador de aserción no-nulo *'!!'* convierte cualquier valor a un tipo no nulo y lanza una excepción *'NullPointerException'* si el valor es nulo.

```
val length: Int = name!!.length
```

Igualdad

En Kotlin hay tenemos la **igualdad estructural** y la **igualdad referencial**.

La igualdad estructural se comprueba con la operación *'=='* y la parte contraria *'!='* y se utiliza para comprobar si dos valores o variables son iguales (*equals()*)

```
if (a == b) {
    // ...
} else {
    // ...
}
```

La igualdad referencial se comprueba con la operación *'==='* y su contraparte *'!=='* y evalúa a *true* si y sólo si dos referencias apuntan al mismo objeto.

[Standard Library Functions]

Son funciones que proporciona Kotlin para aumentar la biblioteca estándar de Java.

[Apply]

'*apply*' es una función de extensión de la biblioteca estándar de Kotlin declarada en '*Any*', por lo que puede ser invocada en cualquier tipo de instancia. '*apply*' acepta una expresión lambda que es invocada y el receptor es la instancia donde es llamada. La función '*apply*' devuelve una instancia del original.

Su uso principal es hacer que el código que necesita inicializar una instancia sea más legible permitiendo que las funciones y las propiedades se llamen directamente dentro de la función antes de devolver el valor en sí.

```
data class Person(var firstName: String, var lastName : String)
var person = Person("John", "Doe")

person.apply { this.firstName = "Bruce" }
print(person) // => Person(firstName=Bruce, lastName=Doe)

// 'apply' retorna la instancia original.
person.apply { this.firstName = "Bruce" }.firstName = "Steve"
print(person) // => Person(firstName=Steve, lastName=Doe)
```

[Let]

La función '*let*' toma el objeto sobre el que se invoca como parámetro y devuelve el resultado de la expresión lambda. Es útil cuando desea ejecutar algún código en un objeto antes de devolver algún valor diferente y no necesita mantener una referencia al original:

```
fun main(args: Array<String>) {
    var str = "Hello World"
    str.let { println("$it!!") } // => Hello World!!
    println(str) // => Hello World
}

var strLength = str.let { "$it function".length } // devuelve el resultado de la
expresión lambda
println("strLength is $strLength") // => strLength is 25
```

[With]

La función **'with'** es una función de nivel superior diseñada para los casos en los que desea llamar a múltiples funciones en un objeto y no desea repetir el receptor cada vez. La función **'with'** acepta un receptor y un cierre para operar en dicho receptor:

```
data class Person(var firstName: String, var lastName : String)
var person = Person("John", "Doe")

with(person)
{
    firstName = "Bruce"
    lastName = "Doe"
}

// notación sin 'with'
person.firstName = "John"
person.lastName = "Doe"
```

La última expresión en un bloque **'with'** se retorna como resultado:

```
var name = with(person)
{
    firstName = "John"
    lastName = "Doe"
    "$firstName $lastName" // se retorna este valor y se almacena en 'name'
}
println(name) // => John Doe
```

[Run]

'Run' es una función que combina las características de **'with'** y **'let'**. Esto significa que se pasa una expresión lambda a la función **'run'** y la instancia del objeto es el receptor. El valor de retorno de la expresión lambda se usa como valor de retorno:

```
person.run {
    this.firstName = "Bruce"
}
print(person) // => Person(firstName=Bruce, lastName=Doe)
```

La diferencia clave entre **'let'** y **'run'** es que con **'run'** el receptor es la instancia, mientras que en **'let'**, el argumento de la expresión lambda es la instancia.

[Repeat]

Esta función acepta un entero y una función literal. La función literal será invocada las veces indicadas por el valor entero.

```
repeat(10, { println("Hello") })
```

[Lazy]

La función '*lazy*' es una función cuya utilidad es envolver funciones costosas en términos de rendimiento o de recursos y que serán invocadas cuando sean requeridas por primera vez. La ventaja de utilizar esta función proporcionada por la biblioteca estándar de Kotlin es que el compilador mantendrá la invocación sincronizada evitando que sea invocada más de una vez.

```
fun readStringFromDatabase(): String = ... // expensive operation
val lazyString = lazy { readStringFromDatabase() }
```

[Use]

La función '*use*' es similar a la declaración '*try-with-resources*' presente en Java 7. La función '*use*' se define como una función de extensión de la interfaz '*Closeable*'. Ejecuta la función y luego 'cierra' el recurso de forma segura.

Assertions

Kotlin proporciona un conjunto de funciones que nos permiten agregar una cantidad limitada de **especificaciones formales** a nuestro código. Una especificación formal es una aserción que siempre debe ser verdadera o falsa en la ubicación cuando se ejecuta la aserción. Estos también se conocen como contratos o diseño por contrato:

- *_require()* y *_requireNotNull()* lanza una excepción de tipo '*IllegalArgumentException*' y se utiliza para garantizar que los argumentos cumplan el contrato.
- *'assert()'* lanza una excepción '*AssertionException*' y se utiliza para garantizar que nuestro estado interno es consistente.
- *'check()'* y *'error()'* lanza una excepción '*IllegalStateException*' y también se usa para mantener la consistencia del estado interno.

Estas funciones son similares. La clave que las diferencia es el tipo de excepción que se plantea.

```
fun neverEmpty(str: String) {
    require(str.length > 0, { "String should not be empty" })
}
```

```

    println(str)
}

fun foo(k: Int, value: Boolean) {
    require(k > 10, { "k should be greater than 10" }) // => throws an
    IllegalArgumentException
    requireNotNull(k) // => throws an IllegalArgumentException if the value is
    null.
    check(value) // => throws an IllegalStateException if the value is false
    if (k == 20) error("Error: k == 20") // => throws an IllegalStateException
}

```

Excepciones

En Kotlin todas las excepciones son subclases de la clase *'Throwable'*. Cada excepción tiene un mensaje, un seguimiento de la pila y una causa opcional. **Kotlin no tiene *'checked exceptions'* a diferencia de Java, que realiza la distinción entre tipos de excepciones.**

Para lanzar un objeto de excepción, se utiliza la palabra clave *'throw'*:

```
throw Exception("Message")
```

Para capturar una excepción lanzada se utiliza un bloque *'try'*:

```

try {
    // some code
}
catch (e: SomeException) {
    // handler
}
finally {
    // optional finally block
}

```

Puede haber 0 o más bloques *'catch'*. Los bloques *'finally'* son opcionales y puede omitirse. Sin embargo, tiene que haber al menos un bloque *'catch'* o *'finally'*.

Al igual que muchas otras instrucciones en Kotlin, *'try'* es una expresión y por tanto puede devolver un valor:

```
val a: Int? = try { parseInt(input) } catch (e: NumberFormatException) { null }
```

El valor devuelto por un **'try'** que actúa como expresión es la última expresión en el bloque **'try'** o la última expresión en el bloque **'catch'**. El contenido del bloque **'finally'** no afecta al resultado de la expresión.

'throw' es una expresión en Kotlin, así que se puede usar, por ejemplo, como parte de una *'Elvis expression'*:

```
val s = person.name ?: throw IllegalArgumentException("Name required")
```

El tipo de retorno de una expresión **'throw'** es el tipo especial **'Nothing'**. Este tipo no tiene valores y se utiliza para marcar ubicaciones del código que nunca se pueden alcanzar.

```
fun fail(message: String): Nothing {
    throw IllegalArgumentException(message)
}
```

Cuando llame a la función del ejemplo anterior, el compilador sabrá que la ejecución no continúa más allá de la llamada:

```
val s = person.name ?: fail("Name required")
println(s)    // 's' is known to be initialized at this point
```

Otro caso en el que puede encontrar este tipo es la inferencia de tipos. La variante *'nullable'* de este tipo, **'Nothing?'**, tiene exactamente un valor posible, que es el valor **'null'**. Si se usa el valor nulo para inicializar un valor de un tipo inferido y no hay otra información que se pueda usar para determinar un tipo más específico, el compilador inferirá el tipo **'Nothing?'**:

```
val x = null           // 'x' tiene el tipo `Nothing?`
val l = listOf(null)   // 'l' tiene el tipo `List<Nothing?>
```

Anotaciones



Las **anotaciones** permiten a los desarrolladores agregar un significado adicional a las clases, interfaces, parámetros, etc., en el momento de la compilación. Las anotaciones pueden ser utilizadas por el compilador o por su propio código a través de la reflexión en tiempo de ejecución. Dependiendo del valor de la anotación, el significado del programa o los datos puede cambiar.

[@JvmStatic]

Kotlin representa funciones de nivel de paquete (funciones fuera de una clase) como métodos estáticos. Kotlin también puede generar métodos estáticos para funciones definidas en *'objects'* y *'companion objects'* si anota esas funciones como *'@JvmStatic'*. Si usa esta anotación, el compilador generará tanto un método estático en la clase envolvente del objeto como un método de instancia en el propio objeto.

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

// Ahora 'foo()' es estático en Java pero no 'bar()'
C.foo(); // correcto
C.bar(); // error: 'bar()' no es un método estático
C.Companion.foo(); // correcto
C.Companion.bar(); // la única forma de invocar a 'bar()'

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

// In Java:
Obj.foo(); // correcto
Obj.bar(); // error
```

```
Obj.INSTANCE.bar(); // correcto, una llamada a través de la instancia 'Singleton'
Obj.INSTANCE.foo(); // correcto
```

[@Throws]

Dado que todas las excepciones en Kotlin son '*unchecked exceptions*', no es necesario agregar una lista de posibles excepciones a las firmas de métodos como las que hay en Java. Sin embargo, es posible que deseamos informar a los usuarios de Java que nuestra API produce excepciones en ciertas situaciones. Podemos hacer esto utilizando la anotación '*@Throws*', que se utiliza para indicar al compilador que genere cláusulas de lanzamiento en los métodos generados.

```
@Throws(FileNotFoundException::class)
fun fileExists(path: String) {
    // ...
}
```

[@JvmOverloads]

Dada una función con parámetros por defecto, '*@JvmOverloads*' hará que el compilador cree múltiples métodos sobrecargados para cada parámetro predeterminado.

[@JvmName]

Podemos cambiar el nombre del fichero creado por Kotlin con la anotación '*@JvmName*':

```
// example.kt (sin @JvmName)
package demo

class Foo

fun bar() { ... }

// En Java
new demo.Foo();
demo.ExampleKt.bar();
```

```
// Usamos la anotación '@JvmName' al principio del fichero para indicar al
compilador el nombre del fichero
@file:JvmName("DemoUtils")

package demo

class Foo

fun bar() { ... }

// Ahora en Java
new demo.Foo();
demo.DemoUtils.bar();
```

[@JvmMultifileClass]

Además de indicarle al compilador el nombre del fichero con '*@JvmName*' podemos indicarle que combine todas las funciones de nivel superior de varios ficheros en Kotlin en una única clase Java con la anotación '*@JvmMultifileClass*'.

[Reflection]



Reflection es el nombre dado a la inspección del código en tiempo de ejecución en lugar de tiempo de compilación. Puede usarse para crear instancias de clases, buscar funciones e invocarlas, inspeccionar anotaciones, buscar campos y descubrir parámetros y genéricos, todo sin conocer esos detalles en el momento de la compilación.

Por ejemplo, si necesitamos persistir tipos en una base de datos y a priori no conocemos el tipo de datos podemos utilizar la reflexión para conocer el tipo de datos en tiempo de ejecución y crear la SQL apropiada a ese tipo.

Para usar la reflexión en Kotlin hay que importar el paquete *kotlin.reflect*.

'*KClass*' es el tipo central utilizado en la reflexión de Kotlin. Cada tipo tiene una instancia de '*KClass*' en tiempo de ejecución que contiene detalles de las funciones, propiedades, anotaciones, etc., para ese tipo. Para obtener una instancia de '*KClass*' para cualquier tipo, usamos la sintaxis especial '*::class*' en una instancia de ese tipo:

```
val name = "George"
val kclass = name::class // => class kotlin.String

data class Person(val firstName: String, val lastName: String)
println(Person::class.qualifiedName) // => Person
println(Person::class.isData) // => true
```

Podemos obtener una referencia a la clase utilizando el '*fully qualified name or FQN*' de la clase y la API 'reflection' de Java. Si el compilador no encuentra la clase lanza una '*ClassNotFoundException*':

```
package com.example
data class Person(val firstName: String, val lastName: String)

val kClass = Class.forName("com.example.Person").kotlin // => class
com.example.Personal
```

Para crear instancias de tipo sin conocer el tipo en tiempo de ejecución podemos invocar la función `'createInstance()'` en una referencia de `'KClass'`. Podemos usar esta función con clases sin parámetros o con parámetros opcionales, es decir, que tengan valor por defecto:

```
class PositiveInteger(value: Int = 0)

fun createInteger(kclass: KClass<PositiveInteger>): PositiveInteger {
    return kclass.createInstance()
}
```

Podemos devolver una lista de todos los constructores declarados en un tipo dado usando la propiedad `'constructors'` disponible en el tipo `'KClass'`. Podemos instanciar una clase usando el constructor con la instrucción `'call'` o `'callBy'`:

```
class Person constructor(val firstName: String, val lastName: String)

fun <T : Any> printConstructors(kclass: KClass<T>) {
    kclass.constructors.forEach {
        println(it.parameters)
    }
}

printConstructors(Person::class) // Muestra el/los constructor/es de la clase 'Person'

// Recupera el primer constructor. Si no encuentra ninguno lanza una excepción.
val constructor = Person::class.constructors.first()
val person = constructor.call("John", "Doe") // Invocar al constructor con 'call'
println(person.firstName) // => John
```

Además de los constructores de una clase, también podemos acceder y listar las funciones de una clase con la propiedad `'functions'` disponible en el tipo `'KClass'`:

```
class Person constructor(val firstName: String, val lastName: String) {
    fun getName(): String {
        return "$firstName $lastName"
    }
}

fun <T : Any> printFunctions(kclass: KClass<T>) {
    kclass.functions.forEach {
        println(it.name)
    }
}

printFunctions(Person::class) // => getName equals hashCode toString
```

```
val function = Person::class.functions.find { it.name == "getName" }  
val person = Person("John", "Doe")  
function?.call(person) // => John Doe
```

Coroutines



(todo)

Testing



KotlinTest es el framework para probar y testear el código en Kotlin. Añadir la dependencia a Gradle: `testCompile 'io.kotlintest:kotlintest:x.y.z'`.

Normalmente, para mantener ordenada la estructura del proyecto los ficheros de test se ubican en `src/test/kotlin`

Una especificación o 'spec' es simplemente la manera en que las pruebas se presentan en los archivos de clase. Hay varias especificaciones diferentes disponibles como **FunSpec**, ****StringSpec***, **ShouldSpec**. etc...

La especificación **FunSpec** permite crear pruebas similares al estilo *jUnit*. Para escribir un test unitario invocamos la función 'test' que toma dos parámetros. El primer parámetro es una descripción de la prueba unitaria y el segundo es una función literal que contiene el cuerpo de la prueba. La descripción o nombre de la prueba aparecerá en la salida, así que permite saber que prueba/s han pasado la prueba y cuáles han fallado.

```
class StringTestWithFunSpec : FunSpec() {
    init {
        test("String.startsWith should be true for a prefix") {
            "helloworld".startsWith("hello") shouldBe true
        }
        test("String.endsWith should be true for a prefix") {
            "helloworld".endsWith("world") shouldBe true
        }
    }
}
```

La especificación **StringSpec** es la especificación recomendada por los autores de Kotlin y es la especificación más simple y compacta ya que reduce la sintaxis al mínimo. Se escribe una cadena seguida de una expresión lambda para probar el código:

```
class StringTestWithStringSpec : StringSpec() {
    init {
        "strings.length should return size of string" {
```

```

        "hello".length shouldBe 5
        "hello" shouldBe haveLength(5)
    }
}

```

La especificación **ShouldSpec** es similar a **FunSpec** pero usa la palabra clave '*should*' en vez de '*test*':

```

class StringTestWithShouldSpec : ShouldSpec() {
    init {
        should("return the length of the string") {
            "sammy".length shouldBe 5
            "".length shouldBe 0
        }
        // Nested form
        "String.length" {
            should("return the length of the string") {
                "sammy".length shouldBe 5
                "".length shouldBe 0
            }
        }
    }
}

```

La especificación **WordSpec** usa también la palabra clave '*should*'. Esta especificación permite anidar las pruebas:

```

class StringTestWithWordSpec : WordSpec() {
    init {
        "String.length" should {
            "return the length of the string" {
                "sammy".length shouldBe 5
                "".length shouldBe 0
            }
        }
    }
}

```

La especificación **BehaviorSpec** utiliza las palabras clave '*given*', '*when*' y '*then*' para crear pruebas unitarias más cercanas al lenguaje natural:

```

class StringTestWithBehaviorSpec : BehaviorSpec() {
    init {

```

```

given("a stack") {
    val stack = Stack<String>()
    `when`("an item is pushed") {
        stack.push("kotlin")
        then("the stack should not be empty") {
            stack.isEmpty() shouldBe true
        }
    }
    `when`("the stack is popped") {
        stack.pop()
        then("it should be empty") {
            stack.isEmpty() shouldBe false
        }
    }
}
}
}

```

La especificación **FeatureSpec** es similar a la especificación **BehaviorSpec** pero utiliza las palabras clave *'feature'* y *'scenario'*:

```

class StringTestWithFeatureSpec : FeatureSpec() {
    init {
        feature("Hello World") {
            scenario("should starts with 'Hello'") {
                "Hello World".startsWith("Hello")
            }
            scenario("should ends with 'World'") {
                "Hello World".endsWith("World")
            }
        }
    }
}

```

Los **matchers** prueban alguna propiedad, indicada por el nombre del **matcher**, más allá de la simple igualdad. Por ejemplo, un comparador puede verificar si una cadena está vacía o si un entero es positivo.

```

// [String matchers]
class StringTestWithDifferentMatchers : StringSpec() {
    init {
        "Tests string prefixes" {
            "Hello".startsWith("He") shouldBe true
            "Hello" should startWith("He")
        }
        "Tests substrings" {
            "Hello" should include("el")
        }
    }
}

```

```

    }
    "Test string suffixes" {
        "Hello".endsWith("llo") shouldBe true
        "Hello" shouldBe endWith("llo")
    }
    "Tests the length of a string" {
        "Hello".length shouldBe 5
        "Hello" shouldBe haveLength(5)
    }
    "Tests the equality using a regular expression" {
        "Hello" shouldBe match("He...")
    }
}

// [Collection matchers]
class CollectionTestWithDifferentMatchers : StringSpec() {
    private val listWithDifferentIntegers = listOf(1, 2, 3, 4, 5)
    private val mapWithKeyAndValues = mapOf<Int, String>(1 to "Hello", 2 to
"World")

    init {
        "Tests that a collection should contain the given element" {
            listWithDifferentIntegers shouldBe contain(3)
        }
        "Test the size of the collection" {
            listWithDifferentIntegers shouldBe haveSize<Int>(5)
        }
        "Tests that the collections should be sorted" {
            listWithDifferentIntegers shouldBe sorted<Int>()
        }
        "Tests that the collection has a single element that is equal to the given
element" {
            listWithDifferentIntegers shouldNotBe singleElement(2)
        }
        "Tests that the collection contains all the given elements. The order of
these elements does not matter." {
            listWithDifferentIntegers shouldBe containsAll(1, 2, 4)
        }
        "Tests whether the collection is empty or not" {
            listWithDifferentIntegers shouldNotBe beEmpty<Int>()
        }
        "Tests whether the map contains mapping from a key to any value" {
            mapWithKeyAndValues shouldBe haveKey(2)
        }
        "Tests whether the map contains the value for at least one key" {
            mapWithKeyAndValues shouldBe haveValue("Hello")
        }
        "Tests that the map contains the exact mapping of the key to the value" {
            mapWithKeyAndValues shouldBe contain(2, "World")
        }
    }
}

```



```

    }
}

// [Floating point matchers]
// En valores en punto flotante más que la igualdad absoluta se utiliza la
'tolerancia' que es el valor mínimo entre dos valores que satisfacen el criterio
de igualdad
class FloatNumberTestWithTolerance : StringSpec() {
    private val randomDouble = 18.005
    private val enoughDouble = 18.006

    init {
        "Test if two numbers are equals" {
            randomDouble shouldNotBe equals(enoughDouble)
            randomDouble shouldBe (enoughDouble plusOrMinus 0.01)
        }
    }
}

// [Exception matchers]
// 'shouldThrow' fallará si se lanza una excepción diferente
class ExceptionTest : StringSpec() {
    init {
        "Testing IllegalArgumentException" {
            shouldThrow<IllegalArgumentException> {
                addNumberToTwo(10.0) shouldEqual 10.5
            }
        }
    }
}

@Throws(IllegalArgumentException::class)
fun addNumberToTwo(a: Any): Int {
    if (a !is Int) {
        throw IllegalArgumentException("Number must be an integer")
    }
    return 2 + a
}

```

Los **matchers** se pueden combinar usando los operadores de la lógica booleana como **'and'** y **'or'**:

```

class CombiningMatchers : StringSpec() {
    init {
        "Combining matchers" {
            "Hello World" should (startWith("Hel") and endWith("rld"))
        }
    }
}

```

Un **inspector** en KotlinTest es la forma más fácil de probar el contenido de *'collections'*:

```
val kings = listOf("Stephen I", "Henry I", "Henry II", "Henry III", "William I",
"William III")

class InspectorTests : StringSpec() {
    init {
        "all kings should have a regal number" {
            forAll(kings) {
                it should endWith("I")
            }
        }
        "only one king has the name Stephen" {
            forOne(kings) {
                it should startWith("Stephen")
            }
        }
        "some kings have regal number II" {
            forSome(kings) {
                it should endWith("II")
            }
        }
        "at least one King has the name Henry" {
            forAtLeastOne(kings) {
                it should startWith("Henry")
            }
        }
    }
}
```

A veces es posible que sea necesario ejecutar algo de código, antes de que se ejecuten las pruebas o después de que se completen todas las pruebas (sean exitosas o no). Esto se puede lograr mediante el uso de la clase abstracta *'ProjectConfig'*. Para usar esto, simplemente se crea un objeto que extienda de esta clase abstracta y asegurarse que esté en la ruta de la clase. KotlinTest lo encontrará automáticamente y lo invocará:

```
object codeExecutionBeforeAndAfterTestCases : ProjectConfig() {
    override fun beforeAll() {
        // ...code
    }

    override fun afterAll() {
        // ...code
    }
}
```

Java Interop



Calling Java from Kotlin

Kotlin está diseñado teniendo en cuenta la interoperabilidad de Java. El código Java existente puede llamarse desde Kotlin de una manera natural, y el código Kotlin también se puede usar desde Java sin problemas.

Casi todo el código de Java se puede utilizar sin problemas:

```
import java.util.*

fun demo(source: List<Int>) {
    val list = ArrayList<Int>()
    // 'for'-loops work for Java collections:
    for (item in source) {
        list.add(item)
    }
    // Operator conventions work as well:
    for (i in 0..source.size - 1) {
        list[i] = source[i] // get and set are called
    }
}
```

'Getters' and 'Setters'

Los métodos que siguen las convenciones de Java para *'getters'* y *'setters'* (métodos sin argumentos con nombres que comienzan con 'get' y métodos con argumentos únicos con nombres que comienzan con 'set') se representan como **propiedades** en Kotlin.

Los métodos de acceso booleanos (donde el nombre del *'getter'* comienza con 'is' y el nombre del *'setter'* comienza con 'set') se representan como propiedades que tienen el mismo nombre que el método *'getter'*:

```
import java.util.Calendar

fun calendarDemo() {
    val calendar = Calendar.getInstance()
    if (calendar.firstDayOfWeek == Calendar.SUNDAY) { // call getFirstDayOfWeek()
        calendar.firstDayOfWeek = Calendar.MONDAY // call setFirstDayOfWeek()
    }
    if (!calendar.isLenient) { // call isLenient()
        calendar.isLenient = true // call setLenient()
    }
}
```

Si la clase Java solo tiene un *'setter'*, no será visible como una propiedad en Kotlin, ya que Kotlin no admite propiedades que tengan únicamente el método *'setter'*.

'Void' como retorno

Si un método Java devuelve *'void'*, devolverá *'Unit'* cuando se llame desde Kotlin. Si, por casualidad, alguien usa ese valor de retorno, el compilador de Kotlin lo asignará en el sitio de la llamada, ya que el valor en sí mismo se conoce de antemano (es *'Unit'*).

Escapar palabras clave en Kotlin

Algunas de las palabras clave de Kotlin son identificadores válidos en Java, como por ejemplo *'in'*, *'object'*, *'is'*, etc... Si una biblioteca de Java usa una palabra clave de Kotlin para un método, se puede escapar usando las comillas invertidas (```):

```
// Java
public class Date {
    public void when(str:String) { .... }
}

// Kotlin
date.`when`("2016")
```

Null-Safety

Cualquier referencia en Java puede ser nula, lo que hace que los requisitos de Kotlin de seguridad con los valores nulos no sean prácticos para los objetos procedentes de Java. Los tipos de declaraciones de Java se tratan especialmente en Kotlin y se llaman *'platform types'*. Los controles nulos son relajados para tales tipos, por lo que las garantías de seguridad para ellos son las mismas que en Java.

```
val list = ArrayList<String>() // non-null (constructor result)
list.add("Item")
val size = list.size // non-null (primitive int)
val item = list[0] // platform type inferred (ordinary Java object)

item.substring(1) // allowed, may throw an exception if item == null
```

'Checked exceptions'

Kotlin no tiene '*checked exceptions*'. Por lo tanto, los métodos Java que tienen '*checked exceptions*' se tratan de la misma manera que el resto de métodos.

Calling Kotlin from Java

Al igual que Java se puede usar sin problemas en Kotlin, Kotlin se puede usar fácilmente desde Java.

Top-level functions

La JVM no admite funciones de nivel superior. Por lo tanto, para hacer que funcionen con Java, el compilador Kotlin crea una clase Java con el nombre del paquete. Las funciones se definen luego como métodos estáticos Java en esta clase, que deben ser instanciados antes de su uso.

```
// Kotlin
package org.example.utils
fun cube(n: Int): Int = n * n * n

// Java
import org.example.utils.Utils;
UtilsKt.cube(3);
```

Como se indica en la sección de "Anotaciones", podemos indicar al compilador el nombre del fichero con la anotación '@JvmName':

```
// Kotlin
@file:JvmName("Utils")
package org.example.utils
fun cube(n: Int): Int = n * n * n

// Java
```

```
import org.example.utils.Utills;
Utills.cube(3);
```

Default parameters

la JVM no tiene soporte para los parámetros por defecto. Por lo tanto, cuando una función se define con los valores predeterminados, el compilador debe crear una sola función sin los parámetros predeterminados. Sin embargo, podemos indicarle al compilador que cree múltiples sobrecargas de la función para cada parámetro predeterminado con la anotación

'*@JvmOverloads*'. Luego, los usuarios de Java pueden ver las diversas funciones y elegir cuál es la más adecuada. Esta anotación funciona tanto para constructores, funciones o métodos estáticos:

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

// Java
// Constructors:
Foo(int x, double y)
Foo(int x)

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

Objects and static methods

Los '*named objects*' y los '*companion objects*' se generan como instancias '*singleton*' de una clase. Sin embargo, podemos indicar al compilador que genere la función como una método estático en Java con la anotación '*@JvmStatic*':

```
// Kotlin
object Console {
    fun clear() : Unit { } // Normal
    @JvmStatic fun exit() : Unit { } // Con anotación
}

// Java
Console.INSTANCE.clear() // Normal
Console.exit() // Con anotación
```

Checked exceptions

En Java, solo podemos detectar las '*checked exceptions*' si están declaradas en el método, incluso si el cuerpo del método lanza esa excepción. Por lo tanto, si tenemos una función que se utilizará desde Java y queremos permitir que las personas detecten una excepción, debemos informar al compilador para que agregue la excepción a la firma del método. Para ello usamos la anotación '@Throws':

```
// Kotlin
@Throws(IOException::class)
fun createDirectory(file: File) {
    if (file.exists()) throw IOException("Directory already exists")
    file.createNewFile()
}

// Java
try {
    UtilsKt.createDirectory(new File("file.txt"));
} catch (IOException e) {
    // handle exception here
}
```

Summary



BASICS

"Hello World" program

```
fun main(args: Array<String>) {  
    println("Hello, World")  
}
```

Declaring function

```
fun sum(a: Int, b: Int): Int {  
    return a + b  
}
```

Single-expression function

```
fun sum(a: Int, b: Int) = a + b
```

Declaring variables

```
val name = "Marcin" // Can't be changed  
var age = 5 // Can be changed  
age++
```

Variables with nullable types


```

var name: String? = null
val length: Int
length = name?.length ?: 0
// length, or 0 if name is null
length = name?.length ?: return
// length, or return when name is null
length = name?.length ?: throw Error()
// length, or throw error when name is null

```

CONTROL STRUCTURES

'If' as an expression

```

fun bigger(a: Int, b: Int) = if (a > b) a else b

```

'For' loop

```

val list = listOf("A", "B", "C")
for (element in list) {
    println(element)
}

```

'When' expression

```

fun numberTypeName(x: Number) = when(x) {
    0 -> "Zero" // Equality check
    in 1..4 -> "Four or less" // Range check
    5, 6, 7 -> "Five to seven" // Multiple values
    is Byte -> "Byte" // Type check
    else -> "Some number"
}

```

'When' expression with predicates

```
fun signAsString(x: Int) = when {
    x < 0 -> "Negative"
    x == 0 -> "Zero"
    else -> "Positive"
}
```

CLASSES

Primary constructor

```
// val declares a read-only property, var a mutable one
class Person(val name: String, var age: Int)
// name is read-only, age is mutable
```

Inheritance

```
open class Person(val name: String) {
    open fun hello() = "Hello, I am $name"
    // Final by default so we need open
}

class PolishPerson(name: String) : Person(name) {
    override fun hello() = "Dzień dobry, jestem $name"
}
```

Properties with accessors

```
class Person(var name: String, var surname: String) {
    var fullName: String
    get() = "$name $surname"
    set(value) {
        val (first, rest) = value.split(" ", limit = 2)
        name = first
        surname = rest
    }
}
```

Data classes

```
data class Person(val name: String, var age: Int)
val mike = Person("Mike", 23)

// Modifier data adds:
// 1. toString that displays all primary constructor properties
print(mike.toString()) // Person(name=Mike, age=23)

// 2. equals that compares all primary constructor properties
print(mike == Person("Mike", 23)) // True
print(mike == Person("Mike", 21)) // False

// 3. hashCode that is based on all primary constructor properties
val hash = mike.hashCode()
print(hash == Person("Mike", 23).hashCode()) // True
print(hash == Person("Mike", 21).hashCode()) // False

// 4. component1, component2 etc. that allows deconstruction
val (name, age) = mike
print("$name $age") // Mike 23

// 5. copy that returns copy of object with concrete properties changed
val jake = mike.copy(name = "Jake")
```

COLLECTION LITERALS

```
listOf(1,2,3,4) // List<Int>
mutableListOf(1,2,3,4) // MutableList<Int>

setOf("A", "B", "C") // Set<String>
mutableSetOf("A", "B", "C") // MutableSet<String>

arrayOf('a', 'b', 'c') // Array<Char>

mapOf(1 to "A", 2 to "B") // Map<Int, String>
mutableMapOf(1 to "A", 2 to "B")
// MutableMap<Int, String>

sequenceOf(4,3,2,1) // Sequence<Int>

1 to "A" // Pair<Int, String>

List(4) { it * 2 } // List<Int>
generateSequence(4) { it + 2 } // Sequence<Int>
```

COLLECTION PROCESSING

```

students
  .filter { it.passing && it.averageGrade > 4.0 }
  // Only passing students
  .sortedByDescending { it.averageGrade }
  // Starting from ones with biggest grades
  .take(10) // Take first 10
  .sortedWith(compareBy({ it.surname }, { it.name }))
  // Sort by surname and then name

generateSequence(0) { it + 1 }
  // Infinite sequence of next numbers starting on 0
  .filter { it % 2 == 0 } // Keep only even
  .map { it * 3 } // Triple every one
  .take(100) // Take first 100
  .average() // Count average

// Most important functions for collection processing
val l = listOf(1,2,3,4)
//filter - returns only elements matched by predicate
l.filter { it % 2 == 0 } // [2, 4]

// map - returns elements after transformation
l.map { it * 2 } // [2, 4, 6, 8]

// flatMap - returns elements yielded from results of trans.
l.flatMap { listOf(it, it + 10) } // [1, 11, 2, 12, 3, 13, 4, 14]

// fold/reduce - accumulates elements
l.fold(0.0) { acc, i -> acc + i } // 10.0
l.reduce { acc, i -> acc * i } // 24

// forEach/forEach - performs an action on every element
l.forEach { print(it) } // Prints 1234, returns Unit
l.forEach { print(it) } // Prints 1234, returns [1, 2, 3, 4]

// partition - splits into pair of lists
val (even, odd) = l.partition { it % 2 == 0 }
print(even) // [2, 4]
print(odd) // [1, 3]

// min/max/minBy/maxBy
l.min() // 1, possible because we can compare Int
l.minBy { -it } // 4
l.max() // 4, possible because we can compare Int

```

```

1.maxBy { -it } // 1

// first/firstBy
1.first() // 1
1.first { it % 2 == 0 } // 2 (first even number)

// count - count elements matched by predicate
1.count { it % 2 == 0 } // 2

// sorted/sortedBy - returns sorted collection
listOf(2,3,1,4).sorted() // [1, 2, 3, 4]
1.sortedBy { it % 2 } // [2, 4, 1, 3]

// groupBy - group elements on collection by key
1.groupBy { it % 2 } // Map: {1=[1, 3], 0=[2, 4]}

// distinct/distinctBy - returns only unique elements
listOf(1,1,2,2).distinct() // [1, 2]

```

Mutable vs immutable collection processing functions

```

val list = mutableListOf(3,4,2,1)
val sortedResult = list.sorted() // Returns sorted
println(sortedResult) // [1, 2, 3, 4]
println(list) // [3, 4, 2, 1]

val sortResult = list.sort() // Sorts mutable collection
println(sortResult) // kotlin.Unit
println(list) // [1, 2, 3, 4]

```

EXTENSION FUNCTIONS TO ANY OBJECT

	Returns 'Receiver'	Returns 'Results of lambda'
Reference to receiver: 'it'	also	let
Reference to receiver: 'this'	apply	run/with

```

val dialog = Dialog().apply {
    title = "Dialog title"
    onClick { print("Clicked") }
}

```

FUNCTIONS

Function types

`() -> Unit` - takes no arguments and returns nothing (Unit). `(Int, Int) -> Int` - takes two arguments of type Int and returns Int. `() -> Unit) -> Int` - takes another function and returns Int. `(Int) -> () -> Unit` - takes argument of type Int and returns function.

Function literals

```
// Simple lambda expression
val add: (Int, Int) -> Int = { i, j -> i + j }

val printAndDouble: (Int) -> Int = {
    println(it)
    // When single parameter, we can reference it using `it`
    it * 2 // In lambda, last expression is returned
}

// Anonymous function alternative
val printAndDoubleFun: (Int) -> Int = fun(i: Int): Int {
    println(i) // Single argument can't be referenced by `it`
    return i * 2 // Needs return like any function
}

val i = printAndDouble(10) // 10
print(i) // 20
```

Extension functions

```
fun Int.isEven() = this % 2 == 0
print(2.isEven()) // true

fun List<Int>.average() = 1.0 * sum() / size
print(listOf(1, 2, 3, 4).average()) // 2.5
```

DELEGATES

```

// Lazy - calculates value before first usage
val i by lazy { print("init "); 10 }
print(i) // Prints: init 10
print(i) // Prints: 10

// notNull - returns last setted value, or throws error if no value has been set

// observable/vetoable - calls function every time value changes. In vetoable
function also decides if new value should be set.
var name by observable("Unset") { p, old, new ->
    println("${p.name} changed $old -> $new")
}
name = "Marcin"
// Prints: name changed Unset -> Marcin

// Map/MutableMap - finds value on map by property name
val map = mapOf("a" to 10)
val a by map
print(a) // Prints: 10

```

VISIBILITY MODIFIERS

Modifier	Class members	Top-level
Public (default)	Visible everywhere	Visible everywhere
Private	Visible only in the same class	Visible in the same class
Protected	Visible only in the same class and subclasses	Not allowed
Internal	Visible in the same module if class is accessible	Visible in the same module

PDF version: [Kotlin\[ES\].pdf](#)

Reference



- <https://kotlinlang.org/docs/reference/>
- <https://code.tutsplus.com/series/kotlin-from-scratch--cms-1209>
- <https://www.packtpub.com/application-development/programming-kotlin>
- <https://learnxinyminutes.com/docs/kotlin/>
- <https://gist.github.com/dodyg/5823184>
- <https://gist.github.com/dodyg/5616605>
- <https://github.com/Zhuinden/guide-to-kotlin>

License



Esta obra está bajo una [licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional](#).