EBook Gratis

APRENDIZAJE Kotlin

Free unaffiliated eBook created from
Stack Overflow contributors.

#kotlin

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: kotlin

It is an unofficial and free Kotlin ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Kotlin.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con Kotlin

Observaciones

Kotlin es un lenguaje de programación orientado a objetos de tipo estático desarrollado por JetBrains que se dirige principalmente a la JVM. Kotlin se ha desarrollado con el objetivo de ser rápido de compilar, compatible con versiones anteriores, muy seguro para el tipo, y 100% interoperable con Java. Kotlin también se ha desarrollado con el objetivo de proporcionar muchas de las funciones que buscan los desarrolladores de Java. El compilador estándar de Kotlin permite compilarlo tanto en el bytecode de Java para la JVM como en JavaScript.

Compilando kotlin

Kotlin tiene un complemento IDE estándar para Eclipse e IntelliJ. Kotlin también se puede compilar usando Maven , usando Ant y usando Gradle , o por medio de la línea de comandos .

Vale la pena señalar que en `$ kotlinc Main.kt` generará un archivo de clase java, en este caso `MainKt.class` (`MainKt.class` cuenta que el Kt se adjunta al nombre de la clase). Sin embargo, si uno fuera a ejecutar el archivo de clase utilizando `$ java MainKt` java se lanzará la siguiente excepción:

```
Exception in thread "main" java.lang.NoClassDefFoundError: kotlin/jvm/internal/Intrinsics
    at MainKt.main(Main.kt)
Caused by: java.lang.ClassNotFoundException: kotlin.jvm.internal.Intrinsics
    at java.net.URLClassLoader.findClass(URLClassLoader.java:381)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:424)
    at sun.misc.Launcher$AppClassLoader.loadClass(Launcher.java:335)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:357)
    ... 1 more
```

Para ejecutar el archivo de clase resultante utilizando Java, se debe incluir el archivo jar de tiempo de ejecución Kotlin en la ruta de la clase actual.

```
java -cp .:/path/to/kotlin/runtime/jar/kotlin-runtime.jar MainKt
```

Versiones

Versión	Fecha de lanzamiento
1.0.0	2016-02-15
1.0.1	2016-03-16
1.0.2	2016-05-13
1.0.3	2016-06-30

Versión	Fecha de lanzamiento
1.0.4	2016-09-22
1.0.5	2016-11-08
1.0.6	2016-12-27
1.1.0	2017-03-01
1.1.1	2017-03-14
1.1.2	2017-04-25
1.1.3	2017-06-23

Examples

Hola Mundo

Todos los programas de Kotlin comienzan en la función `main` . Aquí hay un ejemplo de un programa simple "Hello World" de Kotlin:

```
package my.program

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

Coloque el código anterior en un archivo llamado `Main.kt` (este nombre de archivo es completamente arbitrario)

Cuando se dirige a la JVM, la función se compilará como un método estático en una clase con un nombre derivado del nombre de archivo. En el ejemplo anterior, la clase principal a ejecutar sería `my.program.MainKt` .

Para cambiar el nombre de la clase que contiene funciones de nivel superior para un archivo en particular, coloque la siguiente anotación en la parte superior del archivo sobre la declaración del paquete:

```
@file:JvmName("MyApp")
```

En este ejemplo, la clase principal a ejecutar ahora sería `my.program.MyApp` .

Ver también:

- Funciones de nivel de paquete que incluyen la anotación `@JvmName` .
- Anotación de objetivos de uso de sitio

Hola mundo usando una declaración de objeto

Alternativamente, puede usar una Declaración de Objeto que contiene la función principal para un programa Kotlin.

```
package my.program

object App {
    @JvmStatic fun main(args: Array<String>) {
        println("Hello World")
    }
}
```

El nombre de la clase que ejecutará es el nombre de su objeto, en este caso es `my.program.App` .

La ventaja de este método sobre una función de nivel superior es que el nombre de la clase que se ejecutará es más evidente, y cualquier otra función que agregue se incluye en la `App` la clase. También tiene una instancia única de la `App` para almacenar el estado y hacer otro trabajo.

Ver también:

- Métodos estáticos incluyendo la anotación `@JvmStatic`

Hola mundo usando un objeto compañero

Similar a usar una Declaración de Objeto, puede definir la función `main` de un programa Kotlin usando un Objeto Acompañante de una clase.

```
package my.program

class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            println("Hello World")
        }
    }
}
```

El nombre de la clase que ejecutará es el nombre de su clase, en este caso es `my.program.App` .

La ventaja de este método sobre una función de nivel superior es que el nombre de la clase que se ejecutará es más evidente, y cualquier otra función que agregue se incluye en la `App` la clase. Esto es similar al ejemplo de la `Object Declaration` , aparte de que usted tiene el control de crear instancias de cualquier clase para realizar un trabajo adicional.

Una pequeña variación que crea una instancia de la clase para hacer el "hola" real:

```
class App {
    companion object {
        @JvmStatic fun main(args: Array<String>) {
            App().run()
        }
```

```
    }

    fun run() {
        println("Hello World")
    }
}
```

Ver también:

- Métodos estáticos incluyendo la anotación @JvmStatic

Principales métodos utilizando varargs.

Todos estos estilos de métodos principales también se pueden usar con varargs :

```
package my.program

fun main(vararg args: String) {
    println("Hello, world!")
}
```

Compilar y ejecutar el código Kotlin en la línea de comandos

Como Java, proporcione dos comandos diferentes para compilar y ejecutar código Java. Igual que Kotlin también te proporcionan diferentes comandos.

`javac` para compilar archivos java. `java` para ejecutar archivos java.

Igual que `kotlinc` para compilar archivos `kotlin` para ejecutar archivos kotlin.

Lectura de entrada desde la línea de comandos

Los argumentos pasados desde la consola se pueden recibir en el programa Kotlin y se pueden usar como entrada. Puede pasar N (1 2 3 y así sucesivamente) números de argumentos desde el símbolo del sistema.

Un ejemplo simple de un argumento de línea de comando en Kotlin.

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

    println("Enter Two number")
    var (a, b) = readLine()!!.split(' ') // !! this operator use for
NPE(NullPointerException).

    println("Max number is : ${maxNum(a.toInt(), b.toInt())}")
}



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

    var max = if (a > b) {
        println("The value of a is $a");
```

```
        a
    } else {
        println("The value of b is $b")
        b
    }

    return max;

}
```

Aquí, ingrese dos números desde la línea de comando para encontrar el número máximo. Salida:

```
Enter Two number
71 89 // Enter two number from command line

The value of b is 89
Max number is: 89
```

Por !! Operador Por favor, compruebe Null Safety .

Nota: el ejemplo anterior compila y ejecuta en Intellij.

Lea Empezando con Kotlin en línea: https://riptutorial.com/es/kotlin/topic/490/empezando-con-kotlin

Capítulo 2: Advertencias de Kotlin

Examples

Llamando a un toString () en un tipo anulable

Una cosa a tener en cuenta cuando se utiliza el método `toString` en Kotlin es el manejo de null en combinación con el `String?` .

Por ejemplo, desea obtener texto de un `EditText` en Android.

Tendrías un trozo de código como:

```
// Incorrect:
val text = view.textField?.text.toString() ?: ""
```

Se esperaría que si el campo no existiera, el valor fuera una cadena vacía, pero en este caso es `"null"` .

```
// Correct:
val text = view.textField?.text?.toString() ?: ""
```

Lea Advertencias de Kotlin en línea: https://riptutorial.com/es/kotlin/topic/6608/advertencias-de-kotlin

Capítulo 3: Anotaciones

Examples

Declarar una anotación

Las anotaciones son medios para adjuntar metadatos al código. Para declarar una anotación, coloque el modificador de anotación delante de una clase:

```
annotation class Strippable
```

Las anotaciones pueden tener meta-anotaciones:

```
    @Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
    annotation class Strippable
```

Las anotaciones, como otras clases, pueden tener constructores:

```
annotation class Strippable(val importanceValue: Int)
```

Pero a diferencia de otras clases, se limita a los siguientes tipos:

- tipos que corresponden a los tipos primitivos de Java (Int, Long, etc.);
- instrumentos de cuerda
- clases (Foo :: clase)
- enums
- otras anotaciones
- matrices de los tipos enumerados anteriormente

Meta-anotaciones

Al declarar una anotación, se puede incluir metainformación utilizando las siguientes meta-anotaciones:

- `@Target` : especifica los posibles tipos de elementos que se pueden anotar con la anotación (clases, funciones, propiedades, expresiones, etc.)
- `@Retention` especifica si la anotación se almacena en los archivos de clase compilados y si es visible a través de la reflexión en tiempo de ejecución (de forma predeterminada, ambos son verdaderos).
- `@Repeatable` permite usar la misma anotación en un solo elemento varias veces.
- `@MustBeDocumented` especifica que la anotación es parte de la API pública y debe incluirse en la clase o firma de método que se muestra en la documentación de la API generada.

Ejemplo:

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION,
        AnnotationTarget.VALUE_PARAMETER, AnnotationTarget.EXPRESSION)
@Retention(AnnotationRetention.SOURCE)
@MustBeDocumented
annotation class Fancy
```

Lea Anotaciones en línea: https://riptutorial.com/es/kotlin/topic/4074/anotaciones

Capítulo 4: Arrays

Examples

Arreglos Genéricos

Los arrays genéricos en Kotlin están representados por `Array<T>` .

Para crear una matriz vacía, use la función de fábrica `emptyArray<T>()` :

```
val empty = emptyArray<String>()
```

Para crear una matriz con un tamaño dado y valores iniciales, use el constructor:

```
var strings = Array<String>(size = 5, init = { index -> "Item #$index" })
print(Arrays.toString(a)) // prints "[Item #0, Item #1, Item #2, Item #3, Item #4]"
print(a.size) // prints 5
```

Las matrices tienen funciones `get(index: Int): T` y `set(index: Int, value: T)` :

```
strings.set(2, "ChangedItem")
print(strings.get(2)) // prints "ChangedItem"

// You can use subscription as well:
strings[2] = "ChangedItem"
print(strings[2]) // prints "ChangedItem"
```

Arreglos de primitivos

Estos tipos **no se** heredan de `Array<T>` para evitar el boxeo, sin embargo, tienen los mismos atributos y métodos.

Tipo kotlin	Función de fábrica	Tipo JVM
`BooleanArray`	`booleanArrayOf(true, false)`	`boolean[]`
`ByteArray`	`byteArrayOf(1, 2, 3)`	`byte[]`
`CharArray`	`charArrayOf('a', 'b', 'c')`	`char[]`
`DoubleArray`	`doubleArrayOf(1.2, 5.0)`	`double[]`
`FloatArray`	`floatArrayOf(1.2, 5.0)`	`float[]`
`IntArray`	`intArrayOf(1, 2, 3)`	`int[]`
`LongArray`	`longArrayOf(1, 2, 3)`	`long[]`
`ShortArray`	`shortArrayOf(1, 2, 3)`	`short[]`

Extensiones

`average()` se define para `Byte` , `Int` , `Long` , `Short` , `Double` , `Float` y siempre devuelve `Double` :

```
val doubles = doubleArrayOf(1.5, 3.0)
print(doubles.average()) // prints 2.25

val ints = intArrayOf(1, 4)
println(ints.average()) // prints 2.5
```

`component1()` , `component2()` , ... `component5()` devuelve un elemento de la matriz

`getOrNull(index: Int)` devuelve nulo si el índice está fuera de los límites, de lo contrario, un elemento de la matriz

`first()` , `last()`

`toHashSet()` devuelve un `HashSet<T>` de todos los elementos

`sortedArray()` , `sortedArrayDescending()` crea y devuelve una nueva matriz con elementos ordenados de la corriente

`sort()` , `sortDescending` ordena la matriz en el lugar

`min()` , `max()`

Iterar Array

Puede imprimir los elementos de la matriz utilizando el bucle igual que el bucle mejorado de Java, pero necesita cambiar la palabra clave de `:` a `in` .

```
val asc = Array(5, { i -> (i * i).toString() })
for(s : String in asc){
    println(s);
}
```

También puede cambiar el tipo de datos en el bucle.

```
val asc = Array(5, { i -> (i * i).toString() })
for(s in asc){
    println(s);
}
```

Crear una matriz

```
val a = arrayOf(1, 2, 3) // creates an Array<Int> of size 3 containing [1, 2, 3].
```

Crear una matriz utilizando un cierre

```
val a = Array(3) { i -> i * 2 } // creates an Array<Int> of size 3 containing [0, 2, 4]
```

Crear una matriz sin inicializar

```
val a = arrayOfNulls<Int>(3) // creates an Array<Int?> of [null, null, null]
```

La matriz devuelta siempre tendrá un tipo anulable. Las matrices de elementos no anulables no se pueden crear sin inicializar.

Lea Arrays en línea: https://riptutorial.com/es/kotlin/topic/5722/arrays

Capítulo 5: Bucles en Kotlin

Observaciones

En Kotlin, los bucles se compilan en bucles optimizados siempre que sea posible. Por ejemplo, si recorre un rango de números, el código de bytes se compilará en un bucle correspondiente basado en valores int simples para evitar la sobrecarga de creación de objetos.

Examples

Repetir una acción x veces.

```
repeat(10) { i ->
    println("This line will be printed 10 times")
    println("We are on the ${i + 1}. loop iteration")
}
```

Bucle sobre iterables

Puede realizar un bucle sobre cualquier iterable utilizando el bucle for estándar:

```
val list = listOf("Hello", "World", "!")
for(str in list) {
    print(str)
}
```

Muchas cosas en Kotlin son iterables, como rangos de números:

```
for(i in 0..9) {
    print(i)
}
```

Si necesita un índice mientras está iterando:

```
for((index, element) in iterable.withIndex()) {
    print("$element at index $index")
}
```

También hay un enfoque funcional para iterar incluido en la biblioteca estándar, sin construcciones de lenguaje aparentes, utilizando la función forEach:

```
iterable.forEach {
    print(it.toString())
}
```

`it` en este ejemplo dispone implícitamente el elemento actual, ver las funciones lambda

Mientras bucles

Los bucles while y do-while funcionan como lo hacen en otros idiomas:

```
while(condition) {
    doSomething()
}

do {
    doSomething()
} while (condition)
```

En el bucle do-while, el bloque de condición tiene acceso a valores y variables declarados en el cuerpo del bucle.

Romper y continuar

Romper y continuar las palabras clave funcionan como lo hacen en otros idiomas.

```
while(true) {
    if(condition1) {
        continue // Will immediately start the next iteration, without executing the rest of
the loop body
    }
    if(condition2) {
        break // Will exit the loop completely
    }
}
```

Si tiene bucles anidados, puede etiquetar las sentencias de bucle y calificar las sentencias de interrupción y continuación para especificar qué bucle desea continuar o dividir:

```
outer@ for(i in 0..10) {
    inner@ for(j in 0..10) {
        break       // Will break the inner loop
        break@inner // Will break the inner loop
        break@outer // Will break the outer loop
    }
}
```

Sin embargo, este enfoque no funcionará para la construcción funcional `forEach` .

Iterando sobre un mapa en Kotlin

```
//iterates over a map, getting the key and value at once

var map = hashMapOf(1 to "foo", 2 to "bar", 3 to "baz")

for ((key, value) in map) {
    println("Map[$key] = $value")
}
```

Recursion

También es posible realizar bucles a través de la recursión en Kotlin como en la mayoría de los lenguajes de programación.

```
fun factorial(n: Long): Long = if (n == 0) 1 else n * factorial(n - 1)

println(factorial(10)) // 3628800
```

En el ejemplo anterior, la función `factorial` se llamará repetidamente por sí misma hasta que se cumpla la condición dada.

Construcciones funcionales para iteración.

La biblioteca estándar de Kotlin también proporciona numerosas funciones útiles para trabajar iterativamente sobre colecciones.

Por ejemplo, la función de `map` se puede utilizar para transformar una lista de elementos.

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.map { "Number $it" }
```

Una de las muchas ventajas de este estilo es que permite encadenar operaciones de manera similar. Solo se requeriría una pequeña modificación si, por ejemplo, la lista anterior fuera necesaria para ser filtrada para números pares. La función de `filter` puede ser utilizada.

```
val numbers = listOf(1, 2, 3, 4, 5, 6, 7, 8, 9, 0)
val numberStrings = numbers.filter { it % 2 == 0 }.map { "Number $it" }
```

Lea Bucles en Kotlin en línea: https://riptutorial.com/es/kotlin/topic/2727/bucles-en-kotlin

Capítulo 6: Colecciones

Introducción

A diferencia de muchos idiomas, Kotlin distingue entre colecciones mutables e inmutables (listas, conjuntos, mapas, etc.). El control preciso sobre cuándo se pueden editar las colecciones es útil para eliminar errores y para diseñar buenas API.

Sintaxis

- listOf, mapOf y setOf devuelve objetos de solo lectura que no puede agregar ni eliminar elementos.
- Si desea agregar o eliminar elementos, tiene que usar arrayListOf, hashMapOf, hashSetOf, linkedMapOf (LinkedHashMap), linkedSetOf (LinkedHashSet), mutableListOpp una colección de personas de la empresa.), sortedMapOf o sortedSetOf
- Cada colección tiene métodos como las funciones first (), last (), get () y lambda, como filtrar, mapear, unir, reducir y muchos otros.

Examples

Usando la lista

```
// Create a new read-only List<String>
val list = listOf("Item 1", "Item 2", "Item 3")
println(list) // prints "[Item 1, Item 2, Item 3]"
```

Usando el mapa

```
// Create a new read-only Map<Integer, String>
val map = mapOf(Pair(1, "Item 1"), Pair(2, "Item 2"), Pair(3, "Item 3"))
println(map) // prints "{1=Item 1, 2=Item 2, 3=Item 3}"
```

Usando set

```
// Create a new read-only Set<String>
val set = setOf(1, 3, 5)
println(set) // prints "[1, 3, 5]"
```

Lea Colecciones en línea: https://riptutorial.com/es/kotlin/topic/8846/colecciones

Capítulo 7: Configurando la compilación de Kotlin

Examples

Configuracion gradle

`kotlin-gradle-plugin` se usa para compilar el código Kotlin con Gradle. Básicamente, su versión debe corresponder a la versión de Kotlin que desea utilizar. Por ejemplo, si desea usar Kotlin `1.0.3` , también necesita aplicar `kotlin-gradle-plugin` versión `1.0.3` .

Es una buena idea externalizar esta versión en `gradle.properties` o en `ExtraPropertiesExtension` :

```
buildscript {
   ext.kotlin_version = '1.0.3'

   repositories {
     mavenCentral()
   }

   dependencies {
     classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
   }
}
```

Entonces necesitas aplicar este plugin a tu proyecto. La forma en que lo hace difiere cuando se dirige a diferentes plataformas:

JVM de orientación

```
apply plugin: 'kotlin'
```

Apuntando a android

```
apply plugin: 'kotlin-android'
```

Apuntando a js

```
apply plugin: 'kotlin2js'
```

Estas son las rutas por defecto:

- Fuentes de kotlin: `src/main/kotlin`
- fuentes de java: `src/main/java`

- Pruebas de kotlin: `src/test/kotlin`
- Pruebas de java: `src/test/java`
- recursos de tiempo de ejecución: `src/main/resources`
- recursos de prueba: `src/test/resources`

Es posible que deba configurar `SourceSets` si está utilizando un diseño de proyecto personalizado.

Finalmente, deberá agregar la dependencia de la biblioteca estándar de Kotlin a su proyecto:

```
dependencies {
    compile "org.jetbrains.kotlin:kotlin-stdlib:$kotlin_version"
}
```

Si desea utilizar Kotlin Reflection, también deberá agregar la `compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`

Utilizando Android Studio

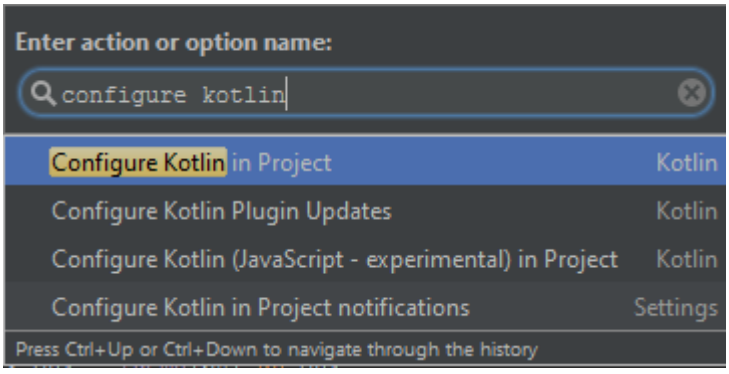
Android Studio puede configurar Kotlin automáticamente en un proyecto de Android.

Instalar el complemento

Para instalar el complemento Kotlin, vaya a Archivo> Configuración> Editor> Complementos> Instalar el complemento JetBrains ...> Kotlin> Instalar, luego reinicie Android Studio cuando se le solicite.

Configurar un proyecto

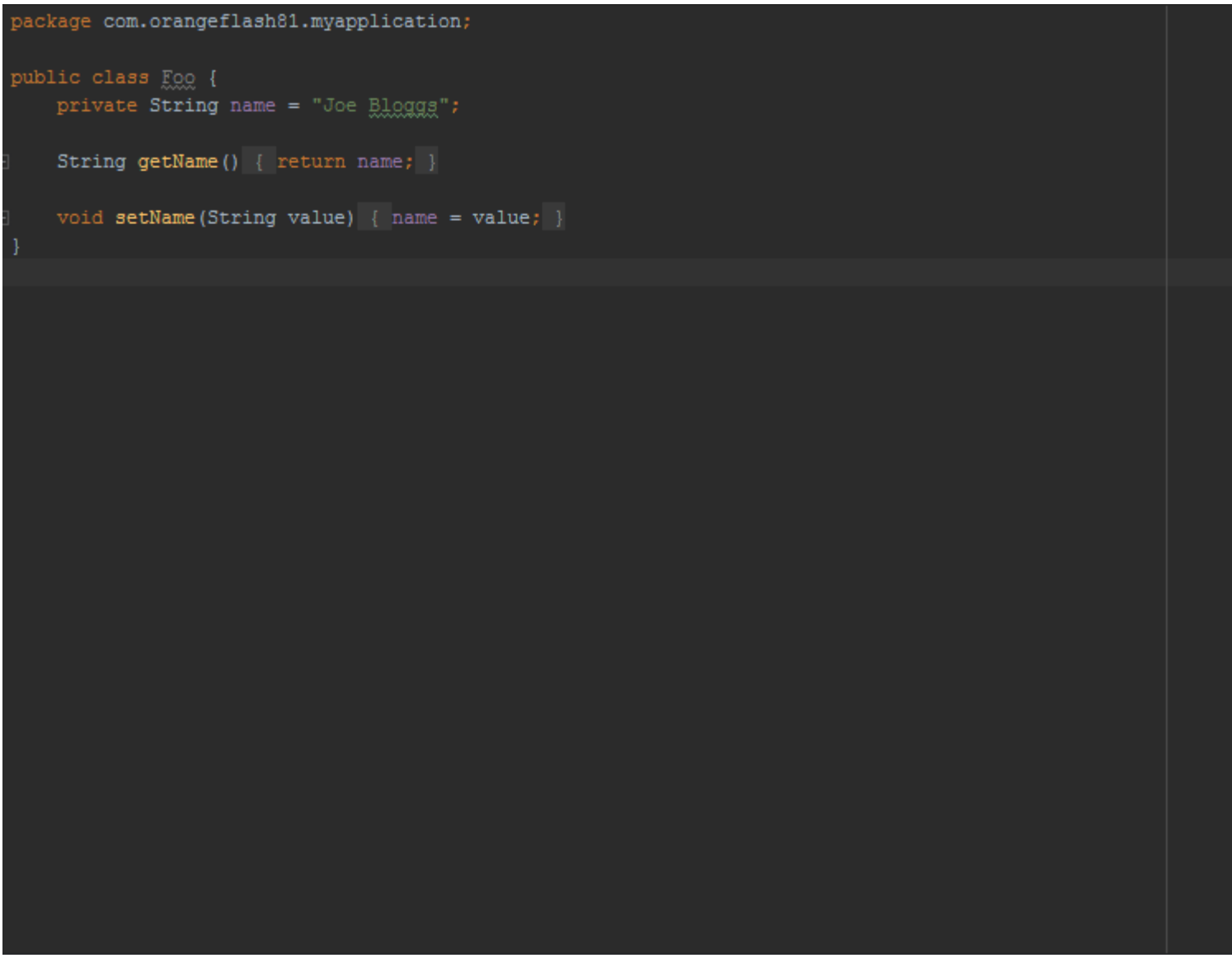
Cree un proyecto de Android Studio normalmente, luego presione `Ctrl` + `Shift` + `A`. En el cuadro de búsqueda, escriba "Configurar Kotlin en el proyecto" y presione Entrar.



Android Studio alterará sus archivos de Gradle para agregar todas las dependencias necesarias.

Convertir Java

Para convertir sus archivos Java a archivos Kotlin, presione `Ctrl` + `Shift` + `A` y busque "Convertir archivo Java a archivo Kotlin". Esto cambiará la extensión del archivo actual a `.kt` y convertirá el código a Kotlin.



Migración de Gradle usando el script Groovy al script Kotlin

Pasos:

- clonar el proyecto gradle-script-kotlin
- Copie / pegue del proyecto clonado a su proyecto:
  - `build.gradle.kts`
  - `gradlew`
  - `gradlew.bat`
  - `settings.gradle`
- actualice el contenido de `build.gradle.kts` según sus necesidades, puede usar como inspiración los scripts en el proyecto que acaba de clonar o en uno de sus ejemplos
- ahora abra Intellij y abra su proyecto, en la ventana del explorador, debe ser reconocido como un proyecto de Gradle, si no, expréselo primero.
- después de abrir, deje que Intellij funcione, abra `build.gradle.kts` y verifique si hay algún

error. Si el resaltado no funciona y / o todo está marcado en rojo, cierre y vuelva a abrir Intellij

- Abre la ventana de Gradle y actualízala.

Si está en Windows, puede encontrar este error , descargue la distribución completa de Gradle 3.3 y utilícela en su lugar. Relacionados .

OSX y Ubuntu funcionan fuera de la caja.

Bono pequeño: si desea evitar todas las molestias de publicación en Maven y similares, use Jitpack , las líneas para agregar son casi idénticas en comparación con Groovy. Puedes inspirarte en este proyecto mío.

Lea Configurando la compilación de Kotlin en línea:
https://riptutorial.com/es/kotlin/topic/2501/configurando-la-compilacion-de-kotlin

Capítulo 8: coroutines

Introducción

Ejemplos de implementación experimental (todavía) de Kotlin de coroutines

Examples

Coroutine simple que demora 1 segundo pero no bloquea.

(del documento oficial)

```
fun main(args: Array<String>) {
    launch(CommonPool) { // create new coroutine in common thread pool
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello,") // main function continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}
```

resultado

```
Hello,
World!
```

Lea coroutines en línea: https://riptutorial.com/es/kotlin/topic/10936/coroutines

Capítulo 9: Declaraciones condicionales

Observaciones

A diferencia del `switch` de Java, la instrucción `when` no tiene un comportamiento de caída. Esto significa que, si una rama coincide, el flujo de control regresa después de su ejecución y no se requiere una declaración de `break` . Si desea combinar los comportamientos para varios argumentos, puede escribir varios argumentos separados por comas:

```
when (x) {
    "foo", "bar" -> println("either foo or bar")
    else -> println("didn't match anything")
}
```

Examples

Declaración if estándar

```
val str = "Hello!"
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

Las ramas else son opcionales en las sentencias if normales.

Declaración-if como una expresión

Las sentencias if pueden ser expresiones:

```
val str = if (condition) "Condition met!" else "Condition not met!"
```

Tenga en cuenta que `else` -branch no es opcional si la declaración- `if` se usa como una expresión.

Esto también se puede hacer con una variante multilínea con corchetes y múltiples declaraciones `else if` .

```
val str = if (condition1){
    "Condition1 met!"
   } else if (condition2) {
    "Condition2 met!"
   } else {
    "Conditions not met!"
   }
```

> CONSEJO: Kotlin puede inferir el tipo de variable para usted, pero si desea estar seguro del tipo, simplemente anótelo en la variable como: `val str: String =` esto forzará el tipo y hará que sea más fácil de leer.

When-statement en lugar de if-else-if chains

La instrucción when es una alternativa a una instrucción if con varias sucursales else-if-branch:

```
when {
    str.length == 0 -> print("The string is empty!")
    str.length > 5  -> print("The string is short!")
    else            -> print("The string is long!")
}
```

El mismo código escrito usando una cadena *if-else-if* :

```
if (str.length == 0) {
    print("The string is empty!")
} else if (str.length > 5) {
    print("The string is short!")
} else {
    print("The string is long!")
}
```

Al igual que con la sentencia if, la rama else es opcional, y puede agregar tantas o tan pocas ramas como desee. También puedes tener ramas multilínea:

```
when {
    condition -> {
        doSomething()
        doSomeMore()
    }
    else -> doSomethingElse()
}
```

Cuando coinciden argumentos de declaración

Cuando se le da un argumento, el `when` -statement coincide con el argumento en contra de las sucursales en secuencia. La coincidencia se realiza utilizando el operador `==` que realiza verificaciones nulas y compara los operandos utilizando la función `equals` . La primera coincidencia será ejecutada.

```
when (x) {
    "English" -> print("How are you?")
    "German" -> print("Wie geht es dir?")
    else -> print("I don't know that language yet :(")
}
```

La instrucción when también conoce algunas opciones de comparación más avanzadas:

```
val names = listOf("John", "Sarah", "Tim", "Maggie")
```

```
when (x) {
    in names -> print("I know that name!")
    !in 1..10 -> print("Argument was not in the range from 1 to 10")
    is String -> print(x.length) // Due to smart casting, you can use String-functions here
}
```

Cuando-declaración como expresión

Como si, cuando también se puede utilizar como una expresión:

```
val greeting = when (x) {
    "English" -> "How are you?"
    "German" -> "Wie geht es dir?"
    else -> "I don't know that language yet :("
}
print(greeting)
```

Para ser utilizada como una expresión, la instrucción when debe ser exhaustiva, es decir, tener una rama else o cubrir todas las posibilidades con las ramas de otra manera.

Cuando-declaración con enumeraciones

`when` se puede utilizar para hacer coincidir los valores de `enum` :

```
enum class Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
}

fun doOnDay(day: Day) {
    when(day) {
        Day.Sunday ->     // Do something
        Day.Monday, Day.Tuesday ->     // Do other thing
        Day.Wednesday ->  // ...
        Day.Thursday ->   // ...
        Day.Friday ->     // ...
        Day.Saturday ->   // ...
    }
}
```

Como puede ver en la segunda línea de casos (`Monday` y `Tuedsay`), también es posible combinar dos o más valores de `enum` .

Si sus casos no son exhaustivos, la compilación mostrará un error. Puedes usar `else` para manejar casos predeterminados:

```
fun doOnDay(day: Day) {
    when(day) {
        Day.Monday ->     // Work
```

```
        Day.Tuesday ->    // Work hard
        Day.Wednesday ->  // ...
        Day.Thursday ->   //
        Day.Friday ->     //
        else ->           // Party on weekend
    }
}
```

Aunque se puede hacer lo mismo usando la construcción `if-then-else` , `when` se cuida de los valores de `enum` faltan y los hace más naturales.

Haga clic aquí para obtener más información sobre kotlin `enum`

Lea Declaraciones condicionales en línea:
https://riptutorial.com/es/kotlin/topic/2685/declaraciones-condicionales

Capítulo 10: Delegación de clase

Introducción

Una clase Kotlin puede implementar una interfaz delegando sus métodos y propiedades a otro objeto que implementa esa interfaz. Esto proporciona una forma de componer el comportamiento utilizando asociación en lugar de herencia.

Examples

Delegar un método a otra clase.

```
interface Foo {
    fun example()
}

class Bar {
    fun example() {
        println("Hello, world!")
    }
}

class Baz(b : Bar) : Foo by b

Baz(Bar()).example()
```

El ejemplo se imprime `Hello, world!`

Lea Delegación de clase en línea: https://riptutorial.com/es/kotlin/topic/10575/delegacion-de-clase

Capítulo 11: Edificio DSL

Introducción

Concéntrese en los detalles de sintaxis para diseñar DSL internos en Kotlin.

Examples

Infix enfoque para construir DSL

Si usted tiene:

```
infix fun <T> T?.shouldBe(expected: T?) = assertEquals(expected, this)
```

Puede escribir el siguiente código similar a DSL en sus pruebas:

```
@Test
fun test() {
  100.plusOne() shouldBe 101
}
```

Anulando el método de invocación para construir DSL

Si usted tiene:

```
class MyExample(val i: Int) {
  operator fun <R> invoke(block: MyExample.() -> R) = block()
  fun Int.bigger() = this > i
}
```

puede escribir el siguiente código similar a DSL en su código de producción:

```
fun main2(args: Array<String>) {
    val ex = MyExample(233)
    ex {
        // bigger is defined in the context of `ex`
        // you can only call this method inside this context
        if (777.bigger()) kotlin.io.println("why")
    }
}
```

Utilizando operadores con lambdas.

Si usted tiene:

```
val r = Random(233)
infix inline operator fun Int.rem(block: () -> Unit) {
```

```
  if (r.nextInt(100) < this) block()
}
```

Puede escribir el siguiente código similar a DSL:

```
20 % { println("The possibility you see this message is 20%") }
```

Usando extensiones con lambdas.

Si usted tiene:

```
operator fun <R> String.invoke(block: () -> R) = {
  try { block.invoke() }
  catch (e: AssertException) { System.err.println("$this\n${e.message}") }
}
```

Puede escribir el siguiente código similar a DSL:

```
"it should return 2" {
   parse("1 + 1").buildAST().evaluate() shouldBe 2
}
```

Si se siente confundido con `shouldBe` arriba, vea el ejemplo de `Infix approach to build DSL`.

Lea Edificio DSL en línea: https://riptutorial.com/es/kotlin/topic/10042/edificio-dsl

Capítulo 12: Enumerar

Observaciones

Al igual que en Java, las clases de enumeración en Kotlin tienen métodos sintéticos que permiten enumerar las constantes de enumeración definidas y obtener una constante de enumeración por su nombre. Las firmas de estos métodos son las siguientes (asumiendo que el nombre de la clase enum es `EnumClass`):

```
EnumClass.valueOf(value: String): EnumClass
EnumClass.values(): Array<EnumClass>
```

El método `valueOf()` lanza una `IllegalArgumentException` si el nombre especificado no coincide con ninguna de las constantes de enumeración definidas en la clase.

Cada constante de enumeración tiene propiedades para obtener su nombre y posición en la declaración de clase de enumeración:

```
val name: String
val ordinal: Int
```

Las constantes de enumeración también implementan la interfaz Comparable, siendo el orden natural el orden en el que se definen en la clase enum.

Examples

Inicialización

Enumerar clases como cualquier otra clase puede tener un constructor y ser inicializado

```
enum class Color(val rgb: Int) {
    RED(0xFF0000),
    GREEN(0x00FF00),
    BLUE(0x0000FF)
}
```

Funciones y propiedades en enumeraciones

Las clases de enumeración también pueden declarar miembros (es decir, propiedades y funciones). Se debe colocar un punto y coma (`;`) entre el último objeto de enumeración y la primera declaración del miembro.

Si un miembro es `abstract` , los objetos enumerados deben implementarlo.

```
enum class Color {
    RED {
```

```
        override val rgb: Int = 0xFF0000
    },
    GREEN {
        override val rgb: Int = 0x00FF00
    },
    BLUE {
        override val rgb: Int = 0x0000FF
    }

    ;

    abstract val rgb: Int

    fun colorString() = "#%06X".format(0xFFFFFF and rgb)
}
```

Enumeración simple

```
enum class Color {
  RED, GREEN, BLUE
}
```

Cada enum constante es un objeto. Las constantes enum están separadas por comas.

Mutabilidad

Las enumeraciones pueden ser mutables, esta es otra forma de obtener un comportamiento singleton:

```
enum class Planet(var population: Int = 0) {
    EARTH(7 * 100000000),
    MARS();

    override fun toString() = "$name[population=$population]"
}

 println(Planet.MARS) // MARS[population=0]
 Planet.MARS.population = 3
 println(Planet.MARS) // MARS[population=3]
```

Lea Enumerar en línea: https://riptutorial.com/es/kotlin/topic/2286/enumerar

Capítulo 13: Equivalentes de flujo de Java 8

Introducción

Kotlin proporciona muchos métodos de extensión en colecciones e iterables para aplicar operaciones de estilo funcional. Un tipo de `Sequence` dedicado permite la composición perezosa de varias de estas operaciones.

Observaciones

Sobre la pereza

Si desea procesar de forma perezosa una cadena, puede convertirla a una `Sequence` utilizando `asSequence()` antes de la cadena. Al final de la cadena de funciones, normalmente también terminas con una `Sequence` . Luego puede usar `toList()` , `toSet()` , `toMap()` o alguna otra función para materializar la `Sequence` al final.

```
// switch to and from lazy
val someList = items.asSequence().filter { ... }.take(10).map { ... }.toList()

// switch to lazy, but sorted() brings us out again at the end
val someList = items.asSequence().filter { ... }.take(10).map { ... }.sorted()
```

¿Por qué no hay tipos?

Notará que los ejemplos de Kotlin no especifican los tipos. Esto se debe a que Kotlin tiene inferencia de tipo completo y es completamente seguro en el momento de la compilación. Más que Java porque también tiene tipos anulables y puede ayudar a prevenir el temido NPE. Así que esto en Kotlin:

```
val someList = people.filter { it.age <= 30 }.map { it.name }
```

es lo mismo que:

```
val someList: List<String> = people.filter { it.age <= 30 }.map { it.name }
```

Debido a que Kotlin sabe qué es la `people` , y que `people.age` es `Int` por lo tanto, la expresión de filtro solo permite la comparación con un `Int` , y que `people.name` es una `String` por lo tanto, el paso del `map` produce una `List<String>` (`List` de `String` de solo lectura).

Ahora, si las `people` posiblemente fueran `null` , como en una `List<People>?` entonces:

```
val someList = people?.filter { it.age <= 30 }?.map { it.name }
```

Devuelve una `List<String>?` eso tendría que estar marcado con un valor nulo (*o usar uno de los otros operadores de Kotlin para valores que aceptan valores* nulos *, vea esta forma idiomática de Kotlin para tratar los valores que aceptan valores nulos y también la forma idiomática de manejar una lista vacía o nula en Kotlin*)

Reutilizando corrientes

En Kotlin, depende del tipo de colección si se puede consumir más de una vez. Una `Sequence` genera un nuevo iterador cada vez y, a menos que afirme "usar solo una vez", se puede restablecer al inicio cada vez que se actúe. Por lo tanto, mientras lo siguiente falla en la secuencia de Java 8, pero funciona en Kotlin:

```
// Java:
Stream<String> stream =
Stream.of("d2", "a2", "b1", "b3", "c").filter(s -> s.startsWith("b"));

stream.anyMatch(s -> true);    // ok
stream.noneMatch(s -> true);   // exception
```

```
// Kotlin:
val stream = listOf("d2", "a2", "b1", "b3", "c").asSequence().filter { it.startsWith('b' ) }

stream.forEach(::println) // b1, b2

println("Any B ${stream.any { it.startsWith('b') }}") // Any B true
println("Any C ${stream.any { it.startsWith('c') }}") // Any C false

stream.forEach(::println) // b1, b2
```

Y en Java para obtener el mismo comportamiento:

```
// Java:
Supplier<Stream<String>> streamSupplier =
    () -> Stream.of("d2", "a2", "b1", "b3", "c")
          .filter(s -> s.startsWith("a"));

streamSupplier.get().anyMatch(s -> true);   // ok
streamSupplier.get().noneMatch(s -> true);  // ok
```

Por lo tanto, en Kotlin, el proveedor de los datos decide si se puede restablecer y proporcionar un nuevo iterador o no. Pero si desea restringir intencionalmente una `Sequence` a una iteración de tiempo, puede usar la función `constrainOnce()` para la `Sequence` siguiente manera:

```
val stream = listOf("d2", "a2", "b1", "b3", "c").asSequence().filter { it.startsWith('b' ) }
        .constrainOnce()

stream.forEach(::println) // b1, b2
stream.forEach(::println) // Error:java.lang.IllegalStateException: This sequence can be
consumed only once.
```

Ver también:

- Referencia de API para funciones de extensión para Iterable
- Referencia de API para funciones de extensión para Array
- Referencia de API para funciones de extensión para Lista
- Referencia de API para funciones de extensión a Map

Examples

Acumular nombres en una lista

```
// Java:
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());
```

```
// Kotlin:
val list = people.map { it.name }  // toList() not needed
```

Convertir elementos en cadenas y concatenarlos, separados por comas.

```
// Java:
String joined = things.stream()
                      .map(Object::toString)
                      .collect(Collectors.joining(", "));
```

```
// Kotlin:
val joined = things.joinToString() // ", " is used as separator, by default
```

Calcular la suma de los salarios de los empleados

```
// Java:
int total = employees.stream()
                     .collect(Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val total = employees.sumBy { it.salary }
```

Grupo de empleados por departamento.

```
// Java:
Map<Department, List<Employee>> byDept
     = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment));
```

```
// Kotlin:
val byDept = employees.groupBy { it.department }
```

Calcular la suma de los salarios por departamento

```
// Java:
```

```
Map<Department, Integer> totalByDept
     = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment,
                     Collectors.summingInt(Employee::getSalary)));
```

```
// Kotlin:
val totalByDept = employees.groupBy { it.dept }.mapValues { it.value.sumBy { it.salary }}
```

Partición de los estudiantes en pasar y fallando

```
// Java:
Map<Boolean, List<Student>> passingFailing =
     students.stream()
             .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

```
// Kotlin:
val passingFailing = students.partition { it.grade >= PASS_THRESHOLD }
```

Nombres de miembros masculinos

```
// Java:
List<String> namesOfMaleMembersCollect = roster
    .stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

```
// Kotlin:
val namesOfMaleMembers = roster.filter { it.gender == Person.Sex.MALE }.map { it.name }
```

Grupo de nombres de miembros en la lista por género

```
// Java:
Map<Person.Sex, List<String>> namesByGender =
      roster.stream().collect(
        Collectors.groupingBy(
            Person::getGender,
            Collectors.mapping(
                Person::getName,
                Collectors.toList())));
```

```
// Kotlin:
val namesByGender = roster.groupBy { it.gender }.mapValues { it.value.map { it.name } }
```

Filtrar una lista a otra lista

```
// Java:
List<String> filtered = items.stream()
    .filter( item -> item.startsWith("o") )
    .collect(Collectors.toList());
```

```
// Kotlin:
val filtered = items.filter { item.startsWith('o') }
```

Encontrando la cadena más corta de una lista

```
// Java:
String shortest = items.stream()
    .min(Comparator.comparing(item -> item.length()))
    .get();
```

```
// Kotlin:
val shortest = items.minBy { it.length }
```

Diferentes tipos de transmisiones # 2: usar perezosamente el primer elemento si existe

```
// Java:
Stream.of("a1", "a2", "a3")
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

Diferentes tipos de transmisiones # 3: iterar un rango de enteros

```
// Java:
IntStream.range(1, 4).forEach(System.out::println);
```

```
// Kotlin:  (inclusive range)
(1..3).forEach(::println)
```

Diferentes tipos de transmisiones # 4: iterar una matriz, mapear los valores, calcular el promedio

```
// Java:
Arrays.stream(new int[] {1, 2, 3})
    .map(n -> 2 * n + 1)
    .average()
    .ifPresent(System.out::println); // 5.0
```

```
// Kotlin:
arrayOf(1,2,3).map { 2 * it + 1}.average().apply(::println)
```

Diferentes tipos de flujos n. ° 5: iterar perezosamente una lista de cadenas, mapear los valores, convertir a Int, encontrar máx.

```
// Java:
Stream.of("a1", "a2", "a3")
    .map(s -> s.substring(1))
    .mapToInt(Integer::parseInt)
    .max()
    .ifPresent(System.out::println);  // 3
```

```
// Kotlin:
sequenceOf("a1", "a2", "a3")
    .map { it.substring(1) }
    .map(String::toInt)
    .max().apply(::println)
```

Diferentes tipos de flujos n. ° 6: iteren perezosamente un flujo de Ints, mapee los valores, imprima resultados

```
// Java:
IntStream.range(1, 4)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:  (inclusive range)
(1..3).map { "a$it" }.forEach(::println)
```

Diferentes tipos de transmisiones # 7: iteraciones perezosas dobles, mapa a Int, mapa a Cadena, imprimir cada

```
// Java:
Stream.of(1.0, 2.0, 3.0)
    .mapToInt(Double::intValue)
    .mapToObj(i -> "a" + i)
    .forEach(System.out::println);

// a1
// a2
// a3
```

```
// Kotlin:
sequenceOf(1.0, 2.0, 3.0).map(Double::toInt).map { "a$it" }.forEach(::println)
```

Contando elementos en una lista después de aplicar el filtro

```
// Java:
long count = items.stream().filter( item -> item.startsWith("t")).count();
```

```
// Kotlin:
val count = items.filter { it.startsWith('t') }.size
```

```
// but better to not filter, but count with a predicate
val count = items.count { it.startsWith('t') }
```

Cómo funcionan las secuencias - filtre, mayúsculas, luego ordene una lista

```
// Java:
List<String> myList = Arrays.asList("a1", "a2", "b1", "c2", "c1");

myList.stream()
      .filter(s -> s.startsWith("c"))
      .map(String::toUpperCase)
     .sorted()
     .forEach(System.out::println);

// C1
// C2
```

```
// Kotlin:
val list = listOf("a1", "a2", "b1", "c2", "c1")
list.filter { it.startsWith('c') }.map (String::toUpperCase).sorted()
        .forEach (::println)
```

Diferentes tipos de transmisiones # 1: ansiosos por usar el primer elemento si existe

```
// Java:
Arrays.asList("a1", "a2", "a3")
    .stream()
    .findFirst()
    .ifPresent(System.out::println);
```

```
// Kotlin:
listOf("a1", "a2", "a3").firstOrNull()?.apply(::println)
```

o cree una función de extensión en String llamada ifPresent:

```
// Kotlin:
inline fun String?.ifPresent(thenDo: (String)->Unit) = this?.apply { thenDo(this) }

// now use the new extension function:
listOf("a1", "a2", "a3").firstOrNull().ifPresent(::println)
```

Ver también: función `apply()`

Ver también: Funciones de extensión.

Vea también: `?.` Operador de Safe Call y, en general, nulabilidad: http://stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-with-nullable-values-referencing-o/34498563 # 34498563

Recopile el ejemplo # 5: encuentre personas mayores de edad, una cadena

con formato de salida

```
// Java:
String phrase = persons
        .stream()
        .filter(p -> p.age >= 18)
        .map(p -> p.name)
        .collect(Collectors.joining(" and ", "In Germany ", " are of legal age."));

System.out.println(phrase);
// In Germany Max and Peter and Pamela are of legal age.
```

```
// Kotlin:
val phrase = persons
        .filter { it.age >= 18 }
        .map { it.name }
        .joinToString(" and ", "In Germany ", " are of legal age.")

println(phrase)
// In Germany Max and Peter and Pamela are of legal age.
```

Y como nota al margen, en Kotlin podemos crear clases de datos simples y crear una instancia de los datos de prueba de la siguiente manera:

```
// Kotlin:
// data class has equals, hashcode, toString, and copy methods automagically
data class Person(val name: String, val age: Int)

val persons = listOf(Person("Tod", 5), Person("Max", 33),
                     Person("Frank", 13), Person("Peter", 80),
                     Person("Pamela", 18))
```

Reúna el ejemplo # 6: agrupe a las personas por edad, edad de impresión y nombres juntos

```
// Java:
Map<Integer, String> map = persons
        .stream()
        .collect(Collectors.toMap(
                p -> p.age,
                p -> p.name,
                (name1, name2) -> name1 + ";" + name2));

System.out.println(map);
// {18=Max, 23=Peter;Pamela, 12=David}
```

Ok, un caso más de interés aquí para Kotlin. Primero, las respuestas incorrectas para explorar las variaciones de la creación de un `Map` partir de una colección / secuencia:

```
// Kotlin:
val map1 = persons.map { it.age to it.name }.toMap()
println(map1)
// output: {18=Max, 23=Pamela, 12=David}
// Result: duplicates overridden, no exception similar to Java 8
```

```
val map2 = persons.toMap({ it.age }, { it.name })
println(map2)
// output: {18=Max, 23=Pamela, 12=David}
// Result: same as above, more verbose, duplicates overridden

val map3 = persons.toMapBy { it.age }
println(map3)
// output: {18=Person(name=Max, age=18), 23=Person(name=Pamela, age=23), 12=Person(name=David,
age=12)}
// Result: duplicates overridden again

val map4 = persons.groupBy { it.age }
println(map4)
// output: {18=[Person(name=Max, age=18)], 23=[Person(name=Peter, age=23), Person(name=Pamela,
age=23)], 12=[Person(name=David, age=12)]}
// Result: closer, but now have a Map<Int, List<Person>> instead of Map<Int, String>

val map5 = persons.groupBy { it.age }.mapValues { it.value.map { it.name } }
println(map5)
// output: {18=[Max], 23=[Peter, Pamela], 12=[David]}
// Result: closer, but now have a Map<Int, List<String>> instead of Map<Int, String>
```

Y ahora para la respuesta correcta:

```
// Kotlin:
val map6 = persons.groupBy { it.age }.mapValues { it.value.joinToString(";") { it.name } }

println(map6)
// output: {18=Max, 23=Peter;Pamela, 12=David}
// Result: YAY!!
```

Solo necesitábamos unir los valores coincidentes para colapsar las listas y proporcionar un transformador para `joinToString` para pasar de la instancia de `Person` al `Person.name` .

Recopile el ejemplo # 7a - Asigne nombres, únase junto con delimitador

```
// Java (verbose):
Collector<Person, StringJoiner, String> personNameCollector =
Collector.of(
        () -> new StringJoiner(" | "),          // supplier
        (j, p) -> j.add(p.name.toUpperCase()),  // accumulator
        (j1, j2) -> j1.merge(j2),               // combiner
        StringJoiner::toString);                // finisher

String names = persons
        .stream()
        .collect(personNameCollector);

System.out.println(names);  // MAX | PETER | PAMELA | DAVID

// Java (concise)
String names = persons.stream().map(p -> p.name.toUpperCase()).collect(Collectors.joining(" |
"));
```

```
// Kotlin:
val names = persons.map { it.name.toUpperCase() }.joinToString(" | ")
```

Ejemplo de recopilación # 7b: recopilación con SummarizingInt

```
// Java:
IntSummaryStatistics ageSummary =
    persons.stream()
           .collect(Collectors.summarizingInt(p -> p.age));

System.out.println(ageSummary);
// IntSummaryStatistics{count=4, sum=76, min=12, average=19.000000, max=23}
```

```
// Kotlin:

// something to hold the stats...
data class SummaryStatisticsInt(var count: Int = 0,
                                var sum: Int = 0,
                                var min: Int = Int.MAX_VALUE,
                                var max: Int = Int.MIN_VALUE,
                                var avg: Double = 0.0) {
    fun accumulate(newInt: Int): SummaryStatisticsInt {
        count++
        sum += newInt
        min = min.coerceAtMost(newInt)
        max = max.coerceAtLeast(newInt)
        avg = sum.toDouble() / count
        return this
    }
}

// Now manually doing a fold, since Stream.collect is really just a fold
val stats = persons.fold(SummaryStatisticsInt()) { stats, person ->
stats.accumulate(person.age) }

println(stats)
// output: SummaryStatisticsInt(count=4, sum=76, min=12, max=23, avg=19.0)
```

Pero es mejor crear una función de extensión, 2 en realidad para que coincida con los estilos en Kotlin stdlib:

```
// Kotlin:
inline fun Collection<Int>.summarizingInt(): SummaryStatisticsInt
        = this.fold(SummaryStatisticsInt()) { stats, num -> stats.accumulate(num) }

inline fun <T: Any> Collection<T>.summarizingInt(transform: (T)->Int): SummaryStatisticsInt =
        this.fold(SummaryStatisticsInt()) { stats, item -> stats.accumulate(transform(item)) }
```

Ahora tiene dos formas de usar las nuevas funciones de `summarizingInt` :

```
val stats2 = persons.map { it.age }.summarizingInt()

// or

val stats3 = persons.summarizingInt { it.age }
```

Y todos estos producen los mismos resultados. También podemos crear esta extensión para trabajar en `Sequence` y para tipos primitivos apropiados.

Lea Equivalentes de flujo de Java 8 en línea:
https://riptutorial.com/es/kotlin/topic/707/equivalentes-de-flujo-de-java-8

Capítulo 14: Excepciones

Examples

Cogiendo la excepción con try-catch-finally

La captura de excepciones en Kotlin es muy similar a Java

```
try {
    doSomething()
}
catch(e: MyException) {
    handle(e)
}
finally {
    cleanup()
}
```

También puedes atrapar múltiples excepciones.

```
try {
    doSomething()
}
catch(e: FileSystemException) {
    handle(e)
}
catch(e: NetworkException) {
    handle(e)
}
catch(e: MemoryException) {
    handle(e)
}
finally {
    cleanup()
}
```

`try` también es una expresión y puede devolver valor

```
val s: String? = try { getString() } catch (e: Exception) { null }
```

Kotlin no ha comprobado las excepciones, por lo que no tiene que detectar ninguna excepción.

```
fun fileToString(file: File) : String {
    //readAllBytes throws IOException, but we can omit catching it
    fileContent = Files.readAllBytes(file)
    return String(fileContent)
}
```

Lea Excepciones en línea: https://riptutorial.com/es/kotlin/topic/7246/excepciones

Capítulo 15: Extensiones Kotlin para Android

Introducción

Kotlin tiene una inyección de vista incorporada para Android, que permite omitir el enlace manual o la necesidad de marcos como ButterKnife. Algunas de las ventajas son una mejor sintaxis, una mejor escritura estática y, por lo tanto, son menos propensos a errores.

Examples

Configuración

Comience con un proyecto gradle debidamente configurado .

En su **proyecto, local** (no en el nivel superior), `build.gradle` declaración del complemento de extensiones debajo de su complemento Kotlin, en el nivel de sangría de nivel superior.

```
buildscript {
    ...
}

apply plugin: "com.android.application"
...
apply plugin: "kotlin-android"
apply plugin: "kotlin-android-extensions"
...
```

Usando vistas

Suponiendo que tenemos una actividad con un diseño de ejemplo llamado `activity_main.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/my_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="My button"/>
</LinearLayout>
```

Podemos usar las extensiones de Kotlin para llamar al botón sin ningún enlace adicional como:

```
import kotlinx.android.synthetic.main.activity_main.my_button

class MainActivity: Activity() {
    override fun onCreate(savedInstanceBundle: Bundle?) {
        super.onCreate(savedInstanceBundle)
```

```
        setContentView(R.layout.activity_main)
        // my_button is already casted to a proper type of "Button"
        // instead of being a "View"
        my_button.setText("Kotlin rocks!")
    }
}
```

También puede importar todos los identificadores que aparecen en el diseño con una notación `*`

```
// my_button can be used the same way as before
import kotlinx.android.synthetic.main.activity_main.*
```

Las vistas sintéticas no se pueden usar fuera de Actividades / Fragmentos / Vistas con ese diseño inflado:

```
import kotlinx.android.synthetic.main.activity_main.my_button

class NotAView {
    init {
        // This sample won't compile!
        my_button.setText("Kotlin rocks!")
    }
}
```

Sabores del producto

Las extensiones de Android también funcionan con múltiples sabores de productos de Android. Por ejemplo, si tenemos sabores en `build.gradle` así:

```
android {
    productFlavors {
        paid {
            ...
        }
        free {
            ...
        }
    }
}
```

Y por ejemplo, solo el sabor libre tiene un botón de compra:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/buy_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Buy full version"/>
</LinearLayout>
```

Podemos unir específicamente al sabor:

```
import kotlinx.android.synthetic.free.main_activity.buy_button
```

Un entusiasta oyente por llamar la atención, cuando la vista está completamente dibujada ahora es tan simple y asombroso con la extensión de Kotlin.

```
mView.afterMeasured {
  // inside this block the view is completely drawn
  // you can get view's height/width, it.height / it.width
}
```

Bajo el capó

```
inline fun View.afterMeasured(crossinline f: View.() -> Unit) {
viewTreeObserver.addOnGlobalLayoutListener(object : ViewTreeObserver.OnGlobalLayoutListener {
    override fun onGlobalLayout() {
        if (measuredHeight > 0 && measuredWidth > 0) {
            viewTreeObserver.removeOnGlobalLayoutListener(this)
            f()
        }
    }
})
}
```

Lea Extensiones Kotlin para Android en línea:
https://riptutorial.com/es/kotlin/topic/9474/extensiones-kotlin-para-android

Capítulo 16: Funciones

Sintaxis

- ***Nombre*** divertido (***Params***) = ...
- ***Nombre*** divertido (***Params***) {...}
- ***Nombre*** divertido (***Params***): ***Tipo*** {...}
- fun < ***Type Argument*** > ***Name*** (***Params***): ***Type*** {...}
- ***Nombre de*** diversión en línea (***Parámetros***): ***Escriba*** {...}
- { ***ArgName*** : ***ArgType*** -> ...}
- { ***ArgName*** -> ...}
- { ***ArgNames*** -> ...}
- {(***ArgName*** : ***ArgType***): ***Type*** -> ...}

Parámetros

Parámetro	Detalles
Nombre	Nombre de la función
Parámetros	Valores dados a la función con un nombre y tipo: ***`Name : Type`***
Tipo	Tipo de retorno de la función.
Tipo de argumento	Tipo de parámetro utilizado en la programación genérica (no necesariamente tipo de retorno)
ArgName	Nombre del valor dado a la función.
ArgType	Especificador de tipo para ***ArgName***
ArgNames	Lista de ArgName separados por comas

Examples

Funciones que toman otras funciones

Como se ve en "Funciones Lambda", las funciones pueden tomar otras funciones como un parámetro. El "tipo de función" que deberá declarar funciones que aceptan otras funciones es el siguiente:

```
# Takes no parameters and returns anything
() -> Any?

# Takes a string and an integer and returns ReturnType
```

```
(arg1: String, arg2: Int) -> ReturnType
```

Por ejemplo, podría usar el tipo más vago, `() -> Any?` , para declarar una función que ejecuta una función lambda dos veces:

```
fun twice(x: () -> Any?) {
    x(); x();
}

fun main() {
    twice {
        println("Foo")
    } # => Foo
      # => Foo
}
```

Funciones Lambda

Las funciones Lambda son funciones anónimas que generalmente se crean durante una llamada de función para actuar como un parámetro de función. Se declaran mediante expresiones circundantes con {llaves}: si se necesitan argumentos, estos se colocan antes de una flecha `->` .

```
{ name: String ->
    "Your name is $name" //This is returned
}
```

La última declaración dentro de una función lambda es automáticamente el valor de retorno.

Los tipos son opcionales, si coloca la lambda en un lugar donde el compilador puede inferir los tipos.

Múltiples argumentos:

```
{ argumentOne:String, argumentTwo:String ->
    "$argumentOne - $argumentTwo"
}
```

Si la función lambda sólo necesita un argumento, a continuación, la lista de argumentos puede ser omitido y el argumento solo se refiere al uso `it` en su lugar.

```
{ "Your name is $it" }
```

Si el único argumento de una función es una función lambda, los paréntesis se pueden omitir completamente de la llamada a la función.

```
# These are identical
listOf(1, 2, 3, 4).map { it + 2 }
listOf(1, 2, 3, 4).map({ it + 2 })
```

Referencias de funciones

Podemos hacer referencia a una función sin realmente llamarla prefijando el nombre de la función con `::` . Esto se puede pasar a una función que acepta alguna otra función como parámetro.

```
fun addTwo(x: Int) = x + 2
listOf(1, 2, 3, 4).map(::addTwo) # => [3, 4, 5, 6]
```

Las funciones sin receptor se convertirán a `(ParamTypeA, ParamTypeB, ...) -> ReturnType` donde `ParamTypeA` , `ParamTypeB` ... son el tipo de parámetros de la función y `ReturnType1 es el tipo de valor de retorno de la función.

```
fun foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
println(::foo::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
```

Las funciones con un receptor (ya sea una función de extensión o una función miembro) tienen una sintaxis diferente. Debe agregar el nombre de tipo del receptor antes de los dos puntos dobles:

```
class Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function4<Foo, Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo, Foo0, Foo1, Foo2) -> Bar
// takes 4 parameters, with receiver as first and actual parameters following, in their order

// this function can't be called like an extension function, though
val ref = Foo::foo
Foo().ref(Foo0(), Foo1(), Foo2()) // compile error

class Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

Sin embargo, cuando el receptor de una función es un objeto, el receptor se omite de la lista de parámetros, porque este es y solo es una instancia de ese tipo.

```
object Foo
fun Foo.foo(p0: Foo0, p1: Foo1, p2: Foo2): Bar {
    //...
}
val ref = Foo::foo
println(ref::class.java.genericInterfaces[0])
// kotlin.jvm.functions.Function3<Foo0, Foo1, Foo2, Bar>
// Human readable type: (Foo0, Foo1, Foo2) -> Bar
// takes 3 parameters, receiver not needed
```

```
object Bar {
    fun bar()
}
print(Bar::bar) // works on member functions, too.
```

Desde kotlin 1.1, la referencia de función también puede estar *limitada* a una variable, que luego se denomina *referencia de función limitada* .

1.1.0

```
fun makeList(last: String?): List<String> {
    val list = mutableListOf("a", "b", "c")
    last?.let(list::add)
    return list
}
```

Tenga en cuenta que este ejemplo se proporciona solo para mostrar cómo funciona la referencia de función acotada. Es una mala práctica en todos los demás sentidos.

Sin embargo, hay un caso especial. Una función de extensión declarada como miembro no puede ser referenciada.

```
class Foo
class Bar {
    fun Foo.foo() {}
    val ref = Foo::foo // compile error
}
```

Funciones básicas

Las funciones se declaran utilizando la palabra clave `fun` , seguida de un nombre de función y cualquier parámetro. También puede especificar el tipo de retorno de una función, que por defecto es `Unit` . El cuerpo de la función está encerrado entre llaves `{}` . Si el tipo de devolución es distinto de `Unit` , el cuerpo debe emitir una declaración de devolución para cada rama de terminación dentro del cuerpo.

```
fun sayMyName(name: String): String {
    return "Your name is $name"
}
```

Una versión abreviada de la misma:

```
fun sayMyName(name: String): String = "Your name is $name"
```

Y el tipo se puede omitir ya que se puede inferir:

```
fun sayMyName(name: String) = "Your name is $name"
```

Funciones abreviadas

Si una función contiene solo una expresión, podemos omitir los corchetes y usar un igual en su lugar, como una asignación de variable. El resultado de la expresión se devuelve automáticamente.

```
fun sayMyName(name: String): String = "Your name is $name"
```

Funciones en línea

Las funciones se pueden declarar en línea usando el prefijo en `inline` , y en este caso actúan como macros en C, en lugar de ser llamadas, se reemplazan por el código del cuerpo de la función en el momento de la compilación. Esto puede llevar a beneficios de rendimiento en algunas circunstancias, principalmente cuando las lambdas se utilizan como parámetros de función.

```
inline fun sayMyName(name: String) = "Your name is $name"
```

Una diferencia de las macros de C es que las funciones en línea no pueden acceder al ámbito desde el que se llaman:

```
inline fun sayMyName() = "Your name is $name"

fun main() {
    val name = "Foo"
    sayMyName() # => Unresolved reference: name
}
```

Funciones del operador

Kotlin nos permite proporcionar implementaciones para un conjunto predefinido de operadores con representación simbólica fija (como `+` o `*`) y precedencia fija. Para implementar un operador, proporcionamos una función miembro o una función de extensión con un nombre fijo, para el tipo correspondiente. Las funciones que sobrecargan a los operadores deben estar marcadas con el modificador de `operator` :

```
data class IntListWrapper (val wrapped: List<Int>) {
    operator fun get(position: Int): Int = wrapped[position]
}

val a = IntListWrapper(listOf(1, 2, 3))
a[1] // == 2
```

Más funciones del operador se pueden encontrar aquí

Lea Funciones en línea: https://riptutorial.com/es/kotlin/topic/1280/funciones

Capítulo 17: Fundamentos de Kotlin

Introducción

Este tema cubre los conceptos básicos de Kotlin para principiantes.

Observaciones

1. El archivo Kotlin tiene una extensión .kt.
2. Todas las clases en Kotlin tienen una superclase común Cualquiera, que es un super predeterminado para una clase sin supertipos declarados (similar a Objeto en Java).
3. Las variables se pueden declarar como val (inmutable-asignar una vez) o var (el valor de mutables se puede cambiar)
4. No se necesita punto y coma al final de la declaración.
5. Si una función no devuelve ningún valor útil, su tipo de retorno es Unidad. También es opcional. 6.La igualdad de referencia se verifica mediante la operación ===. a === b se evalúa como verdadero si y solo si a y b apuntan al mismo objeto.

Examples

Ejemplos basicos

1.La declaración de tipo de retorno de la unidad es opcional para las funciones. Los siguientes códigos son equivalentes.

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello ${name}")
}

fun printHello(name: String?) {
    ...
}
```

2. Funciones de expresión simple: cuando una función devuelve una expresión única, se pueden omitir las llaves y el cuerpo se especifica después de = symbol

```
fun double(x: Int): Int = x * 2
```

Declarar explícitamente el tipo de retorno es opcional cuando el compilador puede inferirlo

```
fun double(x: Int) = x * 2
```

3. Interpolación de cadenas: Usar valores de cadena es fácil.

```
In java:
    int num=10
    String s  = "i =" + i;

In Kotlin
    val num = 10
    val s = "i = $num"
```

4. En Kotlin, el sistema de tipos distingue entre las referencias que pueden contener nulas (referencias que admiten nulos) y las que no (referencias no nulas). Por ejemplo, una variable regular de tipo String no puede contener null:

```
var a: String = "abc"
a = null // compilation error
```

Para permitir valores nulos, podemos declarar una variable como cadena anulable, ¿cadena escrita ?:

```
var b: String? = "abc"
b = null // ok
```

5.En Kotlin, == realmente verifica la igualdad de valores. Por convención, una expresión como a == b se traduce a

```
   a?.equals(b) ?: (b === null)
```

Lea Fundamentos de Kotlin en línea: https://riptutorial.com/es/kotlin/topic/10648/fundamentos-de-kotlin

Capítulo 18: Gamas

Introducción

Las expresiones de rango se forman con funciones rangeTo que tienen la forma de operador .. que se complementa con in y! In. El rango se define para cualquier tipo comparable, pero para los tipos primitivos integrales tiene una implementación optimizada

Examples

Tipos de rangos integrales

Los rangos de tipos integrales (IntRange, LongRange, CharRange) tienen una característica adicional: pueden ser iterados. El compilador se encarga de convertir esto de manera análoga al bucle for-indexado de Java, sin sobrecarga adicional

```
for (i in 1..4) print(i) // prints "1234"
for (i in 4..1) print(i) // prints nothing
```

función downTo ()

¿Si quieres iterar sobre números en orden inverso? Es sencillo. Puede usar la función downTo () definida en la biblioteca estándar

```
for (i in 4 downTo 1) print(i) // prints "4321"
```

función de paso

¿Es posible iterar sobre números con un paso arbitrario, no igual a 1? Claro, la función step () te ayudará.

```
for (i in 1..4 step 2) print(i) // prints "13"
for (i in 4 downTo 1 step 2) print(i) // prints "42"
```

hasta la función

Para crear un rango que no incluya su elemento final, puede usar la función until:

```
for (i in 1 until 10) { // i in [1, 10), 10 is excluded
println(i)
}
```

Lea Gamas en línea: https://riptutorial.com/es/kotlin/topic/10121/gamas

Capítulo 19: Genéricos

Introducción

Una lista puede contener números, palabras o realmente cualquier cosa. Por eso llamamos a la lista *genérica* .

Los genéricos se usan básicamente para definir qué tipos puede contener una clase y qué tipo tiene un objeto actualmente.

Sintaxis

- clase *ClassName* < ***TypeName*** >
- clase *ClassName* <*>
- *ClassName* <en ***UpperBound*** >
- *ClassName* <out ***LowerBound*** >
- *Nombre de la* clase < ***TypeName*** : ***UpperBound*** >

Parámetros

Parámetro	Detalles
Escribe un nombre	Tipo Nombre del parámetro genérico
UpperBound	Tipo Covariante
Límite inferior	Tipo Contravariante
Nombre de la clase	Nombre de la clase

Observaciones

El límite superior implícito es anulable

En Kotlin Generics, el límite superior del tipo de parámetro `T` sería `Any?` . Por lo tanto para esta clase:

```
class Consumer<T>
```

El parámetro de tipo `T` es realmente `T: Any?` . Para hacer un límite superior no anulable, explícitamente específico `T: Any` . Por ejemplo:

```
class Consumer<T: Any>
```

Examples

Variación del sitio de la declaración

La variación del sitio de la declaración se puede considerar como una declaración de la variación del sitio de uso de una vez por todas.

```
class Consumer<in T> { fun consume(t: T) { ... } }

fun charSequencesConsumer() : Consumer<CharSequence>() = ...

val stringConsumer : Consumer<String> = charSequenceConsumer() // OK since in-projection
val anyConsumer : Consumer<Any> = charSequenceConsumer() // Error, Any cannot be passed

val outConsumer : Consumer<out CharSequence> = ... // Error, T is `in`-parameter
```

Los ejemplos generalizados de varianza del sitio de declaración son `List<out T>` , que es inmutable, de modo que `T` solo aparece como el tipo de valor de retorno, y `Comparator<in T>` , que solo recibe `T` como argumento.

Varianza del sitio de uso

La varianza del sitio de uso es similar a los comodines de Java:

Out-proyección:

```
val takeList : MutableList<out SomeType> = ... // Java: List<? extends SomeType>

val takenValue : SomeType = takeList[0] // OK, since upper bound is SomeType

takeList.add(takenValue) // Error, lower bound for generic is not specified
```

En proyección:

```
val putList : MutableList<in SomeType> = ... // Java: List<? super SomeType>

val valueToPut : SomeType = ...
putList.add(valueToPut) // OK, since lower bound is SomeType

putList[0] // This expression has type Any, since no upper bound is specified
```

Proyección de estrellas

```
val starList : MutableList<*> = ... // Java: List<?>

starList[0] // This expression has type Any, since no upper bound is specified
starList.add(someValue) // Error, lower bound for generic is not specified
```

Ver también:

- Variante de interoperabilidad de genéricos al llamar a Kotlin desde Java.

Lea Genéricos en línea: https://riptutorial.com/es/kotlin/topic/1147/genericos

Capítulo 20: Herencia de clase

Introducción

Cualquier lenguaje de programación orientado a objetos tiene alguna forma de herencia de clase. Déjame revisar:

Imagina que tienes que programar un montón de frutas: `Apples` , `Oranges` y `Pears` . Todos ellos difieren en tamaño, forma y color, por eso tenemos diferentes clases.

Pero digamos que sus diferencias no importan por un segundo y usted solo quiere una `Fruit` , ¿no importa cuál exactamente? ¿Qué tipo de retorno tendría `getFruit()` ?

La respuesta es clase `Fruit` . ¡Creamos una nueva clase y hacemos que todas las frutas hereden de ella!

Sintaxis

- abrir {clase base}
- class {Clase derivada}: {Clase base} ({Init Arguments})
- anular {definición de función}
- {DC-Object} es {Clase base} == verdadero

Parámetros

Parámetro	Detalles
Clase base	Clase que se hereda de
Clase derivada	Clase que hereda de la clase base.
Argumentos Iniciales	Argumentos pasados al constructor de la clase base
Definición de la función	Función en la clase derivada que tiene un código diferente al mismo en la clase base
Objeto DC	"Objeto de clase derivado" Objeto que tiene el tipo de la clase derivada

Examples

Conceptos básicos: la palabra clave 'abrir'

En Kotlin, las clases son **finales por defecto, lo** que significa que no se pueden heredar de.

Para permitir la herencia en una clase, use la palabra clave `open` .

```
open class Thing {
    // I can now be extended!
}
```

> **Nota: las** clases abstractas, las clases selladas y las interfaces estarán `open` de forma predeterminada.

Heredando campos de una clase

Definiendo la clase base:

```
open class BaseClass {
    val x = 10
}
```

Definiendo la clase derivada:

```
class DerivedClass: BaseClass() {
    fun foo() {
        println("x is equal to " + x)
    }
}
```

Usando la subclase:

```
fun main(args: Array<String>) {
    val derivedClass = DerivedClass()
    derivedClass.foo() // prints: 'x is equal to 10'
}
```

Heredando métodos de una clase.

Definiendo la clase base:

```
open class Person {
    fun jump() {
        println("Jumping...")
    }
}
```

Definiendo la clase derivada:

```
class Ninja: Person() {
    fun sneak() {
        println("Sneaking around...")
    }
}
```

El Ninja tiene acceso a todos los métodos en persona

```
fun main(args: Array<String>) {
    val ninja = Ninja()
    ninja.jump()  // prints: 'Jumping...'
    ninja.sneak() // prints: 'Sneaking around...'
}
```

Anulando propiedades y métodos

Propiedades de reemplazo (tanto de solo lectura como mutables):

```
abstract class Car {
    abstract val name: String;
    open var speed: Int = 0;
}

class BrokenCar(override val name: String) : Car() {
    override var speed: Int
        get() = 0
        set(value) {
            throw UnsupportedOperationException("The car is bloken")
        }
}

fun main(args: Array<String>) {
    val car: Car = BrokenCar("Lada")
    car.speed = 10
}
```

Métodos de anulación:

```
interface Ship {
    fun sail()
    fun sink()
}

object Titanic : Ship {

    var canSail = true

    override fun sail() {
        sink()
    }

    override fun sink() {
        canSail = false
    }
}
```

Lea Herencia de clase en línea: https://riptutorial.com/es/kotlin/topic/5622/herencia-de-clase

Capítulo 21: Instrumentos de cuerda

Examples

Elementos de cuerda

Los elementos de cadena son caracteres a los que se puede acceder mediante la `string[index]` operación de indexación `string[index]` .

```
val str = "Hello, World!"
println(str[1]) // Prints e
```

Los elementos de cadena se pueden iterar con un bucle for.

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

Literales de cuerda

Kotlin tiene dos tipos de literales de cadena:

- Cuerda escapada
- Cuerda cruda

Cadena escapada maneja caracteres especiales al escapar de ellos. El escape se hace con una barra invertida. Se admiten las siguientes secuencias de escape: `\t` , `\b` , `\n` , `\r` , `\'` , `\"` , `\\` y `\$` . Para codificar cualquier otro carácter, use la sintaxis de secuencia de escape de Unicode: `\uFF00` .

```
val s = "Hello, world!\n"
```

La cadena sin formato delimitada por una comilla triple `"""` no contiene escapes y puede contener nuevas líneas y cualquier otro carácter

```
val text = """
    for (c in "foo")
        print(c)
"""
```

Los espacios en blanco iniciales se pueden eliminar con la función trimMargin () .

```
val text = """
    |Tell me and I forget.
    |Teach me and I remember.
    |Involve me and I learn.
    |(Benjamin Franklin)
    """.trimMargin()
```

El prefijo de margen predeterminado es carácter de canalización `|` , esto se puede establecer como un parámetro para trimMargin; por ejemplo, `trimMargin(">")` .

Plantillas de cadena

Tanto las cadenas de escape como las cadenas sin formato pueden contener expresiones de plantilla. La expresión de plantilla es un fragmento de código que se evalúa y su resultado se concatena en una cadena. Comienza con un signo de dólar `$` y consta de un nombre variable:

```
val i = 10
val s = "i = $i" // evaluates to "i = 10"
```

O una expresión arbitraria entre llaves:

```
val s = "abc"
val str = "$s.length is ${s.length}" // evaluates to "abc.length is 3"
```

Para incluir un signo de dólar literal en una cadena, escápelo con una barra invertida:

```
val str = "\$foo" // evaluates to "$foo"
```

La excepción son las cadenas sin formato, que no admiten el escape. En cadenas sin formato, puede utilizar la siguiente sintaxis para representar un signo de dólar.

```
val price = """
${'$'}9.99
"""
```

Igualdad de cuerdas

En Kotlin, las cuerdas se comparan con el operador `==` que las selecciona para su igualdad estructural.

```
val str1 = "Hello, World!"
val str2 = "Hello," + " World!"
println(str1 == str2) // Prints true
```

La igualdad referencial se verifica con el operador `===` .

```
val str1 = """
    |Hello, World!
    """.trimMargin()

val str2 = """
    #Hello, World!
    """.trimMargin("#")

val str3 = str1

println(str1 == str2) // Prints true
```

```
println(str1 === str2) // Prints false
println(str1 === str3) // Prints true
```

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/kotlin/topic/8285/instrumentos-de-cuerda

Capítulo 22: Interfaces

Observaciones

Consulte también: Documentación de referencia de Kotlin para interfaces: Interfaces

Examples

Interfaz básica

Una interfaz de Kotlin contiene declaraciones de métodos abstractos e implementaciones de métodos predeterminados, aunque no pueden almacenar el estado.

```
interface MyInterface {
    fun bar()
}
```

Esta interfaz ahora puede ser implementada por una clase de la siguiente manera:

```
class Child : MyInterface {
   override fun bar() {
       print("bar() was called")
   }
}
```

Interfaz con implementaciones por defecto.

Una interfaz en Kotlin puede tener implementaciones predeterminadas para funciones:

```
interface MyInterface {
    fun withImplementation() {
      print("withImplementation() was called")
    }
}
```

Las clases que implementen dichas interfaces podrán usar esas funciones sin volver a implementarlas.

```
class MyClass: MyInterface {
    // No need to reimplement here
}
val instance = MyClass()
instance.withImplementation()
```

Propiedades

Las implementaciones predeterminadas también funcionan para los que obtienen y establecen

propiedades:

```
interface MyInterface2 {
    val helloWorld
        get() = "Hello World!"
}
```

Las implementaciones de accesores de interfaz no pueden usar campos de respaldo

```
interface MyInterface3 {
    // this property won't compile!
    var helloWorld: Int
        get() = field
        set(value) { field = value }
}
```

Implementaciones multiples

Cuando varias interfaces implementan la misma función, o todas ellas se definen con una o más implementaciones, la clase derivada debe resolver manualmente la llamada apropiada

```
interface A {
    fun notImplemented()
    fun implementedOnlyInA() { print("only A") }
    fun implementedInBoth() { print("both, A") }
    fun implementedInOne() { print("implemented in A") }
}

interface B {
    fun implementedInBoth() { print("both, B") }
    fun implementedInOne() // only defined
}

class MyClass: A, B {
    override fun notImplemented() { print("Normal implementation") }

    // implementedOnlyInA() can by normally used in instances

    // class needs to define how to use interface functions
    override fun implementedInBoth() {
        super<B>.implementedInBoth()
        super<A>.implementedInBoth()
    }

    // even if there's only one implementation, there multiple definitions
    override fun implementedInOne() {
        super<A>.implementedInOne()
        print("implementedInOne class implementation")
    }
}
```

Propiedades en interfaces

Puedes declarar propiedades en interfaces. Dado que una interfaz no puede tener un estado, solo

puede declarar una propiedad como abstracta o proporcionando una implementación predeterminada para los usuarios.

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

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

class Child : MyInterface {
    override val property: Int = 29
}
```

Conflictos al implementar múltiples interfaces con implementaciones predeterminadas

Cuando se implementa más de una interfaz que tiene métodos del mismo nombre que incluyen implementaciones predeterminadas, es ambiguo para el compilador qué implementación se debe usar. En el caso de un conflicto, el desarrollador debe anular el método conflictivo y proporcionar una implementación personalizada. Esa implementación puede optar por delegar a las implementaciones por defecto o no.

```
interface FirstTrait {
    fun foo() { print("first") }
    fun bar()
}

interface SecondTrait {
    fun foo() { print("second") }
    fun bar() { print("bar") }
}

class ClassWithConflict : FirstTrait, SecondTrait {
    override fun foo() {
        super<FirstTrait>.foo()  // delegate to the default implementation of FirstTrait
        super<SecondTrait>.foo() // delegate to the default implementation of SecondTrait
    }

    // function bar() only has a default implementation in one interface and therefore is ok.
}
```

súper palabra clave

```
interface MyInterface {
    fun funcOne() {
        //optional body
        print("Function with default implementation")
    }
}
```

Si el método en la interfaz tiene su propia implementación predeterminada, podemos usar la palabra clave super para acceder a él.

```
super.funcOne()
```

Lea Interfaces en línea: https://riptutorial.com/es/kotlin/topic/900/interfaces

Capítulo 23: JUIT

Examples

Reglas

Para agregar una regla JUnit a un dispositivo de prueba:

```
@Rule @JvmField val myRule = TemporaryFolder()
```

La anotación `@JvmField` es necesaria para exponer el campo de respaldo con la misma visibilidad (pública) que la propiedad `myRule` (ver respuesta). Las reglas de JUnit requieren que el campo de la regla anotada sea público.

Lea JUIT en línea: https://riptutorial.com/es/kotlin/topic/6973/juit

Capítulo 24: Kotlin para desarrolladores de Java

Introducción

La mayoría de las personas que vienen a Kotlin tienen un fondo de programación en Java.

Este tema recopila ejemplos que comparan Java con Kotlin, resaltando las diferencias más importantes y las gemas que Kotlin ofrece sobre Java.

Examples

Declarando variables

En Kotlin, las declaraciones de variables son un poco diferentes a las de Java:

```
val i : Int = 42
```

- Comienzan con cualquiera `val` o `var` , hacer la declaración `final` ("ue **val")** o iable **var.**
- El tipo se indica después del nombre, separado por `:`
- Gracias a la *inferencia de tipos* de Kotlin, la declaración explícita de tipos puede obtenerse si hay una asignación con un tipo que el compilador puede detectar inequívocamente

Java	Kotlin
`int i = 42;`	`var i = 42` (*o* `var i : Int = 42`)
`final int i = 42;`	`val i = 42`

Hechos rápidos

- Kotlin no necesita `;` para terminar las declaraciones
- Kotlin es **nulo seguro**
- Kotlin es **100% Java interoperable.**
- Kotlin **no** tiene **primitivas** (pero optimiza sus contrapartes de objetos para la JVM, si es posible)
- Las clases de Kotlin tienen **propiedades, no campos**
- Kotlin ofrece **clases de datos** con métodos `equals` / `hashCode` generados automáticamente y `hashCode` campo
- Kotlin solo tiene excepciones en tiempo de ejecución, **no** tiene excepciones **verificadas**
- Kotlin **no** tiene **una `new` palabra clave** . La creación de objetos se realiza simplemente llamando al constructor como cualquier otro método.

- Kotlin soporta **sobrecargas de operadores** (limitadas). Por ejemplo, el acceso a un valor de un mapa puede escribirse como: `val a = someMap["key"]`
- Kotlin no solo se puede compilar en un código de bytes para la JVM, sino también en **Java Script** , lo que le permite escribir tanto el código de backend como el de frontend en Kotlin
- Kotlin es **totalmente compatible con Java 6** , lo cual es especialmente interesante en lo que respecta al soporte de dispositivos Android (no tan antiguos)
- Kotlin es un lenguaje **soportado oficialmente para el desarrollo de Android**
- Las colecciones de Kotlin tienen una distinción integrada entre las colecciones **mutables e inmutables** .
- Kotlin apoya a los **coroutines** (experimental)

Igualdad e identidad

Kotlin usa `==` para la igualdad (es decir, las llamadas son `equals` internamente) y `===` para la identidad referencial.

Java	Kotlin
`a.equals(b);`	`a == b`
`a == b;`	`a === b`
`a != b;`	`a !== b`

Consulte: https://kotlinlang.org/docs/reference/equality.html

SI, TRY y otros son expresiones, no declaraciones

En Kotlin, `if` , `try` y otros son expresiones (por lo que devuelven un valor) en lugar de declaraciones (nulas).

Entonces, por ejemplo, Kotlin no tiene el *operador Elvis* ternario de Java, pero puede escribir algo como esto:

```
val i = if (someBoolean) 33 else 42
```

Aún más desconocida, pero igualmente expresiva, es la *expresión de* `try` :

```
val i = try {
    Integer.parseInt(someString)
}
catch (ex : Exception)
{
    42
}
```

Lea Kotlin para desarrolladores de Java en línea:
https://riptutorial.com/es/kotlin/topic/10099/kotlin-para-desarrolladores-de-java

Capítulo 25: Lambdas basicas

Sintaxis

- Parámetros explícitos:
- {parametersName: ParameterType, otherParameterName: OtherParameterType -> anExpression ()}
- Parámetros inferidos:
- adición val: (Int, Int) -> Int = {a, b -> a + b}
- Un solo parámetro `it` taquigrafía
- cuadrado de val: (Int) -> Int = {it * it}
- Firma:
- () -> ResultType
- (InputType) -> ResultType
- (InputType1, InputType2) -> ResultType

Observaciones

Los parámetros de tipo de entrada se pueden omitir cuando se pueden omitir cuando se pueden inferir del contexto. Por ejemplo, digamos que tiene una función en una clase que toma una función:

```
data class User(val fistName: String, val lastName: String) {
    fun username(userNameGenerator: (String, String) -> String) =
        userNameGenerator(firstName, secondName)
}
```

Puede utilizar esta función pasando un lambda, y como los parámetros ya están especificados en la firma de la función, no es necesario volver a declararlos en la expresión lambda:

```
val user = User("foo", "bar")
println(user.userName { firstName, secondName ->
     "${firstName.toUppercase}"_"${secondName.toUppercase}"
 }) // prints FOO_BAR
```

Esto también se aplica cuando está asignando un lambda a una variable:

```
//valid:
val addition: (Int, Int) = { a, b -> a + b }
```

```
//valid:
val addition = { a: Int, b: Int -> a + b }
//error (type inference failure):
val addition = { a, b -> a + b }
```

Cuando la lambda toma un parámetro, y el tipo se puede inferir del contexto, puede referirse al parámetro por `it` .

```
listOf(1,2,3).map { it * 2 } // [2,4,6]
```

Examples

Lambda como parámetro para filtrar la función.

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }
```

Lambda pasó como una variable

```
val isOfAllowedAge = { user: User -> user.age > MINIMUM_AGE }
val allowedUsers = users.filter(isOfAllowedAge)
```

Lambda para benchmarking una función llamada

Cronómetro de propósito general para medir el tiempo que tarda una función en ejecutarse:

```
object Benchmark {
    fun realtime(body: () -> Unit): Duration {
        val start = Instant.now()
        try {
            body()
        } finally {
            val end = Instant.now()
            return Duration.between(start, end)
        }
    }
}
```

Uso:

```
val time = Benchmark.realtime({
    // some long-running code goes here ...
})
println("Executed the code in $time")
```

Lea Lambdas basicas en línea: https://riptutorial.com/es/kotlin/topic/5878/lambdas-basicas

Capítulo 26: loguearse en kotlin

Observaciones

Pregunta relacionada: Forma idiomática de registro en Kotlin

Examples

kotlin.logging

```
class FooWithLogging {
  companion object: KLogging()

  fun bar() {
    logger.info { "hello $name" }
  }

  fun logException(e: Exception) {
    logger.error(e) { "Error occured" }
  }
}
```

Usando el framework kotlin.logging

Lea loguearse en kotlin en línea: https://riptutorial.com/es/kotlin/topic/3258/loguearse-en-kotlin

Capítulo 27: Métodos de extensión

Sintaxis

- fun TypeName.extensionName (params, ...) {/ * body * /} // Declaración
- fun <T: Any> TypeNameWithGenerics <T> .extensionName (params, ...) {/ * body * /} // Declaración con genéricos
- myObj.extensionName (args, ...) // invocación

Observaciones

Las extensiones se resuelven **estáticamente** . Esto significa que el método de extensión que se utilizará está determinado por el tipo de referencia de la variable a la que está accediendo; No importa cuál sea el tipo de la variable en el tiempo de ejecución, siempre se llamará al mismo método de extensión. Esto se debe a que **declarar un método de extensión en realidad no agrega un miembro al tipo de receptor** .

Examples

Extensiones de nivel superior

Los métodos de extensión de nivel superior no están contenidos dentro de una clase.

```
fun IntArray.addTo(dest: IntArray) {
    for (i in 0 .. size - 1) {
        dest[i] += this[i]
    }
}
```

Encima se define un método de extensión para el tipo `IntArray` . Tenga en cuenta que se accede al objeto para el que se define el método de extensión (llamado **receptor**) con la palabra clave `this` .

Esta extensión se puede llamar así:

```
val myArray = intArrayOf(1, 2, 3)
intArrayOf(4, 5, 6).addTo(myArray)
```

Posible trampa: las extensiones se resuelven de forma estática

El método de extensión a llamar se determina en tiempo de compilación en función del tipo de referencia de la variable a la que se accede. No importa cuál sea el tipo de la variable en el tiempo de ejecución, siempre se llamará al mismo método de extensión.

```
open class Super
```

```
class Sub : Super()

fun Super.myExtension() = "Defined for Super"

fun Sub.myExtension() = "Defined for Sub"

fun callMyExtension(myVar: Super) {
    println(myVar.myExtension())
}

callMyExtension(Sub())
```

El ejemplo anterior imprimirá `"Defined for Super"` , porque el tipo declarado de la variable `myVar` es `Super` .

Muestra que se extiende por mucho tiempo para representar una cadena humana legible

Dado cualquier valor de tipo `Int` o `Long` para representar una cadena legible por humanos:

```
fun Long.humanReadable(): String {
    if (this <= 0) return "0"
    val units = arrayOf("B", "KB", "MB", "GB", "TB", "EB")
    val digitGroups = (Math.log10(this.toDouble())/Math.log10(1024.0)).toInt();
    return DecimalFormat("#,##0.#").format(this/Math.pow(1024.0, digitGroups.toDouble())) + "
" + units[digitGroups];
}

fun Int.humanReadable(): String {
    return this.toLong().humanReadable()
}
```

Luego se usa fácilmente como:

```
println(1999549L.humanReadable())
println(someInt.humanReadable())
```

Ejemplo de extensión de Java 7+ clase de ruta

Un caso de uso común para los métodos de extensión es mejorar una API existente. Aquí hay ejemplos de `notExists` agregar `exist` , `notExists` y `deleteRecursively` a la clase Java 7+ `Path` :

```
fun Path.exists(): Boolean = Files.exists(this)
fun Path.notExists(): Boolean = !this.exists()
fun Path.deleteRecursively(): Boolean = this.toFile().deleteRecursively()
```

Que ahora se puede invocar en este ejemplo:

```
val dir = Paths.get(dirName)
if (dir.exists()) dir.deleteRecursively()
```

Usando funciones de extensión para mejorar la legibilidad

En Kotlin puedes escribir código como:

```
val x: Path = Paths.get("dirName").apply {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not
exist")
}
```

Pero el uso de `apply` no es tan claro en cuanto a su intención. A veces es más claro crear una función de extensión similar para, en efecto, cambiar el nombre de la acción y hacerla más evidente. No se debe permitir que esto se salga de control, pero para acciones muy comunes como la verificación:

```
infix inline fun <T> T.verifiedBy(verifyWith: (T) -> Unit): T {
    verifyWith(this)
    return this
}

infix inline fun <T: Any> T.verifiedWith(verifyWith: T.() -> Unit): T {
    this.verifyWith()
    return this
}
```

Ahora puedes escribir el código como:

```
val x: Path = Paths.get("dirName") verifiedWith {
    if (Files.notExists(this)) throw IllegalStateException("The important file does not
exist")
}
```

Que ahora la gente sepa qué esperar dentro del parámetro lambda.

Tenga en cuenta que el parámetro de tipo `T` para `verifiedBy` es el mismo que `T: Any?` lo que significa que incluso los tipos anulables podrán usar esa versión de la extensión. Aunque `verifiedWith` requiere no anulable.

Ejemplo de extensión de clases temporales de Java 8 para representar una cadena con formato ISO

Con esta declaración:

```
fun Temporal.toIsoString(): String = DateTimeFormatter.ISO_INSTANT.format(this)
```

Ahora puedes simplemente:

```
val dateAsString = someInstant.toIsoString()
```

Funciones de extensión a objetos complementarios (apariencia de funciones

estáticas)

Si desea extender una clase como-si es una función estática, por ejemplo, para la clase `Something` agregue la función de aspecto estático desde la `fromString` , esto solo puede funcionar si la clase tiene un objeto complementario y la función de extensión se ha declarado en el objeto complementario. :

```
class Something {
    companion object {}
}

class SomethingElse {
}

fun Something.Companion.fromString(s: String): Something = ...

fun SomethingElse.fromString(s: String): SomethingElse = ...

fun main(args: Array<String>) {
    Something.fromString("") //valid as extension function declared upon the
                             //companion object

    SomethingElse().fromString("") //valid, function invoked on instance not
                                   //statically

    SomethingElse.fromString("") //invalid
}
```

Solución perezosa de la propiedad de la extensión

Supongamos que desea crear una propiedad de extensión que sea costosa de computar. Así que le gustaría almacenar en caché el cálculo, utilizando el delegado propiedad perezosa y se refieren a instancia actual (`this`), pero no puede hacerlo, como se explica en el Kotlin emite KT-9686 y KT-13053 . Sin embargo, hay una solución oficial proporcionada aquí .

En el ejemplo, la propiedad de extensión es `color` . Utiliza un `colorCache` explícito que puede usarse con `this` ya que no es necesario `lazy` :

```
class KColor(val value: Int)

private val colorCache = mutableMapOf<KColor, Color>()

val KColor.color: Color
    get() = colorCache.getOrPut(this) { Color(value, true) }
```

Extensiones para una referencia más fácil Vista desde el código

Puede usar extensiones para la vista de referencia, no más repeticiones después de crear las vistas.

La idea original es de la biblioteca de Anko

Extensiones

```
inline fun <reified T : View> View.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Activity.find(id: Int): T = findViewById(id) as T
inline fun <reified T : View> Fragment.find(id: Int): T = view?.findViewById(id) as T
inline fun <reified T : View> RecyclerView.ViewHolder.find(id: Int): T =
itemView?.findViewById(id) as T

inline fun <reified T : View> View.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Activity.findOptional(id: Int): T? = findViewById(id) as? T
inline fun <reified T : View> Fragment.findOptional(id: Int): T? = view?.findViewById(id) as?
T
inline fun <reified T : View> RecyclerView.ViewHolder.findOptional(id: Int): T? =
itemView?.findViewById(id) as? T
```

Uso

```
val yourButton by lazy { find<Button>(R.id.yourButtonId) }
val yourText by lazy { find<TextView>(R.id.yourTextId) }
val yourEdittextOptional by lazy { findOptional<EditText>(R.id.yourOptionEdittextId) }
```

Lea Métodos de extensión en línea: https://riptutorial.com/es/kotlin/topic/613/metodos-de-extension

Capítulo 28: Modificadores de visibilidad

Introducción

En Kotlin, hay 4 tipos de modificadores de visibilidad disponibles.

Público: Se puede acceder desde cualquier lugar.

Privado: Solo se puede acceder desde el código del módulo.

Protegido: solo se puede acceder a él desde la clase que lo define y desde cualquier clase derivada.

Interno: solo se puede acceder desde el alcance de la clase que lo define.

Sintaxis

- `<visibility modifier> val/var <variable name> = <value>`

Examples

Ejemplo de código

Public: `public val name = "Avijit"`

Privado: `private val name = "Avijit"`

Protegido: `protected val name = "Avijit"`

Interno: `internal val name = "Avijit"`

Lea Modificadores de visibilidad en línea:
https://riptutorial.com/es/kotlin/topic/10019/modificadores-de-visibilidad

Capítulo 29: Modismos

Examples

Creación de DTO (POJOs / POCOs)

Las clases de datos en kotlin son clases creadas para hacer nada más que mantener datos. Tales clases están marcadas como `data` :

```
data class User(var firstname: String, var lastname: String, var age: Int)
```

El código anterior crea una clase de `User` con lo siguiente generado automáticamente:

- Getters y Setters para todas las propiedades (getters solo para `val` s)
- `equals()`
- `hashcode()`
- `toString()`
- `copy()`
- `componentN()` (donde `N` es la propiedad correspondiente en orden de declaración)

Al igual que con una función, los valores predeterminados también se pueden especificar:

```
data class User(var firstname: String = "Joe", var lastname: String = "Bloggs", var age: Int =
20)
```

Más detalles se pueden encontrar aquí Clases de datos .

Filtrando una lista

```
val list = listOf(1,2,3,4,5,6)

//filter out even numbers

val even = list.filter { it % 2 == 0 }

println(even) //returns [2,4]
```

Delegado a una clase sin aportarlo en el constructor público.

Suponga que desea delegar en una clase pero no desea proporcionar la clase delegada a en el parámetro constructor. En su lugar, desea construirlo de forma privada, haciendo que el constructor que llama no lo sepa. Al principio, esto puede parecer imposible porque la delegación de clase permite delegar solo a los parámetros del constructor. Sin embargo, hay una manera de hacerlo, como se indica en esta respuesta :

```
class MyTable private constructor(table: Table<Int, Int, Int>) : Table<Int, Int, Int> by table
{
```

```
    constructor() : this(TreeBasedTable.create()) // or a different type of table if desired
}
```

Con esto, puedes llamar al constructor de `MyTable` así: `MyTable()` . La `Table<Int, Int, Int>` en la que se crearán los delegados de `MyTable` forma privada. El que llama al constructor no sabe nada al respecto.

Este ejemplo se basa en esta pregunta SO .

Serializable y serialVersionUid en Kotlin

Para crear el `serialVersionUID` para una clase en Kotlin, tiene algunas opciones, todas relacionadas con la adición de un miembro al objeto complementario de la clase.

El bytecode más conciso proviene de una `private const val` que se convertirá en una variable estática privada en la clase contenedora, en este caso `MySpecialCase` :

```
class MySpecialCase : Serializable {
    companion object {
        private const val serialVersionUID: Long = 123
    }
}
```

También puede usar estos formularios, **cada uno con el efecto secundario de tener métodos de obtención / establecimiento** que no son necesarios para la serialización ...

```
class MySpecialCase : Serializable {
    companion object {
        private val serialVersionUID: Long = 123
    }
}
```

Esto crea el campo estático, pero también crea un getter y `getSerialVersionUID` en el objeto complementario que no es necesario.

```
class MySpecialCase : Serializable {
    companion object {
        @JvmStatic private val serialVersionUID: Long = 123
    }
}
```

Esto crea el campo estático, pero también crea un getter estático y `getSerialVersionUID` en la clase que contiene `MySpecialCase` que no es necesario.

Pero todo funciona como un método para agregar el `serialVersionUID` a una clase `Serializable` .

Métodos fluidos en Kotlin

Los métodos fluidos en Kotlin pueden ser los mismos que en Java:

```
fun doSomething() {
   someOtherAction()
   return this
}
```

Pero también puede hacer que sean más funcionales creando una función de extensión como:

```
fun <T: Any> T.fluently(func: ()->Unit): T {
    func()
    return this
}
```

Lo que entonces permite funciones más obviamente obvias:

```
fun doSomething() {
   return fluently { someOtherAction() }
}
```

Utilice let o también para simplificar el trabajo con objetos anulables.

`let` Kotlin cree un enlace local desde el objeto al que fue llamado. Ejemplo:

```
val str = "foo"
str.let {
    println(it) // it
}
```

Esto imprimirá `"foo"` y devolverá la `Unit` .

La diferencia entre `let` y `also` es que puede devolver cualquier valor de un bloque `let` . `also` en la otra parte siempre reutrn la `Unit` .

Ahora, ¿por qué esto es útil, preguntas? Porque si llama a un método que puede devolver `null` y desea ejecutar algún código solo cuando el valor de retorno no es `null` , puede usar `let` o `also` esto:

```
val str: String? = someFun()
str?.let {
    println(it)
}
```

Este fragmento de código solo ejecutará el bloque `let` cuando `str` no sea `null` . Tenga en cuenta el operador de seguridad `null` (`?`).

Utilice aplicar para inicializar objetos o para lograr el encadenamiento de métodos.

La documentación de `apply` dice lo siguiente:

> llama al bloque de funciones especificado con `this` valor como su receptor y devuelve

`this` valor.

Mientras que el kdoc no es tan útil `apply` es de hecho una función útil. En términos sencillos, `apply` establece un alcance en el que `this` está vinculado al objeto al que llamó `apply` . Esto le permite ahorrar algo de código cuando necesita llamar a múltiples métodos en un objeto que luego devolverá. Ejemplo:

```
File(dir).apply { mkdirs() }
```

Esto es lo mismo que escribir esto:

```
fun makeDir(String path): File {
    val result = new File(path)
    result.mkdirs()
    return result
}
```

Lea Modismos en línea: https://riptutorial.com/es/kotlin/topic/2273/modismos

Capítulo 30: Objetos singleton

Introducción

Un *objeto* es un tipo especial de clase, que se puede declarar utilizando la palabra clave `object` . Los objetos son similares a Singletons (un patrón de diseño) en Java. También funciona como la parte estática de java. Los principiantes que cambian de java a kotlin pueden usar ampliamente esta función, en lugar de estática o singletons.

Examples

Utilizar como repalcement de métodos estáticos / campos de java

```
object CommonUtils {

    var anyname: String ="Hello"

    fun dispMsg(message: String) {
        println(message)
    }
}
```

Desde cualquier otra clase, simplemente invoca la variable y las funciones de esta manera:

```
CommonUtils.anyname
CommonUtils.dispMsg("like static call")
```

Utilizar como un singleton

Los objetos de Kotlin son en realidad solo singletons. Su principal ventaja es que no tiene que usar `SomeSingleton.INSTANCE` para obtener la instancia del singleton.

En java tu singleton se ve así:

```
public enum SharedRegistry {
    INSTANCE;
    public void register(String key, Object thing) {}
}

public static void main(String[] args) {
    SharedRegistry.INSTANCE.register("a", "apple");
    SharedRegistry.INSTANCE.register("b", "boy");
    SharedRegistry.INSTANCE.register("c", "cat");
    SharedRegistry.INSTANCE.register("d", "dog");
}
```

En kotlin, el código equivalente es

```
object SharedRegistry {
    fun register(key: String, thing: Object) {}
}

fun main(Array<String> args) {
    SharedRegistry.register("a", "apple")
    SharedRegistry.register("b", "boy")
    SharedRegistry.register("c", "cat")
    SharedRegistry.register("d", "dog")
}
```

Obviamente es menos verboso de usar.

Lea Objetos singleton en línea: https://riptutorial.com/es/kotlin/topic/10152/objetos-singleton

Capítulo 31: Parámetros Vararg en Funciones

Sintaxis

- **Palabra clave Vararg** : `vararg` se utiliza en una declaración de método para indicar que se aceptará un número variable de parámetros.
- **Operador de propagación** : un asterisco (`*`) antes de una matriz que se utiliza en llamadas a funciones para "desplegar" los contenidos en parámetros individuales.

Examples

Conceptos básicos: Uso de la palabra clave vararg

Defina la función utilizando la palabra clave `vararg` .

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

Ahora puede pasar tantos parámetros (del tipo correcto) a la función como desee.

```
printNumbers(0, 1)                // Prints "0" "1"
printNumbers(10, 20, 30, 500)     // Prints "10" "20" "30" "500"
```

> **Notas:** Los parámetros Vararg *deben* ser el último parámetro en la lista de parámetros.

Operador de propagación: pasar matrices a funciones vararg

Las matrices se pueden pasar a funciones vararg utilizando el **Operador de propagación** , `*` .

Suponiendo que exista la siguiente función ...

```
fun printNumbers(vararg numbers: Int) {
    for (number in numbers) {
        println(number)
    }
}
```

Puedes **pasar una matriz** a la función así ...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(*numbers)

// This is the same as passing in (1, 2, 3)
```

El operador de propagación también se puede utilizar **en medio** de los parámetros ...

```
val numbers = intArrayOf(1, 2, 3)
printNumbers(10, 20, *numbers, 30, 40)

// This is the same as passing in (10, 20, 1, 2, 3, 30, 40)
```

Lea Parámetros Vararg en Funciones en línea:
https://riptutorial.com/es/kotlin/topic/5835/parametros-vararg-en-funciones

Capítulo 32: Propiedades delegadas

Introducción

Kotlin puede delegar la implementación de una propiedad a un objeto controlador. Se incluyen algunos manejadores estándar, como la inicialización perezosa o propiedades observables. También se pueden crear controladores personalizados.

Examples

Inicialización perezosa

```
val foo : Int by lazy { 1 + 1 }
println(foo)
```

El ejemplo imprime `2` .

Propiedades observables

```
var foo : Int by Delegates.observable("1") { property, oldValue, newValue ->
    println("${property.name} was changed from $oldValue to $newValue")
}
foo = 2
```

El ejemplo imprime `foo was changed from 1 to 2`

Propiedades respaldadas por el mapa

```
val map = mapOf("foo" to 1)
val foo : String by map
println(foo)
```

El ejemplo imprime `1`

Delegación personalizada

```
class MyDelegate {
    operator fun getValue(owner: Any?, property: KProperty<*>): String {
        return "Delegated value"
    }
}

val foo : String by MyDelegate()
println(foo)
```

El ejemplo imprime `Delegated value`

Delegado Se puede usar como una capa para reducir la placa de caldera

Considere el sistema de tipo nulo de Kotlin y `WeakReference<T>` .

Entonces, digamos que tenemos que guardar algún tipo de referencia y queríamos evitar las fugas de memoria, aquí es donde entra en `WeakReference` .

Tomemos por ejemplo esto:

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: WeakReference<MyMemoryExpensiveClass>? = null

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let {
                it.get()?.let(block)
            }
        }
    }

    init {
        reference = WeakReference(this)
    }
}
```

Ahora esto es solo con una WeakReference. Para reducir esta repetición, podemos usar un delegado de propiedades personalizado para ayudarnos así:

```
class WeakReferenceDelegate<T>(initialValue: T? = null) : ReadWriteProperty<Any, T?> {
    var reference = WeakReference(initialValue)
        private set

    override fun getValue(thisRef: Any, property: KProperty<*>): T? = reference.get()

    override fun setValue(thisRef: Any, property: KProperty<*>, value: T?) {
        reference = WeakReference(value)
    }
}
```

¡Así que ahora podemos usar variables que están envueltas con `WeakReference` al igual que las variables normales que `WeakReference` !

```
class MyMemoryExpensiveClass {
    companion object {
        var reference: MyMemoryExpensiveClass? by
WeakReferenceDelegate<MyMemoryExpensiveClass>()

        fun doWithReference(block: (MyMemoryExpensiveClass) -> Unit) {
            reference?.let(block)
        }
    }

    init {
        reference = this
    }
```

```
}
```

Lea Propiedades delegadas en línea: https://riptutorial.com/es/kotlin/topic/10571/propiedades-delegadas

Capítulo 33: RecyclerView en Kotlin

Introducción

Solo quiero compartir mi pequeño conocimiento y código de RecyclerView utilizando Kotlin.

Examples

Clase principal y adaptador

Supongo que tiene conocimiento de la sintaxis de **Kotlin** y de cómo usarla, simplemente agregue **RecyclerView** en el archivo **activity_main.xml** y establezca con la clase de adaptador.

```
class MainActivity : AppCompatActivity(){

        lateinit var mRecyclerView : RecyclerView
        val mAdapter : RecyclerAdapter = RecyclerAdapter()

        override fun onCreate(savedInstanceState: Bundle?) {
            super.onCreate(savedInstanceState)
            setContentView(R.layout.activity_main)
            val toolbar = findViewById(R.id.toolbar) as Toolbar
            setSupportActionBar(toolbar)

            mRecyclerView = findViewById(R.id.recycler_view) as RecyclerView

            mRecyclerView.setHasFixedSize(true)
            mRecyclerView.layoutManager = LinearLayoutManager(this)
            mAdapter.RecyclerAdapter(getList(), this)
            mRecyclerView.adapter = mAdapter
        }

        private fun getList(): ArrayList<String> {
            var list : ArrayList<String> = ArrayList()
            for  (i in 1..10) { // equivalent of 1 <= i && i <= 10
                println(i)
                list.add("$i")
            }
            return list
        }
    }
```

este es su clase de **adaptador de** vista de reciclador y cree el archivo **main_item.xml** que desee

```
class RecyclerAdapter : RecyclerView.Adapter<RecyclerAdapter.ViewHolder>() {

    var mItems: ArrayList<String>  = ArrayList()
    lateinit var mClick : OnClick

    fun RecyclerAdapter(item : ArrayList<String>, mClick : OnClick){
        this.mItems = item
        this.mClick = mClick;
    }
```

```
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        val item = mItems[position]
        holder.bind(item, mClick, position)
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val layoutInflater = LayoutInflater.from(parent.context)
        return ViewHolder(layoutInflater.inflate(R.layout.main_item, parent, false))
    }

    override fun getItemCount(): Int {
        return mItems.size
    }

    class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
        val card = view.findViewById(R.id.card) as TextView
        fun bind(str: String, mClick: OnClick, position: Int){
            card.text = str
            card.setOnClickListener { view ->
                mClick.onClickListner(position)
            }
        }
    }
}
```

Lea RecyclerView en Kotlin en línea: https://riptutorial.com/es/kotlin/topic/10143/recyclerview-en-kotlin

Capítulo 34: Reflexión

Introducción

Reflexión es la capacidad de un lenguaje para inspeccionar el código en tiempo de ejecución en lugar de compilarlo.

Observaciones

La reflexión es un mecanismo para realizar introspecciones de construcciones de lenguaje (clases y funciones) en el tiempo de ejecución.

Cuando se dirige a la plataforma JVM, las características de reflexión de tiempo de ejecución se distribuyen en JAR separado: `kotlin-reflect.jar` . Esto se hace para reducir el tamaño del tiempo de ejecución, cortar las funciones no utilizadas y hacer posible apuntar a otras plataformas (como JS).

Examples

Hacer referencia a una clase

Para obtener una referencia a un objeto `KClass` que representa alguna clase, use dos puntos dobles:

```
val c1 = String::class
val c2 = MyClass::class
```

Haciendo referencia a una función

Las funciones son ciudadanos de primera clase en Kotlin. Puede obtener una referencia en él usando dos puntos dobles y luego pasarlo a otra función:

```
fun isPositive(x: Int) = x > 0

val numbers = listOf(-2, -1, 0, 1, 2)
println(numbers.filter(::isPositive)) // [1, 2]
```

Interoperación con la reflexión de Java.

Para obtener un objeto `Class` de Java de `KClass` de Kotlin, use la propiedad de extensión `.java` :

```
val stringKClass: KClass<String> = String::class
val c1: Class<String> = stringKClass.java

val c2: Class<MyClass> = MyClass::class.java
```

El compilador optimizará este último ejemplo para no asignar una instancia de `KClass` intermedia.

Obtención de valores de todas las propiedades de una clase.

Dada clase de `Example` extiende la clase `BaseExample` con algunas propiedades:

```
open class BaseExample(val baseField: String)

class Example(val field1: String, val field2: Int, baseField: String):
    BaseExample(baseField) {

    val field3: String
        get() = "Property without backing field"

    val field4 by lazy { "Delegated value" }

    private val privateField: String = "Private value"
}
```

Uno puede hacerse con todas las propiedades de una clase:

```
val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

Ejecutar este código hará que se genere una excepción. La propiedad `private val privateField` se declara privada y llamar a `member.get(example)` no se realizará correctamente. Una forma de manejarlo es filtrar propiedades privadas. Para hacer eso, tenemos que verificar el modificador de visibilidad del Java getter de una propiedad. En el caso de `private val` el captador no existe, por lo que podemos asumir el acceso privado.

La función de ayuda y su uso podrían verse así:

```
fun isFieldAccessible(property: KProperty1<*, *>): Boolean {
    return property.javaGetter?.modifiers?.let { !Modifier.isPrivate(it) } ?: false
}

val example = Example(field1 = "abc", field2 = 1, baseField = "someText")

example::class.memberProperties.filter { isFieldAccessible(it) }.forEach { member ->
    println("${member.name} -> ${member.get(example)}")
}
```

Otro enfoque es hacer que las propiedades privadas sean accesibles mediante la reflexión:

```
example::class.memberProperties.forEach { member ->
    member.isAccessible = true
    println("${member.name} -> ${member.get(example)}")
}
```

Establecer valores de todas las propiedades de una clase

Como ejemplo, queremos establecer todas las propiedades de cadena de una clase de muestra

```
class TestClass {
    val readOnlyProperty: String
        get() = "Read only!"

    var readWriteString = "asd"
    var readWriteInt = 23

    var readWriteBackedStringProperty: String = ""
        get() = field + '5'
        set(value) { field = value + '5' }

    var readWriteBackedIntProperty: Int = 0
        get() = field + 1
        set(value) { field = value - 1 }

    var delegatedProperty: Int by TestDelegate()

    private var privateProperty = "This should be private"

    private class TestDelegate {
        private var backingField = 3

        operator fun getValue(thisRef: Any?, prop: KProperty<*>): Int {
            return backingField
        }

        operator fun setValue(thisRef: Any?, prop: KProperty<*>, value: Int) {
            backingField += value
        }
    }
}
```

Obtener propiedades mutables se basa en obtener todas las propiedades, filtrando las propiedades mutables por tipo. También necesitamos verificar la visibilidad, ya que la lectura de propiedades privadas resulta en una excepción de tiempo de ejecución.

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            System.out.println("${prop.name} -> ${prop.get(instance)}")
        }
```

Para establecer todas las propiedades de `String` en `"Our Value"` , también podemos filtrar por el tipo de retorno. Dado que Kotlin se basa en Java VM, Type Erasure está en vigencia y, por lo tanto, las Propiedades que devuelven tipos genéricos como `List<String>` serán las mismas que `List<Any>` . Lamentablemente, la reflexión no es una bala de oro y no hay una forma sensata de evitar esto, por lo que debe tener cuidado en sus casos de uso.

```
val instance = TestClass()
TestClass::class.memberProperties
        .filter{ prop.visibility == KVisibility.PUBLIC }
        // We only want strings
```

```
        .filter{ it.returnType.isSubtypeOf(String::class.starProjectedType) }
        .filterIsInstance<KMutableProperty<*>>()
        .forEach { prop ->
            // Instead of printing the property we set it to some value
            prop.setter.call(instance, "Our Value")
        }
```

Lea Reflexión en línea: https://riptutorial.com/es/kotlin/topic/2402/reflexion

Capítulo 35: Regex

Examples

Modismos para la concordancia de expresiones regulares en cuando la expresión

Usando locales inmutables:

Utiliza menos espacio horizontal pero más espacio vertical que la plantilla de "temporarios anónimos". Es preferible sobre la plantilla de "temporarios anónimos" si la expresión `when` está en un bucle; en ese caso, las definiciones de expresiones regulares deben colocarse fuera del bucle.

```
import kotlin.text.regex

var string = /* some string */

val regex1 = Regex( /* pattern */ )
val regex2 = Regex( /* pattern */ )
/* etc */

when {
    regex1.matches(string) -> /* do stuff */
    regex2.matches(string) -> /* do stuff */
    /* etc */
}
```

Usando temporarios anónimos:

Utiliza menos espacio vertical pero más espacio horizontal que la plantilla de "locales inmutables". No debe utilizarse si entonces `when` expresión está en un bucle.

```
import kotlin.text.regex

var string = /* some string */

when {
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    Regex( /* pattern */ ).matches(string) -> /* do stuff */
    /* etc */
}
```

Usando el patrón de visitante:

Tiene la ventaja de emular estrechamente el "argumento de pleno derecho" `when` sintaxis. Esto es beneficioso porque indica más claramente el argumento de la expresión `when` , y también evita

ciertos errores de programación que podrían surgir al tener que repetir el argumento `when` en todos los `whenEntry` . Ya sea la plantilla de "locales inmutables" o "temporarios anónimos" se puede usar con esta implementación como patrón de visitante.

```
import kotlin.text.regex

var string = /* some string */

when (RegexWhenArgument(string)) {
    Regex( /* pattern */ ) -> /* do stuff */
    Regex( /* pattern */ ) -> /* do stuff */
    /* etc */
}
```

Y la definición mínima de la clase contenedora para el argumento de expresión `when` :

```
class RegexWhenArgument (val whenArgument: CharSequence) {
    operator fun equals(whenEntry: Regex) = whenEntry.matches(whenArgument)
    override operator fun equals(whenEntry: Any?) = (whenArgument == whenEntry)
}
```

Introducción a las expresiones regulares en Kotlin.

Esta publicación muestra cómo usar la mayoría de las funciones en la clase `Regex` , trabajar con nulos relacionados con las funciones `Regex` y cómo las cadenas en bruto facilitan la escritura y lectura de patrones de expresiones regulares.

La clase RegEx

Para trabajar con expresiones regulares en Kotlin, debe usar la `Regex(pattern: String)` e invocar funciones como `find(..)` o `replace(..)` en ese objeto de expresión regular.

Un ejemplo sobre cómo usar la clase `Regex` que devuelve verdadero si la cadena de `input` contiene c o d:

```
val regex = Regex(pattern = "c|d")
val matched = regex.containsMatchIn(input = "abc")    // matched: true
```

Lo esencial a entender con todas las funciones `Regex` es que el resultado se basa en hacer coincidir el `pattern` regex y la cadena de `input` . Algunas de las funciones requieren una coincidencia completa, mientras que el resto solo requiere una coincidencia parcial. La función `containsMatchIn(..)` utilizada en el ejemplo requiere una coincidencia parcial y se explica más adelante en esta publicación.

Seguridad nula con expresiones regulares.

Tanto `find(..)` como `matchEntire(..)` devolverán un `MatchResult?` objeto. El `?` el carácter después

de `MatchResult` es necesario para que Kotlin maneje el nulo de forma segura .

Un ejemplo que demuestra cómo Kotlin maneja nulo de forma segura desde una función `Regex` , cuando la función `find(..)` devuelve null:

```
val matchResult =
    Regex("c|d").find("efg")           // matchResult: null
val a = matchResult?.value             // a: null
val b = matchResult?.value.orEmpty()   // b: ""
a?.toUpperCase()                       // Still needs question mark. => null
b.toUpperCase()                        // Accesses the function directly. => ""
```

Con la función `orEmpty()` , `b` no puede ser nulo y el `?` el carácter no es necesario cuando llamas funciones en `b` .

Si no se preocupan por esta manipulación segura de los valores nulos, Kotlin le permite trabajar con valores nulos como en Java con el `!!` caracteres:

```
a!!.toUpperCase()                      // => KotlinNullPointerException
```

Cuerdas crudas en patrones regex

Kotlin proporciona una mejora sobre Java con una cadena sin formato que hace posible escribir patrones de expresiones regulares puras sin doble barra diagonal inversa, que son necesarios con una cadena Java. Una cadena en bruto se representa con una comilla triple:

```
"""\d{3}-\d{3}-\d{4}""" // raw Kotlin string
"\\d{3}-\\d{3}-\\d{4}"  // standard Java string
```

find (entrada: CharSequence, startIndex: Int): MatchResult?

La cadena de `input` se comparará con el `pattern` en el objeto `Regex` . ¿Devuelve un `Matchresult?` objeto con el primer texto coincidente después del `startIndex` , o `null` si el patrón no coincide con la cadena de `input` . La cadena de resultados se recupera de `MatchResult?` propiedad del `value` del objeto. El parámetro `startIndex` es opcional con el valor predeterminado 0.

Para extraer el primer número de teléfono válido de una cadena con detalles de contacto:

```
val phoneNumber :String? = Regex(pattern = """\d{3}-\d{3}-\d{4}""")
    .find(input = "phone: 123-456-7890, e..")?.value // phoneNumber: 123-456-7890
```

Sin un número de teléfono válido en la cadena de `input` , la variable `phoneNumber` será `null` .

findAll (input: CharSequence, startIndex: Int): secuencia

Devuelve todas las coincidencias de la cadena de `input` que coincide con el `pattern` expresiones regulares.

Para imprimir todos los números separados por espacios, desde un texto con letras y dígitos:

```
val matchedResults = Regex(pattern = """\d+""").findAll(input = "ab12cd34ef")
val result = StringBuilder()
for (matchedText in matchedResults) {
    result.append(matchedText.value + " ")
}

println(result) // => 12 34
```

La variable `matchedResults` es una secuencia con objetos `MatchResult` . Con una cadena de `input` sin dígitos, la función `findAll(..)` devolverá una secuencia vacía.

matchEntire (input: CharSequence): MatchResult?

Si todos los caracteres en la cadena de `input` coinciden con el `pattern` expresiones regulares, se devolverá una cadena igual a la `input` . De lo contrario, se devolverá `null` .

Devuelve la cadena de entrada si toda la cadena de entrada es un número:

```
val a = Regex("""\d+""").matchEntire("100")?.value             // a: 100
val b = Regex("""\d+""").matchEntire("100 dollars")?.value     // b: null
```

partidos (entrada: CharSequence): booleano

Devuelve verdadero si toda la cadena de entrada coincide con el patrón de expresiones regulares. Falso de lo contrario.

Comprueba si dos cadenas contienen solo dígitos:

```
val regex = Regex(pattern = """\d+""")
regex.matches(input = "50")             // => true
regex.matches(input = "50 dollars")     // => false
```

contieneMatchIn (entrada: CharSequence):

Boolean

Devuelve verdadero si parte de la cadena de entrada coincide con el patrón de expresiones regulares. Falso de lo contrario.

Probar si dos cadenas contienen al menos un dígito:

```
Regex("""\d+""").containsMatchIn("50 dollars")       // => true
Regex("""\d+""").containsMatchIn("Fifty dollars")    // => false
```

split (entrada: CharSequence, limit: Int): Lista

Devuelve una nueva lista sin todas las coincidencias de expresiones regulares.

Para devolver listas sin dígitos:

```
val a = Regex("""\d+""").split("ab12cd34ef")     // a: [ab, cd, ef]
val b = Regex("""\d+""").split("This is a test") // b: [This is a test]
```

Hay un elemento en la lista para cada división. La primera cadena de `input` tiene tres números. Eso resulta en una lista con tres elementos.

reemplazar (entrada: CharSequence, reemplazo: cadena): cadena

Reemplaza todas las coincidencias del `pattern` expresiones regulares en la cadena de `input` con la cadena de reemplazo.

Para reemplazar todos los dígitos en una cadena con una x:

```
val result = Regex("""\d+""").replace("ab12cd34ef", "x") // result: abxcdxef
```

Lea Regex en línea: https://riptutorial.com/es/kotlin/topic/8364/regex

Capítulo 36: Seguridad nula

Examples

Tipos anulables y no anulables

Los tipos normales, como `String` , no son anulables. Para que sean capaces de mantener valores nulos, tiene que denotar explícitamente al poner un `?` detrás de ellos: `String?`

```
var string        : String = "Hello World!"
var nullableString: String? = null

string = nullableString    // Compiler error: Can't assign nullable to non-nullable type.
nullableString = string    // This will work however!
```

Operador de llamada segura

Para acceder a las funciones y propiedades de los tipos anulables, debe utilizar operadores especiales.

El primero, `?.` , le proporciona la propiedad o función a la que intenta acceder o le da un valor nulo si el objeto es nulo:

```
val string: String? = "Hello World!"
print(string.length)   // Compile error: Can't directly access property of nullable type.
print(string?.length)  // Will print the string's length, or "null" if the string is null.
```

Idioma: llamar a múltiples métodos en el mismo objeto sin verificar

Una forma elegante de llamar a múltiples métodos de un objeto con comprobación nula es usando las `apply` de Kotlin de la siguiente manera:

```
obj?.apply {
    foo()
    bar()
}
```

Esto llamará `foo` y `bar` en `obj` (que es `this` en el bloque de `apply`) solo si `obj` no es nulo, omitiendo todo el bloque de lo contrario.

Para poner una variable anulable dentro del alcance como una referencia no anulable sin convertirla en el receptor implícito de llamadas de función y propiedad, puede usar `let` lugar de `apply` :

```
nullable?.let { notnull ->
```

```
    notnull.foo()
    notnull.bar()
}
```

`notnull` podría tener un nombre, o incluso `notnull` y usarse a través del parámetro lambda implícito `it` .

Moldes inteligentes

Si el compilador puede inferir que un objeto no puede ser nulo en un cierto punto, ya no tiene que usar los operadores especiales:

```
var string: String? = "Hello!"
print(string.length)  // Compile error
if(string != null) {
    // The compiler now knows that string can't be null
    print(string.length)  // It works now!
}
```

> **Nota:** el compilador no le permitirá realizar una conversión inteligente de variables mutables que podrían modificarse entre la comprobación nula y el uso previsto.
>
> Si se puede acceder a una variable desde fuera del alcance del bloque actual (debido a que son miembros de un objeto no local, por ejemplo), debe crear una referencia local nueva que luego pueda realizar una conversión inteligente y usar.

Elimina los nulos de un iterable y un array

A veces necesitamos cambiar el tipo de `Collection<T?>` `Collections<T>` . En ese caso, `filterNotNull` es nuestra solución.

```
val a: List<Int?> = listOf(1, 2, 3, null)
val b: List<Int> = a.filterNotNull()
```

Null Coalescing / Elvis Operator

A veces es deseable evaluar una expresión anulable de una manera si no es así. El operador de elvis, `?:` , Se puede usar en Kotlin para tal situación.

Por ejemplo:

```
val value: String = data?.first() ?: "Nothing here."
```

La expresión anterior devuelve `"Nothing here"` si los `data?.first()` o los `data` sí mismos arrojan un valor `null` o el resultado de los `data?.first()` .

También es posible lanzar excepciones utilizando la misma sintaxis para abortar la ejecución del código.

```
val value: String = data?.second()
    ?: throw IllegalArgumentException("Value can't be null!")
```

> Recordatorio: las nullPointerExceptions se pueden lanzar mediante el operador de aserción (por ejemplo, `data!!.second()!!`)

Afirmación

`!!` los sufijos ignoran la nulabilidad y devuelven una versión no nula de ese tipo. `KotlinNullPointerException` lanzará `KotlinNullPointerException` si el objeto es un `null` .

```
val message: String? = null
println(message!!) //KotlinNullPointerException thrown, app crashes
```

Operador Elvis (? :)

En Kotlin, podemos declarar una variable que puede contener una `null reference` . Supongamos que tenemos una referencia anulable `a` , podemos decir "si `a` no es nulo, utilícelo; de lo contrario, use un valor no nulo `x` "

```
var a: String? = "Nullable String Value"
```

Ahora, `a` puede ser nulo. Entonces, cuando necesitamos acceder al valor de `a` , debemos realizar una verificación de seguridad, ya sea que contenga valor o no. Podemos realizar esta verificación de seguridad por convencional `if...else` declaración.

```
val b: Int = if (a != null) a.length else -1
```

Pero aquí viene el operador avanzado `Elvis` (operador Elvis `?:` . Más arriba `if...else` puede expresarse con el operador de Elvis de la siguiente manera:

```
val b = a?.length ?: -1
```

Si la expresión a la izquierda de `?:` (Aquí: `a?.length`) no es nula, el operador elvis la devuelve, de lo contrario, devuelve la expresión a la derecha (aquí: `-1`). La expresión del lado derecho solo se evalúa si el lado izquierdo es nulo.

Lea Seguridad nula en línea: https://riptutorial.com/es/kotlin/topic/2080/seguridad-nula

Capítulo 37: Tipo de alias

Introducción

Con los alias de tipo, podemos dar un alias a otro tipo. Es ideal para dar un nombre a tipos de funciones como `(String) -> Boolean` Tipo `(String) -> Boolean` o genérico como `Pair<Person, Person>` .

Los alias de tipo soportan genéricos. Un alias puede reemplazar un tipo con genéricos y un alias puede ser genéricos.

Sintaxis

- **typealias** *nombre-alias* **=** *tipo-existente*

Observaciones

Los alias de tipo son una característica del compilador. No se agrega nada en el código generado para la JVM. Todos los alias serán reemplazados por el tipo real.

Examples

Tipo de función

```
typealias StringValidator = (String) -> Boolean
typealias Reductor<T, U, V> = (T, U) -> V
```

Tipo genérico

```
typealias Parents = Pair<Person, Person>
typealias Accounts = List<Account>
```

Lea Tipo de alias en línea: https://riptutorial.com/es/kotlin/topic/9453/tipo-de-alias

Capítulo 38: Tipo de constructores seguros

Observaciones

Un *generador de tipos seguros* es un concepto, en lugar de una característica de lenguaje, por lo que no está estrictamente formalizado.

Una estructura típica de un constructor de tipo seguro

Una función de constructor único generalmente consta de 3 pasos:

1. Crea un objeto.
2. Ejecuta lambda para inicializar el objeto.
3. Agregue el objeto a estructurar o devuélvalo.

Constructores seguros en las bibliotecas de Kotlin

El concepto de constructores de tipo seguro se usa ampliamente en algunas bibliotecas y marcos de Kotlin, por ejemplo:

- Anko
- Wasabi
- Ktor
- Especulación

Examples

Generador de estructura de árbol de tipo seguro

Los constructores se pueden definir como un conjunto de funciones de extensión que toman expresiones lambda con receptores como argumentos. En este ejemplo, se está construyendo un menú de un `JFrame` :

```
import javax.swing.*

fun JFrame.menuBar(init: JMenuBar.() -> Unit) {
    val menuBar = JMenuBar()
    menuBar.init()
    setJMenuBar(menuBar)
}

fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit) {
    val menu = JMenu(caption)
    menu.init()
    add(menu)
}
```

```
fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit) {
    val menuItem = JMenuItem(caption)
    menuItem.init()
    add(menuItem)
}
```

Estas funciones se pueden usar para construir una estructura de árbol de objetos de una manera fácil:

```
class MyFrame : JFrame() {
    init {
        menuBar {
            menu("Menu1") {
                menuItem("Item1") {
                    // Initialize MenuItem with some Action
                }
                menuItem("Item2") {}
            }
            menu("Menu2") {
                menuItem("Item3") {}
                menuItem("Item4") {}
            }
        }
    }
}
```

Lea Tipo de constructores seguros en línea: https://riptutorial.com/es/kotlin/topic/6010/tipo-de-constructores-seguros

Creditos

S. No	Capítulos	Contributors
1	Empezando con Kotlin	babedev, Community, cyberscientist, ganesshkumar, Ihor Kucherenko, Jayson Minard, mnoronha, neworld, Parker Hoyes, Ruckus T-Boom, Sach, Sean Reilly, Sheigutn, Simón Oroño, UnKnown, Urko Pineda
2	Advertencias de Kotlin	Grigory Konushev, Spidfire
3	Anotaciones	Brad Larson, Caelum, Héctor, Mood, piotrek1543, Sapan Zaveri
4	Arrays	egor.zhdan, Sam, UnKnown
5	Bucles en Kotlin	Ben Leggiero, JaseAnderson, mayojava, razzledazzle, Robin
6	Colecciones	Ascension
7	Configurando la compilación de Kotlin	Aaron Christiansen, elect, madhead
8	coroutines	Jemo Mgebrishvili
9	Declaraciones condicionales	Abdullah, Alex Facciorusso, jpmcosta, Kirill Rakhman, Robin, Spidfire
10	Delegación de clase	Sam
11	Edificio DSL	Dmitriy L, ice1000
12	Enumerar	David Soroko, Kirill Rakhman, SerCe
13	Equivalentes de flujo de Java 8	Brad, Gerson, Jayson Minard, Piero Divasto, Sam
14	Excepciones	Brad Larson, jereksel, Sapan Zaveri
15	Extensiones Kotlin para Android	Jemo Mgebrishvili, Ritave
16	Funciones	Aaron Christiansen, baha, Caelum, glee8e, Jayson Minard, KeksArmee, madhead, Spidfire
17	Fundamentos de Kotlin	Shinoo Goyal

18	Gamas	Nihal Saxena
19	Genéricos	hotkey, Jayson Minard, KeksArmee
20	Herencia de clase	byxor, KeksArmee, piotrek1543, Slav
21	Instrumentos de cuerda	Januson, Sam
22	Interfaces	Divya, Jan Vladimir Mostert, Jayson Minard, Ritave, Robin
23	JUIT	jenglert
24	Kotlin para desarrolladores de Java	Thorsten Schleinzer
25	Lambdas basicas	memoizr, Rich Kuzsma
26	loguearse en kotlin	Konrad Jamrozik, olivierlemasle, oshai
27	Métodos de extensión	Dávid Tímár, Jayson Minard, Kevin Robatel, Konrad Jamrozik, olivierlemasle, Parker Hoyes, razzledazzle
28	Modificadores de visibilidad	Avijit Karmakar
29	Modismos	Aaron Christiansen, Adam Arold, Brad Larson, Héctor, Jayson Minard, Konrad Jamrozik, madhead, mayojava, razzledazzle, Sapan Zaveri, Serge Nikitin, yole
30	Objetos singleton	Divya, glee8e
31	Parámetros Vararg en Funciones	byxor, piotrek1543, Sam
32	Propiedades delegadas	Sam, Seaskyways
33	RecyclerView en Kotlin	Mohit Suthar
34	Reflexión	atok, Kirill Rakhman, madhead, Ritave, Sup
35	Regex	Espen, Travis
36	Seguridad nula	KeksArmee, Kirill Rakhman, piotrek1543, razzledazzle, Robin, SerCe, Spidfire, technerd, Thorsten Schleinzer
37	Tipo de alias	Kevin Robatel

38	Tipo de constructores seguros	Slav