EBook Gratuito

APPRENDIMENTO Kotlin

Free unaffiliated eBook created from
Stack Overflow contributors.

#kotlin

Sommario

Di

Capitolo 1: Iniziare con Kotlin

Osservazioni

Kotlin è un linguaggio di programmazione orientato agli oggetti con funzionalità statiche sviluppato da JetBrains che si rivolge principalmente alla JVM. Kotlin è stato sviluppato con l'obiettivo di essere veloce da compilare, retrocompatibile, molto sicuro da usare e al 100% interoperabile con Java. Kotlin è anche sviluppato con l'obiettivo di fornire molte delle funzionalità richieste dagli sviluppatori Java. Il compilatore standard di Kotlin consente di compilare sia in bytecode Java per JVM che in JavaScript.

Compilando Kotlin

Kotlin ha un plugin IDE standard per Eclipse e IntelliJ. Kotlin può anche essere compilato usando Maven , usando Ant e usando Gradle , o tramite la riga di comando .

Vale la pena notare in `$ kotlinc Main.kt` restituirà un file di classe java, in questo caso `MainKt.class` (Notare il Kt aggiunto al nome della classe). Tuttavia, se si dovesse eseguire il file di classe utilizzando `$ java MainKt` java, verrà `$ java MainKt` la seguente eccezione:

```
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
```

Per eseguire il file di classe risultante utilizzando Java, è necessario includere il file jar runtime di Kotlin nel percorso di classe corrente.

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

Versioni

Versione	Data di rilascio
1.0.0	2016/02/15
1.0.1	2016/03/16
1.0.2	2016/05/13
1.0.3	2016/06/30

Versione	Data di rilascio
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

Ciao mondo

Tutti i programmi Kotlin iniziano dalla funzione `main` . Ecco un esempio di un semplice programma Kotlin "Hello World":

```
package my.program

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

Inserisci il codice precedente in un file chiamato `Main.kt` (questo nome file è interamente arbitrario)

Quando si esegue il targeting della JVM, la funzione verrà compilata come metodo statico in una classe con un nome derivato dal nome file. Nell'esempio sopra, la classe principale da eseguire sarebbe `my.program.MainKt` .

Per modificare il nome della classe che contiene funzioni di primo livello per un determinato file, posiziona la seguente annotazione nella parte superiore del file sopra l'istruzione del pacchetto:

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

In questo esempio, la classe principale da eseguire sarebbe ora `my.program.MyApp` .

Guarda anche:

- Funzioni a livello di pacchetto includendo `@JvmName` annotazione `@JvmName` .
- Annotazione degli obiettivi del sito di utilizzo

Ciao mondo usando una dichiarazione di oggetti

In alternativa è possibile utilizzare una dichiarazione oggetto che contiene la funzione principale per un programma Kotlin.

```
package my.program

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

Il nome della classe che verrà eseguito è il nome del tuo oggetto, in questo caso è `my.program.App` .

Il vantaggio di questo metodo su una funzione di primo livello è che il nome della classe per l'esecuzione è più evidente, ed eventuali altre funzioni che si aggiungono si ambito nella classe `App` . Quindi hai anche un'istanza singleton di `App` per memorizzare lo stato e fare altro lavoro.

Guarda anche:

- Metodi statici inclusa l'annotazione `@JvmStatic`

Hello World utilizza un oggetto Companion

Simile all'utilizzo di una dichiarazione oggetto, è possibile definire la funzione `main` di un programma Kotlin utilizzando un oggetto compagno di una classe.

```
package my.program

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

Il nome della classe che verrà eseguito è il nome della classe, in questo caso è `my.program.App` .

Il vantaggio di questo metodo su una funzione di primo livello è che il nome della classe per l'esecuzione è più evidente, ed eventuali altre funzioni che si aggiungono si ambito nella classe `App` . Questo è simile all'esempio della `Object Declaration` , a parte il controllo sull'istanza di qualsiasi classe per svolgere ulteriori attività.

Una leggera variazione che istanzia la classe a fare il vero "ciao":

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

    fun run() {
```

```
        println("Hello World")
    }
}
```

Guarda anche:

- Metodi statici inclusa l'annotazione @JvmStatic

Metodi principali che utilizzano vararg

Tutti questi stili principali del metodo possono essere utilizzati anche con vararg :

```
package my.program

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

Compilare ed eseguire il codice Kotlin nella riga di comando

Poiché java fornisce due diversi comandi per compilare ed eseguire il codice Java. Lo stesso di Kotlin fornisce anche comandi diversi.

`javac` per compilare i file java. `java` per eseguire file java.

Lo stesso di `kotlinc` per compilare i file `kotlin` per eseguire i file kotlin.

Lettura dell'input da Command Line

Gli argomenti passati dalla console possono essere ricevuti nel programma Kotlin e possono essere utilizzati come input. È possibile passare N (1 2 3 e così via) numeri di argomenti dal prompt dei comandi.

Un semplice esempio di un argomento da linea di comando in 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;

}
```

Qui, inserisci due numeri dalla riga di comando per trovare il numero massimo. Produzione :

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

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

Per !! Operatore Si prega di verificare Null Safety .

Nota: l'esempio precedente viene compilato ed eseguito su Intellij.

Leggi Iniziare con Kotlin online: https://riptutorial.com/it/kotlin/topic/490/iniziare-con-kotlin

Capitolo 2: annotazioni

Examples

Dichiarazione di un'annotazione

Le annotazioni sono mezzi per allegare metadati al codice. Per dichiarare un'annotazione, metti il modificatore di annotazione davanti a una classe:

```
annotation class Strippable
```

Le annotazioni possono avere meta-anotazioni:

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

Le annotazioni, come altre classi, possono avere costruttori:

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

Ma a differenza di altre classi, è limitato ai seguenti tipi:

- tipi che corrispondono ai tipi primitivi di Java (Int, Long ecc.);
- stringhe
- classi (Foo :: classe)
- enumerazioni
- altre annotazioni
- matrici dei tipi sopra elencati

Meta-annotazioni

Quando si dichiara un'annotazione, è possibile includere meta-informazioni utilizzando le seguenti meta-annotazioni:

- `@Target` : specifica i possibili tipi di elementi che possono essere annotati con l'annotazione (classi, funzioni, proprietà, espressioni, ecc.)
- `@Retention` specifica se l'annotazione è archiviata nei file di classe compilati e se è visibile tramite reflection in fase di esecuzione (per impostazione predefinita, entrambi sono veri).
- `@Repeatable` consente di utilizzare la stessa annotazione su un singolo elemento più volte.
- `@MustBeDocumented` specifica che l'annotazione fa parte dell'API pubblica e deve essere inclusa nella firma della classe o del metodo indicata nella documentazione dell'API generata.

Esempio:

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

Leggi annotazioni online: https://riptutorial.com/it/kotlin/topic/4074/annotazioni

Capitolo 3: Array

Examples

Array generici

Gli array generici in Kotlin sono rappresentati da `Array<T>` .

Per creare un array vuoto, utilizzare la funzione `emptyArray<T>()` :

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

Per creare un array con dimensioni e valori iniziali specifici, utilizzare il costruttore:

```
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
```

Le matrici hanno `get(index: Int): T` e `set(index: Int, value: T)` funzioni:

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

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

Matrici di primitivi

Questi tipi **non** ereditano dalla `Array<T>` per evitare la boxe, tuttavia hanno gli stessi attributi e metodi.

Tipo Kotlin	Funzione di fabbrica	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[]`

estensioni

`average()` è definito per `Byte` , `Int` , `Long` , `Short` , `Double` , `Float` e restituisce sempre `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()` restituisce un elemento dell'array

`getOrNull(index: Int)` restituisce null se index è fuori limite, altrimenti un elemento dell'array

`first()` , `last()`

`toHashSet()` restituisce un `HashSet<T>` di tutti gli elementi

`sortedArray()` , `sortedArrayDescending()` crea e restituisce un nuovo array con elementi ordinati di corrente

`sort()` , `sortDescending` ordina l'array sul posto

`min()` , `max()`

Iterate Array

È possibile stampare gli elementi dell'array utilizzando il loop stesso del ciclo avanzato Java, ma è necessario modificare la parola chiave da `:` a `in` .

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

È inoltre possibile modificare il tipo di dati in ciclo for.

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

Crea un array

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

Crea un array usando una chiusura

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

Crea una matrice non inizializzata

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

L'array restituito avrà sempre un tipo nullable. Le matrici di elementi non annullabili non possono essere create non inizializzate.

Leggi Array online: https://riptutorial.com/it/kotlin/topic/5722/array

Capitolo 4: collezioni

introduzione

A differenza di molte lingue, Kotlin distingue tra collezioni mutabili e immutabili (elenchi, insiemi, mappe, ecc.). Un controllo preciso su quando le collezioni possono essere modificate è utile per eliminare i bug e per progettare buone API.

Sintassi

- listOf, mapOf e setOf restituiscono oggetti di sola lettura che non è possibile aggiungere o rimuovere elementi.
- Se vuoi aggiungere o rimuovere elementi devi usare arrayListOf, hashMapOf, hashSetOf, linkedMapOf (LinkedHashMap), linkedSetOf (LinkedHashSet), mutableListOf (La raccolta di Kotlin MultableList), mutableMapOf (La raccolta di Kotlin MultableMap), mutableSetOf (La raccolta di Kotlin MultableSet), sortedMapOf o sortSetOf
- Ogni raccolta ha metodi come first (), last (), get () e funzioni lambda come filtro, mappa, join, riduci e molti altri.

Examples

Utilizzando 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]"
```

Utilizzando la mappa

```
// 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}"
```

Utilizzando set

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

Leggi collezioni online: https://riptutorial.com/it/kotlin/topic/8846/collezioni

Capitolo 5: Configurazione della build di Kotlin

Examples

Configurazione gradle

`kotlin-gradle-plugin` è usato per compilare il codice di Kotlin con Gradle. Fondamentalmente, la sua versione dovrebbe corrispondere alla versione di Kotlin che si desidera utilizzare. Ad esempio, se si desidera utilizzare Kotlin `1.0.3` , è necessario anche abilitare `kotlin-gradle-plugin` versione `1.0.3` .

È una buona idea esternalizzare questa versione in `gradle.properties` o in `ExtraPropertiesExtension` :

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

   repositories {
     mavenCentral()
   }

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

Quindi devi applicare questo plugin al tuo progetto. Il modo in cui lo fai è diverso quando scegli come target piattaforme diverse:

Targeting JVM

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

Targeting per Android

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

Targeting JS

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

Questi sono i percorsi predefiniti:

- fonti kotlin: `src/main/kotlin`
- sorgenti java: `src/main/java`
- test di kotlin: `src/test/kotlin`
- test java: `src/test/java`
- risorse di runtime: `src/main/resources`
- risorse di prova: `src/test/resources`

Potrebbe essere necessario configurare i `SourceSets` se si utilizza il layout del progetto personalizzato.

Infine, dovrai aggiungere la dipendenza della libreria standard di Kotlin al tuo progetto:

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

Se vuoi usare Kotlin Reflection dovrai anche `compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`

Utilizzando Android Studio

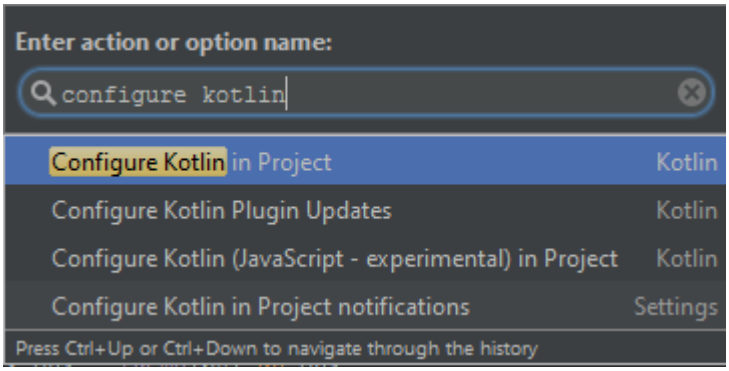
Android Studio può configurare Kotlin automaticamente in un progetto Android.

Installa il plugin

Per installare il plugin Kotlin, vai su File> Impostazioni> Editor> Plugin> Installa plugin JetBrains ...> Kotlin> Installa, quindi riavvia Android Studio quando richiesto.

Configura un progetto

Crea un progetto Android Studio come normale, quindi premi `Ctrl` + `Maiusc` + `A`. Nella casella di ricerca, digita "Configura Kotlin in Progetto" e premi Invio.

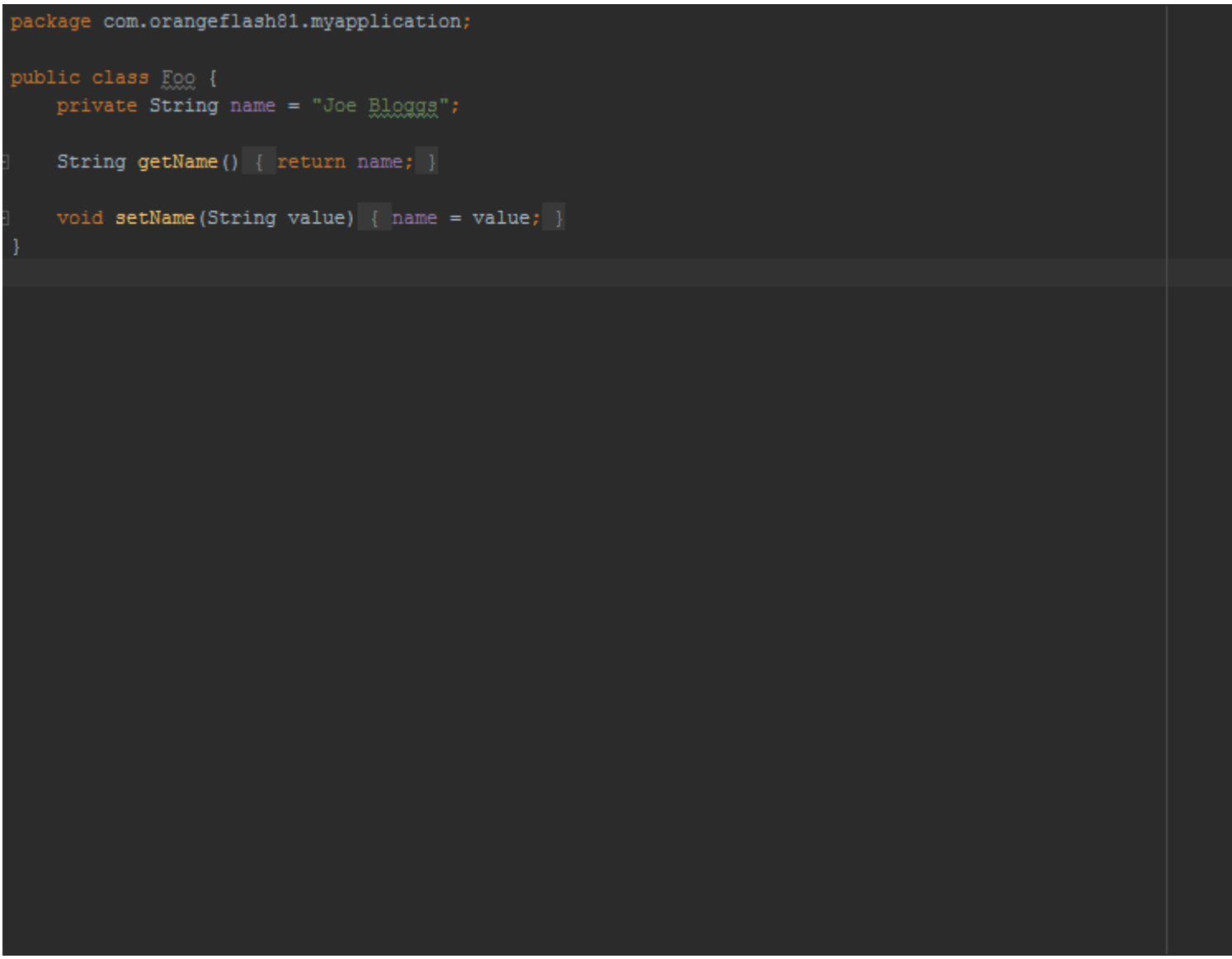


Android Studio modificherà i tuoi file Gradle per aggiungere tutte le dipendenze necessarie.

Conversione di Java

Per convertire i file Java in file Kotlin, premi `Ctrl` + `Maiusc` + `A` e trova "Converti file Java in file

Kotlin". Questo cambierà l'estensione del file corrente in `.kt` e convertirà il codice in Kotlin.



Migrazione da Gradle usando lo script Groovy per lo script Kotlin

passi:

- clonare il progetto gradle-script-kotlin
- copia / incolla dal progetto clonato al tuo progetto:
    - `build.gradle.kts`
    - `gradlew`
    - `gradlew.bat`
    - `settings.gradle`
- aggiornare il contenuto di `build.gradle.kts` base alle proprie esigenze, è possibile utilizzare come ispirazione gli script nel progetto appena clonato o in uno dei suoi esempi
- ora apri Intellij e apri il tuo progetto, nella finestra di explorer, dovrebbe essere riconosciuto come un progetto Gradle, altrimenti lo espandi prima.

- dopo l'apertura, intellij funziona, apri `build.gradle.kts` e controlla se ci sono errori. Se l'evidenziazione non funziona e / o è tutto contrassegnato in rosso, quindi chiudere e riaprire Intellij

- apri la finestra Gradle e aggiornala

Se sei su Windows, potresti incontrare questo bug , scaricare la distribuzione Gradle 3.3 completa e usarla invece di quella fornita. Correlato

OSX e Ubuntu funzionano immediatamente.

Piccolo bonus, se vuoi evitare tutte le seccature di pubblicità su Maven e simili, usa Jitpack , le linee da aggiungere sono quasi identiche a quelle di Groovy. Puoi trarre ispirazione da questo mio progetto .

Leggi Configurazione della build di Kotlin online:
https://riptutorial.com/it/kotlin/topic/2501/configurazione-della-build-di-kotlin

Capitolo 6: coroutine

introduzione

Esempi dell'implementazione sperimentale (ancora) di Kotlin delle coroutine

Examples

Coroutine semplice con ritardo di 1 secondo ma non blocchi

(dal documento ufficiale)

```
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
}
```

risultato

```
Hello,
World!
```

Leggi coroutine online: https://riptutorial.com/it/kotlin/topic/10936/coroutine

Capitolo 7: Costruttori sicuri di tipo

Osservazioni

Un *builder sicuro per i tipi* è un concetto, piuttosto che una funzionalità linguistica, quindi non è strettamente formalizzato.

Una tipica struttura di un costruttore sicuro per i tipi

Una singola funzione di generatore di solito consiste in 3 passaggi:

1. Crea un oggetto.
2. Esegui lambda per inizializzare l'oggetto.
3. Aggiungi l'oggetto alla struttura o restituiscilo.

Costruttori sicuri di tipo nelle librerie Kotlin

Il concetto di builder sicuri per i tipi è ampiamente usato in alcune librerie e framework di Kotlin, ad es .:

- Anko
- Wasabi
- KTOR
- Spec

Examples

Costruttore di strutture ad albero sicuro per tipo

I builder possono essere definiti come un insieme di funzioni di estensione che assumono espressioni lambda con i ricevitori come argomenti. In questo esempio, viene creato un menu di 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)
}
```

Queste funzioni possono quindi essere utilizzate per costruire una struttura ad albero di oggetti in modo semplice:

```
class MyFrame : JFrame() {
    init {
        menuBar {
            menu("Menu1") {
                menuItem("Item1") {
                    // Initialize MenuItem with some Action
                }
                menuItem("Item2") {}
            }
            menu("Menu2") {
                menuItem("Item3") {}
                menuItem("Item4") {}
            }
        }
    }
}
```

Leggi Costruttori sicuri di tipo online: https://riptutorial.com/it/kotlin/topic/6010/costruttori-sicuri-di-tipo

Capitolo 8: Delegazione di classe

introduzione

Una classe Kotlin può implementare un'interfaccia delegando i suoi metodi e proprietà a un altro oggetto che implementa tale interfaccia. Questo fornisce un modo per comporre il comportamento usando l'associazione piuttosto che l'ereditarietà.

Examples

Delegare un metodo a un'altra classe

```
interface Foo {
    fun example()
}

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

class Baz(b : Bar) : Foo by b

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

L'esempio stampa `Hello, world!`

Leggi Delegazione di classe online: https://riptutorial.com/it/kotlin/topic/10575/delegazione-di-classe

Capitolo 9: Dichiarazioni condizionali

Osservazioni

In contrasto con l' `switch` di Java, l'istruzione `when` non ha un comportamento fall-through. Ciò significa che se un ramo è abbinato, il flusso di controllo ritorna dopo la sua esecuzione e non è richiesta alcuna istruzione `break` . Se si desidera combinare i parametri per più argomenti, è possibile scrivere più argomenti separati da virgole:

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

Examples

If-statement standard

```
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!")
}
```

Gli altri rami sono opzionali nelle normali dichiarazioni if.

If-statement come espressione

Le dichiarazioni if possono essere espressioni:

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

Notare che `else` branch non è opzionale se l'espressione `if` viene usata come espressione.

Questo può essere fatto anche con una variante a più righe con parentesi graffe e più istruzioni `else if` .

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

SUGGERIMENTO: Kotlin può dedurre il tipo di variabile per te, ma se vuoi essere sicuro del tipo basta annotarlo sulla variabile come: `val str: String =` questo imporrà il tipo e renderà più facile la lettura.

Quando-invece di catene if-else-if

L'istruzione when è un'alternativa a un'istruzione if con più rami if if:

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

Lo stesso codice scritto usando una catena *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!")
}
```

Proprio come con l'if-statement, il else-branch è opzionale, e puoi aggiungere tutte le ramificazioni che desideri. Puoi anche avere rami multilinea:

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

Corrispondenza dell'argomento when-statement

Quando viene fornito un argomento, lo stato- `when` corrisponde all'argomento contro i rami in sequenza. La corrispondenza viene eseguita utilizzando l'operatore `==` che esegue controlli nulli e confronta gli operandi utilizzando la funzione `equals` . Il primo corrispondente verrà eseguito.

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

L'istruzione when conosce anche alcune opzioni di corrispondenza più avanzate:

```
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
}
```

Quando-affermazione come espressione

Come se, quando può essere usato anche come espressione:

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

Per essere usato come espressione, la dichiarazione quando deve essere esaustiva, ovvero avere un altro ramo o coprire tutte le possibilità con i rami in un altro modo.

Quando-dichiarazione con enum

`when` può essere usato per abbinare i valori `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 ->   // ...
    }
}
```

Come puoi vedere nella seconda linea (`Monday` e `Tuedsay`) è anche possibile combinare due o più valori `enum` .

Se i tuoi casi non sono esaustivi, la compilazione mostrerà un errore. Puoi utilizzare `else` per gestire i casi predefiniti:

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

```
        Day.Thursday ->   //
        Day.Friday ->     //
        else ->           // Party on weekend
    }
}
```

Anche se lo stesso può essere fatto usando il costrutto `if-then-else` , `when` prende cura dei valori di `enum` mancanti e lo rende più naturale.

Controlla qui per ulteriori informazioni su kotlin `enum`

Leggi Dichiarazioni condizionali online: https://riptutorial.com/it/kotlin/topic/2685/dichiarazioni-condizionali

Capitolo 10: Digita gli alias

introduzione

Con gli alias di tipo, possiamo dare un alias ad un altro tipo. È ideale per dare un nome a tipi di funzioni come `(String) -> Boolean` o tipo generico come `Pair<Person, Person>` .

Digitare alias di supporto generici. Un alias può sostituire un tipo con generici e un alias può essere generico.

Sintassi

- **typealias** *alias-name* **=** *tipo esistente*

Osservazioni

Digitare alias è una funzionalità del compilatore. Nulla viene aggiunto nel codice generato per JVM. Tutti gli alias verranno sostituiti dal tipo reale.

Examples

Tipo di funzione

```
typealias StringValidator = (String) -> Boolean
typealias Reductor<T, U, V> = (T, U) -> V
```

Tipo generico

```
typealias Parents = Pair<Person, Person>
typealias Accounts = List<Account>
```

Leggi Digita gli alias online: https://riptutorial.com/it/kotlin/topic/9453/digita-gli-alias

Capitolo 11: eccezioni

Examples

Eccezionale cattura con try-catch-finally

Le eccezioni di cattura in Kotlin sono molto simili a Java

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

Puoi anche catturare più eccezioni

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

`try` è anche un'espressione e può restituire valore

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

Kotlin non ha verificato le eccezioni, quindi non devi rilevare eccezioni.

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

Leggi eccezioni online: https://riptutorial.com/it/kotlin/topic/7246/eccezioni

Capitolo 12: Edificio DSL

introduzione

Concentrati sui dettagli della sintassi per progettare i DSL interni in Kotlin.

Examples

Approccio Infix per creare DSL

Se hai:

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

puoi scrivere il seguente codice simile a DSL nei tuoi test:

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

Sovrascrivere il metodo invoke per creare DSL

Se hai:

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

puoi scrivere il seguente codice simile a DSL nel tuo codice di produzione:

```
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")
    }
}
```

Usando operatori con lambda

Se hai:

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

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

Puoi scrivere il seguente codice simile a DSL:

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

Usando le estensioni con lambda

Se hai:

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

Puoi scrivere il seguente codice simile a DSL:

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

Se ti senti confuso con `shouldBe` sopra, consulta l'esempio di `Infix approach to build DSL`.

Leggi Edificio DSL online: https://riptutorial.com/it/kotlin/topic/10042/edificio-dsl

Capitolo 13: enum

Osservazioni

Proprio come in Java, le classi enum di Kotlin hanno metodi sintetici che consentono di elencare le costanti enum definite e ottenere una costante enum con il suo nome. Le firme di questi metodi sono le seguenti (assumendo che il nome della classe enum sia `EnumClass`):

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

Il metodo `valueOf()` genera un `IllegalArgumentException` se il nome specificato non corrisponde a nessuna delle costanti enum definite nella classe.

Ogni costante enum ha proprietà per ottenere il proprio nome e posizione nella dichiarazione della classe enum:

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

Le costanti enum implementano anche l'interfaccia Comparable, in cui l'ordine naturale è l'ordine in cui sono definiti nella classe enum.

Examples

Inizializzazione

Classi Enum come qualsiasi altra classe può avere un costruttore e essere inizializzata

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

Funzioni e proprietà nelle enumerazioni

Le classi Enum possono anche dichiarare membri (cioè proprietà e funzioni). Un punto e virgola (`;`) deve essere inserito tra l'ultimo oggetto enum e la prima dichiarazione membro.

Se un membro è `abstract` , gli oggetti enum devono 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)
}
```

Enum semplice

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

Ogni costante enum è un oggetto. Le costanti Enum sono separate da virgole.

Mutabilità

Le enumerazioni possono essere modificabili, questo è un altro modo per ottenere un comportamento 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]
```

Leggi enum online: https://riptutorial.com/it/kotlin/topic/2286/enum

Capitolo 14: Equivalenti Java 8 Stream

introduzione

Kotlin fornisce molti metodi di estensione su raccolte e iterabili per l'applicazione di operazioni in stile funzionale. Un tipo di `Sequence` dedicato consente la composizione lenta di diverse di tali operazioni.

Osservazioni

A proposito di pigrizia

Se si desidera elaborare una catena pigro, è possibile convertire in una `Sequence` utilizzando `asSequence()` prima della catena. Alla fine della catena di funzioni, di solito si ottiene anche una `Sequence` . Quindi puoi usare `toList()` , `toSet()` , `toMap()` o qualche altra funzione per materializzare la `Sequence` alla fine.

```
// 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()
```

Perché non ci sono tipi?!?

Noterai che gli esempi di Kotlin non specificano i tipi. Questo perché Kotlin ha l'inferenza di tipo completo ed è completamente sicuro al momento della compilazione. Più che Java perché ha anche i tipi nullable e può aiutare a prevenire il temuto NPE. Quindi questo in Kotlin:

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

equivale a:

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

Poiché Kotlin sa cosa sono le `people` e che `people.age` è `Int` l'espressione di filtro consente solo il confronto con un `Int` e that `people.name` è una `String` pertanto il passo della `map` produce un `List<String>` (`List` di `String` di sola lettura).

Ora, se le `people` fossero forse `null` , come in una `List<People>?` poi:

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

Restituisce un `List<String>?` che avrebbe bisogno di essere controllato con il nulla (*o usare uno*

degli altri operatori di Kotlin per valori nullable, vedi questo modo idiomatico di Kotlin per gestire valori nullable e anche il modo Idiomatic di gestire l'elenco nullable o vuoto in Kotlin)

Riutilizzo di flussi

In Kotlin, dipende dal tipo di raccolta se può essere consumata più di una volta. Una `Sequence` genera un nuovo iteratore ogni volta e, a meno che non asserisca "usa una sola volta", può reimpostarsi all'inizio ogni volta che viene eseguito. Pertanto, mentre il seguente non funziona nello stream di Java 8, ma funziona in 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
```

E in Java per ottenere lo stesso comportamento:

```
// 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
```

Pertanto in Kotlin il fornitore dei dati decide se resettare e fornire un nuovo iteratore oppure no. Ma se vuoi vincolare intenzionalmente una `Sequence` ad una iterazione una sola volta, puoi usare la funzione `constrainOnce()` per `Sequence` come segue:

```
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.
```

Guarda anche:

- Riferimento API per le funzioni di estensione per Iterable

- Riferimento API per le funzioni di estensione per Array
- Riferimento API per le funzioni di estensione per List
- Riferimento API per le funzioni di estensione alla mappa

Examples

Accumula nomi in una lista

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

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

Converti elementi in stringhe e concatenali, separati da virgole

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

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

Calcolo della somma degli stipendi del dipendente

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

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

Raggruppa dipendenti per dipartimento

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

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

Calcolo della somma degli stipendi per dipartimento

```
// 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 }}
```

Studenti di partizione in passaggio e in mancanza

```
// 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 }
```

Nomi di membri maschili

```
// 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 }
```

Nomi di gruppo di membri in roster per genere

```
// 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 } }
```

Filtra un elenco in un altro elenco

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

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

Trovare una stringa più corta è una lista

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

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

Diversi tipi di stream n. 2: usare pigramente il primo oggetto se esiste

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

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

Diversi tipi di stream # 3 - iterate una gamma di numeri interi

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

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

Diversi tipi di stream # 4 - iterare un array, mappare i valori, calcolare la media

```
// 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)
```

Diversi tipi di stream # 5 - lazily itera un elenco di stringhe, mappa i valori, converti in Int, trova max

```
// 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)
```

Diversi tipi di stream # 6 - iterare pigramente un flusso di Ints, mappare i valori, stampare i risultati

```
// 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)
```

Diversi tipi di flussi n. 7: ripetizione pigramente Doppio, mappa in Int, mappa in stringa, stampa ciascuno

```
// 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)
```

È in corso il conteggio degli elementi in un elenco dopo il 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') }
```

Funzionamento dei flussi: filtro, maiuscolo, quindi ordinare un elenco

```
// 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)
```

Diversi tipi di stream n. 1: desiderosi di utilizzare il primo oggetto se esiste

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

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

oppure, creare una funzione di estensione su String chiamata 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)
```

Vedi anche: funzione `apply()`

Vedi anche: Funzioni di estensione

Vedi anche: `?.` Operatore di chiamate sicure e in generale nullability: http://stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-with-nullable-values-referencing-o/34498563 # 34498563

Raccogli l'esempio n. 5: trova persone con età legale, stringa formattata di output

```
// 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.
```

E come nota a margine, in Kotlin possiamo creare semplici classi di dati e creare un'istanza dei dati di test come segue:

```
// 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))
```

Raccogli l'esempio n. 6: raggruppa persone per età, età di stampa e nomi insieme

```
// 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 più interessante qui per Kotlin. Prima le risposte sbagliate per esplorare le varianti della creazione di una `Map` da una raccolta / sequenza:

```
// 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>
```

E ora per la risposta corretta:

```
// 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!!
```

Avevamo solo bisogno di unire i valori corrispondenti per comprimere gli elenchi e fornire un trasformatore per `joinToString` a `joinToString` per passare `joinToString` `Person` a `Person.name` .

Raccogli l'esempio # 7a - Nomi delle mappe, unisci insieme al delimitatore

```
// 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(" | ")
```

Raccogli l'esempio # 7b - Raccogli 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)
```

Ma è meglio creare una funzione di estensione, 2 in realtà per abbinare gli stili in 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)) }
```

Ora hai due modi per usare le nuove funzioni `summarizingInt` :

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

// or

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

E tutti questi producono gli stessi risultati. Possiamo anche creare questa estensione per lavorare su `Sequence` e per i tipi primitivi appropriati.

Leggi Equivalenti Java 8 Stream online: https://riptutorial.com/it/kotlin/topic/707/equivalenti-java-8-stream

Capitolo 15: Eredità di classe

introduzione

Qualsiasi linguaggio di programmazione orientato agli oggetti ha una qualche forma di ereditarietà di classe. Fammi rivedere:

Immagina di dover programmare un mucchio di frutta: `Apples` , `Oranges` e `Pears` . Differiscono per dimensioni, forma e colore, ecco perché abbiamo classi diverse.

Ma diciamo che le loro differenze non contano per un secondo e tu vuoi solo un `Fruit` , non importa quale sia esattamente? Quale tipo di `getFruit()` avrebbe `getFruit()` ?

La risposta è classe `Fruit` . Creiamo una nuova classe e ne ereditiamo tutti i frutti!

Sintassi

- apri {Base Class}
- class {Derived Class}: {Base Class} ({Init Arguments})
- override {Definizione funzione}
- {DC-Object} è {Base Class} == true

Parametri

Parametro	Dettagli
Classe Base	Classe ereditata da
Classe derivata	Classe che eredita dalla classe base
Argomenti di init	Argomenti passati al costruttore di Base Class
Definizione della funzione	Funzione nella classe derivata con codice diverso da quello della classe base
DC-Object	Oggetto "Derivato oggetto di classe" con il tipo di Classe derivata

Examples

Nozioni di base: la parola chiave 'aperta'

In Kotlin, le classi sono **definitive di default, il** che significa che non possono essere ereditate da.

Per consentire l'ereditarietà su una classe, usa la parola chiave `open` .

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

> **Nota: le** classi astratte, le classi sigillate e le interfacce saranno `open` per impostazione predefinita.

Ereditare i campi da una classe

Definire la classe base:

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

Definizione della classe derivata:

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

Utilizzando la sottoclasse:

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

Ereditare metodi da una classe

Definire la classe base:

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

Definizione della classe derivata:

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

Il Ninja ha accesso a tutti i metodi in Persona

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

Sovrascrittura di proprietà e metodi

Proprietà di override (sia di sola lettura che mutabili):

```
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
}
```

Metodi di sovrascrittura:

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

object Titanic : Ship {

    var canSail = true

    override fun sail() {
        sink()
    }

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

Leggi Eredità di classe online: https://riptutorial.com/it/kotlin/topic/5622/eredita-di-classe

Capitolo 16: Estensioni Android Kotlin

introduzione

Kotlin ha un'iniezione di vista integrata per Android, che consente di saltare l'associazione manuale o la necessità di strutture come ButterKnife. Alcuni dei vantaggi sono una sintassi migliore, una digitazione statica migliore e quindi una minore inclinazione agli errori.

Examples

Configurazione

Inizia con un progetto gradle correttamente configurato .

Nella dichiarazione del plugin `build.gradle` **locale del progetto** (non di livello superiore) `build.gradle` estensioni sotto il tuo plugin Kotlin, al livello di indentazione di primo livello.

```
buildscript {
    ...
}

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

Utilizzo di viste

Supponendo che abbiamo un'attività con un layout di esempio chiamato `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>
```

Possiamo usare le estensioni di Kotlin per chiamare il pulsante senza alcun legame aggiuntivo in questo modo:

```
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!")
    }
}
```

Puoi anche importare tutti gli ID che appaiono nel layout con una notazione `*`

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

Le viste sintetiche non possono essere utilizzate al di fuori di Attività / Frammenti / Viste con tale layout gonfiato:

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

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

Sapori del prodotto

Le estensioni Android funzionano anche con più Android Product Flavors. Per esempio se abbiamo sapori in `build.gradle` in `build.gradle` modo:

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

Ad esempio, solo il sapore gratuito ha un pulsante di acquisto:

```
<?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>
```

Possiamo legare al gusto in particolare:

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

L'ascoltatore doloroso per essere avvisato, quando la vista è completamente disegnata ora è così semplice e impressionante con l'estensione di Kotlin

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

Sotto il cappuccio

```
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()
        }
    }
})
}
```

Leggi Estensioni Android Kotlin online: https://riptutorial.com/it/kotlin/topic/9474/estensioni-android-kotlin

Capitolo 17: funzioni

Sintassi

- fun ***Name*** (***Params***) = ...
- ***nome*** divertente (***Params***) {...}
- Fun ***Name*** (***Params***): ***Digitare*** {...}
- divertente < ***Tipo argomento*** > ***Nome*** (***Params***): ***Tipo*** {...}
- Inline divertimento ***Nome (Parametri): Type*** {...}
- { ***ArgName*** : ***ArgType*** -> ...}
- { ***ArgName*** -> ...}
- { ***ArgNames*** -> ...}
- {(***ArgName*** : ***ArgType***): ***Type*** -> ...}

Parametri

Parametro	Dettagli
Nome	Nome della funzione
Parametri	Valori assegnati alla funzione con un nome e tipo: `Name : Type`
genere	Restituisce il tipo di funzione
Digita argomento	Digitare il parametro utilizzato nella programmazione generica (non necessariamente il tipo restituito)
ArgName	Nome del valore assegnato alla funzione
ArgType	***Specifier di*** tipo per ***ArgName***
ArgNames	Elenco di ArgName separato da virgole

Examples

Funzioni che assumono altre funzioni

Come visto in "Funzioni Lambda", le funzioni possono assumere altre funzioni come parametro. Il "tipo di funzione" che devi dichiarare funzioni che assumono altre funzioni è il seguente:

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

# Takes a string and an integer and returns ReturnType
(arg1: String, arg2: Int) -> ReturnType
```

Ad esempio, potresti usare il tipo più vago, `() -> Any?` , per dichiarare una funzione che esegue due volte una funzione lambda:

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

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

Funzioni Lambda

Le funzioni Lambda sono funzioni anonime che vengono solitamente create durante una chiamata di funzione per fungere da parametro di funzione. Sono dichiarati dalle espressioni circostanti con {parentesi} - se sono necessari argomenti, questi vengono messi prima di una freccia `->` .

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

L'ultima istruzione all'interno di una funzione lambda è automaticamente il valore di ritorno.

I tipi sono opzionali, se metti il lambda su un posto dove il compilatore può inferire i tipi.

Argomenti multipli:

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

Se la funzione lambda necessita solo argomento, allora la lista argomento può essere omesso e il singolo argomento indicate con `it` invece.

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

Se l'unico argomento di una funzione è una funzione lambda, le parentesi possono essere completamente omesse dalla chiamata di funzione.

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

Riferimenti di funzione

Possiamo fare riferimento a una funzione senza effettivamente chiamarla anteponendo il nome della funzione a `::` . Questo può quindi essere passato a una funzione che accetta qualche altra

funzione come parametro.

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

Le funzioni senza ricevitore saranno convertite in `(ParamTypeA, ParamTypeB, ...) -> ReturnType` dove `ParamTypeA` , `ParamTypeB` ... sono il tipo di parametri della funzione e `ReturnType1 è il tipo di valore restituito dalla funzione.

```
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
```

Le funzioni con un ricevitore (che si tratti di una funzione di estensione o di una funzione membro) hanno una sintassi diversa. Devi aggiungere il nome del tipo del ricevitore prima dei due punti:

```
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.
```

Tuttavia, quando il ricevitore di una funzione è un oggetto, il ricevitore viene omesso dall'elenco dei parametri, poiché questi sono e sono solo un'istanza di tale 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.
```

Dal momento che kotlin 1.1, il riferimento alla funzione può anche essere *limitato* a una variabile, che viene quindi chiamata *riferimento a* una *funzione limitata* .

1.1.0

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

Nota questo esempio è dato solo per mostrare come funziona la funzione di riferimento limitata. È una cattiva pratica in tutti gli altri sensi.

C'è un caso speciale, però. Una funzione di estensione dichiarata come membro non può essere referenziata.

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

Funzioni base

Le funzioni sono dichiarate usando la `fun` parola chiave, seguita da un nome di funzione e da qualsiasi parametro. È anche possibile specificare il tipo di ritorno di una funzione, che per impostazione predefinita è `Unit` . Il corpo della funzione è racchiuso tra parentesi graffe `{}` . Se il tipo di ritorno è diverso `Unit` , il corpo deve emettere una dichiarazione di ritorno per ogni ramo di chiusura all'interno del corpo.

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

Una versione abbreviata dello stesso:

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

E il tipo può essere omesso poiché può essere dedotto:

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

Funzioni di stenografia

Se una funzione contiene solo un'espressione, possiamo omettere le parentesi graffe e usare invece un uguaglianza, come un'assegnazione di variabile. Il risultato dell'espressione viene restituito automaticamente.

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

Funzioni in linea

Le funzioni possono essere dichiarate in linea usando il prefisso `inline` e, in questo caso, si comportano come i macro in C, invece di essere chiamate, vengono sostituite dal codice del corpo della funzione in fase di compilazione. Ciò può portare a benefici prestazionali in alcune circostanze, principalmente laddove i lambda sono utilizzati come parametri di funzione.

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

Una differenza rispetto alle macro C è che le funzioni inline non possono accedere all'ambito da cui vengono chiamate:

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

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

Funzioni dell'operatore

Kotlin ci consente di fornire implementazioni per un insieme predefinito di operatori con una rappresentazione simbolica fissa (come `+` o `*`) e una precedenza fissa. Per implementare un operatore, forniamo una funzione membro o una funzione di estensione con un nome fisso, per il tipo corrispondente. Funzioni che gli operatori di overload devono essere contrassegnati con il modificatore `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
```

Altre funzioni dell'operatore possono essere trovate qui

Leggi funzioni online: https://riptutorial.com/it/kotlin/topic/1280/funzioni

Capitolo 18: Generics

introduzione

Una lista può contenere numeri, parole o qualsiasi cosa. Ecco perché chiamiamo la lista *generica* .

Generics sono fondamentalmente usati per definire quali tipi una classe può contenere e quale tipo un oggetto detiene attualmente.

Sintassi

- class *ClassName* < ***TypeName*** >
- class *ClassName* <*>
- *ClassName* <in ***UpperBound*** >
- *ClassName* <out ***LowerBound*** >
- Classe *Nome* ***<TypeName: upper bound>***

Parametri

Parametro	Dettagli
TypeName	Tipo Nome del parametro generico
Limite superiore	Tipo covariante
Limite inferiore	Tipo controvariante
Nome della classe	Nome della classe

Osservazioni

Implied Upper Bound è Nullable

In Kotlin Generics, il limite superiore del parametro di tipo `T` sarebbe `Any?` . Pertanto per questa classe:

```
class Consumer<T>
```

Il parametro type `T` è davvero `T: Any?` . Per rendere un limite superiore non nullable, `T: Any` esplicitamente specifico. Per esempio:

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

Examples

Variante del sito di dichiarazione

La varianza del sito di dichiarazione può essere considerata come dichiarazione della varianza del sito di utilizzo una volta per tutte i siti di utilizzo.

```
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
```

Gli esempi diffusi di varianza del sito di dichiarazione sono `List<out T>` , che è immutabile in modo che `T` appaia solo come tipo di valore di ritorno, e `Comparator<in T>` , che riceve solo `T` come argomento.

Scostamento del sito d'uso

La varianza del sito di utilizzo è simile ai caratteri jolly Java:

Out-proiezione:

```
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
```

In proiezione:

```
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
```

Star-proiezione

```
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
```

Guarda anche:

- Interoperabilità di Variant Generics quando si chiama Kotlin da Java.

Leggi Generics online: https://riptutorial.com/it/kotlin/topic/1147/generics

Capitolo 19: idiomi

Examples

Creazione di DTO (POJO / POCO)

Le classi di dati in kotlin sono classi create per non fare altro che contenere i dati. Tali classi sono contrassegnate come `data` :

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

Il codice sopra crea una classe `User` con i seguenti generati automaticamente:

- Getter e setter per tutte le proprietà (getter solo per `val` s)
- `equals()`
- `hashcode()`
- `toString()`
- `copy()`
- `componentN()` (dove `N` è la proprietà corrispondente in ordine di dichiarazione)

Proprio come con una funzione, è possibile specificare anche i valori predefiniti:

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

Maggiori dettagli possono essere trovati qui Classi di dati .

Filtrare un elenco

```
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]
```

Delegare a una classe senza fornirla nel costruttore pubblico

Si supponga di voler delegare a una classe ma non si desidera fornire la classe delegata al parametro costruttore. Invece, si vuole costruirlo privatamente, rendendo il chiamante del costruttore inconsapevole. All'inizio questo potrebbe sembrare impossibile perché la delega delle classi consente di delegare solo ai parametri del costruttore. Tuttavia, c'è un modo per farlo, come indicato in questa risposta :

```
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 questo, puoi semplicemente chiamare il costruttore di `MyTable` questo modo: `MyTable()` . La `Table<Int, Int, Int>` a cui i delegati `MyTable` verranno creati privatamente. Il chiamante del costruttore non ne sa nulla.

Questo esempio è basato su questa domanda SO .

Serializable e serialVersionUid in Kotlin

Per creare `serialVersionUID` per una classe in Kotlin hai a disposizione alcune opzioni che prevedono l'aggiunta di un membro all'oggetto companion della classe.

Il bytecode più conciso proviene da un `private const val` che diventerà una variabile statica privata sulla classe `MySpecialCase` , in questo caso `MySpecialCase` :

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

È inoltre possibile utilizzare questi moduli, **ciascuno con un effetto collaterale di avere metodi getter / setter** che non sono necessari per la serializzazione ...

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

Questo crea il campo statico ma crea anche un `getSerialVersionUID` sull'oggetto companion che non è necessario.

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

Questo crea il campo statico ma crea anche un getter statico e `getSerialVersionUID` sulla classe contenente `MySpecialCase` che non è necessaria.

Ma tutto funziona come metodo per aggiungere `serialVersionUID` ad una classe `Serializable` .

Metodi fluidi in Kotlin

I metodi fluenti in Kotlin possono essere gli stessi di Java:

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

Ma puoi anche renderli più funzionali creando una funzione di estensione come:

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

Che consente quindi più ovviamente le funzioni fluenti:

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

Usare let o anche per semplificare il lavoro con oggetti nullable

`let` in Kotlin crea un binding locale dall'oggetto su cui è stato chiamato. Esempio:

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

Questo stamperà `"foo"` e restituirà `Unit` .

La differenza tra `let` e `also` è che puoi restituire qualsiasi valore da un blocco `let` . `also` nell'altra mano sarà sempre Reutrn `Unit` .

Ora, perché questo è utile, chiedi? Perché se chiami un metodo che può restituire `null` e vuoi eseguire qualche codice solo quando quel valore di ritorno non è `null` puoi usare `let` o `also` così:

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

Questo pezzo di codice eseguirà solo il blocco `let` quando `str` non è `null` . Notare l'operatore di sicurezza `null` (`?`).

Utilizzare applica per inizializzare oggetti o per ottenere il concatenamento del metodo

La documentazione di `apply` dice quanto segue:

> chiama il blocco funzione specificato con `this` valore come ricevitore e restituisce `this` valore.

Mentre il kdoc non è così utile `apply` è davvero una funzione utile. In parole povere si `apply` stabilisce un ambito in cui `this` è legato all'oggetto che hai chiamato `apply` . Ciò ti consente di risparmiare del codice quando devi chiamare più metodi su un oggetto che poi restituirai in seguito. Esempio:

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

Questo è come scrivere questo:

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

Leggi idiomi online: https://riptutorial.com/it/kotlin/topic/2273/idiomi

Capitolo 20: interfacce

Osservazioni

Vedi anche: Documentazione di riferimento di Kotlin per Interfacce: Interfacce

Examples

Interfaccia di base

Un'interfaccia di Kotlin contiene dichiarazioni di metodi astratti e implementazioni di metodi predefiniti anche se non possono memorizzare lo stato.

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

Questa interfaccia può ora essere implementata da una classe come segue:

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

Interfaccia con implementazioni predefinite

Un'interfaccia in Kotlin può avere implementazioni predefinite per le funzioni:

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

Le classi che implementano tali interfacce saranno in grado di utilizzare tali funzioni senza reimplementare

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

Proprietà

Le implementazioni di default funzionano anche per getter e setter di proprietà:

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

Le implementazioni degli accessor di interfaccia non possono utilizzare i campi di supporto

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

Implementazioni multiple

Quando più interfacce implementano la stessa funzione, o tutte definiscono con una o più implementazioni, la classe derivata deve risolvere manualmente la chiamata corretta

```
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")
    }
}
```

Proprietà nelle interfacce

È possibile dichiarare le proprietà nelle interfacce. Poiché un'interfaccia non può avere uno stato, puoi dichiarare una proprietà solo come astratta o fornendo un'implementazione predefinita per gli accessor.

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

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

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

Conflitti durante l'implementazione di più interfacce con implementazioni predefinite

Quando si implementa più di un'interfaccia che ha metodi con lo stesso nome che includono implementazioni predefinite, è ambiguo per il compilatore quale implementazione dovrebbe essere utilizzata. In caso di conflitto, lo sviluppatore deve sovrascrivere il metodo in conflitto e fornire un'implementazione personalizzata. Tale implementazione può scegliere di delegare o meno alle implementazioni predefinite.

```
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.
}
```

super parola chiave

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

Se il metodo nell'interfaccia ha la sua implementazione predefinita, possiamo usare la parola chiave super per accedervi.

```
super.funcOne()
```

Leggi interfacce online: https://riptutorial.com/it/kotlin/topic/900/interfacce

Capitolo 21: Intervalli

introduzione

Le espressioni di intervallo sono formate con le funzioni rangeTo che hanno la forma dell'operatore .. che è completata da in e! In. L'intervallo è definito per qualsiasi tipo comparabile, ma per i tipi primitivi integrali ha un'implementazione ottimizzata

Examples

Gamme di tipo integrale

Le gamme di tipi integrali (IntRange, LongRange, CharRange) hanno una caratteristica in più: possono essere ripetute. Il compilatore si occupa di convertirlo in modo analogo al ciclo for indexato di Java, senza costi aggiuntivi

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

funzione downTo ()

se vuoi ripetere i numeri in ordine inverso? È semplice. È possibile utilizzare la funzione downTo () definita nella libreria standard

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

funzione step ()

È possibile iterare su numeri con step arbitrario, non uguale a 1? Certo, la funzione step () ti aiuterà

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

fino alla funzione

Per creare un intervallo che non include il suo elemento finale, puoi utilizzare la funzione until:

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

Leggi Intervalli online: https://riptutorial.com/it/kotlin/topic/10121/intervalli

Capitolo 22: JUnit

Examples

Regole

Per aggiungere una regola JUnit a un dispositivo di prova:

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

L'annotazione `@JvmField` è necessaria per esporre il campo di supporto con la stessa visibilità (pubblica) della proprietà `myRule` (vedi risposta). Le regole di JUnit richiedono che il campo della regola annotata sia pubblico.

Leggi JUnit online: https://riptutorial.com/it/kotlin/topic/6973/junit

Capitolo 23: Kotlin Caveats

Examples

Chiamando a toString () su un tipo nullable

Una cosa da tenere a mente quando si usa il metodo `toString` in Kotlin è la gestione di null in combinazione con `String?` .

Ad esempio, si desidera ottenere il testo da un `EditText` in Android.

Avresti un pezzo di codice come:

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

Ci si aspetterebbe che se il campo non esistesse il valore sarebbe una stringa vuota ma in questo caso è `"null"` .

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

Leggi Kotlin Caveats online: https://riptutorial.com/it/kotlin/topic/6608/kotlin-caveats

Capitolo 24: Kotlin per sviluppatori Java

introduzione

La maggior parte delle persone che vengono a Kotlin hanno un background di programmazione in Java.

Questo argomento raccoglie esempi di confronto tra Java e Kotlin, evidenziando le differenze più importanti e quelle gemme che Kotlin offre su Java.

Examples

Dichiarazione delle variabili

In Kotlin, le dichiarazioni variabili hanno un aspetto leggermente diverso da quello di Java:

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

- Iniziano con uno `val` o `var` , che effettua la dichiarazione `final` (**"val** ue") o **var** iAble.
- Il tipo è indicato dopo il nome, separato da un `:`
- Grazie *all'inferenza di tipo* di Kotlin, la dichiarazione di tipo esplicita può essere sottomessa se esiste un'assegnazione con un tipo che il compilatore è in grado di rilevare senza ambiguità

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

I fatti in breve

- Kotlin non ha bisogno `;` finire le dichiarazioni
- Kotlin è **sicuro**
- Kotlin è al **100% Java interoperabile**
- Kotlin **non** ha **primitive** (ma ottimizza le loro controparti oggetto per la JVM, se possibile)
- Le classi di Kotlin hanno **proprietà, non campi**
- Kotlin offre **classi di dati** con metodi `equals` / `hashCode` generati automaticamente e `hashCode` campo
- Kotlin ha solo eccezioni di runtime, **nessuna eccezione verificata**
- Kotlin **non** ha **`new` parole chiave** . La creazione di oggetti avviene semplicemente chiamando il costruttore come qualsiasi altro metodo.
- Kotlin supporta (limitato) il **sovraccarico dell'operatore** . Ad esempio, l'accesso a un valore

di una mappa può essere scritto come: `val a = someMap["key"]`
- Kotlin può essere compilato non solo in codice byte per la JVM, ma anche in **Java Script** , consentendo di scrivere sia il codice backend che il frontend in Kotlin
- Kotlin è **completamente compatibile con Java 6** , che è particolarmente interessante per quanto riguarda il supporto dei (non così) vecchi dispositivi Android
- Kotlin è una lingua **ufficialmente supportata per lo sviluppo Android**
- Le collezioni di Kotlin hanno una disconnessione incorporata tra **collezioni mutevoli e immutabili** .
- Kotlin supporta **Coroutine** (sperimentale)

Uguaglianza e identità

Kotlin usa `==` per l'uguaglianza (vale a dire, le chiamate è `equals` internamente) e `===` per l'identità referenziale.

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

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

IF, TRY e altri sono espressioni, non affermazioni

In Kotlin, `if` , `try` e gli altri sono espressioni (quindi restituiscono un valore) piuttosto che dichiarazioni (nulle).

Quindi, ad esempio, Kotlin non ha l' *operatore Elvis* ternario di Java, ma puoi scrivere qualcosa del genere:

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

Ancora più non familiare, ma ugualmente espressivo, è l' *espressione* `try` :

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

Leggi Kotlin per sviluppatori Java online: https://riptutorial.com/it/kotlin/topic/10099/kotlin-per-sviluppatori-java

Capitolo 25: Lambdas di base

Sintassi

- Parametri espliciti:
- {parameterName: ParameterType, otherParameterName: OtherParameterType -> anExpression ()}
- Parametri dedotti:
- aggiunta val: (Int, Int) -> Int = {a, b -> a + b}
- Singolo parametro `it` stenografia
- val square: (Int) -> Int = {it * it}
- Firma:
- () -> ResultType
- (InputType) -> ResultType
- (InputType1, InputType2) -> ResultType

Osservazioni

I parametri del tipo di input possono essere omessi quando possono essere omessi quando possono essere dedotti dal contesto. Ad esempio, diciamo che hai una funzione su una classe che accetta una funzione:

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

È possibile utilizzare questa funzione passando un lambda e poiché i parametri sono già specificati nella firma della funzione non è necessario ripeterli nuovamente nell'espressione lambda:

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

Questo vale anche quando assegni un lambda a una variabile:

```
//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 }
```

Quando lambda accetta un parametro e il tipo può essere dedotto dal contesto, puoi fare riferimento al parametro `it` .

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

Examples

Lambda come parametro per filtrare la funzione

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

Lambda è passato come una variabile

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

Lambda per il benchmarking di una chiamata di funzione

Cronometro per uso generico per la tempistica della durata di una funzione da eseguire:

```
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")
```

Leggi Lambdas di base online: https://riptutorial.com/it/kotlin/topic/5878/lambdas-di-base

Capitolo 26: logging in kotlin

Osservazioni

Domanda correlata: modalità idiomatica di registrazione in Kotlin

Examples

kotlin.logging

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

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

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

Utilizzando il framework kotlin.logging

Leggi logging in kotlin online: https://riptutorial.com/it/kotlin/topic/3258/logging-in-kotlin

Capitolo 27: Loop in Kotlin

Osservazioni

In Kotlin, i loop vengono compilati in loop ottimizzati, laddove possibile. Ad esempio, se si esegue iterazione su un intervallo numerico, il bytecode verrà compilato in un ciclo corrispondente basato su valori int interi per evitare il sovraccarico della creazione dell'oggetto.

Examples

Ripeti un'azione x volte

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

In loop su iterables

Puoi eseguire il loop su qualsiasi iterable usando il ciclo for standard:

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

Un sacco di cose in Kotlin sono iterabili, come gli intervalli di numeri:

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

Se è necessario un indice durante l'iterazione:

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

Esiste anche un approccio funzionale all'iterazione incluso nella libreria standard, senza costrutti del linguaggio apparente, utilizzando la funzione forEach:

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

`it` in questo esempio tiene implicitamente l'elemento corrente, vedi funzioni lambda

Mentre cicli

I cicli while e do-while funzionano come fanno in altre lingue:

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

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

Nel ciclo do-while, il blocco delle condizioni ha accesso ai valori e alle variabili dichiarate nel corpo del ciclo.

Rompi e continua

Rompere e continuare le parole chiave funzionano come fanno in altre lingue.

```
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
    }
}
```

Se hai loop nidificati, puoi etichettare le istruzioni loop e qualificare le istruzioni break e continue per specificare il ciclo che vuoi continuare o interrompere:

```
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
    }
}
```

`forEach` , questo approccio non funzionerà per il costrutto funzionale `forEach` .

Iterare su una mappa in 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")
}
```

ricorsione

Il looping tramite ricorsione è anche possibile in Kotlin come nella maggior parte dei linguaggi di programmazione.

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

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

Nell'esempio sopra, la funzione `factorial` sarà chiamata ripetutamente da sola fino a quando non viene soddisfatta la condizione data.

Costrutti funzionali per l'iterazione

La Kotlin Standard Library fornisce anche numerose utili funzioni per lavorare in modo iterativo sulle raccolte.

Ad esempio, la funzione `map` può essere utilizzata per trasformare un elenco di elementi.

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

Uno dei molti vantaggi di questo stile è che consente di concatenare le operazioni in modo simile. Sarebbe necessaria solo una piccola modifica se, per esempio, la lista sopra fosse necessaria per essere filtrata per i numeri pari. La funzione `filter` può essere utilizzata.

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

Leggi Loop in Kotlin online: https://riptutorial.com/it/kotlin/topic/2727/loop-in-kotlin

Capitolo 28: Metodi di estensione

Sintassi

- fun TypeName.extensionName (params, ...) {/ * body * /} // Dichiarazione
- fun <T: Any> TypeNameWithGenerics <T> .extensionName (params, ...) {/ * body * /} // Dichiarazione con Generics
- myObj.extensionName (args, ...) // invocazione

Osservazioni

Le estensioni sono risolte **staticamente** . Ciò significa che il metodo di estensione da utilizzare è determinato dal tipo di riferimento della variabile a cui si sta accedendo; non importa quale sia il tipo della variabile in fase di esecuzione, verrà sempre chiamato lo stesso metodo di estensione. Questo perché **dichiarare un metodo di estensione non aggiunge effettivamente un membro al tipo di destinatario** .

Examples

Estensioni di primo livello

I metodi di estensione di primo livello non sono contenuti in una classe.

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

Sopra un metodo di estensione è definito per il tipo `IntArray` . Si noti che l'oggetto per cui è definito il metodo di estensione (chiamato **ricevitore**) è accessibile usando la parola chiave `this` .

Questa estensione può essere chiamata in questo modo:

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

Potenziale trappola: le estensioni vengono risolte staticamente

Il metodo di estensione da chiamare viene determinato al momento della compilazione in base al tipo di riferimento della variabile a cui si accede. Non importa quale sia il tipo della variabile in fase di esecuzione, verrà sempre chiamato lo stesso metodo di estensione.

```
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())
```

L'esempio precedente stamperà `"Defined for Super"` , perché il tipo dichiarato della variabile `myVar` è `Super` .

Esempio che si estende a lungo per rendere una stringa leggibile dall'uomo

Dato qualsiasi valore di tipo `Int` o `Long` per rendere una stringa leggibile dall'uomo:

```
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()
}
```

Quindi facilmente utilizzabile come:

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

Esempio di estensione della classe Java 7+ Path

Un caso di utilizzo comune per i metodi di estensione è il miglioramento di un'API esistente. Ecco alcuni esempi di aggiunta di `exist` , `notExists` e `deleteRecursively` al Java 7+ `Path` classe:

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

Quale ora può essere invocato in questo esempio:

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

Utilizzo delle funzioni di estensione per migliorare la leggibilità

In Kotlin puoi scrivere codice come:

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

Ma l'uso di `apply` non è chiaro per quanto riguarda il tuo intento. A volte è più chiaro creare una funzione di estensione simile per rinominare in effetti l'azione e renderla più evidente. Questo non dovrebbe essere permesso di sfuggire di mano, ma per azioni molto comuni come la verifica:

```
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
}
```

Ora puoi scrivere il codice come:

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

Che ora facciamo sapere alle persone cosa aspettarsi all'interno del parametro lambda.

Si noti che il parametro di tipo `T` per `verifiedBy` è uguale a `T: Any?` il che significa che anche i tipi nullable saranno in grado di usare quella versione dell'estensione. Sebbene `verifiedWith` richiede non nullable.

Esempio di estensione delle classi temporali di Java 8 per il rendering di una stringa formattata ISO

Con questa dichiarazione:

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

Ora puoi semplicemente:

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

Funzioni di estensione agli oggetti companion (aspetto delle funzioni statiche)

Se si desidera estendere una classe come-se si è una funzione statica, ad esempio per la classe `Something` aggiunge la funzione di ricerca statica `fromString` , questo può funzionare solo se la classe ha un oggetto associato e che la funzione di estensione è stata dichiarata sull'oggetto

associato :

```
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
}
```

Soluzione di proprietà Lazy extension

Si supponga di voler creare una proprietà di estensione che è costosa da calcolare. Pertanto, si desidera memorizzare nella cache il calcolo, utilizzando il delegato della proprietà lazy e fare riferimento all'istanza corrente (`this`), ma non è possibile farlo, come spiegato nei problemi Kotlin KT-9686 e KT-13053 . Tuttavia, v'è una soluzione ufficiale qui fornite .

Nell'esempio, la proprietà dell'estensione è a `color` . Usa un `colorCache` esplicito che può essere usato con `this` dato che non è necessario alcun `lazy` :

```
class KColor(val value: Int)

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

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

Estensioni per un più facile riferimento Visualizza dal codice

È possibile utilizzare le estensioni per la vista di riferimento, non più codice standard dopo aver creato le viste.

L'idea originale è di Anko Library

estensioni

```
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) }
```

Leggi Metodi di estensione online: https://riptutorial.com/it/kotlin/topic/613/metodi-di-estensione

Capitolo 29: Modificatori di visibilità

introduzione

In Kotlin sono disponibili 4 tipi di modificatori di visibilità.

Pubblico: accessibile da qualsiasi luogo.

Privato: questo è possibile accedere solo dal codice del modulo.

Protetto: è possibile accedervi solo dalla classe che lo definisce e da eventuali classi derivate.

Interno: questo è accessibile solo dall'ambito della classe che lo definisce.

Sintassi

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

Examples

Esempio di codice

Pubblico: `public val name = "Avijit"`

Privato: `private val name = "Avijit"`

Protetto: `protected val name = "Avijit"`

Internal: `internal val name = "Avijit"`

Leggi Modificatori di visibilità online: https://riptutorial.com/it/kotlin/topic/10019/modificatori-di-visibilita

Capitolo 30: Nozioni di base di Kotlin

introduzione

Questo argomento copre le basi di Kotlin per i principianti.

Osservazioni

1. Il file Kotlin ha un'estensione .kt.
2. Tutte le classi in Kotlin hanno una superclasse comune Any, che è un super predefinito per una classe senza dichiarare supertipi (simile a Object in Java).
3. Le variabili possono essere dichiarate come val (immutable- assign once once) o var (mutables- valore può essere modificato)
4. Il punto e virgola non è necessario alla fine della dichiarazione.
5. Se una funzione non restituisce alcun valore utile, il suo tipo di ritorno è Unit.It è anche facoltativo. 6. L'uguaglianza regi- strale viene verificata dall'operazione ===. a === b restituisce true se e solo se a e b puntano allo stesso oggetto.

Examples

Esempi di base

1. La dichiarazione del tipo di ritorno dell'Unità è opzionale per le funzioni. I seguenti codici sono equivalenti.

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

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

2. Funzioni Single-Expression: quando una funzione restituisce una singola espressione, le parentesi graffe possono essere omesse e il corpo viene specificato dopo = simbolo

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

Dichiarare esplicitamente che il tipo di reso è facoltativo quando questo può essere dedotto dal compilatore

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

3. Interpolazione a stringhe: l'uso di valori stringa è facile.

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

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

4. In Kotlin, il sistema tipo distingue tra riferimenti che possono contenere null (riferimenti nullable) e quelli che non possono contenere (riferimenti non nulli). Ad esempio, una variabile regolare di tipo String non può contenere null:

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

Per consentire i null, possiamo dichiarare una variabile come stringa nullable, scritta String ?:

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

5.In Kotlin, == controlla effettivamente l'uguaglianza dei valori. Per convenzione, viene tradotta un'espressione come a == b

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

Leggi Nozioni di base di Kotlin online: https://riptutorial.com/it/kotlin/topic/10648/nozioni-di-base-di-kotlin

Capitolo 31: Nulla sicurezza

Examples

Tipi Nullable e Non-Nullable

I tipi normali, come `String` , non sono annullabili. Per renderli in grado di contenere valori nulli, devi indicarlo esplicitamente mettendo un `?` dietro di loro: `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!
```

Operatore di chiamate sicure

Per accedere a funzioni e proprietà di tipi nullable, è necessario utilizzare operatori speciali.

Il primo, `?.` , ti fornisce la proprietà o la funzione a cui stai tentando di accedere, oppure ti dà null se l'oggetto è nullo:

```
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: chiamare più metodi sullo stesso oggetto con controllo null

Un modo elegante per chiamare più metodi di un oggetto con controllo null sta usando Kotlin's `apply` questo modo:

```
obj?.apply {
    foo()
    bar()
}
```

Questo chiamerà `foo` e `bar` su `obj` (che è `this` nel blocco `apply`) solo se `obj` non è nullo, altrimenti salterà l'intero blocco.

Per portare una variabile nullable nell'ambito come riferimento non nullable senza renderlo il ricevitore implicito di chiamate di proprietà e di proprietà, è possibile utilizzare `let` invece di `apply` :

```
nullable?.let { notnull ->
    notnull.foo()
    notnull.bar()
}
```

`notnull` potrebbe essere chiamato nulla, o addirittura lasciato fuori e utilizzato attraverso il parametro lambda implicita `it` .

Cast intelligenti

Se il compilatore può dedurre che un oggetto non può essere nullo ad un certo punto, non devi più usare gli operatori speciali:

```
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:** il compilatore non consente di modificare le variabili mutabili intelligenti che potrebbero essere potenzialmente modificate tra il controllo null e l'utilizzo previsto.
>
> Se una variabile è accessibile dall'esterno del blocco corrente (perché sono membri di un oggetto non locale, ad esempio), è necessario creare un nuovo riferimento locale che è possibile utilizzare come cast intelligente.

Elimina i null da un Iterable e array

A volte abbiamo bisogno di cambiare tipo dalla `Collection<T?>` `Collections<T>` . In tal caso, `filterNotNull` è la nostra soluzione.

```
val a: List<Int?> = listOf(1, 2, 3, null)
val b: List<Int> = a.filterNotNull()
```

Operatore Null Coalescing / Elvis

A volte è preferibile valutare un'espressione nullable in modo if else. L'operatore elvis, `?:` , Può essere utilizzato in Kotlin per una situazione del genere.

Per esempio:

```
val value: String = data?.first() ?: "Nothing here."
```

L'espressione sopra restituisce `"Nothing here"` se `data?.first()` o `data` stessa producono un valore `null` altrimenti il risultato di `data?.first()` .

È anche possibile generare eccezioni utilizzando la stessa sintassi per interrompere l'esecuzione del codice.

```
val value: String = data?.second()
    ?: throw IllegalArgumentException("Value can't be null!")
```

> Promemoria: NullPointerExceptions può essere lanciato usando l' operatore di

asserzione (es. `data!!.second()!!`)

Asserzione

`!!` i suffissi ignorano il nullability e restituiscono una versione non nulla di quel tipo. `KotlinNullPointerException` verrà generato se l'oggetto è `null` .

```
val message: String? = null
println(message!!) //KotlinNullPointerException thrown, app crashes
```

Operatore Elvis (? :)

In Kotlin, possiamo dichiarare una variabile che può contenere `null reference` . Supponiamo di avere un riferimento nullable `a` , possiamo dire "se `a` non è nullo, usalo, altrimenti usa un valore non nullo `x` "

```
var a: String? = "Nullable String Value"
```

Ora, `a` può essere nullo. Pertanto, quando è necessario accedere al valore di `a` , è necessario eseguire un controllo di sicurezza, indipendentemente dal fatto che contenga o meno un valore. Possiamo eseguire questo controllo di sicurezza con la convenzionale istruzione `if...else` .

```
val b: Int = if (a != null) a.length else -1
```

Ma ecco che arriva operatore anticipo `Elvis` (Operatore Elvis: `?:`). Sopra `if...else` può essere espresso con l'operatore Elvis come di seguito:

```
val b = a?.length ?: -1
```

Se l'espressione a sinistra di `?:` (Qui: `a?.length`) non è nulla, l'operatore elvis la restituisce, altrimenti restituisce l'espressione a destra (qui: `-1`). L'espressione sul lato destro viene valutata solo se il lato sinistro è nullo.

Leggi Nulla sicurezza online: https://riptutorial.com/it/kotlin/topic/2080/nulla-sicurezza

Capitolo 32: Oggetti singleton

introduzione

Un *oggetto* è un tipo speciale di classe, che può essere dichiarato usando la parola chiave `object` . Gli oggetti sono simili a Singletons (uno schema di progettazione) in java. Funziona anche come parte statica di java. I principianti che passano da java a kotlin possono utilizzare questa funzione in modo massiccio, al posto di statici o singleton.

Examples

Utilizzare come repalcement di metodi / campi statici di java

```
object CommonUtils {

    var anyname: String ="Hello"

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

Da qualsiasi altra classe, invoca semplicemente la variabile e le funzioni in questo modo:

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

Utilizzare come un singleton

Gli oggetti Kotlin sono in realtà solo singoletti. Il suo vantaggio principale è che non devi usare `SomeSingleton.INSTANCE` per ottenere l'istanza del singleton.

In java il tuo singleton ha questo aspetto:

```
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");
}
```

In kotlin, il codice equivalente è

```
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")
}
```

È molto meno verboso da usare.

Leggi Oggetti singleton online: https://riptutorial.com/it/kotlin/topic/10152/oggetti-singleton

Capitolo 33: Parametri vararg in funzioni

Sintassi

- **Parola chiave Vararg** : `vararg` viene utilizzato in una dichiarazione di metodo per indicare che verrà accettato un numero variabile di parametri.
- **Operatore di spargimento** : un asterisco (`*`) prima di un array utilizzato nelle chiamate di funzione per "spiegare" il contenuto in singoli parametri.

Examples

Nozioni di base: utilizzo della parola chiave vararg

Definire la funzione usando la parola chiave `vararg` .

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

Ora puoi passare tutti i parametri (del tipo corretto) nella funzione che desideri.

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

> **Note: i** parametri Vararg *devono* essere l'ultimo parametro nella lista parametri.

Spread Operator: passaggio degli array alle funzioni vararg

Le matrici possono essere passate in funzioni vararg usando l' **operatore di propagazione** , `*` .

Supponendo che esista la seguente funzione ...

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

Puoi **passare un array** alla funzione in questo modo ...

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

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

L'operatore di spread può essere utilizzato anche **nel mezzo** dei parametri ...

```
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)
```

Leggi Parametri vararg in funzioni online: https://riptutorial.com/it/kotlin/topic/5835/parametri-vararg-in-funzioni

Capitolo 34: Proprietà delegate

introduzione

Kotlin può delegare l'implementazione di una proprietà a un oggetto gestore. Sono inclusi alcuni gestori standard, ad esempio l'inizializzazione pigra o le proprietà osservabili. È anche possibile creare gestori personalizzati.

Examples

Inizializzazione pigra

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

L'esempio stampa `2` .

Proprietà osservabili

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

L'esempio stampa `foo was changed from 1 to 2`

Proprietà con supporto della mappa

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

L'esempio stampa `1`

Delegazione personalizzata

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

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

L'esempio stampa il `Delegated value`

Delegato Può essere utilizzato come strato per ridurre la piastra di riscaldamento

Considera il sistema di tipo null di Kotlin e `WeakReference<T>` .

Quindi diciamo che dobbiamo salvare una sorta di riferimento e che volevamo evitare perdite di memoria, ecco dove entra `WeakReference` .

prendi ad esempio questo:

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

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

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

Ora questo è solo con un WeakReference. Per ridurre questo boilerplate, possiamo usare un delegato di proprietà personalizzato per aiutarci in questo modo:

```
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)
    }
}
```

Così ora possiamo usare le variabili che sono avvolte con `WeakReference` proprio come normali variabili nullable!

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

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

    init {
        reference = this
```

```
    }
}
```

Leggi Proprietà delegate online: https://riptutorial.com/it/kotlin/topic/10571/proprieta-delegate

Capitolo 35: RecyclerView in Kotlin

introduzione

Voglio solo condividere la mia piccola conoscenza e il codice di RecyclerView usando Kotlin.

Examples

Classe principale e adattatore

Presumo che tu sia a conoscenza della sintassi di **Kotlin** e delle modalità di utilizzo, basta aggiungere **RecyclerView** nel file **activity_main.xml** e impostare con la classe adapter.

```
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
        }
    }
```

questa è la classe **dell'adattatore di** visualizzazione **recycler** e crea il file **main_item.xml** come desideri

```
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)
            }
        }
    }
}
```

Leggi RecyclerView in Kotlin online: https://riptutorial.com/it/kotlin/topic/10143/recyclerview-in-kotlin

Capitolo 36: regex

Examples

Idiomi per la corrispondenza di Regex in When Expression

Usando i locali immutabili:

Utilizza meno spazio orizzontale ma più spazio verticale rispetto al modello "anonymous temporaries". Preferibile sul modello "anonymous temporaries" se l'espressione `when` è in un loop - in questo caso, le definizioni regex devono essere posizionate all'esterno del ciclo.

```
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 i temporari anonimi:

Utilizza meno spazio verticale ma più spazio orizzontale rispetto al modello "immutabili locali". Non dovrebbe essere usato se poi `when` espressione è in un ciclo.

```
import kotlin.text.regex

var string = /* some string */

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

Utilizzando il modello di visitatore:

Ha il vantaggio di emulare da vicino il "argument-ful" `when` sintassi. Ciò è utile perché indica più chiaramente l'argomento dell'espressione `when` , e preclude anche alcuni errori del programmatore che potrebbero derivare dal dover ripetere l'argomento `when` in ogni `whenEntry` . O il modello

"immutable locals" o "anonymous temporaries" può essere utilizzato con questa implementazione del pattern visitor.

```
import kotlin.text.regex

var string = /* some string */

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

E la definizione minima della classe wrapper per l'argomento di espressione `when` :

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

Introduzione alle espressioni regolari in Kotlin

Questo post mostra come utilizzare la maggior parte delle funzioni nella classe `Regex` , lavorare con null in modo sicuro correlato alle funzioni `Regex` e come le stringhe raw rendano più facile scrivere e leggere i modelli regex.

La classe RegEx

Per lavorare con le espressioni regolari in Kotlin, devi usare la classe `Regex(pattern: String)` e invocare funzioni come `find(..)` o `replace(..)` su quell'oggetto regex.

Un esempio su come utilizzare la classe `Regex` che restituisce true se la stringa di `input` contiene c o d:

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

La cosa essenziale da capire con tutte le funzioni di `Regex` è che il risultato è basato sulla corrispondenza tra il `pattern` regex e la stringa di `input` . Alcune funzioni richiedono una corrispondenza completa, mentre il resto richiede solo una corrispondenza parziale. La funzione `containsMatchIn(..)` utilizzata nell'esempio richiede una corrispondenza parziale e viene illustrata più avanti in questo post.

Sicurezza nulla con espressioni regolari

Sia `find(..)` che `matchEntire(..)` restituiranno `MatchResult?` oggetto. Il `?` carattere dopo `MatchResult` è necessario affinchè Kotlin gestisca il null in modo sicuro .

Un esempio che dimostra come Kotlin gestisce nettamente in sicurezza da una funzione `Regex` , quando la funzione `find(..)` restituisce 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 funzione `orEmpty()` , `b` non può essere nullo e il `?` il carattere non è necessario quando si chiamano le funzioni su `b` .

Se non ti interessa questa gestione sicura dei valori nulli, Kotlin ti permette di lavorare con valori null come in Java con `!!` personaggi:

```
a!!.toUpperCase()                      // => KotlinNullPointerException
```

Stringhe raw nei pattern regex

Kotlin fornisce un miglioramento rispetto a Java con una stringa non elaborata che consente di scrivere schemi regex puri senza doppi backslash, necessari con una stringa Java. Una stringa grezza è rappresentata con una citazione tripla:

```
"""\d{3}-\d{3}-\d{4}""" // raw Kotlin string
"\\d{3}-\\d{3}-\\d{4}"  // standard Java string
```

find (input: CharSequence, startIndex: Int): MatchResult?

La stringa di `input` verrà confrontata con il `pattern` nell'oggetto `Regex` . Restituisce un `Matchresult?` oggetto con il primo testo abbinato dopo `startIndex` , o `null` se il modello non corrisponde alla stringa di `input` . La stringa del risultato viene recuperata da `MatchResult?` proprietà del `value` dell'oggetto. Il parametro `startIndex` è facoltativo con il valore predefinito 0.

Per estrarre il primo numero di telefono valido da una stringa con i dettagli del contatto:

```
val phoneNumber :String? = Regex(pattern = """\d{3}-\d{3}-\d{4}""")
    .find(input = "phone: 123-456-7890, e..")?.value // phoneNumber: 123-456-7890
```

Con nessun numero di telefono valido nella stringa di `input` , la variabile `phoneNumber` sarà `null` .

findAll (input: CharSequence, startIndex: Int):

Sequence

Restituisce tutte le corrispondenze dalla stringa di `input` che corrisponde al `pattern` regex.

Per stampare tutti i numeri separati con lo spazio, da un testo con lettere e cifre:

```
val matchedResults = Regex(pattern = """\d+""").findAll(input = "ab12cd34ef")
val result = StringBuilder()
for (matchedText in matchedResults) {
    result.append(matchedText.value + " ")
}

println(result) // => 12 34
```

La variabile `matchedResults` è una sequenza con oggetti `MatchResult` . Con una stringa di `input` senza cifre, la funzione `findAll(..)` restituirà una sequenza vuota.

matchEntire (input: CharSequence): MatchResult?

Se tutti i caratteri nella stringa di `input` corrispondono al `pattern` regex, verrà restituita una stringa uguale `input` . Altrimenti, verrà restituito `null` .

Restituisce la stringa di input se l'intera stringa di input è un numero:

```
val a = Regex("""\d+""").matchEntire("100")?.value            // a: 100
val b = Regex("""\d+""").matchEntire("100 dollars")?.value    // b: null
```

matches (input: CharSequence): Boolean

Restituisce true se l'intera stringa di input corrisponde al modello regex. Falso altrimenti.

Verifica se due stringhe contengono solo cifre:

```
val regex = Regex(pattern = """\d+""")
regex.matches(input = "50")             // => true
regex.matches(input = "50 dollars")     // => false
```

containsMatchIn (input: CharSequence): Boolean

Restituisce true se parte della stringa di input corrisponde al modello regex. Falso altrimenti.

Verifica se due stringhe contengono almeno una cifra:

```
Regex("""\d+""").containsMatchIn("50 dollars")       // => true
Regex("""\d+""").containsMatchIn("Fifty dollars")    // => false
```

split (input: CharSequence, limit: Int): List

Restituisce una nuova lista senza tutte le corrispondenze regolari.

Per restituire liste senza cifre:

```
val a = Regex("""\d+""").split("ab12cd34ef")     // a: [ab, cd, ef]
val b = Regex("""\d+""").split("This is a test") // b: [This is a test]
```

C'è un elemento nell'elenco per ogni divisione. La prima stringa di `input` ha tre numeri. Ciò si traduce in una lista con tre elementi.

replace (input: CharSequence, replacement: String): String

Sostituisce tutte le corrispondenze del `pattern` regex nella stringa di `input` con la stringa di sostituzione.

Per sostituire tutte le cifre in una stringa con una x:

```
val result = Regex("""\d+""").replace("ab12cd34ef", "x") // result: abxcdxef
```

Leggi regex online: https://riptutorial.com/it/kotlin/topic/8364/regex

Capitolo 37: Riflessione

introduzione

Reflection è la capacità di una lingua di ispezionare il codice in fase di runtime anziché la compilazione del tempo.

Osservazioni

Reflection è un meccanismo per analizzare i costrutti del linguaggio (classi e funzioni) al runtime.

Durante il targeting della piattaforma JVM, le funzioni di riflessione runtime sono distribuite in JAR separati: `kotlin-reflect.jar` . Questo viene fatto per ridurre le dimensioni del runtime, tagliare le funzionalità non utilizzate e rendere possibile il targeting di altre piattaforme (come JS).

Examples

Fare riferimento a una classe

Per ottenere un riferimento a un oggetto `KClass` che rappresenta alcuni doppi punti di classe:

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

Riferimento a una funzione

Le funzioni sono cittadini di prima classe a Kotlin. È possibile ottenere un riferimento su di esso utilizzando doppio punto e quindi passarlo a un'altra funzione:

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

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

Interoperando con la riflessione di Java

Per ottenere un oggetto di `Class` Java da Kotlin, `KClass` usa la proprietà di estensione `.java` :

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

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

L'ultimo esempio verrà ottimizzato dal compilatore per non allocare un'istanza di `KClass` intermedia.

Ottenere valori di tutte le proprietà di una classe

Dato `Example` classe che estende la classe `BaseExample` con alcune proprietà:

```
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"
}
```

Si possono ottenere tutte le proprietà di una classe:

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

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

L'esecuzione di questo codice causerà il lancio di un'eccezione. La proprietà `private val privateField` è dichiarata privata e la chiamata `member.get(example)` su di essa non avrà successo. Un modo per gestirlo per filtrare le proprietà private. Per farlo dobbiamo controllare il modificatore di visibilità del getter Java di una proprietà. Nel caso di `private val` il getter non esiste quindi possiamo assumere un accesso privato.

La funzione di supporto e il suo utilizzo potrebbero assomigliare a questo:

```
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)}")
}
```

Un altro approccio è rendere accessibili le proprietà private utilizzando la riflessione:

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

Impostazione dei valori di tutte le proprietà di una classe

Ad esempio vogliamo impostare tutte le proprietà stringa di una classe di esempio

```
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
        }
    }
}
```

Ottenere le proprietà mutabili si basa sull'acquisizione di tutte le proprietà, filtrando le proprietà mutabili per tipo. Dobbiamo anche controllare la visibilità, poiché la lettura delle proprietà private risulta in un'eccezione di run-time.

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

Per impostare tutte le proprietà `String` su `"Our Value"` possiamo anche filtrare in base al tipo restituito. Poiché Kotlin è basato su Java VM, Type Erasure è in effetti, e quindi le proprietà che restituiscono tipi generici come `List<String>` saranno uguali a `List<Any>` . Purtroppo la riflessione non è una pallottola d'oro e non esiste un modo ragionevole per evitarlo, quindi è necessario prestare attenzione ai casi d'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")
        }
```

Leggi Riflessione online: https://riptutorial.com/it/kotlin/topic/2402/riflessione

Capitolo 38: stringhe

Examples

Elementi di stringa

Gli elementi di String sono caratteri a cui è possibile accedere tramite la `string[index]` operazioni di indicizzazione `string[index]` .

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

Gli elementi stringa possono essere iterati con un ciclo for.

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

String letterali

Kotlin ha due tipi di stringhe letterali:

- Stringa sfuggita
- Stringa grezza

La stringa di escape gestisce i caratteri speciali facendoli sfuggire. L'escape è fatto con una barra rovesciata. Sono supportate le seguenti sequenze di escape: `\t` , `\b` , `\n` , `\r` , `\'` , `\"` , `\\` e `\$` . Per codificare qualsiasi altro carattere, utilizzare la sintassi della sequenza di escape Unicode: `\uFF00` .

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

Stringa raw delimitata da una virgola tripla `"""` , non contiene escape e può contenere righe nuove e qualsiasi altro carattere

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

Gli spazi bianchi principali possono essere rimossi con la funzione trimMargin () .

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

Il prefisso di margine predefinito è pipe character `|` , questo può essere impostato come parametro su trimMargin; es. `trimMargin(">")` .

Modelli di stringa

Entrambe le stringhe di escape e le stringhe non elaborate possono contenere espressioni di template. L'espressione template è un pezzo di codice che viene valutato e il suo risultato è concatenato in stringa. Inizia con un segno di dollaro `$` e consiste in un nome di variabile:

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

O un'espressione arbitraria in parentesi graffe:

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

Per includere un simbolo di dollaro letterale in una stringa, sfuggire utilizzando una barra rovesciata:

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

L'eccezione è costituita da stringhe non formattate, che non supportano l'escape. Nelle stringhe non formattate è possibile utilizzare la seguente sintassi per rappresentare un segno di dollaro.

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

Uguaglianza delle stringhe

In Kotlin le stringhe vengono confrontate con l'operatore `==` che esegue il checking per la loro uguaglianza strutturale.

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

L'uguaglianza referenziale viene verificata con l'operatore `===` .

```
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
```

Leggi stringhe online: https://riptutorial.com/it/kotlin/topic/8285/stringhe

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare 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	annotazioni	Brad Larson, Caelum, Héctor, Mood, piotrek1543, Sapan Zaveri
3	Array	egor.zhdan, Sam, UnKnown
4	collezioni	Ascension
5	Configurazione della build di Kotlin	Aaron Christiansen, elect, madhead
6	coroutine	Jemo Mgebrishvili
7	Costruttori sicuri di tipo	Slav
8	Delegazione di classe	Sam
9	Dichiarazioni condizionali	Abdullah, Alex Facciorusso, jpmcosta, Kirill Rakhman, Robin, Spidfire
10	Digita gli alias	Kevin Robatel
11	eccezioni	Brad Larson, jereksel, Sapan Zaveri
12	Edificio DSL	Dmitriy L, ice1000
13	enum	David Soroko, Kirill Rakhman, SerCe
14	Equivalenti Java 8 Stream	Brad, Gerson, Jayson Minard, Piero Divasto, Sam
15	Eredità di classe	byxor, KeksArmee, piotrek1543, Slav
16	Estensioni Android Kotlin	Jemo Mgebrishvili, Ritave
17	funzioni	Aaron Christiansen, baha, Caelum, glee8e, Jayson Minard, KeksArmee, madhead, Spidfire

18	Generics	hotkey, Jayson Minard, KeksArmee
19	idiomi	Aaron Christiansen, Adam Arold, Brad Larson, Héctor, Jayson Minard, Konrad Jamrozik, madhead, mayojava, razzledazzle, Sapan Zaveri, Serge Nikitin, yole
20	interfacce	Divya, Jan Vladimir Mostert, Jayson Minard, Ritave, Robin
21	Intervalli	Nihal Saxena
22	JUnit	jenglert
23	Kotlin Caveats	Grigory Konushev, Spidfire
24	Kotlin per sviluppatori Java	Thorsten Schleinzer
25	Lambdas di base	memoizr, Rich Kuzsma
26	logging in kotlin	Konrad Jamrozik, olivierlemasle, oshai
27	Loop in Kotlin	Ben Leggiero, JaseAnderson, mayojava, razzledazzle, Robin
28	Metodi di estensione	Dávid Tímár, Jayson Minard, Kevin Robatel, Konrad Jamrozik, olivierlemasle, Parker Hoyes, razzledazzle
29	Modificatori di visibilità	Avijit Karmakar
30	Nozioni di base di Kotlin	Shinoo Goyal
31	Nulla sicurezza	KeksArmee, Kirill Rakhman, piotrek1543, razzledazzle, Robin, SerCe, Spidfire, technerd, Thorsten Schleinzer
32	Oggetti singleton	Divya, glee8e
33	Parametri vararg in funzioni	byxor, piotrek1543, Sam
34	Proprietà delegate	Sam, Seaskyways
35	RecyclerView in Kotlin	Mohit Suthar
36	regex	Espen, Travis
37	Riflessione	atok, Kirill Rakhman, madhead, Ritave, Sup
38	stringhe	Januson, Sam