eBook Gratuit

APPRENEZ Kotlin

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#kotlin

Table des matières

À propos

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

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

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

Chapitre 1: Démarrer avec Kotlin

Remarques

Kotlin est un langage de programmation orienté objet de type statique développé par JetBrains et destiné principalement à la JVM. Kotlin a été développé dans le but d'être rapide à compiler, rétrocompatible, très sûr et parfaitement compatible avec Java. Kotlin est également développé dans le but de fournir un grand nombre des fonctionnalités souhaitées par les développeurs Java. Le compilateur standard de Kotlin lui permet d'être compilé à la fois en Java bytecode pour la JVM et en JavaScript.

Compiler Kotlin

Kotlin dispose d'un plug-in IDE standard pour Eclipse et IntelliJ. Kotlin peut également être compilé en utilisant Maven , en utilisant Ant et en utilisant Gradle , ou via la ligne de commande .

Il est intéressant de noter que `$ kotlinc Main.kt` affichera un fichier de classe Java, dans ce cas `MainKt.class` (notez le Kt ajouté au nom de la classe). Cependant, si l'on devait exécuter le fichier de classe en utilisant `$ java MainKt` java lancera l'exception suivante:

```
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
```

Pour exécuter le fichier de classe résultant en utilisant Java, il faut inclure le fichier JAR runt-time de Kotlin dans le chemin de la classe en cours.

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

Versions

Version	Date de sortie
1.0.0	2016-02-15
1.0.1	2016-03-16
1.0.2	2016-05-13
1.0.3	2016-06-30

Version	Date de sortie
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

Bonjour le monde

Tous les programmes Kotlin commencent à la fonction `main` . Voici un exemple de programme Kotlin "Hello World" simple:

```
package my.program

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

Placez le code ci-dessus dans un fichier nommé `Main.kt` (ce nom de fichier est totalement arbitraire)

Lorsque vous ciblez la JVM, la fonction sera compilée en tant que méthode statique dans une classe avec un nom dérivé du nom de fichier. Dans l'exemple ci-dessus, la classe principale à exécuter serait `my.program.MainKt` .

Pour modifier le nom de la classe contenant les fonctions de niveau supérieur pour un fichier particulier, placez l'annotation suivante en haut du fichier au-dessus de la déclaration de package:

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

Dans cet exemple, la classe principale à exécuter serait désormais `my.program.MyApp` .

Voir également:

- Fonctions de niveau package, y `@JvmName` annotation `@JvmName` .
- Cibles d'utilisation des annotations

Bonjour tout le monde en utilisant une déclaration d'objet

Vous pouvez également utiliser une déclaration d'objet contenant la fonction principale d'un programme Kotlin.

```
package my.program

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

Le nom de la classe que vous allez exécuter est le nom de votre objet, dans ce cas, `my.program.App` .

L'avantage de cette méthode par rapport à une fonction de niveau supérieur est que le nom de la classe à exécuter est plus évident et que toutes les autres fonctions que vous ajoutez sont intégrées à la classe `App` . Vous avez ensuite également une instance singleton `App` pour stocker l'état et faire d'autres travaux.

Voir également:

- Méthodes statiques incluant l'annotation `@JvmStatic`

Bonjour tout le monde en utilisant un objet compagnon

Semblable à l'utilisation d'une déclaration d'objet, vous pouvez définir la fonction `main` d'un programme Kotlin en utilisant un objet compagnon d'une classe.

```
package my.program

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

Le nom de la classe que vous allez exécuter est le nom de votre classe, en l'occurrence `my.program.App` .

L'avantage de cette méthode par rapport à une fonction de niveau supérieur est que le nom de la classe à exécuter est plus évident et que toutes les autres fonctions que vous ajoutez sont intégrées à la classe `App` . Ceci est similaire à l'exemple de `Object Declaration` , sauf que vous contrôlez l'instanciation de classes pour effectuer d'autres tâches.

Une légère variation qui instancie la classe pour faire le bonjour:

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

```
        }
    }

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

Voir également:

- Méthodes statiques incluant l'annotation @JvmStatic

Principales méthodes utilisant varargs

Tous ces styles de méthode principaux peuvent également être utilisés avec varargs :

```
package my.program

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

Compiler et exécuter le code Kotlin en ligne de commande

Comme java fournit deux commandes différentes pour compiler et exécuter le code Java. Comme Kotlin vous fournit également différentes commandes.

`javac` pour compiler des fichiers java. `java` pour exécuter des fichiers java.

Identique à `kotlinc` pour compiler les fichiers `kotlin` pour exécuter les fichiers kotlin.

Lecture de l'entrée de la ligne de commande

Les arguments transmis depuis la console peuvent être reçus dans le programme Kotlin et peuvent être utilisés en entrée. Vous pouvez passer N (1 2 3 et ainsi de suite) nombre d'arguments à partir de l'invite de commande.

Un exemple simple d'argument de ligne de commande dans 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;

}
```

Ici, entrez deux nombres à partir de la ligne de commande pour trouver le nombre maximum.
Sortie:

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

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

Pour !! Opérateur Veuillez vérifier Null Safety .

Remarque: l'exemple ci-dessus est compilé et exécuté sur Intellij.

Lire Démarrer avec Kotlin en ligne: https://riptutorial.com/fr/kotlin/topic/490/demarrer-avec-kotlin

Chapitre 2: Alias de type

Introduction

Avec les alias de type, nous pouvons donner un alias à un autre type. C'est idéal pour donner un nom aux types de fonctions comme `(String) -> Boolean` ou type générique comme `Pair<Person, Person>` .

Les alias de type prennent en charge les génériques. Un alias peut remplacer un type par des génériques et un alias peut être un générique.

Syntaxe

- **typealias** *nom-alias* **=** *type-existant*

Remarques

Les alias de type sont une fonctionnalité du compilateur. Rien n'est ajouté dans le code généré pour la JVM. Tous les alias seront remplacés par le type réel.

Examples

Type de fonction

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

Type générique

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

Lire Alias de type en ligne: https://riptutorial.com/fr/kotlin/topic/9453/alias---de-type

Chapitre 3: Annotations

Examples

Déclarer une annotation

Les annotations permettent d'attacher des métadonnées au code. Pour déclarer une annotation, placez le modificateur d'annotation devant une classe:

```
annotation class Strippable
```

Les annotations peuvent avoir des méta-anotations:

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

Les annotations, comme les autres classes, peuvent avoir des constructeurs:

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

Mais contrairement aux autres classes, se limite aux types suivants:

- les types correspondant aux types primitifs Java (Int, Long etc.);
- cordes
- classes (Foo :: class)
- énumérations
- autres annotations
- tableaux des types listés ci-dessus

Méta-annotations

Lors de la déclaration d'une annotation, meta-info peut être inclus à l'aide des méta-annotations suivantes:

- `@Target` : spécifie les types d'éléments pouvant être annotés avec l'annotation (classes, fonctions, propriétés, expressions, etc.)
- `@Retention` spécifie si l'annotation est stockée dans les fichiers de classe compilés et si elle est visible lors de la réflexion au moment de l'exécution (par défaut, les deux sont vrais).
- `@Repeatable` permet d'utiliser la même annotation sur un seul élément plusieurs fois.
- `@MustBeDocumented` spécifie que l'annotation fait partie de l'API publique et doit être incluse dans la signature de classe ou de méthode indiquée dans la documentation de l'API générée.

Exemple:

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

Lire Annotations en ligne: https://riptutorial.com/fr/kotlin/topic/4074/annotations

Chapitre 4: Bâtiment DSL

Introduction

Concentrez-vous sur les détails de la syntaxe pour concevoir des DSL internes dans Kotlin.

Examples

Approche Infix pour construire DSL

Si tu as:

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

vous pouvez écrire le code de type DSL suivant dans vos tests:

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

Remplacer la méthode invoke pour construire DSL

Si tu as:

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

vous pouvez écrire le code de type DSL suivant dans votre code de production:

```
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")
    }
}
```

Utiliser des opérateurs avec lambdas

Si tu as:

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

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

Vous pouvez écrire le code de type DSL suivant:

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

Utiliser des extensions avec lambdas

Si tu as:

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

Vous pouvez écrire le code de type DSL suivant:

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

Si vous vous sentez confus avec `shouldBe` ci-dessus, consultez l'exemple de l' `Infix approach to build DSL` .

Lire Bâtiment DSL en ligne: https://riptutorial.com/fr/kotlin/topic/10042/batiment-dsl

Chapitre 5: Boucles à Kotlin

Remarques

Dans Kotlin, les boucles sont compilées dans la mesure du possible avec des boucles optimisées. Par exemple, si vous effectuez une itération sur une plage de nombres, le bytecode sera compilé dans une boucle correspondante basée sur des valeurs int simples afin d'éviter la surcharge de la création d'objets.

Examples

Répéter une action x fois

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

En boucle sur les itérables

Vous pouvez effectuer une boucle sur n'importe quelle itération en utilisant la norme for-loop:

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

Beaucoup de choses dans Kotlin sont itérables, comme les plages de numéros:

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

Si vous avez besoin d'un index en itérant:

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

Il existe également une approche fonctionnelle de l'itération incluse dans la bibliothèque standard, sans apparences de langage, à l'aide de la fonction forEach:

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

`it` dans cet exemple implicitement l'élément tient en cours, voir Fonctions Lambda

Pendant que les boucles

Les boucles while et do-while fonctionnent comme dans d'autres langues:

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

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

Dans la boucle do-while, le bloc de conditions a accès aux valeurs et aux variables déclarées dans le corps de la boucle.

Pause et continuer

Les mots-clés break et continue fonctionnent comme dans d'autres langues.

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

Si vous avez des boucles imbriquées, vous pouvez étiqueter les instructions de boucle et qualifier les instructions break et continue pour spécifier la boucle que vous souhaitez continuer ou interrompre:

```
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
    }
}
```

Cette approche ne fonctionnera pas pour la construction fonctionnelle de `forEach` .

Itérer sur une carte à 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")
}
```

Récursivité

La boucle via la récursivité est également possible dans Kotlin comme dans la plupart des langages de programmation.

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

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

Dans l'exemple ci-dessus, la fonction `factorial` sera appelée à plusieurs reprises jusqu'à ce que la condition donnée soit remplie.

Constructions fonctionnelles pour itération

La bibliothèque standard de Kotlin fournit également de nombreuses fonctions utiles pour travailler itérativement sur des collections.

Par exemple, la fonction de `map` peut être utilisée pour transformer une liste d'éléments.

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

L'un des nombreux avantages de ce style est qu'il permet de chaîner les opérations d'une manière similaire. Seule une modification mineure serait nécessaire si, par exemple, la liste ci-dessus devait être filtrée pour les nombres pairs. La fonction de `filter` peut être utilisée.

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

Lire Boucles à Kotlin en ligne: https://riptutorial.com/fr/kotlin/topic/2727/boucles-a-kotlin

Chapitre 6: Collections

Introduction

Contrairement à de nombreux langages, Kotlin distingue les collections mutables et immuables (listes, ensembles, cartes, etc.). Un contrôle précis sur le moment exact où les collections peuvent être éditées est utile pour éliminer les bogues et pour concevoir de bonnes API.

Syntaxe

- listOf, mapOf et setOf renvoie les objets en lecture seule que vous ne pouvez pas ajouter ou supprimer.
- Si vous souhaitez ajouter ou supprimer des éléments, vous devez utiliser arrayListOf, hashMapOf, hashSetOf, linkedMapOf (LinkedHashMap), linkedSetOf (LinkedHashSet), mutableListOf (la collection Kotlin MultableList), mutableMapOf (la collection Kotlin MultableMap), mutableSetOf (la collection Kotlin MultableSet).), TRIMAPOF ou TriSetOf
- Chaque collection a des méthodes comme first (), last (), get () et les fonctions lambda comme filter, map,

Examples

Utiliser la liste

```
// 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]"
```

Utiliser la carte

```
// 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}"
```

En utilisant set

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

Lire Collections en ligne: https://riptutorial.com/fr/kotlin/topic/8846/collections

Chapitre 7: Configuration de la construction de Kotlin

Examples

Configuration de Gradle

`kotlin-gradle-plugin` est utilisé pour compiler le code Kotlin avec Gradle. Fondamentalement, sa version devrait correspondre à la version de Kotlin que vous souhaitez utiliser. Par exemple, si vous souhaitez utiliser Kotlin `1.0.3` , vous devez `kotlin-gradle-plugin` version `1.0.3` `kotlin-gradle-plugin` .

C'est une bonne idée d'externaliser cette version dans `gradle.properties` ou dans `ExtraPropertiesExtension` :

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

   repositories {
     mavenCentral()
   }

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

Ensuite, vous devez appliquer ce plug-in à votre projet. La façon dont vous procédez diffère lorsque vous ciblez différentes plates-formes:

JVM ciblé

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

Ciblage Android

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

Ciblage JS

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

Ce sont les chemins par défaut:

- sources de kotlin: `src/main/kotlin`
- sources java: `src/main/java`
- tests de kotlin: `src/test/kotlin`
- tests java: `src/test/java`
- ressources d'exécution: `src/main/resources`
- ressources de test: `src/test/resources`

Vous devrez peut-être configurer `SourceSets` si vous utilisez la mise en page de projet personnalisée.

Enfin, vous devrez ajouter la dépendance à la bibliothèque standard Kotlin à votre projet:

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

Si vous voulez utiliser Kotlin Reflection, vous devrez également ajouter `compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`

Utiliser Android Studio

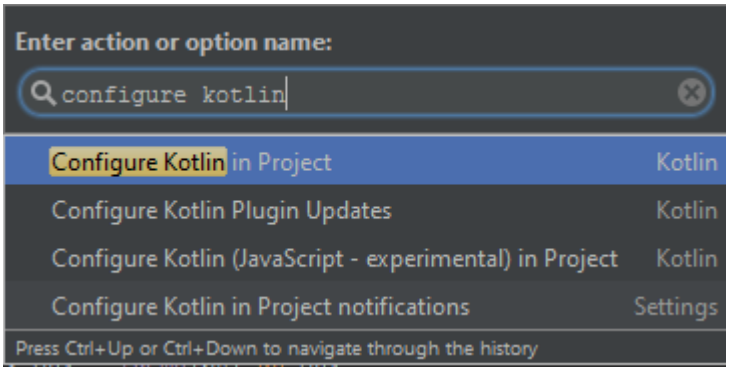
Android Studio peut configurer Kotlin automatiquement dans un projet Android.

Installer le plugin

Pour installer le plug-in Kotlin, sélectionnez Fichier> Paramètres> Editeur> Plug-ins> Installer le plug-in JetBrains ...> Kotlin> Installer, puis redémarrez Android Studio lorsque vous y êtes invité.

Configurer un projet

Créez un projet Android Studio normalement, puis appuyez sur `Ctrl` + `Maj` + `A`. Dans la zone de recherche, tapez "Configurer Kotlin dans le projet" et appuyez sur Entrée.

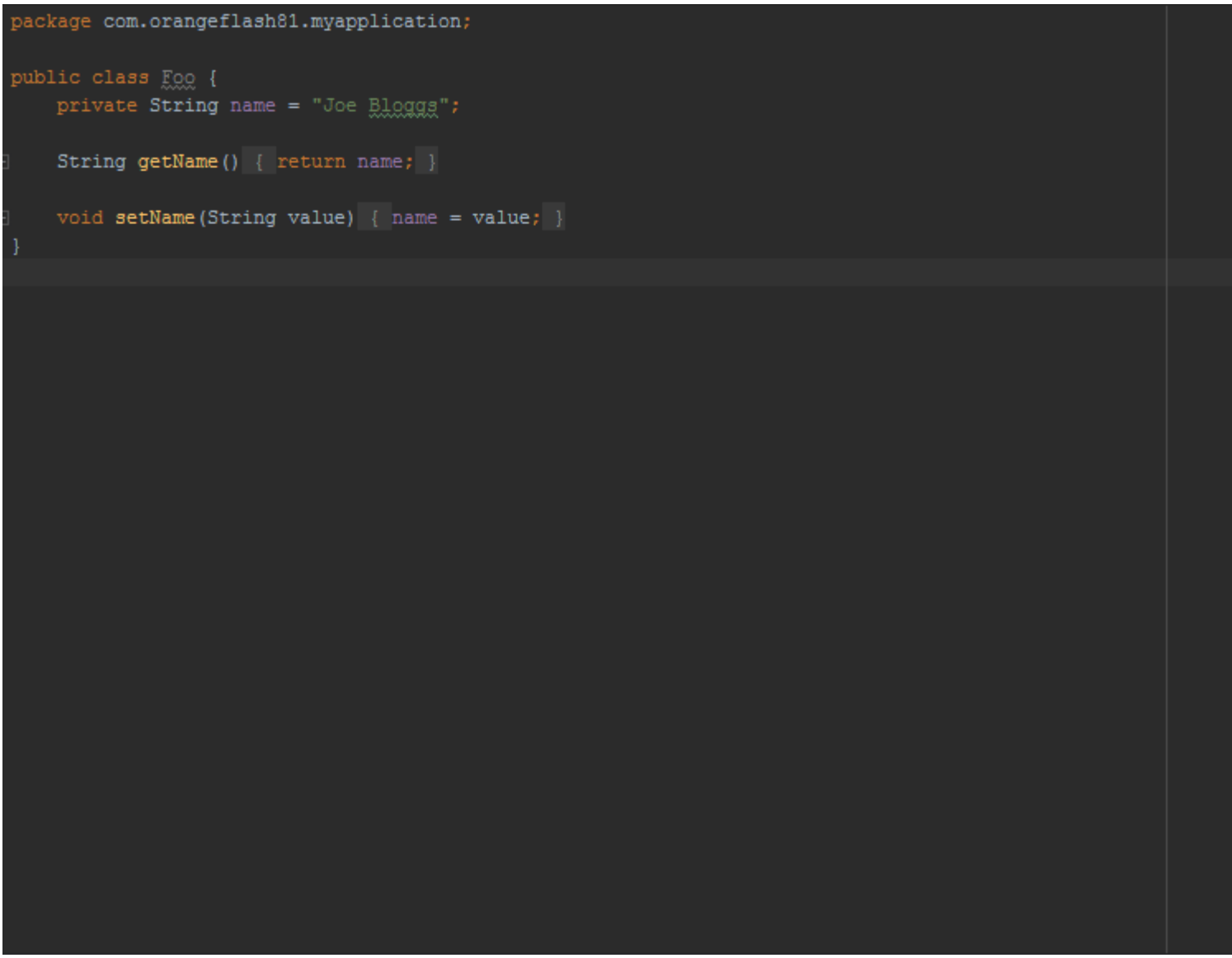


Android Studio va modifier vos fichiers Gradle pour ajouter toutes les dépendances nécessaires.

Conversion de Java

Pour convertir vos fichiers Java en fichiers Kotlin, appuyez sur `Ctrl` + `Maj` + `A` et trouvez "Convertir

le fichier Java en fichier Kotlin". Cela changera l'extension du fichier actuel en `.kt` et convertira le code en Kotlin.



Migration de Gradle à l'aide du script Groovy vers le script Kotlin

Pas:

- cloner le projet gradle-script-kotlin
- copier / coller du projet cloné à votre projet:
  - `build.gradle.kts`
  - `gradlew`
  - `gradlew.bat`
  - `settings.gradle`
- mettre à jour le contenu du `build.gradle.kts` fonction de vos besoins, vous pouvez utiliser comme source d'inspiration les scripts du projet que vous venez de cloner ou dans l'un de ses exemples

- Maintenant, ouvrez Intellij et ouvrez votre projet, dans la fenêtre de l'explorateur, il devrait être reconnu comme un projet Gradle, sinon développez-le d'abord.
- Après ouverture, laissez Intellij fonctionner, ouvrez `build.gradle.kts` et vérifiez s'il y a des erreurs. Si la mise en évidence ne fonctionne pas et / ou si tout est marqué en rouge, fermez et rouvrez Intellij
- ouvrez la fenêtre Gradle et rafraîchissez-la

Si vous êtes sous Windows, vous pouvez rencontrer ce bogue , télécharger la distribution complète de Gradle 3.3 et utiliser celle fournie. Liés

OSX et Ubuntu fonctionnent parfaitement.

Petit bonus, si vous voulez éviter toute la publicité sur Maven et similaire, utilisez Jitpack , les lignes à ajouter sont presque identiques à celles de Groovy. Vous pouvez vous inspirer de ce projet .

Lire Configuration de la construction de Kotlin en ligne:
https://riptutorial.com/fr/kotlin/topic/2501/configuration-de-la-construction-de-kotlin

Chapitre 8: Constructeurs de type sûr

Remarques

Un *générateur de type sécurisé* est un concept, et non une fonctionnalité de langage, de sorte qu'il n'est pas strictement formalisé.

Une structure typique d'un constructeur de type sécurisé

Une seule fonction de constructeur se compose généralement de 3 étapes:

1. Créez un objet.
2. Exécutez lambda pour initialiser l'objet.
3. Ajoutez l'objet pour le structurer ou le renvoyer.

Constructeurs de type sécurisé dans les bibliothèques de Kotlin

Le concept de générateurs sans risque de type est largement utilisé dans certaines bibliothèques et structures de Kotlin, par exemple:

- Anko
- Wasabi
- Ktor
- Spéc

Examples

Générateur d'arborescence sécurisé

Les générateurs peuvent être définis comme un ensemble de fonctions d'extension prenant les expressions lambda avec des récepteurs comme arguments. Dans cet exemple, un menu de `JFrame` est en cours de construction:

```
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)
}
```

Ces fonctions peuvent ensuite être utilisées pour créer facilement une arborescence d'objets:

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

Lire Constructeurs de type sûr en ligne: https://riptutorial.com/fr/kotlin/topic/6010/constructeurs-de-type-sur

Chapitre 9: Cordes

Examples

Éléments de ficelle

Les éléments de chaîne sont des caractères accessibles par la `string[index]` opération d'indexation `string[index]` .

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

Les éléments de chaîne peuvent être itérés avec une boucle for.

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

Littéraux de chaîne

Kotlin a deux types de littéraux de chaîne:

- Ficelle échappé
- Chaîne brute

Chaîne échappée gère les caractères spéciaux en leur échappant. L'échappement se fait avec une barre oblique inverse. Les séquences d'échappement suivantes sont prises en charge: `\t` , `\b` , `\n` , `\r` , `\'` , `\"` , `\\` et `\$` . Pour coder tout autre caractère, utilisez la syntaxe de séquence d'échappement Unicode: `\uFF00` .

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

Chaîne brute délimitée par un triple guillemet `"""` , ne contient aucun échappement et peut contenir des nouvelles lignes et tout autre caractère

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

Les espaces blancs peuvent être supprimés avec la fonction trimMargin () .

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

```
    """.trimMargin()
```

Le préfixe de marge par défaut est le caractère de tuyau `|` , cela peut être défini comme paramètre de trimMargin; eg `trimMargin(">")` .

Modèles de chaînes

Les chaînes d'échappement et les chaînes brutes peuvent contenir des expressions de modèle. L'expression de modèle est un élément de code évalué et son résultat est concaténé en chaîne. Il commence par un signe dollar `$` et se compose soit d'un nom de variable:

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

Ou une expression arbitraire entre accolades:

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

Pour inclure un signe littéral dans une chaîne, échappez-le en utilisant une barre oblique inverse:

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

L'exception concerne les chaînes brutes, qui ne prennent pas en charge l'échappement. Dans les chaînes brutes, vous pouvez utiliser la syntaxe suivante pour représenter un symbole dollar.

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

Égalité de chaîne

Dans Kotlin, les chaînes sont comparées avec l'opérateur `==` qui chect pour leur égalité structurelle.

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

L'égalité référentielle est vérifiée avec l'opérateur `===` .

```
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
```

Lire Cordes en ligne: https://riptutorial.com/fr/kotlin/topic/8285/cordes

Chapitre 10: coroutines

Introduction

Exemples d'implémentation expérimentale (encore) de coroutines de Kotlin

Examples

Coroutine simple qui retarde de 1 seconde mais pas de bloc

(extrait du document officiel)

```
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
}
```

résultat

```
Hello,
World!
```

Lire coroutines en ligne: https://riptutorial.com/fr/kotlin/topic/10936/coroutines

Chapitre 11: Délégation de classe

Introduction

Une classe Kotlin peut implémenter une interface en déléguant ses méthodes et propriétés à un autre objet qui implémente cette interface. Cela permet de composer un comportement en utilisant l'association plutôt que l'héritage.

Examples

Déléguer une méthode à une autre classe

```
interface Foo {
    fun example()
}

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

class Baz(b : Bar) : Foo by b

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

L'exemple imprime `Hello, world!`

Lire Délégation de classe en ligne: https://riptutorial.com/fr/kotlin/topic/10575/delegation-de-classe

Chapitre 12: Des exceptions

Examples

Exception de capture avec try-catch-finally

Catching exceptions dans Kotlin ressemble beaucoup à Java

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

Vous pouvez également intercepter plusieurs exceptions

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

`try` est aussi une expression et peut renvoyer une valeur

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

Kotlin n'a pas vérifié les exceptions, vous n'avez donc aucune exception à prendre.

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

Lire Des exceptions en ligne: https://riptutorial.com/fr/kotlin/topic/7246/des-exceptions

Chapitre 13: Enum

Remarques

Tout comme en Java, les classes enum dans Kotlin ont des méthodes synthétiques permettant de lister les constantes enum définies et d'obtenir une constante enum par son nom. Les signatures de ces méthodes sont les suivantes (en supposant que le nom de la classe enum est `EnumClass`):

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

La méthode `valueOf()` renvoie une `IllegalArgumentException` si le nom spécifié ne correspond à aucune des constantes enum définies dans la classe.

Chaque constante enum a des propriétés pour obtenir son nom et sa position dans la déclaration de la classe enum:

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

Les constantes enum implémentent également l'interface Comparable, l'ordre naturel étant l'ordre dans lequel elles sont définies dans la classe enum.

Examples

Initialisation

Les classes Enum comme toutes les autres classes peuvent avoir un constructeur et être initialisées

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

Fonctions et propriétés dans les énumérations

Les classes Enum peuvent également déclarer des membres (propriétés et fonctions). Un point-virgule (`;`) doit être placé entre le dernier objet enum et la déclaration du premier membre.

Si un membre est `abstract` , les objets enum doivent l'implémenter.

```
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)
}
```

Énumération simple

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

Chaque constante d'énumération est un objet. Les constantes enum sont séparées par des virgules.

Mutabilité

Les énumérations peuvent être mutables, c'est une autre façon d'obtenir un comportement 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]
```

Lire Enum en ligne: https://riptutorial.com/fr/kotlin/topic/2286/enum

Chapitre 14: Expressions conditionnelles

Remarques

Contrairement au `switch` Java, l'instruction `when` n'a pas de comportement de chute. Cela signifie que si une branche est appariée, le flux de contrôle retourne après son exécution et aucune instruction de `break` n'est requise. Si vous voulez combiner les comportements pour plusieurs arguments, vous pouvez écrire plusieurs arguments séparés par des virgules:

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

Examples

Déclaration standard if

```
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!")
}
```

Les autres branches sont facultatives dans les instructions if normales.

Si-déclaration en tant qu'expression

Les déclarations if peuvent être des expressions:

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

Notez que la classe `else` -branch n'est pas facultative si l'instruction `if` est utilisée comme expression.

Cela peut également être fait avec une variante multi-lignes avec des accolades et plusieurs `else if` instructions `else if` .

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

ASTUCE: Kotlin peut déduire le type de la variable pour vous mais si vous voulez être sûr du type, annotez-le simplement sur la variable comme: `val str: String =` cela renforcera le type et le rendra plus lisible.

Quand-instruction au lieu de if-else-if

L'instruction when est une alternative à une instruction if avec plusieurs branches if-if-branches:

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

Même code écrit en utilisant une chaîne *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!")
}
```

Tout comme avec l'instruction if, la branche else est facultative et vous pouvez ajouter autant ou peu de branches que vous le souhaitez. Vous pouvez également avoir des branches multilignes:

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

L'argument quand-instruction correspond

Quand on lui donne un argument, le `when` -statement correspond à l'argument contre les branches dans l'ordre. La correspondance est effectuée à l'aide de l'opérateur `==` qui effectue des vérifications NULL et compare les opérandes à l'aide de la fonction `equals` . Le premier correspondant sera exécuté.

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

L'instruction when connaît également des options de correspondance plus avancées:

```
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
}
```

Quand-déclaration en tant qu'expression

Comme si, quand peut aussi être utilisé comme une expression:

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

Pour être utilisée comme une expression, l'instruction when doit être exhaustive, c'est-à-dire avoir une autre branche ou couvrir toutes les possibilités avec les branches d'une autre manière.

Quand-déclaration avec énumérations

`when` peut-on utiliser des valeurs `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 ->   // ...
    }
}
```

Comme vous pouvez le voir dans la deuxième ligne de cas (`Monday` et `Tuedsay`), il est également possible de combiner plusieurs valeurs `enum` .

Si vos cas ne sont pas exhaustifs, la compilation affichera une erreur. Vous pouvez utiliser `else` pour gérer les cas par défaut:

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

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

Bien que la même chose puisse être faite en utilisant la construction `if-then-else` , `when` prend soin des valeurs `enum` manquantes et le rend plus naturel.

Cliquez ici pour plus d'informations sur kotlin `enum`

Lire Expressions conditionnelles en ligne: https://riptutorial.com/fr/kotlin/topic/2685/expressions-conditionnelles

Chapitre 15: Expressions idiomatiques

Examples

Création de DTO (POJO / POCO)

Les classes de données dans kotlin sont des classes créées pour ne rien faire mais contenir des données. Ces classes sont marquées comme des `data` :

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

Le code ci-dessus crée une classe `User` avec les éléments suivants générés automatiquement:

- Getters et Setters pour toutes les propriétés (seulement pour les `val` s)
- `equals()`
- `hashcode()`
- `toString()`
- `copy()`
- `componentN()` (où `N` est la propriété correspondante dans l'ordre de déclaration)

Tout comme avec une fonction, les valeurs par défaut peuvent également être spécifiées:

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

Plus de détails peuvent être trouvés ici Data Classes .

Filtrer une liste

```
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]
```

Déléguer à une classe sans la fournir dans le constructeur public

Supposons que vous souhaitiez déléguer à une classe mais que vous ne souhaitez pas fournir la classe déléguée dans le paramètre constructeur. Au lieu de cela, vous voulez le construire en privé, en ignorant l'appelant du constructeur. Au début, cela peut sembler impossible car la délégation de classe permet de déléguer uniquement aux paramètres du constructeur. Cependant, il y a un moyen de le faire, comme indiqué dans cette réponse :

```
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
}
```

Avec ceci, vous pouvez simplement appeler le constructeur de `MyTable` comme ceci: `MyTable()` . La `Table<Int, Int, Int>` à laquelle les délégués `MyTable` seront créés en privé. L'appelant constructeur n'en sait rien.

Cet exemple est basé sur cette question SO .

Serializable et serialVersionUid dans Kotlin

Pour créer le `serialVersionUID` d'une classe dans Kotlin, vous disposez de quelques options, toutes impliquant l'ajout d'un membre à l'objet compagnon de la classe.

Le code d'octet le plus concis provient d'une valeur `private const val` qui deviendra une variable statique privée sur la classe contenant, dans ce cas `MySpecialCase` :

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

Vous pouvez également utiliser ces formulaires, **chacun avec un effet secondaire d'avoir des méthodes de lecture / définition** qui ne sont pas nécessaires pour la sérialisation ...

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

Cela crée le champ statique mais crée également un getter `getSerialVersionUID` sur l'objet compagnon qui n'est pas nécessaire.

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

Cela crée le champ statique mais crée également un getter statique ainsi que `getSerialVersionUID` sur la classe contenant `MySpecialCase` qui est inutile.

Mais tout fonctionne comme une méthode pour ajouter `serialVersionUID` à une classe `Serializable` .

Méthodes courant à Kotlin

Les méthodes courantes dans Kotlin peuvent être les mêmes que Java:

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

Mais vous pouvez également les rendre plus fonctionnels en créant une fonction d'extension telle que:

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

Ce qui permet alors des fonctions plus évidentes:

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

Utiliser let ou simplifier le travail avec des objets nullables

`let` dans Kotlin crée une liaison locale à partir de l'objet sur lequel il a été appelé. Exemple:

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

Cela affichera `"foo"` et renverra `Unit` .

La différence entre `let` et `also` est que vous pouvez retourner n'importe quelle valeur d'un bloc `let` . `also` autre part, toujours reutrn `Unit` .

Maintenant, pourquoi c'est utile, vous demandez? Parce que si vous appelez une méthode qui peut renvoyer `null` et que vous souhaitez exécuter du code uniquement lorsque cette valeur de retour n'est pas `null` vous pouvez utiliser `let` ou `also` comme ceci:

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

Ce morceau de code exécutera uniquement le bloc `let` lorsque `str` n'est pas `null` . Notez l'opérateur de sécurité `null` (`?`).

Utiliser apply pour initialiser des objets ou pour réaliser un chaînage de méthode

La documentation de `apply` indique ce qui suit:

> appelle le bloc fonction spécifié avec `this` valeur comme récepteur et renvoie `this` valeur.

Bien que le kdoc ne soit pas très utile, l' `apply` est en effet une fonction utile. Dans les termes de profane, `apply` définit une portée dans laquelle `this` est lié à l'objet que vous avez appelé `apply` . Cela vous permet d'économiser du code lorsque vous devez appeler plusieurs méthodes sur un objet que vous renverrez ensuite. Exemple:

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

C'est la même chose que d'écrire ceci:

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

Lire Expressions idiomatiques en ligne: https://riptutorial.com/fr/kotlin/topic/2273/expressions-idiomatiques

Chapitre 16: Gammes

Introduction

Les expressions de plage sont formées avec les fonctions rangeTo dont la forme opérateur est complétée par in et! In. La plage est définie pour tout type comparable, mais pour les types primitifs intégraux, elle a une implémentation optimisée.

Examples

Gammes de type intégral

Les plages de types intégraux (IntRange, LongRange, CharRange) ont une fonctionnalité supplémentaire: elles peuvent être itérées. Le compilateur prend soin de convertir ceci de manière analogue à Java indexé pour la boucle, sans surcharge supplémentaire

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

fonction downTo ()

si vous voulez parcourir les nombres dans l'ordre inverse? C'est simple. Vous pouvez utiliser la fonction downTo () définie dans la bibliothèque standard

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

fonction step ()

Est-il possible d'itérer sur des nombres avec pas arbitraire, pas égal à 1? Bien sûr, la fonction step () vous aidera

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

jusqu'à fonction

Pour créer une plage qui n'inclut pas son élément de fin, vous pouvez utiliser la fonction jusqu'à:

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

Lire Gammes en ligne: https://riptutorial.com/fr/kotlin/topic/10121/gammes

Chapitre 17: Génériques

Introduction

Une liste peut contenir des nombres, des mots ou n'importe quoi. C'est pourquoi nous appelons la liste *générique* .

Les génériques sont essentiellement utilisés pour définir les types pouvant être conservés par une classe et le type actuellement occupé par un objet.

Syntaxe

- class *ClassName* < ***TypeName*** >
- class *ClassName* <*>
- *ClassName* <dans ***UpperBound*** >
- *ClassName* <out ***LowerBound*** >
- *Nom* de classe ***<TypeName: UpperBound>***

Paramètres

Paramètre	Détails
TypeName	Type Nom du paramètre générique
UpperBound	Type de covariant
Borne inférieure	Type Contravariant
Nom du cours	Nom de la classe

Remarques

La limite supérieure implicite est Nullable

Dans Kotlin Generics, la limite supérieure du type de paramètre `T` serait `Any?` . Par conséquent pour cette classe:

```
class Consumer<T>
```

Le paramètre de type `T` est vraiment `T: Any?` . Pour rendre une limite supérieure non nullable, explicitement spécifique `T: Any` . Par exemple:

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

Examples

Variance du site de déclaration

La variance du site de déclaration peut être considérée comme une déclaration de la variance du site d'utilisation une fois pour tous les sites d'utilisation.

```
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
```

Les exemples les plus répandus de la variance des sites de déclaration sont `List<out T>` , qui est immuable de sorte que `T` n'apparaît que comme type de valeur de retour, et `Comparator<in T>` , qui ne reçoit que `T` en argument.

Variance du site d'utilisation

La variance du site d'utilisation est similaire aux caractères génériques Java:

Projection:

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

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

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

En projection:

```
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
```

Projection étoile

```
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
```

Voir également:

- Interopérabilité Variant Generics lors de l’appel de Kotlin à partir de Java.

Lire Génériques en ligne: https://riptutorial.com/fr/kotlin/topic/1147/generiques

Chapitre 18: Héritage de classe

Introduction

Tout langage de programmation orienté objet possède une forme d'héritage de classe. Laissez moi réviser:

Imaginez que vous deviez programmer une grappe de fruits: `Apples` , `Oranges` et `Pears` . Ils diffèrent tous par la taille, la forme et la couleur, c'est pourquoi nous avons différentes classes.

Mais disons que leurs différences ne comptent pas pour une seconde et que vous voulez juste un `Fruit` , peu importe exactement? Quel type de retour `getFruit()` aurait-il?

La réponse est classe `Fruit` . Nous créons une nouvelle classe et en faisons hériter tous les fruits!

Syntaxe

- ouvrir {Classe de base}
- class {Classe dérivée}: {Classe de base} ({Arguments d'initialisation})
- override {Définition de fonction}
- {DC-Object} est {Classe de base} == true

Paramètres

Paramètre	Détails
Classe de base	Classe héritée de
Classe dérivée	Classe qui hérite de la classe de base
Init Arguments	Arguments passés au constructeur de la classe de base
Définition de fonction	Fonction dans la classe dérivée dont le code est différent de celui de la classe de base
DC-Objet	”Objet de classe dérivé” Objet du type de la classe dérivée

Examples

Basics: le mot clé 'open'

Dans Kotlin, les classes sont **finales par défaut,** ce qui signifie qu'elles ne peuvent pas être héritées.

Pour autoriser l'héritage sur une classe, utilisez le mot clé `open` .

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

> **Remarque:** les classes abstraites, les classes scellées et les interfaces seront `open` par défaut.

Héritage de champs d'une classe

Définir la classe de base:

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

Définir la classe dérivée:

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

En utilisant la sous-classe:

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

Méthodes d'héritage d'une classe

Définir la classe de base:

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

Définir la classe dérivée:

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

Le Ninja a accès à toutes les méthodes en personne

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

Propriétés et méthodes dominantes

Propriétés de substitution (à la fois en lecture seule et mutable):

```
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
}
```

Méthodes primordiales:

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

object Titanic : Ship {

    var canSail = true

    override fun sail() {
        sink()
    }

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

Lire Héritage de classe en ligne: https://riptutorial.com/fr/kotlin/topic/5622/heritage-de-classe

Chapitre 19: Interfaces

Remarques

Voir aussi: documentation de référence Kotlin pour les interfaces: interfaces

Examples

Interface de base

Une interface Kotlin contient des déclarations de méthodes abstraites et des implémentations de méthodes par défaut bien qu'elles ne puissent pas stocker d'état.

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

Cette interface peut maintenant être implémentée par une classe comme suit:

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

Interface avec les implémentations par défaut

Une interface dans Kotlin peut avoir des implémentations par défaut pour les fonctions:

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

Les classes implémentant de telles interfaces pourront utiliser ces fonctions sans les réimplémenter

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

Propriétés

Les implémentations par défaut fonctionnent également pour les getters et les setters de

propriété:

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

Les implémentations d'interface ne peuvent pas utiliser les champs de sauvegarde

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

Implémentations multiples

Lorsque plusieurs interfaces implémentent la même fonction ou qu'elles définissent toutes avec une ou plusieurs implémentations, la classe dérivée doit résoudre manuellement les appels appropriés

```
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")
    }
}
```

Propriétés dans les interfaces

Vous pouvez déclarer des propriétés dans les interfaces. Une interface ne pouvant pas avoir d'état, vous ne pouvez déclarer qu'une propriété comme abstraite ou en fournissant une implémentation par défaut pour les accesseurs.

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

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

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

Conflits lors de l'implémentation de plusieurs interfaces avec des implémentations par défaut

Lors de l'implémentation de plusieurs interfaces ayant des méthodes du même nom incluant des implémentations par défaut, le compilateur est ambigu quant à l'implémentation à utiliser. En cas de conflit, le développeur doit remplacer la méthode en conflit et fournir une implémentation personnalisée. Cette implémentation peut choisir de déléguer ou non aux implémentations par défaut.

```
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 mot clé

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

```
}
```

Si la méthode de l'interface a sa propre implémentation par défaut, nous pouvons utiliser super mot-clé pour y accéder.

```
super.funcOne()
```

Lire Interfaces en ligne: https://riptutorial.com/fr/kotlin/topic/900/interfaces

Chapitre 20: Java 8 équivalents de flux

Introduction

Kotlin fournit de nombreuses méthodes d'extension sur les collections et les itérables pour appliquer des opérations de style fonctionnel. Un type de `Sequence` dédié permet la composition paresseuse de plusieurs de ces opérations.

Remarques

À propos de la paresse

Si vous voulez traiter une chaîne paresseuse, vous pouvez convertir en une `Sequence` utilisant `asSequence()` avant la chaîne. À la fin de la chaîne de fonctions, vous vous retrouvez généralement avec une `Sequence` . Ensuite, vous pouvez utiliser `toList()` , `toSet()` , `toMap()` ou une autre fonction pour matérialiser la `Sequence` à la fin.

```
// 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()
```

Pourquoi n'y a-t-il pas de types?

Vous remarquerez que les exemples Kotlin ne spécifient pas les types. Ceci est dû au fait que Kotlin a une inférence de type complète et est complètement de type sécurisé au moment de la compilation. Plus que Java, car il a également des types nullables et peut aider à empêcher le NPE redouté. Donc ceci à Kotlin:

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

est le même que:

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

Comme Kotlin sait ce que sont les `people` et que `people.age` est `Int` l'expression de filtre permet uniquement la comparaison avec un `Int` , et que `people.name` est une `String` conséquent, l'étape `map` génère une `List<String>` (Listonly `List` of `String`).

Maintenant, si les `people` étaient peut-être `null` , comme dans une `List<People>?` puis:

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

Retourne une `List<String>?` cela devrait être vérifié par la valeur null (*ou utiliser l'un des autres opérateurs Kotlin pour les valeurs nullables, voir cette méthode idiomatique de Kotlin pour traiter les valeurs nulles et également la manière idiomatique de gérer les listes nullables ou vides dans Kotlin*)

Réutiliser les flux

Dans Kotlin, cela dépend du type de collection si elle peut être consommée plus d'une fois. Une `Sequence` génère un nouvel itérateur à chaque fois, et à moins qu'elle n'utilise "une seule fois", elle peut être réinitialisée à chaque fois qu'elle est utilisée. Par conséquent, alors que le suivant échoue dans le flux Java 8, mais fonctionne dans 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
```

Et en Java pour obtenir le même comportement:

```
// 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
```

Par conséquent, dans Kotlin, le fournisseur des données décide s'il peut réinitialiser et fournir un nouvel itérateur ou non. Mais si vous souhaitez contraindre intentionnellement une `Sequence` à une itération unique, vous pouvez utiliser la fonction `constrainOnce()` pour la `Sequence` comme suit:

```
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.
```

Voir également:

- Référence API pour les fonctions d'extension pour Iterable
- Référence API pour les fonctions d'extension pour Array
- Référence API pour les fonctions d'extension pour List
- Référence API pour les fonctions d'extension à Map

Examples

Accumuler des noms dans une liste

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

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

Convertir des éléments en chaînes et les concaténer, séparés par des virgules

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

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

Calculer la somme des salaires des employés

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

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

Employés du groupe par département

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

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

Calculer la somme des salaires par département

```
// 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 }}
```

Partitionnez les élèves en passant et échouant

```
// 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 }
```

Noms de membres masculins

```
// 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 }
```

Noms de groupe des membres inscrits par sexe

```
// 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 } }
```

Filtrer une liste dans une autre liste

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

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

Trouver la plus courte chaîne d'une liste

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

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

Différents types de flux # 2 - utiliser paresseusement le premier élément s'il existe

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

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

Différents types de flux # 3 - itération d'une gamme d'entiers

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

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

Différents types de flux # 4 - itérer un tableau, mapper les valeurs, calculer la moyenne

```
// 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)
```

Différents types de flux # 5 - répétez lentement une liste de chaînes, mappez les valeurs, convertissez-les en Int, recherchez 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)
```

Différents types de flux # 6 - itération lente d'un flux d'Ints, mappage des valeurs, impression des résultats

```
// 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)
```

Différents types de flux # 7 - itération paresseuse Doubles, mapper à Int, mapper à String, imprimer chaque

```
// 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)
```

Comptage d'éléments dans une liste après application du filtre

```
// 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') }
```

Comment les flux fonctionnent - filtrer, majuscule, puis trier une liste

```
// 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)
```

Différents types de flux # 1 - désireux d'utiliser le premier élément s'il existe

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

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

ou, créez une fonction d'extension sur String appelée 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)
```

Voir aussi: fonction `apply()`

Voir aussi: Fonctions d'extension

Voir aussi: `?.` Opérateur d'appel sécurisé , et en nullité générale: http://stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-with-nullable-values-referencing-o/34498563 # 34498563

Recueillir l'exemple n ° 5 - trouver des personnes d'âge légal, sortir des chaînes formatées

```
// 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.
```

Et comme remarque secondaire, dans Kotlin, nous pouvons créer des classes de données simples et instancier les données de test comme suit:

```
// 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))
```

Recueillir l'exemple n ° 6 - regrouper les personnes par âge, imprimer l'âge et les noms ensemble

```
// 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 cas plus intéressant ici pour Kotlin. D'abord, les mauvaises réponses pour explorer les variantes de création d'une `Map` partir d'une collection / séquence:

```
// 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>
```

Et maintenant, pour la bonne réponse:

```
// 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!!
```

Nous devions simplement joindre les valeurs correspondantes pour `joinToString` les listes et fournir un transformateur à `joinToString` pour passer d'une instance `Person` au nom `Person.name` .

Recueillir l'exemple n ° 7a - Noms des cartes, joindre avec un délimiteur

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

Recueillir l'exemple n ° 7b - Recueillir avec 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)
```

Mais il est préférable de créer une fonction d'extension 2 pour faire correspondre les styles de 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)) }
```

Maintenant, vous avez deux façons d'utiliser les nouvelles fonctions de `summarizingInt` :

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

// or

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

Et tout cela produit les mêmes résultats. Nous pouvons également créer cette extension pour travailler sur la `Sequence` et pour les types primitifs appropriés.

Lire Java 8 équivalents de flux en ligne: https://riptutorial.com/fr/kotlin/topic/707/java-8-equivalents-de-flux

Chapitre 21: JUnit

Examples

Règles

Pour ajouter une règle JUnit à un montage de test:

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

L'annotation `@JvmField` est nécessaire pour exposer le champ de sauvegarde avec la même visibilité (public) que la propriété `myRule` (voir la réponse). Les règles JUnit exigent que le champ de règle annoté soit public.

Lire JUnit en ligne: https://riptutorial.com/fr/kotlin/topic/6973/junit

Chapitre 22: Kotlin Android Extensions

Introduction

Kotlin a une injection de vue intégrée pour Android, permettant de sauter la liaison manuelle ou de recourir à des frameworks tels que ButterKnife. Certains des avantages sont une syntaxe plus agréable, un meilleur typage statique et sont donc moins sujets aux erreurs.

Examples

Configuration

Commencez avec un projet de graduation correctement configuré .

Dans votre **projet local** (pas de niveau supérieur) `build.gradle` ajoutez la déclaration du plug-in d'extensions sous votre plug-in Kotlin, au niveau de l'indentation de niveau supérieur.

```
buildscript {
    ...
}

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

Utiliser des vues

En supposant que nous ayons une activité avec un exemple de disposition appelé `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>
```

Nous pouvons utiliser les extensions Kotlin pour appeler le bouton sans aucune liaison supplémentaire, comme ceci:

```
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!")
    }
}
```

Vous pouvez également importer tous les identifiants apparaissant dans la présentation avec une notation `*`

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

Les vues synthétiques ne peuvent être utilisées en dehors des activités / fragments / vues avec cette disposition gonflée:

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

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

Saveurs du produit

Les extensions Android fonctionnent également avec plusieurs arômes de produit Android. Par exemple, si nous avons des `build.gradle` dans `build.gradle` comme ceci:

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

Et par exemple, seule la saveur gratuite a un bouton d'achat:

```
<?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>
```

Nous pouvons nous lier à la saveur en particulier:

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

Un auditeur assidu pour obtenir un avis, lorsque la vue est complètement dessinée maintenant est tellement simple et génial avec l'extension de Kotlin

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

Sous la capuche

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

Lire Kotlin Android Extensions en ligne: https://riptutorial.com/fr/kotlin/topic/9474/kotlin-android-extensions

Chapitre 23: Kotlin Caveats

Examples

Appeler un toString () sur un type nullable

Une chose à surveiller lors de l'utilisation de la méthode `toString` dans Kotlin est la gestion de null en combinaison avec la `String?` .

Par exemple, vous souhaitez obtenir du texte depuis un `EditText` sous Android.

Vous auriez un morceau de code comme:

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

Vous vous attendriez à ce que si le champ n'existait pas, la valeur serait une chaîne vide mais dans ce cas, il s'agit de `"null"` .

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

Lire Kotlin Caveats en ligne: https://riptutorial.com/fr/kotlin/topic/6608/kotlin-caveats

Chapitre 24: Kotlin pour les développeurs Java

Introduction

La plupart des gens qui viennent à Kotlin ont une formation en programmation Java.

Cette rubrique rassemble des exemples comparant Java à Kotlin, en soulignant les différences les plus importantes et les joyaux proposés par Kotlin sur Java.

Examples

Déclaration des variables

Dans Kotlin, les déclarations de variables sont légèrement différentes de celles de Java:

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

- Ils commencent par `val` ou `var` , rendant la déclaration `final` (" **val** ue") ou **var** iable.
- Le type est noté après le nom, séparé par un `:`
- Grâce à l' *inférence de type* de Kotlin, la déclaration de type explicite peut être obéie s'il y a une affectation avec un type que le compilateur est capable de détecter sans ambiguïté

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

Faits rapides

- Kotlin n'a pas besoin `;` mettre fin aux déclarations
- Kotlin est **null-safe**
- Kotlin est **100% interopérable avec Java**
- Kotlin n'a **pas de primitives** (mais optimise ses homologues d'objet pour la JVM, si possible)
- Les classes Kotlin ont des **propriétés, pas des champs**
- Kotlin propose **des classes de données** avec des méthodes `equals` / `hashCode` et des accesseurs de champs générés automatiquement
- Kotlin a uniquement des exceptions d'exécution, **aucune exception vérifiée**
- Kotlin n'a **pas de `new` mot clé** . La création d'objets se fait simplement en appelant le constructeur comme toute autre méthode.

- Kotlin prend en charge la **surcharge** (limitée) de l' **opérateur** . Par exemple, l'accès à une valeur de carte peut être écrit comme `val a = someMap["key"]` : `val a = someMap["key"]`
- Kotlin peut non seulement être compilé en octets pour la machine virtuelle **Java** , mais aussi en **Java Script** , ce qui vous permet d'écrire à la fois du code backend et du code frontal dans Kotlin.
- Kotlin est **entièrement compatible avec Java 6** , ce qui est particulièrement intéressant en ce qui concerne le support des anciens appareils Android
- Kotlin est une langue **officiellement prise en charge pour le développement Android**
- Les collections de Kotlin présentent une distinction intrinsèque entre les collections **mutables et immuables** .
- Kotlin soutient **Coroutines** (expérimental)

Égalité et identité

Kotlin utilise `==` pour l'égalité (c'est-à-dire que les appels sont `equals` interne) et `===` pour l'identité référentielle.

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

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

SI, TRY et d'autres sont des expressions, pas des déclarations

Dans Kotlin, `if` , `try` et d'autres sont des expressions (elles renvoient donc une valeur) plutôt que des instructions (vides).

Ainsi, par exemple, Kotlin n'a pas l' *opérateur Elvis* ternaire de Java, mais vous pouvez écrire quelque chose comme ceci:

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

L' *expression* `try` est encore moins familière, mais tout aussi *expressive* :

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

Lire Kotlin pour les développeurs Java en ligne: https://riptutorial.com/fr/kotlin/topic/10099/kotlin-pour-les-developpeurs-java

Chapitre 25: Lambdas de base

Syntaxe

- Paramètres explicites:
- {parameterName: ParameterType, otherParameterName: OtherParameterType -> anExpression ()}
- Paramètres inférés:
- val addition: (Int, Int) -> Int = {a, b -> a + b}
- Paramètre unique `it` raccourci
- val square: (Int) -> Int = {it * it}
- Signature:
- () -> ResultType
- (InputType) -> ResultType
- (InputType1, InputType2) -> ResultType

Remarques

Les paramètres de type d'entrée peuvent être omis lorsqu'ils peuvent être omis lorsqu'ils peuvent être déduits du contexte. Par exemple, disons que vous avez une fonction sur une classe qui a une fonction:

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

Vous pouvez utiliser cette fonction en passant un lambda, et comme les paramètres sont déjà spécifiés dans la signature de la fonction, il n'est pas nécessaire de les déclarer de nouveau dans l'expression lambda:

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

Cela s'applique également lorsque vous affectez un lambda à une variable:

```
//valid:
```

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

Lorsque le lambda prend un paramètre et que le type peut être déduit du contexte, vous pouvez vous référer au paramètre par `it` .

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

Examples

Lambda comme paramètre pour filtrer la fonction

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

Lambda passé en tant que variable

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

Lambda pour évaluer un appel de fonction

Chronomètre polyvalent pour déterminer la durée d'exécution d'une fonction:

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

Usage:

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

Lire Lambdas de base en ligne: https://riptutorial.com/fr/kotlin/topic/5878/lambdas-de-base

Chapitre 26: Les bases de Kotlin

Introduction

Ce sujet couvre les bases de Kotlin pour les débutants.

Remarques

1. Le fichier Kotlin a une extension .kt.
2. Toutes les classes de Kotlin ont une super-classe commune Any, qui est un super par défaut pour une classe sans supertypes déclarés (similaire à Object en Java).
3. Les variables peuvent être déclarées comme val (immutable- assign une fois) ou var (mutable-value peut être modifié)
4. Le point-virgule n'est pas nécessaire à la fin de l'instruction.
5. Si une fonction ne renvoie aucune valeur utile, son type de retour est Unit.Il est également facultatif. 6. L'égalité référentielle est vérifiée par l'opération ===. un === b est évalué à true si et seulement si a et b pointent vers le même objet.

Examples

Exemples de base

1.La déclaration de type de retour d'unité est facultative pour les fonctions. Les codes suivants sont équivalents.

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

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

2. Fonctions à expression unique: lorsqu'une fonction renvoie une expression unique, les accolades peuvent être omises et le corps est spécifié après = symbole

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

La déclaration explicite du type de retour est facultative lorsque cela peut être déduit par le compilateur

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

3.Interpolation de chaîne: L'utilisation de valeurs de chaîne est facile.

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

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

4.Dans Kotlin, le système de types distingue les références pouvant contenir des valeurs NULL (références nullables) et celles qui ne le peuvent pas (références non NULL). Par exemple, une variable régulière de type String ne peut pas contenir la valeur null:

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

Pour autoriser les valeurs NULL, nous pouvons déclarer une variable en tant que chaîne nullable, écrite String?

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

5.Dans Kotlin, == vérifie réellement l'égalité des valeurs.Par convention, une expression comme == b est traduite en

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

Lire Les bases de Kotlin en ligne: https://riptutorial.com/fr/kotlin/topic/10648/les-bases-de-kotlin

Chapitre 27: Les fonctions

Syntaxe

- fun ***Nom*** (***Params***) = ...
- ***Nom*** fun (***Params***) {...}
- fun ***Nom*** (***Params***): ***Type*** {...}
- fun < ***Type Argument*** > ***Name*** (***Params***): ***Type*** {...}
- inline fun ***Name*** (***Params***): ***tapez*** {...}
- { ***ArgName*** : ***ArgType*** -> ...}
- { ***ArgName*** -> ...}
- { ***ArgNames*** -> ...}
- {(***ArgName*** : ***ArgType***): ***Type*** -> ...}

Paramètres

Paramètre	Détails
prénom	Nom de la fonction
Params	Valeurs données à la fonction avec un nom et un type: ***`Name : Type`***
Type	Type de retour de la fonction
Type Argument	Paramètre de type utilisé dans la programmation générique (pas nécessairement le type de retour)
ArgName	Nom de la valeur donnée à la fonction
ArgType	***Spécificateur de*** type pour ***ArgName***
ArgNames	Liste de ArgName séparés par des virgules

Examples

Fonctions prenant d'autres fonctions

Comme on le voit dans "Fonctions Lambda", les fonctions peuvent prendre d'autres fonctions en paramètre. Le "type de fonction" dont vous aurez besoin pour déclarer des fonctions prenant d'autres fonctions est le suivant:

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

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

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

Par exemple, vous pouvez utiliser le type le plus vague, `() -> Any?` , pour déclarer une fonction qui exécute une fonction lambda deux fois:

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

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

Fonctions Lambda

Les fonctions Lambda sont des fonctions anonymes qui sont généralement créées lors d'un appel de fonction pour agir en tant que paramètre de fonction. Ils sont déclarés par des expressions environnantes avec {accolades} - si des arguments sont nécessaires, ils sont mis avant une flèche `->` .

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

La dernière instruction dans une fonction lambda est automatiquement la valeur de retour.

Les types sont optionnels, si vous placez le lambda sur un emplacement où le compilateur peut en déduire les types.

Plusieurs arguments:

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

Si la fonction lambda ne nécessite qu'un seul argument, la liste des arguments peut être omise et l'argument unique peut être utilisé `it` place.

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

Si le seul argument d'une fonction est une fonction lambda, les parenthèses peuvent être complètement omises de l'appel de fonction.

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

Références de fonction

Nous pouvons référencer une fonction sans l'appeler en préfixant le nom de la fonction avec `::`. Cela peut ensuite être passé à une fonction qui accepte une autre fonction en tant que paramètre.

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

Les fonctions sans récepteur seront converties en `(ParamTypeA, ParamTypeB, ...) -> ReturnType` où `ParamTypeA` , `ParamTypeB` ... sont le type des paramètres de la fonction et `ReturnType1 est le type de la valeur de retour de la fonction.

```
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
```

Fonctions avec un récepteur (que ce soit une fonction d'extension ou une fonction membre) a une syntaxe différente. Vous devez ajouter le nom de type du récepteur avant les deux points suivants:

```
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.
```

Cependant, lorsque le récepteur d'une fonction est un objet, le récepteur est omis de la liste de paramètres, car il s'agit et n'est qu'une instance de ce type.

```
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.
```

Depuis kotlin 1.1, la référence de fonction peut également être *liée* à une variable, appelée alors *référence de fonction bornée* .

1.1.0

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

Notez que cet exemple est donné uniquement pour montrer comment fonctionne la référence de fonction bornée. C'est une mauvaise pratique dans tous les autres sens.

Il y a cependant un cas particulier. Une fonction d'extension déclarée en tant que membre ne peut pas être référencée.

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

Les fonctions de base

Les fonctions sont déclarées à l'aide du mot-clé `fun` , suivi d'un nom de fonction et de tous les paramètres. Vous pouvez également spécifier le type de retour d'une fonction, par défaut `Unit` . Le corps de la fonction est entre accolades `{}` . Si le type de retour est autre que `Unit` , le corps doit émettre une déclaration de retour pour chaque branche de terminaison dans le corps.

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

Une version abrégée du même:

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

Et le type peut être omis car il peut être déduit:

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

Fonctions abrégées

Si une fonction ne contient qu'une seule expression, nous pouvons omettre les accolades et utiliser plutôt une équation comme une affectation de variable. Le résultat de l'expression est renvoyé automatiquement.

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

Fonctions en ligne

Les fonctions peuvent être déclarées en ligne en utilisant le préfixe `inline` , et dans ce cas elles agissent comme des macros dans C - plutôt que d'être appelées, elles sont remplacées par le code du corps de la fonction au moment de la compilation. Cela peut entraîner des avantages en termes de performances dans certaines circonstances, principalement lorsque les lambdas sont utilisés comme paramètres de fonction.

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

Une différence avec les macros C est que les fonctions en ligne ne peuvent pas accéder à la portée à partir de laquelle elles sont appelées:

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

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

Fonctions opérateur

Kotlin nous permet de fournir des implémentations pour un ensemble prédéfini d'opérateurs avec une représentation symbolique fixe (comme `+` ou `*`) et une priorité fixe. Pour implémenter un opérateur, nous fournissons une fonction membre ou une fonction d'extension avec un nom fixe, pour le type correspondant. Les fonctions qui surchargent les opérateurs doivent être marquées avec le modificateur d' `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
```

Vous trouverez plus de fonctions opérateur ici

Lire Les fonctions en ligne: https://riptutorial.com/fr/kotlin/topic/1280/les-fonctions

Chapitre 28: Méthodes d'extension

Syntaxe

- fun TypeName.extensionName (params, ...) {/ * body * /} // Déclaration
- fun <T: Any> TypeNameWithGenerics <T> .extensionName (params, ...) {/ * body * /} // Déclaration avec des génériques
- myObj.extensionName (args, ...) // invocation

Remarques

Les extensions sont résolues de **manière statique** . Cela signifie que la méthode d'extension à utiliser est déterminée par le type de référence de la variable à laquelle vous accédez; Peu importe le type de la variable à l'exécution, la même méthode d'extension sera toujours appelée. En effet, la **déclaration d'une méthode d'extension n'ajoute pas réellement un membre au type de récepteur** .

Examples

Extensions de niveau supérieur

Les méthodes d'extension de niveau supérieur ne sont pas contenues dans une classe.

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

Au-dessus d'une méthode d'extension est défini pour le type `IntArray` . Notez que l'objet pour lequel la méthode d'extension est définie (appelé le **récepteur**) est accessible à l'aide du mot `this` clé `this` .

Cette extension peut être appelée comme ceci:

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

Piège potentiel: les extensions sont résolues de manière statique

La méthode d'extension à appeler est déterminée au moment de la compilation en fonction du type de référence de la variable à laquelle on accède. Quel que soit le type de la variable à l'exécution, la même méthode d'extension sera toujours appelée.

```
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'exemple ci-dessus affichera `"Defined for Super"` , car le type déclaré de la variable `myVar` est `Super` .

Echantillon s'étendant longtemps pour rendre une chaîne lisible par l'homme

Étant donné toute valeur de type `Int` ou `Long` pour rendre une chaîne lisible par un humain:

```
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()
}
```

Puis facilement utilisé comme:

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

Exemple d'extension de la classe Java 7+ Path

Un cas d'utilisation courant des méthodes d'extension consiste à améliorer une API existante. Voici des exemples d'ajout de `exist` , `notExists` et `deleteRecursively` à la classe Java 7+ `Path` :

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

Qui peut maintenant être invoqué dans cet exemple:

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

Utilisation des fonctions d'extension pour améliorer la lisibilité

Dans Kotlin, vous pouvez écrire du code comme:

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

Mais l'utilisation de `apply` n'est pas si claire quant à votre intention. Parfois, il est plus clair de créer une fonction d'extension similaire pour renommer l'action et la rendre plus évidente. Cela ne devrait pas être autorisé à devenir incontrôlable, mais pour des actions très courantes telles que la vérification:

```
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
}
```

Vous pouvez maintenant écrire le code en tant que:

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

Ce qui permet maintenant aux gens de savoir à quoi s’attendre dans le paramètre lambda.

Notez que le paramètre de type `T` pour `verifiedBy` est le même que `T: Any?` ce qui signifie que même les types nullables pourront utiliser cette version de l'extension. Bien que `verifiedWith` nécessite non-nullable.

Exemple d'extension de classes temporelles Java 8 pour le rendu d'une chaîne au format ISO

Avec cette déclaration:

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

Vous pouvez maintenant simplement:

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

Fonctions d'extension aux objets compagnons (apparition de fonctions statiques)

Si vous souhaitez étendre une classe, si vous êtes une fonction statique, par exemple pour la

classe `Something` fonction statique ajouter à la recherche `fromString` , cela ne peut fonctionner que si la classe a un objet compagnon et que la fonction d'extension a été déclarée sur l'objet compagnon :

```
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
}
```

Solution de contournement de propriété d'extension paresseuse

Supposons que vous souhaitiez créer une propriété d'extension coûteuse à calculer. Vous souhaitez donc mettre le calcul en cache en utilisant le délégué de propriété paresseux et faire référence à l'instance actuelle (`this`), mais vous ne pouvez pas le faire, comme expliqué dans les problèmes Kotlin KT-9686 et KT-13053 . Cependant, il existe une solution de contournement officielle fournie ici .

Dans l'exemple, la propriété d'extension est `color` . Il utilise un `colorCache` explicite qui peut être utilisé avec `this` car aucun `lazy` n'est nécessaire:

```
class KColor(val value: Int)

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

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

Extensions pour une référence plus facile Voir du code

Vous pouvez utiliser des extensions pour la vue de référence, plus de passe-partout après avoir créé les vues.

Idée originale de la bibliothèque Anko

Les extensions

```
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
```

Usage

```
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) }
```

Lire Méthodes d'extension en ligne: https://riptutorial.com/fr/kotlin/topic/613/methodes-d-extension

Chapitre 29: Modificateurs de visibilité

Introduction

Dans Kotlin, il existe 4 types de modificateurs de visibilité disponibles.

Public: Vous pouvez y accéder de n'importe où.

Privé: accessible uniquement à partir du code du module.

Protected: accessible uniquement à partir de la classe le définissant et des classes dérivées.

Interne: accessible uniquement à partir de la portée de la classe le définissant.

Syntaxe

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

Examples

Exemple de code

Public: `public val name = "Avijit"`

Privé: `private val name = "Avijit"`

Protégé: `protected val name = "Avijit"`

Interne: `internal val name = "Avijit"`

Lire Modificateurs de visibilité en ligne: https://riptutorial.com/fr/kotlin/topic/10019/modificateurs-de-visibilite

Chapitre 30: Objets singleton

Introduction

Un *objet* est un type spécial de classe, qui peut être déclaré en utilisant le mot-clé `object` . Les objets sont similaires à Singletons (un motif de conception) en Java. Il fonctionne également comme la partie statique de Java. Les débutants qui passent de Java à Kotlin peuvent largement utiliser cette fonctionnalité, à la place de statique, ou de singletons.

Examples

Utiliser comme repalcement des méthodes statiques / champs de java

```
object CommonUtils {

    var anyname: String ="Hello"

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

De toute autre classe, appelez simplement la variable et fonctionne de la manière suivante:

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

Utiliser comme un singleton

Les objets Kotlin ne sont en réalité que des singletons. Son principal avantage est que vous n'avez pas besoin d'utiliser `SomeSingleton.INSTANCE` pour obtenir l'instance du singleton.

En Java, votre singleton ressemble à ceci:

```
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");
}
```

Dans kotlin, le code équivalent est

```
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")
}
```

C'est évidemment moins verbeux à utiliser.

Lire Objets singleton en ligne: https://riptutorial.com/fr/kotlin/topic/10152/objets-singleton

Chapitre 31: Paramètres Vararg dans les fonctions

Syntaxe

- **Vararg Mot clé** : `vararg` est utilisé dans une déclaration de méthode pour indiquer qu'un nombre variable de paramètres sera accepté.
- **Spread Operator** : Un astérisque (`*`) avant un tableau utilisé dans les appels de fonction pour "déployer" le contenu dans des paramètres individuels.

Examples

Notions de base: Utilisation du mot-clé vararg

Définissez la fonction à l'aide du mot clé `vararg` .

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

Vous pouvez maintenant transmettre autant de paramètres (du type correct) à la fonction que vous le souhaitez.

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

> **Notes:** Les paramètres Vararg *doivent* être le dernier paramètre de la liste de paramètres.

Spread Operator: passer des tableaux dans des fonctions vararg

Les tableaux peuvent être transmis aux fonctions vararg à l'aide de l' **opérateur Spread** , `*` .

En supposant que la fonction suivante existe ...

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

Vous pouvez **passer un tableau** dans la fonction comme ça ...

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

```
printNumbers(*numbers)

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

L'opérateur spread peut également être utilisé **au milieu** des paramètres ...

```
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)
```

Lire Paramètres Vararg dans les fonctions en ligne:
https://riptutorial.com/fr/kotlin/topic/5835/parametres-vararg-dans-les-fonctions

Chapitre 32: Propriétés déléguées

Introduction

Kotlin peut déléguer l'implémentation d'une propriété à un objet gestionnaire. Certains gestionnaires standard sont inclus, tels que l'initialisation différée ou les propriétés observables. Des gestionnaires personnalisés peuvent également être créés.

Examples

Initialisation paresseuse

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

L'exemple imprime `2` .

Propriétés observables

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

L'exemple des impressions `foo was changed from 1 to 2`

Propriétés sauvegardées sur la carte

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

L'exemple imprime `1`

Délégation personnalisée

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

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

L'exemple imprimé `Delegated value`

Délégué Peut être utilisé comme couche pour réduire le passe-partout

Considérons le système de type nul de Kotlin et `WeakReference<T>` .

Donc, supposons que nous devions sauvegarder une sorte de référence et que nous voulions éviter les fuites de mémoire, voici où `WeakReference` entre en jeu.

Prenons par exemple ceci:

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

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

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

Maintenant, c'est juste avec un WeakReference. Pour réduire ce niveau, nous pouvons utiliser un délégué de propriété personnalisé pour nous aider ainsi:

```
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)
    }
}
```

Donc maintenant, nous pouvons utiliser des variables qui sont enveloppées avec `WeakReference` tout comme les variables `WeakReference` normales!

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

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

    init {
        reference = this
    }
```

```
}
```

Lire Propriétés déléguées en ligne: https://riptutorial.com/fr/kotlin/topic/10571/proprietes-deleguees

Chapitre 33: RecyclerView à Kotlin

Introduction

Je veux juste partager mes connaissances et mon code de RecyclerView en utilisant Kotlin.

Examples

Classe principale et adaptateur

Je suppose que vous avez connaissance de la syntaxe de **Kotlin** et de son utilisation, ajoutez simplement **RecyclerView** dans le fichier **activity_main.xml** et définissez-le avec la classe d'adaptateur.

```
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
        }
    }
```

celui-ci est votre classe d' **adaptateur de** vue de recycleur et crée le fichier **main_item.xml**

```
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)
            }
        }
    }
}
```

Lire RecyclerView à Kotlin en ligne: https://riptutorial.com/fr/kotlin/topic/10143/recyclerview-a-kotlin

Chapitre 34: Réflexion

Introduction

La réflexion est la capacité d'un langage à inspecter le code à l'exécution au lieu de la compilation.

Remarques

Reflection est un mécanisme d'introspection des constructions de langage (classes et fonctions) au moment de l'exécution.

Lors du ciblage de la plate-forme JVM, les fonctionnalités de réflexion à l'exécution sont distribuées dans un `kotlin-reflect.jar` JAR distinct: `kotlin-reflect.jar` . Ceci est fait pour réduire la taille de l'exécution, réduire les fonctionnalités inutilisées et permettre de cibler d'autres plateformes (comme JS).

Examples

Référencement d'une classe

Pour obtenir une référence à un objet `KClass` représentant des doubles `KClass` utilisation de classe:

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

Référencement d'une fonction

Les fonctions sont des citoyens de première classe à Kotlin. Vous pouvez obtenir une référence à l'aide de deux points et la transmettre à une autre fonction:

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

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

Interaction avec la réflexion Java

Pour obtenir un objet de `Class` Java à partir du `KClass` de Kotlin, utilisez la propriété d'extension `.java` :

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

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

Le dernier exemple sera optimisé par le compilateur pour ne pas allouer une instance `KClass` intermédiaire.

Obtenir des valeurs de toutes les propriétés d'une classe

Étant donné la classe `Example` qui `BaseExample` classe `BaseExample` avec certaines propriétés:

```
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"
}
```

On peut s'emparer de toutes les propriétés d'une classe:

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

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

L'exécution de ce code entraînera la levée d'une exception. La propriété `private val privateField` est déclarée privée et l'appel de `member.get(example)` ne réussira pas. Une façon de gérer cela pour filtrer les propriétés privées. Pour ce faire, nous devons vérifier le modificateur de visibilité du getter Java d'une propriété. En cas de valeur `private val` le getter n'existe pas, nous pouvons donc supposer un accès privé.

La fonction d'assistance et son utilisation peuvent ressembler à ceci:

```
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)}")
}
```

Une autre approche consiste à rendre les propriétés privées accessibles en utilisant la réflexion:

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

Définition des valeurs de toutes les propriétés d'une classe

A titre d'exemple, nous voulons définir toutes les propriétés de chaîne d'une classe d'échantillon

```
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
        }
    }
}
```

Obtenir des propriétés modifiables s'appuie sur l'obtention de toutes les propriétés, en filtrant les propriétés mutables par type. Nous devons également vérifier la visibilité, car la lecture des propriétés privées entraîne une exception d'exécution.

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

Pour définir toutes les propriétés de `String` sur `"Our Value"` nous pouvons également filtrer par le type de retour. Étant donné que Kotlin est basé sur la machine virtuelle Java, Type Erasure est activé et les propriétés renvoyant des types génériques tels que `List<String>` seront donc identiques à `List<Any>` . Malheureusement, la réflexion n'est pas une solution miracle et il n'y a pas de moyen sensé d'éviter cela. Vous devez donc faire attention à vos cas d'utilisation.

```
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")
        }
```

Lire Réflexion en ligne: https://riptutorial.com/fr/kotlin/topic/2402/reflexion

Chapitre 35: Regex

Examples

Idioms pour Regex correspondant à quand expression

Utiliser des locaux immuables:

Utilise moins d'espace horizontal mais plus d'espace vertical que le modèle "anonymous temporaries". Préférable sur le modèle "anonymous temporaries" si l'expression `when` est dans une boucle - dans ce cas, les définitions regex doivent être placées en dehors de la boucle.

```
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 */
}
```

Utiliser des temporaires anonymes:

Utilise moins d'espace vertical mais plus d'espace que le modèle "locaux immuables". Ne doit pas être utilisé si alors `when` expression est en boucle.

```
import kotlin.text.regex

var string = /* some string */

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

En utilisant le modèle de visiteur:

A l'avantage de reproduire étroitement le "argument-ful" `when` syntaxe. Cela est bénéfique car il indique plus clairement l'argument de l'expression `when` , et exclut également certaines erreurs de programmeur qui pourraient découler de la répétition de l'argument `when` dans chaque `whenEntry` .

Le modèle "immuable locals" ou "anonymous temporaries" peut être utilisé avec cette implémentation du pattern visiteur.

```
import kotlin.text.regex

var string = /* some string */

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

Et la définition minimale de la classe wrapper pour l'argument expression `when` :

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

Introduction aux expressions régulières dans Kotlin

Cet article montre comment utiliser la plupart des fonctions de la classe `Regex` , travailler avec des valeurs null liées aux fonctions `Regex` , et comment les chaînes brutes facilitent l'écriture et la lecture des modèles regex.

La classe RegEx

Pour travailler avec des expressions régulières dans Kotlin, vous devez utiliser la classe `Regex(pattern: String)` et appeler des fonctions telles que `find(..)` ou `replace(..)` sur cet objet regex.

Un exemple d'utilisation de la classe `Regex` qui renvoie true si la chaîne d' `input` contient c ou d:

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

La chose essentielle à comprendre avec toutes les fonctions `Regex` est que le résultat est basé sur la correspondance du `pattern` expression régulière et de la chaîne d' `input` . Certaines fonctions nécessitent une correspondance complète, tandis que le reste ne nécessite qu'une correspondance partielle. La fonction `containsMatchIn(..)` utilisée dans l'exemple nécessite une correspondance partielle et est expliquée plus loin dans cette publication.

Sécurité nulle avec des expressions régulières

Les deux `find(..)` et `matchEntire(..) MatchResult?` un `MatchResult?` objet. Le `?` caractère après `MatchResult` est nécessaire pour Kotlin pour gérer NULL en toute sécurité .

Un exemple qui montre comment Kotlin gère NULL en toute sécurité depuis une fonction `Regex` , lorsque la fonction `find(..)` renvoie 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. => ""
```

Avec la fonction `orEmpty()` , `b` ne peut pas être nul et le `?` le caractère est inutile lorsque vous appelez des fonctions sur `b` .

Si vous ne vous souciez pas de cette manipulation sécurisée des valeurs NULL, Kotlin vous permet de travailler avec des valeurs NULL comme en Java avec le `!!` personnages:

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

Chaînes brutes dans les modèles d'expressions rationnelles

Kotlin fournit une amélioration par rapport à Java avec une chaîne brute qui permet d'écrire des modèles de regex purs sans double barre oblique inverse, nécessaires avec une chaîne Java. Une chaîne brute est représentée avec un triple devis:

```
"""\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 chaîne d' `input` sera comparée au `pattern` dans l'objet `Regex` . Il retourne un `Matchresult?` objet avec le premier texte correspondant après le `startIndex` , ou `null` si le motif ne correspond pas à la chaîne d' `input` . La chaîne de résultat est extraite de `MatchResult?` propriété `value` objet. Le paramètre `startIndex` est facultatif avec la valeur par défaut 0.

Pour extraire le premier numéro de téléphone valide d'une chaîne avec ses coordonnées:

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

En l'absence de numéro de téléphone valide dans la chaîne d' `input` , la variable `phoneNumber` sera `null` .

findAll (entrée: CharSequence, startIndex: Int): séquence

Renvoie toutes les correspondances de la chaîne d' `input` correspondant au `pattern` regex.

Pour imprimer tous les nombres séparés par un espace, à partir d'un texte avec des lettres et des chiffres:

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

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

La variable `matchedResults` est une séquence `MatchResult` objets `MatchResult` . Avec une chaîne d' `input` sans chiffres, la fonction `findAll(..)` renvoie une séquence vide.

matchEntire (entrée: CharSequence): MatchResult?

Si tous les caractères de la chaîne d' `input` correspondent au `pattern` expression régulière, une chaîne égale à l' `input` sera renvoyée. Sinon, `null` sera retourné.

Renvoie la chaîne d'entrée si la chaîne d'entrée entière est un nombre:

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

correspond à (input: CharSequence): booléen

Renvoie true si la chaîne d'entrée complète correspond au modèle de regex. Faux sinon.

Teste si deux chaînes ne contiennent que des chiffres:

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

containsMatchIn (input: CharSequence): Booléen

Renvoie true si une partie de la chaîne d'entrée correspond au modèle de regex. Faux sinon.

Testez si deux chaînes contiennent au moins un chiffre:

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

split (input: CharSequence, limite: Int): Liste

Renvoie une nouvelle liste sans toutes les correspondances regex.

Pour retourner des listes sans chiffres:

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

Il y a un élément dans la liste pour chaque division. La première chaîne d' `input` comporte trois chiffres. Cela se traduit par une liste de trois éléments.

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

Remplace toutes les correspondances du `pattern` regex dans la chaîne d' `input` par la chaîne de remplacement.

Pour remplacer tous les chiffres d'une chaîne par un x:

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

Lire Regex en ligne: https://riptutorial.com/fr/kotlin/topic/8364/regex

Chapitre 36: se connecter à kotlin

Remarques

Question connexe: Manière idiomatique de se connecter à Kotlin

Examples

kotlin.logging

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

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

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

Utiliser le framework kotlin.logging

Lire se connecter à kotlin en ligne: https://riptutorial.com/fr/kotlin/topic/3258/se-connecter-a-kotlin

Chapitre 37: Sécurité nulle

Examples

Types nullables et non nullables

Les types normaux, tels que `String` , ne sont pas nullables. Pour les rendre capables de contenir des valeurs nulles, vous devez explicitement le noter en mettant un `?` derrière eux: `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!
```

Opérateur d'appel sécurisé

Pour accéder aux fonctions et propriétés des types nullables, vous devez utiliser des opérateurs spéciaux.

Le premier, `?.` , vous donne la propriété ou la fonction à laquelle vous essayez d'accéder ou vous donne la valeur NULL si l'objet est nul:

```
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.
```

Idiom: appel de plusieurs méthodes sur le même objet vérifié par un objet nul

Une façon élégante d'appeler plusieurs méthodes d'un objet contrôlé nul utilise de Kotlin `apply` comme ceci:

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

Cela appellera `foo` et `bar` sur `obj` (qui est `this` dans le `apply` bloc) que si `obj` est non nul, sauter le bloc entier autrement.

Pour amener une variable nullable dans la portée en tant que référence non nullable sans en faire le récepteur implicite des appels de fonctions et de propriétés, vous pouvez utiliser `let` au lieu d' `apply` :

```
nullable?.let { notnull ->
```

```
    notnull.foo()
    notnull.bar()
}
```

`notnull` pourrait être nommé quoi que ce soit, ou même à l' écart et utilisé par le paramètre implicite lambda `it` .

Smart cast

Si le compilateur peut déduire qu'un objet ne peut pas être nul à un certain point, vous n'avez plus besoin d'utiliser les opérateurs spéciaux:

```
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!
}
```

> **Remarque:** Le compilateur ne vous autorisera pas à convertir intelligemment les variables pouvant être modifiées entre la vérification null et l'utilisation prévue.
>
> Si une variable est accessible en dehors de la portée du bloc actuel (parce que ce sont des membres d'un objet non local, par exemple), vous devez créer une nouvelle référence locale que vous pouvez ensuite convertir et utiliser intelligemment.

Elimine les valeurs NULL d'une Iterable et d'un tableau

Parfois, nous devons changer le type de `Collection<T?>` `Collections<T>` . Dans ce cas, `filterNotNull` est notre solution.

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

Coalescence Nul / Opérateur Elvis

Parfois, il est souhaitable d'évaluer une expression nullable d'une manière sinon. L'opérateur elvis, `?:` , Peut être utilisé dans Kotlin pour une telle situation.

Par exemple:

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

L'expression ci-dessus renvoie `"Nothing here"` si `data?.first()` ou `data` elle-même donne une valeur `null` else le résultat de `data?.first()` .

Il est également possible de lancer des exceptions en utilisant la même syntaxe pour annuler l'exécution du code.

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

> Rappel: NullPointerExceptions peut être lancé avec l' opérateur d'assertion (par exemple, les `data!!.second()!!`)

Affirmation

`!!` les suffixes ignorent la nullité et retournent une version non nulle de ce type. `KotlinNullPointerException` sera levée si l'objet est un `null` .

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

Elvis Operator (? :)

Dans Kotlin, nous pouvons déclarer une variable pouvant contenir une `null reference` . Supposons que nous ayons une référence nullable `a` , nous pouvons dire "si `a` n'est pas nul, utilisez-le, sinon utilisez une valeur non nulle `x` "

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

Maintenant, `a` peut être nul. Ainsi, lorsque nous devons accéder à la valeur de `a` , nous devons effectuer une vérification de sécurité, qu'elle contienne ou non de la valeur. Nous pouvons effectuer cette vérification de sécurité par une instruction `if...else` classique.

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

Mais voici opérateur avance `Elvis` (opérateur Elvis: `?:`). Ci `if...else` dessus `if...else` peut être exprimé avec l'opérateur d'Elvis comme ci-dessous:

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

Si l'expression à gauche de `?:` (Ici: `a?.length`) n'est pas nulle, l'opérateur elvis le renvoie, sinon il renvoie l'expression à droite (ici: `-1`). L'expression côté droit est évaluée uniquement si le côté gauche est nul.

Lire Sécurité nulle en ligne: https://riptutorial.com/fr/kotlin/topic/2080/securite-nulle

Chapitre 38: Tableaux

Examples

Tableaux génériques

Les tableaux génériques de Kotlin sont représentés par `Array<T>` .

Pour créer un tableau vide, utilisez la fonction de fabrique `emptyArray<T>()` :

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

Pour créer un tableau avec une taille et des valeurs initiales données, utilisez le constructeur:

```
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
```

Les tableaux ont `get(index: Int): T` et `set(index: Int, value: T)` fonctions:

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

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

Tableaux de primitifs

Ces types n'héritent **pas** de `Array<T>` pour éviter la boxe, cependant, ils ont les mêmes attributs et méthodes.

Type Kotlin	Fonction d'usine	Type 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[]`

Les extensions

`average()` est défini pour `Byte` , `Int` , `Long` , `Short` , `Double` , `Float` et renvoie toujours le `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()` renvoie un élément du tableau

`getOrNull(index: Int)` renvoie null si l'index est hors limites, sinon un élément du tableau

`first()` , `last()`

`toHashSet()` renvoie un `HashSet<T>` de tous les éléments

`sortedArray()` , `sortedArrayDescending()` crée et retourne un nouveau tableau avec des éléments triés de courant

`sort()` , `sortDescending` trie le tableau sur place

`min()` , `max()`

Itérer Array

Vous pouvez imprimer les éléments du tableau en utilisant la même boucle que la boucle améliorée Java, mais vous devez changer le mot clé de `:` en `in` .

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

Vous pouvez également modifier le type de données en boucle.

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

Créer un tableau

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

Créer un tableau en utilisant une fermeture

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

Créer un tableau non initialisé

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

Le tableau renvoyé aura toujours un type nullable. Les tableaux d'éléments non nullables ne peuvent pas être créés non initialisés.

Lire Tableaux en ligne: https://riptutorial.com/fr/kotlin/topic/5722/tableaux

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec 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	Alias de type	Kevin Robatel
3	Annotations	Brad Larson, Caelum, Héctor, Mood, piotrek1543, Sapan Zaveri
4	Bâtiment DSL	Dmitriy L, ice1000
5	Boucles à Kotlin	Ben Leggiero, JaseAnderson, mayojava, razzledazzle, Robin
6	Collections	Ascension
7	Configuration de la construction de Kotlin	Aaron Christiansen, elect, madhead
8	Constructeurs de type sûr	Slav
9	Cordes	Januson, Sam
10	coroutines	Jemo Mgebrishvili
11	Délégation de classe	Sam
12	Des exceptions	Brad Larson, jereksel, Sapan Zaveri
13	Enum	David Soroko, Kirill Rakhman, SerCe
14	Expressions conditionnelles	Abdullah, Alex Facciorusso, jpmcosta, Kirill Rakhman, Robin, Spidfire
15	Expressions idiomatiques	Aaron Christiansen, Adam Arold, Brad Larson, Héctor, Jayson Minard, Konrad Jamrozik, madhead, mayojava, razzledazzle, Sapan Zaveri, Serge Nikitin, yole
16	Gammes	Nihal Saxena
17	Génériques	hotkey, Jayson Minard, KeksArmee

18	Héritage de classe	byxor, KeksArmee, piotrek1543, Slav
19	Interfaces	Divya, Jan Vladimir Mostert, Jayson Minard, Ritave, Robin
20	Java 8 équivalents de flux	Brad, Gerson, Jayson Minard, Piero Divasto, Sam
21	JUnit	jenglert
22	Kotlin Android Extensions	Jemo Mgebrishvili, Ritave
23	Kotlin Caveats	Grigory Konushev, Spidfire
24	Kotlin pour les développeurs Java	Thorsten Schleinzer
25	Lambdas de base	memoizr, Rich Kuzsma
26	Les bases de Kotlin	Shinoo Goyal
27	Les fonctions	Aaron Christiansen, baha, Caelum, glee8e, Jayson Minard, KeksArmee, madhead, Spidfire
28	Méthodes d'extension	Dávid Tímár, Jayson Minard, Kevin Robatel, Konrad Jamrozik, olivierlemasle, Parker Hoyes, razzledazzle
29	Modificateurs de visibilité	Avijit Karmakar
30	Objets singleton	Divya, glee8e
31	Paramètres Vararg dans les fonctions	byxor, piotrek1543, Sam
32	Propriétés déléguées	Sam, Seaskyways
33	RecyclerView à Kotlin	Mohit Suthar
34	Réflexion	atok, Kirill Rakhman, madhead, Ritave, Sup
35	Regex	Espen, Travis
36	se connecter à kotlin	Konrad Jamrozik, olivierlemasle, oshai
37	Sécurité nulle	KeksArmee, Kirill Rakhman, piotrek1543, razzledazzle, Robin, SerCe, Spidfire, technerd, Thorsten Schleinzer
38	Tableaux	egor.zhdan, Sam, UnKnown