Kostenloses eBook

LERNEN Kotlin

Free unaffiliated eBook created from
Stack Overflow contributors.

#kotlin

Inhaltsverzeichnis

Über

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Kapitel 1: Erste Schritte mit Kotlin

Bemerkungen

Kotlin ist eine statisch typisierte, objektorientierte Programmiersprache, die von JetBrains entwickelt wurde und hauptsächlich auf die JVM abzielt. Kotlin wurde mit dem Ziel entwickelt, schnell zu kompilieren, abwärtskompatibel, sehr typsicher und zu 100% mit Java kompatibel zu sein. Kotlin wurde auch mit dem Ziel entwickelt, viele der von Java-Entwicklern gewünschten Funktionen bereitzustellen. Kotlins Standard-Compiler ermöglicht die Kompilierung sowohl in Java-Bytecode für die JVM als auch in JavaScript.

Kotlin kompilieren

Kotlin verfügt über ein Standard-IDE-Plugin für Eclipse und IntelliJ. Kotlin kann auch mit Maven , Ant und Gradle oder über die Befehlszeile kompiliert werden .

Beachten Sie, dass in `$ kotlinc Main.kt` eine Java- `$ kotlinc Main.kt` wird, in diesem Fall `MainKt.class` . Wenn Sie jedoch die Klassendatei mit `$ java MainKt` wird `$ java MainKt` java die folgende Ausnahme `$ java MainKt` :

```
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
```

Um die resultierende Klassendatei mit Java ausführen zu können, muss die Jot-Datei für die Kotlin-Laufzeit in den aktuellen Klassenpfad eingefügt werden.

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

Versionen

Ausführung	Veröffentlichungsdatum
1.0.0	2016-02-15
1.0.1	2016-03-16
1.0.2	2016-05-13
1.0.3	2016-06-30

Ausführung	Veröffentlichungsdatum
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

Hallo Welt

Alle Kotlin Programme beginnen an der `main` Hier ist ein Beispiel eines einfachen Kotlin "Hello World" -Programms:

```
package my.program

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

Platzieren Sie den obigen Code in einer Datei namens `Main.kt` (dieser Dateiname ist völlig willkürlich)

Beim Targeting der JVM wird die Funktion als statische Methode in einer Klasse mit einem vom Dateinamen abgeleiteten Namen kompiliert. Im obigen Beispiel würde die `my.program.MainKt` .

Um den Namen der Klasse zu ändern, die Funktionen der obersten Ebene für eine bestimmte Datei enthält, platzieren Sie die folgende Anmerkung oben in der Datei über der Paketanweisung:

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

In diesem Beispiel wäre die `my.program.MyApp` jetzt `my.program.MyApp` .

Siehe auch:

- Funktionen auf `@JvmName` einschließlich Annotation `@JvmName` .
- Annotation-Nutzungs-Site-Ziele

Hallo Welt mit einer Objektdeklaration

Sie können alternativ eine Objektdeklaration verwenden , die die Hauptfunktion für ein Kotlin-Programm enthält.

```
package my.program

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

Der Klassenname, den Sie ausführen werden, ist der Name Ihres Objekts. In diesem Fall ist dies `my.program.App` .

Der Vorteil dieser Methode gegenüber einer Top-Level-Funktion besteht darin, dass der auszuführende Klassenname selbstverständlich ist und alle anderen Funktionen, die Sie hinzufügen, in die Klasse `App` einbezogen werden. Sie haben dann auch eine Einzelinstanz von `App` , um den Status zu speichern und andere Arbeiten auszuführen.

Siehe auch:

- Statische Methoden einschließlich der `@JvmStatic` Annotation

Hallo Welt mit einem Companion-Objekt

Ähnlich wie bei der Verwendung einer Objektdeklaration können Sie die `main` eines Kotlin-Programms mithilfe eines Companion-Objekts einer Klasse definieren.

```
package my.program

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

Der Klassenname, den Sie ausführen werden, ist der Name Ihrer Klasse. In diesem Fall ist dies `my.program.App` .

Der Vorteil dieser Methode gegenüber einer Top-Level-Funktion besteht darin, dass der auszuführende Klassenname selbstverständlich ist und alle anderen Funktionen, die Sie hinzufügen, in die Klasse `App` einbezogen werden. Dies ähnelt dem Beispiel für die `Object Declaration` , außer dass Sie die Instanziierung von Klassen für die weitere Arbeit steuern können.

Eine kleine Variation, die die Klasse instanziiert, um das eigentliche "Hallo" zu tun:

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

```
        }
    }

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

Siehe auch:

- Statische Methoden einschließlich der @JvmStatic-Annotation

Hauptmethoden mit Varargs

Alle diese Hauptmethodenstile können auch mit varargs verwendet werden :

```
package my.program

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

Kotlin-Code in der Befehlszeile kompilieren und ausführen

Java bietet zwei verschiedene Befehle zum Kompilieren und Ausführen von Java-Code. Genauso wie Kotlin auch andere Befehle.

`javac` zum Kompilieren von Java-Dateien. `java` , um Java-Dateien auszuführen.

Wie `kotlinc` , um kotlin-Dateien zu kompilieren `kotlinc` , um kotlin-Dateien `kotlin` .

Eingabe von der Befehlszeile lesen

Die von der Konsole übergebenen Argumente können im Kotlin-Programm empfangen und als Eingabe verwendet werden. Sie können N (1 2 3 usw.) Argumentationsnummern an der Eingabeaufforderung übergeben.

Ein einfaches Beispiel für ein Befehlszeilenargument in Kotlin.

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

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

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



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

    var max = if (a > b) {
```

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

    return max;

}
```

Geben Sie hier zwei Zahlen in der Befehlszeile ein, um die maximale Anzahl zu ermitteln. Ausgabe :

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

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

Zum !! Betreiber Bitte überprüfen Sie die Nullsicherheit .

Hinweis: In obigem Beispiel kompilieren und auf Intellij ausführen.

Erste Schritte mit Kotlin online lesen: https://riptutorial.com/de/kotlin/topic/490/erste-schritte-mit-kotlin

Kapitel 2: Anmerkungen

Examples

Annotation deklarieren

Anmerkungen sind Mittel zum Anhängen von Metadaten an Code. Um eine Anmerkung zu deklarieren, setzen Sie den Anmerkungsmodifizierer vor eine Klasse:

```
annotation class Strippable
```

Anmerkungen können Meta-Anmerkungen enthalten:

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

Anmerkungen können wie andere Klassen über Konstruktoren verfügen:

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

Im Gegensatz zu anderen Klassen ist dies jedoch auf folgende Typen beschränkt:

- Typen, die Java-Grundtypen entsprechen (Int, Long usw.);
- Streicher
- Klassen (Foo :: Klasse)
- enums
- andere Anmerkungen
- Arrays der oben aufgeführten Typen

Meta-Anmerkungen

Bei der Deklaration einer Annotation können Meta-Informationen mit den folgenden Meta-Annotationen eingefügt werden:

- `@Target` : `@Target` die möglichen Arten von Elementen an, die mit der Annotation versehen werden können (Klassen, Funktionen, Eigenschaften, Ausdrücke usw.).

- `@Retention` gibt an, ob die Anmerkung in den kompilierten Klassendateien gespeichert ist und ob sie zur Laufzeit durch Reflektion sichtbar ist (standardmäßig sind beide wahr).

- `@Repeatable` ermöglicht die mehrfache Verwendung derselben Annotation für ein einzelnes Element.

- `@MustBeDocumented` gibt an, dass die Annotation Teil der öffentlichen API ist und in der in der generierten API-Dokumentation angegebenen Klasse oder Methodensignatur enthalten sein

sollte.

Beispiel:

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

Anmerkungen online lesen: https://riptutorial.com/de/kotlin/topic/4074/anmerkungen

Kapitel 3: Arrays

Examples

Generische Arrays

Generische Arrays in Kotlin werden durch `Array<T>` .

Um ein leeres Array zu erstellen, verwenden `emptyArray<T>()` Factory-Funktion `emptyArray<T>()` :

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

Verwenden Sie den Konstruktor, um ein Array mit gegebener Größe und Anfangswerten zu erstellen:

```
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
```

Arrays haben `get(index: Int): T` und `set(index: Int, value: T)` Funktionen:

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

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

Arrays von Primitiven

Diese Typen erben **nicht** von `Array<T>` , um Boxen zu vermeiden, sie haben jedoch dieselben Attribute und Methoden.

Kotlin-Typ	Werksfunktion	JVM-Typ
`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[]`

Erweiterungen

`average()` ist für `Byte` , `Int` , `Long` , `Short` , `Double` und `Float` und liefert immer `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()` ein Element des Arrays zurück

`getOrNull(index: Int)` gibt null zurück, wenn der Index außerhalb der Grenzen liegt, andernfalls ein Element des Arrays

`first()` , `last()`

`toHashSet()` gibt ein `HashSet<T>` aller Elemente zurück

`sortedArray()` , `sortedArrayDescending()` erstellt ein neues Array mit sortierten Elementen des aktuellen und gibt dieses zurück

`sort()` , `sortDescending` sortiert das Array an Ort und Stelle

`min()` , `max()`

Array iterieren

Sie können die Array-Elemente mit der Schleife wie bei der erweiterten Java-Schleife drucken. Sie müssen jedoch das Schlüsselwort von `:` in `in` ändern.

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

Sie können den Datentyp auch in der Schleife ändern.

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

Erstellen Sie ein Array

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

Erstellen Sie ein Array mit einem Abschluss

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

Erstellen Sie ein nicht initialisiertes Array

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

Das zurückgegebene Array hat immer einen nullfähigen Typ. Arrays von nicht nullfähigen Objekten können nicht ohne Initialisierung erstellt werden.

Arrays online lesen: https://riptutorial.com/de/kotlin/topic/5722/arrays

Kapitel 4: Ausnahmen

Examples

Ausnahme mit try-catch-finally

Ausnahmen in Kotlin abzufangen sieht Java sehr ähnlich

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

Sie können auch mehrere Ausnahmen abfangen

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

`try` ist auch ein Ausdruck und kann einen Wert zurückgeben

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

Kotlin hat keine Ausnahmen geprüft, so dass Sie keine Ausnahmen erkennen müssen.

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

Ausnahmen online lesen: https://riptutorial.com/de/kotlin/topic/7246/ausnahmen

Kapitel 5: Bedingte Anweisungen

Bemerkungen

Im Gegensatz zu Javas `switch` hat die `when` -Anweisung kein Durchfallverhalten. Das heißt, wenn ein Zweig übereinstimmt, kehrt der Kontrollfluss nach seiner Ausführung zurück und es ist keine `break` Anweisung erforderlich. Wenn Sie die bahaviors für mehrere Argumente kombinieren möchten, können Sie mehrere durch Kommas getrennte Argumente schreiben:

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

Examples

Standard-if-Anweisung

```
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!")
}
```

Die else-Zweige sind in normalen if-Anweisungen optional.

If-Anweisung als Ausdruck

If-Anweisungen können Ausdrücke sein:

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

Beachten Sie, dass die `else` -branch nicht optional ist, wenn die `if` -statement als Ausdruck verwendet wird.

Dies kann auch mit einer mehrzeiligen Variante mit geschweiften Klammern und mehreren `else if` Anweisungen geschehen.

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

TIPP: Kotlin kann den Typ der Variablen für Sie ableiten. Wenn Sie jedoch sichergehen möchten, dass der Typ sicher ist, kommentieren Sie ihn wie folgt für die Variable: `val str: String =` wird der Typ `val str: String =` und das Lesen wird einfacher.

Wenn-Anweisung anstelle von if-else-if-Ketten

Die when-Anweisung ist eine Alternative zu einer if-Anweisung mit mehreren else-if-Zweigen:

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

Gleicher Code, der mit einer *if-else-if*- Kette geschrieben wurde:

```
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!")
}
```

Genau wie bei der if-Anweisung ist der else-Zweig optional, und Sie können beliebig viele oder wenige Zweige hinzufügen. Sie können auch mehrzeilige Zweige haben:

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

Wenn-Anweisungsargument übereinstimmt

Bei Angabe eines Arguments stimmt die `when` -Anweisung mit dem Argument gegen die Zweige in der Reihenfolge überein. Der Abgleich erfolgt mit dem Operator `==` , der Nullprüfungen durchführt und die Operanden mit der Funktion `equals` vergleicht. Der erste passende wird ausgeführt.

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

Die when-Anweisung kennt auch einige erweiterte Übereinstimmungsoptionen:

```
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
}
```

Wann-Anweisung als Ausdruck

Wie wenn, wenn kann auch als Ausdruck verwendet werden:

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

Um als Ausdruck verwendet zu werden, muss die when-Anweisung erschöpfend sein, dh entweder einen anderen Zweig haben oder alle Möglichkeiten mit den Zweigen auf andere Weise abdecken.

Wann-Statement mit Aufzählungszeichen

`when` kann verwendet werden, um `enum` Werte `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 ->   // ...
    }
}
```

Wie Sie in der zweiten `Tuedsay` (`Monday` und `Tuedsay`) sehen können, können Sie auch zwei oder mehrere `enum` kombinieren.

Wenn Ihre Fälle nicht vollständig sind, zeigt das Kompilieren einen Fehler an. Sie können `else` , um Standardfälle zu behandeln:

```
fun doOnDay(day: Day) {
```

```
    when(day) {
        Day.Monday ->      // Work
        Day.Tuesday ->     // Work hard
        Day.Wednesday ->   // ...
        Day.Thursday ->    //
        Day.Friday ->      //
        else ->            // Party on weekend
    }
}
```

Dasselbe kann mit dem `if-then-else` Konstrukt durchgeführt werden, `when` sich um fehlende `enum` kümmert und diese natürlicher macht.

Schauen Sie hier für weitere Informationen über Kotlin `enum`

Bedingte Anweisungen online lesen: https://riptutorial.com/de/kotlin/topic/2685/bedingte-anweisungen

Kapitel 6: Bereiche

Einführung

Bereichsausdrücke werden mit RangeTo-Funktionen gebildet, die die Operator-Form haben .. die durch in und! In ergänzt wird. Der Bereich ist für jeden vergleichbaren Typ definiert, für integrale Primitivtypen jedoch eine optimierte Implementierung

Examples

Integrierte Typenbereiche

Integrale Typenbereiche (IntRange, LongRange, CharRange) haben eine zusätzliche Funktion: Sie können wiederholt werden. Der Compiler kümmert sich um die analoge Konvertierung in die indizierte for-Schleife von Java ohne zusätzlichen Aufwand

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

Funktion downTo ()

Wenn Sie Zahlen in umgekehrter Reihenfolge durchlaufen möchten? Es ist einfach. Sie können die in der Standardbibliothek definierte Funktion downTo () verwenden

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

Stufenfunktion

Ist es möglich, Zahlen mit beliebigen Schritten, die nicht gleich 1 sind, zu durchlaufen? Sicher, die step () - Funktion wird Ihnen helfen

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

bis zur Funktion

Um einen Bereich zu erstellen, der sein Endelement nicht enthält, können Sie die before-Funktion verwenden:

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

Bereiche online lesen: https://riptutorial.com/de/kotlin/topic/10121/bereiche

Kapitel 7: Coroutinen

Einführung

Beispiele für Kotlins experimentelle (noch) Implementierung von Coroutinen

Examples

Einfache Coroutine, deren Verzögerung 1 Sekunde beträgt, aber nicht blockiert

(aus offiziellem Dokument)

```
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
}
```

Ergebnis

```
Hello,
World!
```

Coroutinen online lesen: https://riptutorial.com/de/kotlin/topic/10936/coroutinen

Kapitel 8: Delegierte Eigenschaften

Einführung

Kotlin kann die Implementierung einer Eigenschaft an ein Handler-Objekt delegieren. Einige Standard-Handler sind enthalten, z. B. Lazy-Initialisierung oder beobachtbare Eigenschaften. Es können auch benutzerdefinierte Handler erstellt werden.

Examples

Faule Initialisierung

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

Das Beispiel druckt `2` .

Beobachtbare Eigenschaften

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

Die Beispieldrucke `foo was changed from 1 to 2`

Kartenunterstützte Eigenschaften

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

Das Beispiel druckt `1`

Kundenspezifische Delegation

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

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

Das Beispiel druckt einen `Delegated value`

Delegat Kann als Schicht verwendet werden, um die Boilerplate zu reduzieren

Betrachten Sie das Null-Typ-System von Kotlin und `WeakReference<T>` .

`WeakReference` wir also an, wir müssen eine Art Referenz speichern und wir wollten Speicherlecks vermeiden. Hier kommt `WeakReference` ins `WeakReference` .

Nehmen Sie zum Beispiel folgendes:

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

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

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

Nun ist dies nur mit einer WeakReference. Um diese Boilerplate zu reduzieren, können wir einen benutzerdefinierten Eigenschaftsdelegierten verwenden, der uns wie folgt hilft:

```
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)
    }
}
```

Jetzt können wir Variablen verwenden, die mit `WeakReference` wie normale, nullfähige Variablen `WeakReference` sind!

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

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

    init {
        reference = this
    }
```

```
}
```

Delegierte Eigenschaften online lesen: https://riptutorial.com/de/kotlin/topic/10571/delegierte-eigenschaften

Kapitel 9: DSL-Gebäude

Einführung

Konzentrieren Sie sich auf die Syntaxdetails, um interne DSLs in Kotlin zu entwerfen.

Examples

Infix-Ansatz zum Aufbau von DSL

Wenn Sie haben:

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

In Ihren Tests können Sie den folgenden DSL-ähnlichen Code schreiben:

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

Überschreiben der Aufrufmethode zum Erstellen von DSL

Wenn Sie haben:

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

Sie können den folgenden DSL-ähnlichen Code in Ihren Produktionscode schreiben:

```
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")
    }
}
```

Verwenden von Operatoren mit Lambdas

Wenn Sie haben:

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

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

Sie können den folgenden DSL-ähnlichen Code schreiben:

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

Erweiterungen mit Lambdas verwenden

Wenn Sie haben:

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

Sie können den folgenden DSL-ähnlichen Code schreiben:

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

Wenn Sie sich mit `shouldBe` obigen `shouldBe` verwirrt fühlen sollten, `shouldBe` Sie den `Infix approach to build DSL .`

DSL-Gebäude online lesen: https://riptutorial.com/de/kotlin/topic/10042/dsl-gebaude

Kapitel 10: Enum

Bemerkungen

Genau wie in Java verfügen Enum-Klassen in Kotlin über synthetische Methoden, mit denen die definierten Enumenkonstanten aufgelistet und eine Enumenkonstante anhand ihres Namens abgerufen werden kann. Die Signaturen dieser Methoden lauten wie folgt (vorausgesetzt, der Name der `EnumClass` lautet `EnumClass`):

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

Die Methode `valueOf()` `IllegalArgumentException` eine `IllegalArgumentException` wenn der angegebene Name mit keiner der in der Klasse definierten `IllegalArgumentException` übereinstimmt.

Jede Enumenkonstante verfügt über Eigenschaften, um ihren Namen und ihre Position in der Deklaration der Enumenklassen zu erhalten:

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

Die Enumenkonstanten implementieren auch die Comparable-Schnittstelle, wobei die natürliche Reihenfolge die Reihenfolge ist, in der sie in der Enumenklasse definiert werden.

Examples

Initialisierung

Aufzählungsklassen wie alle anderen Klassen können einen Konstruktor haben und können initialisiert werden

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

Funktionen und Eigenschaften in Enums

Aufzählungsklassen können auch Member deklarieren (dh Eigenschaften und Funktionen). Ein Semikolon (`;`) muss zwischen dem letzten Enum-Objekt und der ersten Memberdeklaration stehen.

Wenn ein Member `abstract` , muss es von den Enum-Objekten implementiert werden.

```
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)
}
```

Einfache enum

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

Jede Enumerationskonstante ist ein Objekt. Aufzählungskonstanten werden durch Kommas getrennt.

Wandlungsfähigkeit

Enumerationen können veränderlich sein. Dies ist eine weitere Möglichkeit, um ein Singleton-Verhalten zu erzielen:

```
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]
```

Enum online lesen: https://riptutorial.com/de/kotlin/topic/2286/enum

Kapitel 11: Erweiterungsmethoden

Syntax

- fun TypeName.extensionName (Parameter, ...) {/ * body * /} // Deklaration
- fun <T: Any> TypeNameWithGenerics <T> .extensionName (params, ...) {/ * body * /} // Deklaration mit Generics
- myObj.extensionName (args, ...) // Aufruf

Bemerkungen

Erweiterungen werden **statisch** aufgelöst. Dies bedeutet, dass die zu verwendende Erweiterungsmethode durch den Referenztyp der Variablen bestimmt wird, auf die Sie zugreifen. Es spielt keine Rolle, welchen Typ die Variable zur Laufzeit hat, es wird immer dieselbe Erweiterungsmethode aufgerufen. Dies liegt daran, dass beim **Deklarieren einer Erweiterungsmethode dem Empfängertyp kein Mitglied tatsächlich hinzugefügt wird** .

Examples

Erweiterungen der obersten Ebene

Erweiterungsmethoden der obersten Ebene sind nicht in einer Klasse enthalten.

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

Oben ist eine Erweiterungsmethode für den Typ `IntArray` . Beachten Sie, dass auf das Objekt, für das die Erweiterungsmethode definiert ist (als **Empfänger bezeichnet**), mit dem Schlüsselwort `this` zugegriffen wird.

Diese Erweiterung kann wie folgt aufgerufen werden:

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

Mögliche Gefahr: Erweiterungen werden statisch aufgelöst

Die aufzurufende Erweiterungsmethode wird zur Kompilierzeit basierend auf dem Referenztyp der Variablen bestimmt, auf die zugegriffen wird. Es spielt keine Rolle, was der Variablentyp zur Laufzeit ist, es wird immer dieselbe Erweiterungsmethode aufgerufen.

```
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())
```

Im obigen Beispiel wird `"Defined for Super"` gedruckt, da der deklarierte Typ der Variablen `myVar` `Super` .

Beispiel, das sich lang ausdehnt, um eine für Menschen lesbare Zeichenfolge darzustellen

Geben Sie einen beliebigen Wert vom Typ `Int` oder `Long` , um eine für Menschen lesbare Zeichenfolge darzustellen:

```
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()
}
```

Dann einfach als:

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

Beispiel für die Erweiterung der Klasse Java 7+ Path

Ein üblicher Anwendungsfall für Erweiterungsmethoden ist die Verbesserung einer vorhandenen API. Hier einige Beispiele für das Hinzufügen von `exist` , `notExists` und `deleteRecursively` zur Java 7+ `Path` Klasse:

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

Was kann nun in diesem Beispiel aufgerufen werden:

```
val dir = Paths.get(dirName)
```

```
if (dir.exists()) dir.deleteRecursively()
```

Erweiterungsfunktionen verwenden, um die Lesbarkeit zu verbessern

In Kotlin könnte man Code schreiben wie:

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

Die Verwendung von `apply` ist jedoch nicht so klar wie Ihre Absicht. Manchmal ist es klarer, eine ähnliche Erweiterungsfunktion zu erstellen, um die Aktion tatsächlich umzubenennen und sie selbstverständlich zu machen. Dies sollte nicht außer Kontrolle geraten, sondern bei sehr häufigen Aktionen wie der Verifizierung:

```
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
}
```

Sie könnten jetzt den Code schreiben als:

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

Was nun die Leute wissen lässt, was innerhalb des Lambda-Parameters zu erwarten ist.

Beachten Sie, dass der Typparameter `T` für `verifiedBy` mit `T: Any?` Das bedeutet, dass auch nullfähige Typen diese Version der Erweiterung verwenden können. Obwohl `verifiedWith` nicht nullfähig ist.

Beispiel zur Erweiterung von Java 8 Temporal-Klassen zum Rendern einer ISO-formatierten Zeichenfolge

Mit dieser Erklärung:

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

Sie können jetzt einfach:

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

Erweiterungsfunktionen für Begleitobjekte (Darstellung statischer Funktionen)

Wenn Sie eine Klasse als statische Funktion erweitern möchten, z. B. für die Klasse `Something` add statisch `fromString` Funktion, kann dies nur funktionieren, wenn die Klasse über ein Begleitobjekt verfügt und die Erweiterungsfunktion für das Begleitobjekt deklariert wurde :

```
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
}
```

Problemumgehung für faul Erweiterungs-Eigenschaften

Angenommen, Sie möchten eine Erweiterungseigenschaft erstellen, deren Berechnung teuer ist. Daher möchten Sie die Berechnung zwischenspeichern, indem Sie den verzögerten Eigenschaftsdelegierten verwenden und auf die aktuelle Instanz (`this`) verweisen. `this` ist jedoch nicht möglich, wie in den Kotlin-Ausgaben KT-9686 und KT-13053 erläutert. Es wird jedoch eine offizielle Problemumgehung bereitgestellt .

Im Beispiel ist die Erweiterungseigenschaft `color` . Es verwendet eine explizite `colorCache` , die mit verwendet werden kann `this` als nicht `lazy` ist notwendig:

```
class KColor(val value: Int)

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

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

Erweiterungen zum leichteren Nachschlagen View from code

Sie können Erweiterungen für die Referenzansicht verwenden, keine Boilerplate mehr, nachdem Sie die Ansichten erstellt haben.

Ursprüngliche Idee stammt von Anko Library

Erweiterungen

```
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
```

Verwendungszweck

```
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) }
```

Erweiterungsmethoden online lesen:
https://riptutorial.com/de/kotlin/topic/613/erweiterungsmethoden

Kapitel 12: Funktionen

Syntax

- Spaß ***Name*** (***Params***) = ...
- lustiger ***Name*** (***Params***) {...}
- Spaß ***Name*** (***Params***): ***Typ*** {...}
- Spaß ***<Typ Argument> Name (Param): Typ*** {...}
- Inline - Spaß - ***Name (Param): Typ*** {...}
- { ***ArgName*** : ***ArgType*** -> ...}
- { ***ArgName*** -> ...}
- { ***ArgNames*** -> ...}
- {(***ArgName*** : ***ArgType***): ***Typ*** -> ...}

Parameter

Parameter	Einzelheiten
Name	Name der Funktion
Params	Werte, die der Funktion mit einem Namen und Typ übergeben werden: `Name : Type`
Art	Rückgabetyp der Funktion
Geben Sie Argument ein	Typparameter, der in der generischen Programmierung verwendet wird (nicht unbedingt Rückgabetyp)
ArgName	Name des Werts, der der Funktion gegeben wurde
ArgType	***Typbezeichner*** für ***ArgName***
ArgNames	Liste der durch Kommas getrennten ArgName

Examples

Funktionen unter anderen Funktionen

Wie in "Lambda-Funktionen" gezeigt, können Funktionen andere Funktionen als Parameter annehmen. Der "Funktionstyp", den Sie zum Deklarieren von Funktionen benötigen, die andere Funktionen benötigen, lautet wie folgt:

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

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

Zum Beispiel könnten Sie den vagen Typ verwenden, `() -> Any?` , um eine Funktion zu deklarieren, die eine Lambda-Funktion zweimal ausführt:

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

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

Lambda-Funktionen

Lambda-Funktionen sind anonyme Funktionen, die normalerweise während eines Funktionsaufrufs als Funktionsparameter erstellt werden. Sie werden durch umgebende Ausdrücke mit {geschweiften Klammern} deklariert. Wenn Argumente benötigt werden, werden diese vor einem Pfeil `->` .

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

Die letzte Anweisung innerhalb einer Lambda-Funktion ist automatisch der Rückgabewert.

Die Typen sind optional, wenn Sie das Lambda an einer Stelle ablegen, an der der Compiler auf die Typen schließen kann.

Mehrere Argumente:

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

Wenn die Lambda - Funktion ein Argument nur braucht, dann kann die Argumentliste weggelassen werden , und das einzige Argument zu verwenden bezeichnet `it` stattdessen.

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

Wenn das einzige Argument einer Funktion eine Lambda-Funktion ist, können Klammern vollständig vom Funktionsaufruf weggelassen werden.

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

Funktionsreferenzen

Wir können auf eine Funktion verweisen, ohne sie tatsächlich aufzurufen, indem Sie den Namen der Funktion mit `::` . Diese kann dann an eine Funktion übergeben werden, die eine andere Funktion als Parameter akzeptiert.

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

Funktionen ohne Empfänger werden in `(ParamTypeA, ParamTypeB, ...) -> ReturnType` `ParamTypeA` , wobei `ParamTypeA` , `ParamTypeB` ... der Typ der Funktionsparameter sind und `ReturnType1 der Typ des Funktionsrückgabewerts.

```
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
```

Funktionen mit einem Empfänger (sei es eine Erweiterungsfunktion oder eine Memberfunktion) haben eine andere Syntax. Sie müssen den Typnamen des Empfängers vor dem Doppelpunkt eingeben:

```
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.
```

Wenn der Empfänger einer Funktion jedoch ein Objekt ist, wird der Empfänger nicht in der Parameterliste aufgeführt, da dies nur eine Instanz eines solchen Typs ist.

```
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.
```

Seit kotlin 1.1 kann die Funktionsreferenz auch auf eine Variable *begrenzt* werden, die dann als *begrenzte Funktionsreferenz bezeichnet wird* .

1.1.0

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

Beachten Sie, dass dieses Beispiel nur dazu dient, zu zeigen, wie die Funktionsreferenz begrenzter Funktionen funktioniert. Es ist schlechte Praxis in allen anderen Sinnen.

Es gibt jedoch einen Sonderfall. Eine als Member deklarierte Erweiterungsfunktion kann nicht referenziert werden.

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

Basisfunktionen

Funktionen werden mit dem Schlüsselwort `fun` deklariert, gefolgt von einem Funktionsnamen und beliebigen Parametern. Sie können auch den Rückgabetyp einer Funktion angeben, der standardmäßig auf `Unit` . Der Hauptteil der Funktion ist in geschweifte Klammern `{}` . Wenn der Rückgabetyp nicht `Unit` , muss der Hauptteil für jede abschließende Verzweigung innerhalb des Hauptteils eine Rückgabeanweisung ausgeben.

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

Eine Kurzfassung des gleichen:

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

Und der Typ kann weggelassen werden, da darauf geschlossen werden kann:

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

Abkürzungsfunktionen

Wenn eine Funktion nur einen Ausdruck enthält, können wir die geschweiften Klammern weglassen und stattdessen ein Gleiches wie eine variable Zuweisung verwenden. Das Ergebnis des Ausdrucks wird automatisch zurückgegeben.

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

Inline-Funktionen

Funktionen können mithilfe des `inline` Präfixes inline deklariert werden. In diesem Fall verhalten sie sich wie C-Makros. Sie werden nicht aufgerufen, sondern werden beim Kompilieren durch den Bodycode der Funktion ersetzt. Dies kann unter Umständen zu Leistungsvorteilen führen, vor allem wenn Lambdas als Funktionsparameter verwendet werden.

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

Ein Unterschied zu C-Makros besteht darin, dass Inline-Funktionen nicht auf den Gültigkeitsbereich zugreifen können, aus dem sie aufgerufen werden:

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

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

Bedienfunktionen

Kotlin ermöglicht es uns, Implementierungen für eine vordefinierte Gruppe von Operatoren mit fester symbolischer Darstellung (wie `+` oder `*`) und fester Priorität bereitzustellen. Um einen Operator zu implementieren, stellen wir eine Member-Funktion oder eine Erweiterungsfunktion mit festem Namen für den entsprechenden Typ bereit. Funktionen, die Überladungsoperatoren mit dem `operator` Modifizierer kennzeichnen müssen:

```
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
```

Weitere Bedienfunktionen finden Sie hier

Funktionen online lesen: https://riptutorial.com/de/kotlin/topic/1280/funktionen

Kapitel 13: Geben Sie Aliase ein

Einführung

Mit Typ-Aliasnamen können wir anderen Typen einen Alias geben. Es ist ideal, um Funktionstypen wie `(String) -> Boolean` oder generischen Typen wie `Pair<Person, Person>` einen Namen zu geben.

Typ-Aliase unterstützen Generics. Ein Alias kann einen Typ durch Generika ersetzen, und ein Alias kann Generika sein.

Syntax

- **typealias** *alias-name* **=** *vorhandener Typ*

Bemerkungen

Typ-Aliase ist eine Funktion des Compilers. Der generierte Code für die JVM wird nicht hinzugefügt. Alle Aliase werden durch den echten Typ ersetzt.

Examples

Funktionsart

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

Generischer Typ

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

Geben Sie Aliase ein online lesen: https://riptutorial.com/de/kotlin/topic/9453/geben-sie-aliase-ein

Kapitel 14: Generics

Einführung

Eine Liste kann Zahlen, Wörter oder wirklich alles enthalten. Deshalb nennen wir die Liste *generisch* .

Generics werden im Allgemeinen verwendet, um zu definieren, welche Typen eine Klasse enthalten kann und welchen Typ ein Objekt derzeit enthält.

Syntax

- Klasse *Klassenname* < ***Typname*** >
- Klasse *ClassName* <*>
- *ClassName* <in ***UpperBound*** >
- *ClassName* <out ***LowerBound*** >
- *Klassenname* ***<Typename: Upperbound>***

Parameter

Parameter	Einzelheiten
Modellname	Typ Name des generischen Parameters
Obere Grenze	Kovarianter Typ
LowerBound	Kontravarianter Typ
Klassenname	Name der Klasse

Bemerkungen

Implizite obere Grenze ist nullfähig

In Kotlin Generics wäre die Obergrenze des Typparameters `T Any?` . Deshalb für diese Klasse:

```
class Consumer<T>
```

Der Typparameter `T` ist wirklich `T: Any?` . Um eine nicht-nullfähige Obergrenze festzulegen, explizit ein bestimmtes `T: Any` . Zum Beispiel:

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

Examples

Deklaration-Site-Abweichung

Deklarationsstellenabweichung kann als Deklaration der Verwendungsstellenabweichung einmal und für alle Verwendungsstellen verstanden werden.

```
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
```

Weit verbreitete Beispiele für die Abweichung von Deklarationsstellen sind `List<out T>` , die unveränderlich ist, sodass `T` nur als Rückgabewerttyp angezeigt wird, und `Comparator<in T>` , der nur `T` als Argument empfängt.

Abweichung der Verwendungsstelle

Die Abweichung bei der Verwendung von Websites ähnelt Java-Platzhaltern:

Out-Projektion:

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

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

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

In-Projektion:

```
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
```

Sternprojektion

```
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
```

Siehe auch:

- Interoperabilität der Variant Generics beim Aufruf von Kotlin aus Java.

Generics online lesen: https://riptutorial.com/de/kotlin/topic/1147/generics

Kapitel 15: Grundlagen von Kotlin

Einführung

Dieses Thema behandelt die Grundlagen von Kotlin für Anfänger.

Bemerkungen

1. Die Kotlin-Datei hat die Erweiterung .kt.
2. Alle Klassen in Kotlin haben eine gemeinsame Oberklasse Any, das ist ein Standard-Super für eine Klasse ohne Supertypen (ähnlich wie Object in Java).
3. Variablen können als val (unveränderlich, einmalig zuweisen) oder var (variabler Wert können geändert werden).
4. Am Ende der Anweisung wird kein Semikolon benötigt.
5. Wenn eine Funktion keinen nützlichen Wert zurückgibt, lautet der Rückgabetyp Unit.It ist ebenfalls optional. 6.Referenzielle Gleichheit wird durch die Operation === geprüft. a === b wird dann als wahr ausgewertet, wenn a und b auf dasselbe Objekt zeigen.

Examples

Grundlegende Beispiele

1.Die Deklaration der Rückgabetypen ist für Funktionen optional. Die folgenden Codes sind gleichwertig.

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

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

2.Einzelausdruck-Funktionen: Wenn eine Funktion einen einzelnen Ausdruck zurückgibt, können die geschweiften Klammern weggelassen werden und der Körper wird nach = Symbol angegeben

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

Das explizite Deklarieren des Rückgabetyps ist optional, wenn dies vom Compiler abgeleitet werden kann

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

3. String-Interpolation: Die Verwendung von String-Werten ist einfach.

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

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

4. In Kotlin unterscheidet das Typsystem zwischen Referenzen, die null (nullfähige Referenzen) enthalten können, und solchen, die dies nicht können (nicht-null-Referenzen). Beispielsweise kann eine reguläre Variable vom Typ String nicht null enthalten:

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

Um Nullen zuzulassen, können wir eine Variable als nullbaren String, geschriebener String, deklarieren:

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

5.In Kotlin prüft == tatsächlich die Gleichheit der Werte. Nach der Konvention wird ein Ausdruck wie a == b in übersetzt

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

Grundlagen von Kotlin online lesen: https://riptutorial.com/de/kotlin/topic/10648/grundlagen-von-kotlin

Kapitel 16: Grundlegende Lambdas

Syntax

- Explizite Parameter:
- {parameterName: ParameterType, otherParameterName: OtherParameterType -> anExpression ()}
- Abgeleitete Parameter:
- Addition: (Int, Int) -> Int = {a, b -> a + b}
- Einzelne Parameter `it` Kurzschrift
- val square: (Int) -> Int = {it * it}
- Unterschrift:
- () -> ResultType
- (Eingabetyp) -> ResultType
- (InputType1, InputType2) -> ResultType

Bemerkungen

Eingabetypparameter können ausgelassen werden, wenn sie ausgelassen werden können, wenn sie aus dem Kontext abgeleitet werden können. Angenommen, Sie haben eine Funktion für eine Klasse, die eine Funktion übernimmt:

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

Sie können diese Funktion verwenden, indem Sie ein Lambda übergeben. Da die Parameter bereits in der Funktionssignatur angegeben sind, müssen Sie sie nicht im Lambda-Ausdruck erneut deklarieren:

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

Dies gilt auch, wenn Sie einer Variablen ein Lambda zuweisen:

```
//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 }
```

Wenn für das Lambda ein Parameter benötigt wird und der Typ aus dem Kontext abgeleitet werden kann, können Sie auf den Parameter durch `it` Bezug nehmen.

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

Examples

Lambda als Parameter für die Filterfunktion

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

Lambda wurde als Variable übergeben

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

Lambda für das Benchmarking eines Funktionsaufrufs

Mehrzweck-Stoppuhr für das Timing der Funktionsdauer einer Funktion:

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

Verwendungszweck:

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

Grundlegende Lambdas online lesen: https://riptutorial.com/de/kotlin/topic/5878/grundlegende-lambdas

Kapitel 17: Java 8-Stream-Entsprechungen

Einführung

Kotlin bietet viele Erweiterungsmethoden für Sammlungen und iterierbare Elemente zum Anwenden funktionaler Operationen. Ein dedizierter `Sequence` ermöglicht die faule Komposition mehrerer solcher Operationen.

Bemerkungen

Über Faulheit

Wenn Sie eine Kette faul verarbeiten möchten, können Sie sie mit `asSequence()` vor der Kette in eine `Sequence asSequence()` . Am Ende der Funktionskette erhalten Sie normalerweise auch eine `Sequence` . Dann können Sie `toList()` , `toSet()` , `toMap()` oder eine andere Funktion verwenden, um die `Sequence` am Ende zu materialisieren.

```
// 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()
```

Warum gibt es keine Typen?!?

Sie werden feststellen, dass die Kotlin-Beispiele die Typen nicht angeben. Dies liegt daran, dass Kotlin die vollständige Typinferenz hat und zur Kompilierzeit vollständig typsicher ist. Mehr als Java, da es auch nullfähige Typen hat und dazu beitragen kann, die gefürchtete NPE zu verhindern. Also das in Kotlin:

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

ist das gleiche wie:

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

Da Kotlin weiß, was `people` sind und dass `people.age Int` ist, `people.age` der Filterausdruck nur Vergleiche mit einem `Int` , und da `people.name` ein `String` erzeugt der `map` eine `List<String>` (Readonly `List` of `String`).

Nun, wenn `people` möglicherweise `null` , als `List<People>?` dann:

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

Gibt eine `List<String>?` das müsste auf null geprüft werden (*oder verwenden Sie einen der anderen Kotlin-Operatoren für nullfähige Werte, siehe diese idiomatische Methode von Kotlin zum Umgang mit nullfähigen Werten sowie die idiomatische Art, nullfähige oder leere Listen in Kotlin zu behandeln*)

Streams wiederverwenden

In Kotlin hängt es von der Art der Sammlung ab, ob sie mehr als einmal konsumiert werden kann. Eine `Sequence` generiert jedes Mal einen neuen Iterator, und wenn sie nicht "nur einmal verwenden" behauptet, kann sie bei jeder Ausführung auf den Start zurückgesetzt werden. Daher schlägt folgendes in Java 8-Stream fehl, funktioniert aber in Kotlin:

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

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

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

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

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

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

Und in Java, um das gleiche Verhalten zu erhalten:

```
// 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
```

Daher entscheidet der Anbieter der Daten in Kotlin, ob er zurücksetzen und einen neuen Iterator bereitstellen kann oder nicht. Wenn Sie jedoch eine `Sequence` absichtlich auf eine Iteration beschränken möchten, können Sie die Funktion `constrainOnce()` für `Sequence` wie folgt verwenden:

```
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.
```

Siehe auch:

- API-Referenz für Erweiterungsfunktionen für Iterable
- API-Referenz für Erweiterungsfunktionen für Array
- API-Referenz für Erweiterungsfunktionen für List
- API-Referenz für Erweiterungsfunktionen für Map

Examples

Namen in einer Liste sammeln

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

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

Konvertieren Sie Elemente in Strings und verketten Sie sie, getrennt durch Kommas

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

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

Berechnen Sie die Summe der Gehälter des Angestellten

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

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

Mitarbeiter nach Abteilungen zusammenfassen

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

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

Berechnen Sie die Summe der Gehälter nach Abteilung

```
// 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 }}
```

Trennen Sie die Schüler in Pass und Misserfolg

```
// 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 }
```

Namen der männlichen Mitglieder

```
// 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 }
```

Gruppennamen der Mitglieder in der Liste nach Geschlecht

```
// 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 } }
```

Filtern Sie eine Liste in eine andere Liste

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

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

Suchen Sie nach der kürzesten Zeichenfolge einer Liste

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

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

Verschiedene Arten von Streams # 2 - faul beim ersten Artikel, falls vorhanden

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

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

Verschiedene Arten von Streams # 3 - eine Reihe von ganzen Zahlen durchlaufen

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

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

Verschiedene Arten von Streams # 4 - Durchlaufen Sie ein Array, ordnen Sie die Werte zu und berechnen Sie den Durchschnitt

```
// 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)
```

Verschiedene Arten von Streams Nr. 5: Durchlaufen Sie faul eine Liste von Strings, ordnen Sie die Werte zu, konvertieren Sie in Int, finden Sie 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)
```

Verschiedene Arten von Streams Nr. 6: Durchlaufen Sie einen Ints-Stream träge, ordnen Sie die Werte zu und drucken Sie die Ergebnisse

```
// 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)
```

Verschiedene Arten von Streams Nr. 7 - Doppelter Durchlauf durchlässig, Zuordnung zu Int, Zuordnung zu Zeichenfolge, Drucken jeweils

```
// 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)
```

Elemente in einer Liste zählen, nachdem der Filter angewendet wurde

```
// 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') }
```

Funktionsweise von Streams: Filtern Sie Großbuchstaben und sortieren Sie dann eine 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)
```

Verschiedene Arten von Streams Nr. 1 - eifrig mit dem ersten Element, falls vorhanden

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

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

oder erstellen Sie eine Erweiterungsfunktion für String mit dem Namen 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)
```

Siehe auch: `apply()` Funktion

Siehe auch: Erweiterungsfunktionen

Siehe auch: `?.` Safe Call-Operator und im Allgemeinen nullability: http://stackoverflow.com/questions/34498562/in-kotlin-what-is-the-idiomatic-way-to-deal-mit-nullable-values-referencing-o/34498563 # 34498563

Sammeln Sie Beispiel 5 - finden Sie Personen im gesetzlichen Alter, geben Sie eine formatierte Zeichenfolge aus

```
// 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.
```

Als Randnotiz können wir in Kotlin einfache Datenklassen erstellen und die Testdaten wie folgt instanziieren:

```
// 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))
```

Sammle Beispiel # 6 - gruppiere Leute nach Alter, Alter und Namen zusammen

```
// 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, ein interessanter Fall hier für Kotlin. Zuerst die falschen Antworten, um Variationen beim Erstellen einer `Map` aus einer Sammlung / Sequenz zu untersuchen:

```
// 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>
```

Und nun zur richtigen Antwort:

```
// 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!!
```

Wir mussten nur die übereinstimmenden Werte zusammenfügen, um die Listen zu `joinToString` , und `joinToString` einen Transformer `joinToString` , um von der `Person` zum `Person` zu `Person.name` .

Sammeln Sie Beispiel # 7a - Kartennamen, verbinden Sie sich mit Trennzeichen

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

Sammeln Sie Beispiel # 7b - Sammeln Sie mit 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)
```

Es ist jedoch besser, eine Erweiterungsfunktion zu erstellen, 2, die tatsächlich mit den Styles in Kotlin stdlib übereinstimmt:

```
// 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)) }
```

Jetzt haben Sie zwei Möglichkeiten , um die neuen verwenden `summarizingInt` Funktionen:

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

// or

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

Und alle diese produzieren die gleichen Ergebnisse. Wir können diese Erweiterung auch erstellen, um in `Sequence` und für entsprechende primitive Typen zu arbeiten.

Java 8-Stream-Entsprechungen online lesen: https://riptutorial.com/de/kotlin/topic/707/java-8-stream-entsprechungen

Kapitel 18: JUnit

Examples

Regeln

So fügen Sie einem Test-Fixture eine JUnit- Regel hinzu :

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

Die Annotation `@JvmField` ist erforderlich, um das Hintergrundfeld mit der gleichen Sichtbarkeit (public) wie die `myRule` Eigenschaft (siehe Antwort) `myRule` . Bei JUnit-Regeln muss das annotierte Regelfeld öffentlich sein.

JUnit online lesen: https://riptutorial.com/de/kotlin/topic/6973/junit

Kapitel 19: Klassendelegation

Einführung

Eine Kotlin-Klasse kann eine Schnittstelle implementieren, indem sie ihre Methoden und Eigenschaften an ein anderes Objekt delegiert, das diese Schnittstelle implementiert. Dies bietet eine Möglichkeit, Verhalten unter Verwendung von Assoziation statt Vererbung zu erstellen.

Examples

Delegieren Sie eine Methode an eine andere Klasse

```
interface Foo {
    fun example()
}

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

class Baz(b : Bar) : Foo by b

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

Das Beispiel druckt `Hello, world!`

Klassendelegation online lesen: https://riptutorial.com/de/kotlin/topic/10575/klassendelegation

Kapitel 20: Klassenvererbung

Einführung

Jede objektorientierte Programmiersprache hat irgendeine Form der Klassenvererbung. Lass mich überarbeiten:

Stellen Sie sich vor, Sie müssten einen Haufen Obst programmieren: `Apples` , `Oranges` und `Pears` . Sie unterscheiden sich alle in Größe, Form und Farbe, deshalb haben wir unterschiedliche Klassen.

Aber lassen Sie uns sagen, dass ihre Unterschiede für eine Sekunde keine Rolle spielen und Sie wollen einfach nur eine `Fruit` , egal welche genau? Welchen Rückgabetyp hätte `getFruit()` ?

Die Antwort ist Klasse `Fruit` . Wir erstellen eine neue Klasse und lassen alle Früchte davon erben!

Syntax

- offen {Basisklasse}
- Klasse {Abgeleitete Klasse}: {Basisklasse} ({Init-Argumente})
- überschreiben {Funktionsdefinition}
- {DC-Object} ist {Base Class} == wahr

Parameter

Parameter	Einzelheiten
Basisklasse	Klasse, von der vererbt wird
Abgeleitete Klasse	Klasse, die von der Basisklasse erbt
Init-Argumente	An den Konstruktor der Basisklasse übergebene Argumente
Funktionsdefinition	Funktion in abgeleiteter Klasse, deren Code sich von der Basisklasse unterscheidet
DC-Objekt	"Abgeleitetes Klassenobjekt" Objekt, das den Typ der abgeleiteten Klasse hat

Examples

Grundlagen: das Schlüsselwort 'open'

In Kotlin sind Klassen **standardmäßig final, dh** sie können nicht von übernommen werden.

Um die Vererbung einer Klasse zuzulassen, verwenden Sie das Schlüsselwort `open` .

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

> **Hinweis:** Abstrakte Klassen, versiegelte Klassen und Schnittstellen sind standardmäßig `open` .

Felder von einer Klasse übernehmen

Definieren der Basisklasse:

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

Definieren der abgeleiteten Klasse:

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

Verwendung der Unterklasse:

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

Methoden von einer Klasse übernehmen

Definieren der Basisklasse:

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

Definieren der abgeleiteten Klasse:

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

Der Ninja hat Zugriff auf alle Methoden in Person

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

Eigenschaften und Methoden überschreiben

Überschreibende Eigenschaften (sowohl schreibgeschützt als auch veränderbar):

```
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
}
```

Überschreibende Methoden:

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

object Titanic : Ship {

    var canSail = true

    override fun sail() {
        sink()
    }

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

Klassenvererbung online lesen: https://riptutorial.com/de/kotlin/topic/5622/klassenvererbung

Kapitel 21: Kotlin Android Extensions

Einführung

Kotlin verfügt über eine integrierte View-Injection für Android, mit der manuelle Bindungen übersprungen werden können oder Rahmen wie ButterKnife erforderlich sind. Einige Vorteile sind eine schönere Syntax, eine bessere statische Typisierung und somit weniger fehleranfällig.

Examples

Aufbau

Beginnen Sie mit einem richtig konfigurierten Gradle-Projekt .

In Ihrem **Projekt-local** (nicht auf oberster Ebene) `build.gradle` Erweiterungen anhängen unter Ihrem Kotlin Plugin, auf der obersten Ebene Einrückungsebene Plugin Erklärung.

```
buildscript {
    ...
}

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

Ansichten verwenden

Angenommen, wir haben eine Aktivität mit einem Beispiellayout namens `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>
```

Wir können Kotlin-Erweiterungen verwenden, um die Schaltfläche ohne weitere Bindung aufzurufen.

```
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!")
    }
}
```

Sie können auch alle im Layout angezeigten IDs mit einer `*` -Notation importieren

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

Synthetische Ansichten können nicht außerhalb von Aktivitäten / Fragmenten / Ansichten mit diesem aufgeblasenen Layout verwendet werden:

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

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

Produktaromen

Android-Erweiterungen funktionieren auch mit mehreren Android-Produktvarianten. Zum Beispiel, wenn wir in `build.gradle` haben:

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

Und zum Beispiel hat nur der freie Geschmack einen Buy-Button:

```
<?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>
```

Wir können uns speziell an den Geschmack binden:

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

Ein schmerzhafter Zuhörer, wenn Sie Kenntnis erhalten, wenn die Ansicht jetzt vollständig gezeichnet ist, ist dies mit Kotlins Erweiterung so einfach und fantastisch

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

Unter der Haube

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

Kotlin Android Extensions online lesen: https://riptutorial.com/de/kotlin/topic/9474/kotlin-android-extensions

Kapitel 22: Kotlin für Java-Entwickler

Einführung

Die meisten Leute, die nach Kotlin kommen, haben einen Programmierhintergrund in Java.

In diesem Thema werden Beispiele zum Vergleich von Java und Kotlin gesammelt. Dabei werden die wichtigsten Unterschiede und die von Kotlin angebotenen Juwelen über Java hervorgehoben.

Examples

Variablen deklarieren

In Kotlin sehen Variablendeklarationen ein bisschen anders aus als Javas:

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

- Sie beginnen mit entweder `val` oder `var` , so dass die Erklärung `final` (**„val** ue“) oder **var** iable.
- Der Typ wird nach dem Namen angegeben, getrennt durch ein `:`
- Dank der *Typinferenz* von Kotlin kann die explizite Typdeklaration ignoriert werden, wenn eine Zuordnung mit einem Typ vorliegt, den der Compiler eindeutig erkennen kann

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

Schnelle Fakten

- Kotlin braucht nicht `;` Anweisungen beenden
- Kotlin ist **null-sicher**
- Kotlin ist zu **100% mit Java kompatibel**
- Kotlin hat **keine Grundelemente** (optimiert jedoch, falls möglich, seine Gegenstücke für die JVM)
- Kotlin-Klassen haben **Eigenschaften, keine Felder**
- Kotlin bietet **Datenklassen** mit automatisch generierten `equals` / `hashCode` Methoden und Feldzugriffsfunktionen an
- Kotlin hat nur Laufzeitausnahmen, **keine markierten Ausnahmen**
- Kotlin hat **kein `new` Keyword** . Das Erstellen von Objekten erfolgt einfach durch Aufrufen des Konstruktors wie jede andere Methode.
- Kotlin unterstützt (begrenzte) **Überladung von Operatoren** . Der Zugriff auf einen

Kartenwert kann beispielsweise wie `val a = someMap["key"]` geschrieben werden: `val a = someMap["key"]`
- Kotlin kann nicht nur in Bytecode für die JVM, sondern auch in **Java Script** kompiliert werden, sodass Sie sowohl Backend- als auch Frontend-Code in Kotlin schreiben können
- Kotlin ist **vollständig kompatibel mit Java 6** , was besonders für die Unterstützung von (nicht so) alten Android-Geräten interessant ist
- Kotlin ist eine **offiziell unterstützte** Sprache **für die Android-Entwicklung**
- Kotlins Kollektionen verfügen über eine integrierte Unterscheidung zwischen **veränderlichen und unveränderlichen Sammlungen** .
- Kotlin unterstützt **Coroutines** (experimentell)

Gleichheit & Identität

Kotlin verwendet `==` für Gleichheit (`===` Aufrufe sind intern `equals`) und `===` für referenzielle Identität.

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

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

IF, TRY und andere sind Ausdrücke und keine Anweisungen

In Kotlin sind `if` , `try` und andere Ausdrücke (sie geben also einen Wert zurück) und keine (leeren) Anweisungen.

Kotlin hat zum Beispiel nicht Javas ternären *Elvis Operator* , aber Sie können so etwas schreiben:

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

Noch ungewohnter, aber ebenso ausdrucksstark ist der `try` *Ausdruck* :

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

Kotlin für Java-Entwickler online lesen: https://riptutorial.com/de/kotlin/topic/10099/kotlin-fur-java-entwickler

Kapitel 23: Kotlin Vorsichtsmaßnahmen

Examples

ToString () für einen nullfähigen Typ aufrufen

Bei der Verwendung der `toString` Methode in Kotlin ist die Handhabung von null in Kombination mit dem `String?` .

Sie möchten beispielsweise Text aus einem `EditText` in Android `EditText` .

Sie hätten einen Code wie:

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

Sie würden erwarten, dass der Wert eine leere Zeichenfolge wäre, wenn das Feld nicht vorhanden wäre. In diesem Fall ist es `"null"` .

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

Kotlin Vorsichtsmaßnahmen online lesen: https://riptutorial.com/de/kotlin/topic/6608/kotlin-vorsichtsma-nahmen

Kapitel 24: Kotlin-Build konfigurieren

Examples

Gradle Konfiguration

`kotlin-gradle-plugin` wird verwendet, um Kotlin-Code mit Gradle zu kompilieren. Grundsätzlich sollte seine Version der Kotlin-Version entsprechen, die Sie verwenden möchten. Wenn Sie beispielsweise Kotlin `1.0.3` möchten, müssen Sie auch die Version `1.0.3` `kotlin-gradle-plugin` .

Es `gradle.properties` diese Version in `gradle.properties` oder in `ExtraPropertiesExtension` :

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

   repositories {
     mavenCentral()
   }

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

Dann müssen Sie dieses Plugin auf Ihr Projekt anwenden. Die Art und Weise, wie Sie dies tun, unterscheidet sich bei der Ausrichtung auf verschiedene Plattformen:

Targeting auf JVM

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

Targeting für Android

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

Targeting auf JS

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

Dies sind die Standardpfade:

- Kotlin-Quellen: `src/main/kotlin`
- Java-Quellen: `src/main/java`
- Kotlin-Tests: `src/test/kotlin`
- Java-Tests: `src/test/java`

- Laufzeitressourcen: `src/main/resources`
- Testressourcen: `src/test/resources`

Sie müssen möglicherweise `SourceSets` konfigurieren, wenn Sie ein benutzerdefiniertes Projektlayout verwenden.

Schließlich müssen Sie die Kotlin-Standardbibliothekabhängigkeit zu Ihrem Projekt hinzufügen:

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

Wenn Sie Kotlin Reflection verwenden möchten, müssen Sie auch `compile "org.jetbrains.kotlin:kotlin-reflect:$kotlin_version"`

Android Studio verwenden

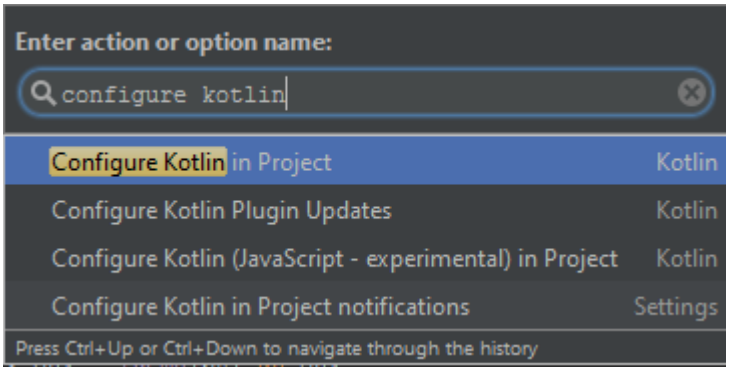
Android Studio kann Kotlin in einem Android-Projekt automatisch konfigurieren.

Installieren Sie das Plugin

Um das Kotlin-Plugin zu installieren, wählen Sie Datei> Einstellungen> Editor> Plugins> JetBrains-Plugin installieren ...> Kotlin> Installieren und starten Sie Android Studio neu, wenn Sie dazu aufgefordert werden.

Projekt konfigurieren

Erstellen Sie ein Android Studio-Projekt wie gewohnt und drücken Sie dann `Strg` + `Umschalttaste` + `A`. Geben Sie im Suchfeld "Configure Kotlin in Project" ein und drücken Sie die Eingabetaste.



Android Studio ändert Ihre Gradle-Dateien, um alle erforderlichen Abhängigkeiten hinzuzufügen.

Java konvertieren

Um Ihre Java-Dateien in Kotlin-Dateien zu konvertieren, drücken Sie `Strg` + `Umschalt` + `A` und suchen Sie nach "Java-Datei in Kotlin-Datei konvertieren". Dadurch wird die Dateierweiterung der aktuellen Datei in `.kt` und der Code in Kotlin konvertiert.

```java
package com.orangeflash81.myapplication;

public class Foo {
    private String name = "Joe Bloggs";

    String getName() { return name; }

    void setName(String value) { name = value; }
}
```

Migration von Gradle mithilfe des Groovy-Skripts zum Kotlin-Skript

Schritte:

- Klonen Sie das Gradle-Script-Kotlin- Projekt
- Kopieren / Einfügen aus dem geklonten Projekt in Ihr Projekt:
    - `build.gradle.kts`
    - `gradlew`
    - `gradlew.bat`
    - `settings.gradle`
- Aktualisieren Sie den Inhalt von `build.gradle.kts` Ihren Anforderungen. Sie können die Skripts im geklonten Projekt oder in einem seiner Beispiele als Inspiration verwenden
- Öffnen Sie jetzt Intellij und öffnen Sie Ihr Projekt. Im Explorer-Fenster sollte es als Gradle-Projekt erkannt werden. Wenn nicht, erweitern Sie es zuerst.
- Lassen Sie Intellij nach dem Öffnen funktionieren, öffnen Sie `build.gradle.kts` und prüfen

Sie, ob ein Fehler vorliegt. Wenn die Hervorhebung nicht funktioniert und / oder alles rot markiert ist, schließen Sie Intellij und öffnen Sie es erneut

- Öffnen Sie das Gradle-Fenster und aktualisieren Sie es

Wenn Sie unter Windows sind, können Sie auf diesen Fehler stoßen, die vollständige Gradle 3.3-Distribution herunterladen und statt der bereitgestellten verwenden. Verwandte

OSX und Ubuntu funktionieren sofort.

Kleiner Bonus, wenn Sie alle Mühen der Veröffentlichung auf Maven und ähnlichem vermeiden möchten, verwenden Sie Jitpack . Die hinzuzufügenden Zeilen sind im Vergleich zu Groovy nahezu identisch. Sie können sich von diesem Projekt von mir inspirieren lassen.

Kotlin-Build konfigurieren online lesen: https://riptutorial.com/de/kotlin/topic/2501/kotlin-build-konfigurieren

Kapitel 25: Null Sicherheit

Examples

Nullable- und Non-Nullable-Typen

Normale Typen wie `String` sind nicht nullfähig. Damit sie in der Lage sind, Nullwerte zu enthalten, muss dies explizit angegeben werden, indem ein `?` dahinter: `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!
```

Safe Call Operator

Um auf Funktionen und Eigenschaften von nullfähigen Typen zuzugreifen, müssen Sie spezielle Operatoren verwenden.

Der erste `?.` gibt Ihnen die Eigenschaft oder Funktion, auf die Sie zugreifen möchten, oder null, wenn das Objekt null ist:

```
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: Aufrufen mehrerer Methoden für dasselbe, auf Null überprüfte Objekt

Eine elegante Methode zum Aufrufen mehrerer Methoden eines nullgeprüften Objekts ist die Verwendung von Kotlins `apply` wie folgt:

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

Dies ruft `foo` und `bar` auf `obj` (was `this` in der `apply` Block) nur dann , wenn `obj` nicht-Null ist, andernfalls den gesamten Block zu überspringen.

Um eine nullfähige Variable als nicht-nullfähige Referenz in den Gültigkeitsbereich zu bringen, ohne sie zum impliziten Empfänger von Funktions- und Eigenschaftsaufrufen zu machen, können Sie `let` statt `apply` :

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

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

`notnull` könnte etwas genannt, oder sogar weggelassen und durch verwenden die impliziten Lambda - Parameter `it` .

Intelligente Besetzungen

Wenn der Compiler feststellen kann, dass ein Objekt an einem bestimmten Punkt nicht null sein kann, müssen Sie keine speziellen Operatoren mehr verwenden:

```
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!
}
```

> **Hinweis:** Mit dem Compiler können Sie keine veränderlichen Variablen für das Umwandeln von Variablen verwenden, die möglicherweise zwischen der Nullprüfung und der beabsichtigten Verwendung geändert werden.
>
> Wenn auf eine Variable außerhalb des Gültigkeitsbereichs des aktuellen Blocks zugegriffen werden kann (z. B. weil sie Mitglieder eines nichtlokalen Objekts ist), müssen Sie eine neue lokale Referenz erstellen, die Sie anschließend mit einem Cast umsetzen und verwenden können.

Beseitigen Sie Nullen aus einem Iterable-Array und einem Array

Manchmal müssen wir den Typ von `Collection<T?>` `Collections<T>` ändern. In diesem Fall ist `filterNotNull` unsere Lösung.

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

Null Coalescing / Elvis Operator

Manchmal ist es wünschenswert, einen auswertbaren Ausdruck auf eine wenn-andere Weise auszuwerten. Der elvis-Operator `?:` Kann für eine solche Situation in Kotlin verwendet werden.

Zum Beispiel:

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

Der obige Ausdruck gibt `"Nothing here"` wenn `data?.first()` oder `data` selbst einen `null` , andernfalls das Ergebnis von `data?.first()`

Es ist auch möglich, Ausnahmen mit derselben Syntax auszulösen, um die Codeausführung

abzubrechen.

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

> Erinnerung: Mit dem Assertion-Operator können NullPointerExceptions ausgelöst werden (zB `data!!.second()!!`)

Behauptung

`!!` Suffixe ignorieren die Nullfähigkeit und geben eine nicht-null-Version dieses Typs zurück. `KotlinNullPointerException` wird ausgelöst, wenn das Objekt eine `null` .

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

Elvis Operator (? :)

In Kotlin können wir eine Variable deklarieren, die eine `null reference` . Angenommen, wir haben eine nullfähige Referenz `a` , wir können sagen: "Wenn `a` nicht null ist, verwenden Sie es, andernfalls verwenden Sie einen Nicht-Nullwert `x` ".

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

Nun, `a` kann null sein. Wenn wir also auf den Wert von `a` zugreifen müssen, müssen wir eine Sicherheitsüberprüfung durchführen, ob der Wert einen Wert enthält oder nicht. Wir können diese Sicherheitsüberprüfung mit einer konventionellen `if...else` Anweisung durchführen.

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

Aber hier kommt Voraus Operator `Elvis` (Operator Elvis: `?:`). Oben `if...else` kann mit dem Elvis-Operator wie folgt ausgedrückt werden:

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

Wenn der Ausdruck links von `?:` (Hier: `a?.length`) nicht null ist, gibt der elvis-Operator ihn zurück, andernfalls den Ausdruck nach rechts (hier: `-1`). Der Ausdruck auf der rechten Seite wird nur ausgewertet, wenn die linke Seite null ist.

Null Sicherheit online lesen: https://riptutorial.com/de/kotlin/topic/2080/null-sicherheit

Kapitel 26: Protokollierung in Kotlin

Bemerkungen

Zugehörige Frage: Idiomatische Art der Protokollierung in Kotlin

Examples

kotlin.logging

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

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

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

Kotlin.logging- Framework verwenden

Protokollierung in Kotlin online lesen: https://riptutorial.com/de/kotlin/topic/3258/protokollierung-in-kotlin

Kapitel 27: RecyclerView in Kotlin

Einführung

Ich möchte nur mein bisschen Wissen und Code von RecyclerView mit Kotlin teilen.

Examples

Hauptklasse und Adapter

Ich gehe davon aus, dass Sie über die Syntax von **Kotlin** und deren Verwendung **Bescheid** wissen. **Fügen Sie** einfach **RecyclerView** in die Datei **activity_main.xml hinzu** und setzen Sie die Adapterklasse ein.

```
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
        }
    }
```

dies ist Ihre **Adapterklasse** Recycler Ansicht und **main_item.xml** - Datei erstellen , was Sie wollen

```
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)
            }
        }
    }
}
```

RecyclerView in Kotlin online lesen: https://riptutorial.com/de/kotlin/topic/10143/recyclerview-in-kotlin

Kapitel 28: Redewendungen

Examples

Erstellen von DTOs (POJOs / POCOs)

Datenklassen in Kotlin sind Klassen, die nur zum Halten von Daten erstellt wurden. Solche Klassen werden als `data` markiert:

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

Der obige Code erstellt eine `User` Klasse mit der folgenden automatischen Generierung:

- Getter und Setter für alle Eigenschaften (Getter nur für `val`)
- `equals()`
- `hashcode()`
- `toString()`
- `copy()`
- `componentN()` (wobei `N` die entsprechende Eigenschaft in der Reihenfolge der Deklaration ist)

Wie bei einer Funktion können auch Standardwerte angegeben werden:

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

Weitere Details finden Sie hier Datenklassen .

Eine Liste filtern

```
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]
```

Delegieren Sie an eine Klasse, ohne diese im öffentlichen Konstruktor anzugeben

Angenommen, Sie möchten an eine Klasse delegieren, aber die an Klasse delegierte Klasse nicht im Konstruktorparameter angeben. Stattdessen möchten Sie es privat konstruieren, sodass der Konstruktor-Aufrufer es nicht bemerkt. Zunächst erscheint dies möglicherweise als unmöglich, da die Delegierung von Klassen nur die Übergabe an Konstruktorparameter zulässt. Es gibt jedoch eine Möglichkeit, wie in dieser Antwort beschrieben :

```
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

}
```

Damit können Sie den Konstruktor von `MyTable` einfach so aufrufen: `MyTable()` . Die `Table<Int, Int, Int>` für die die `MyTable` Delegaten privat erstellt werden. Konstrukteur Anrufer weiß nichts davon.

Dieses Beispiel basiert auf dieser SO-Frage .

Serializable und serialVersionUid in Kotlin

Um die `serialVersionUID` für eine Klasse in Kotlin zu erstellen, `serialVersionUID` Ihnen einige Optionen zur Verfügung, die alle das Hinzufügen eines Members zum Companion-Objekt der Klasse umfassen.

Der prägnanteste Bytecode stammt aus einem `private const val` von `private const val` , der in der `MySpecialCase` Klasse eine private statische Variable wird, in diesem Fall `MySpecialCase` :

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

Sie können diese Formulare auch verwenden, wobei es **jeweils einen Nebeneffekt auf Getter / Setter-Methoden** gibt, die für die Serialisierung nicht erforderlich sind.

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

Dadurch wird das statische Feld erstellt, es wird jedoch auch ein Getter für `getSerialVersionUID` für das Begleitobjekt erstellt, was nicht `getSerialVersionUID` ist.

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

Dadurch wird das statische Feld erstellt, aber auch ein statischer Getter für `getSerialVersionUID` für die enthaltende Klasse `MySpecialCase` . `MySpecialCase` ist nicht `MySpecialCase` .

Alle funktionieren jedoch als Methode zum Hinzufügen der `serialVersionUID` zu einer `Serializable` Klasse.

Fließende Methoden in Kotlin

Fließende Methoden in Kotlin können mit Java identisch sein:

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

Sie können sie jedoch auch funktionaler machen, indem Sie eine Erweiterungsfunktion erstellen, z.

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

Das erlaubt dann offensichtlich offensichtlich fließende Funktionen:

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

Verwenden Sie let oder auch, um die Arbeit mit nullfähigen Objekten zu vereinfachen

`let` in Kotlin wird aus dem Objekt, zu dem es aufgerufen wurde, eine lokale Bindung erstellt. Beispiel:

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

Dadurch wird `"foo"` gedruckt und `Unit` .

Der Unterschied zwischen `let` und `also` ist , *dass Sie* einen *beliebigen Wert von einer zurückgeben kann `let` Block. `also` in der anderen Hand wird die `Unit` immer reutrn.*

Warum ist das nützlich, fragen Sie? Denn wenn Sie eine Methode aufrufen, die `null` und Sie nur Code ausführen möchten, wenn der Rückgabewert nicht `null` , können Sie `let` oder `also` so verwenden:

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

Dieser Code führt den `let` Block nur aus `let` wenn `str` nicht `null` . Beachten Sie den `null` Sicherheitsoperator (`?`).

Verwenden Sie "Apply", um Objekte zu initialisieren oder eine

Methodenverkettung zu erreichen

Die Dokumentation von `apply` lautet wie folgt:

> ruft den angegebenen Funktionsblock mit `this` Wert als Empfänger auf und gibt `this` Wert zurück.

Während das kdoc nicht so hilfreich ist `apply` ist das Anwenden tatsächlich eine nützliche Funktion. In den Begriffen von layman gilt `apply` , in welchem Umfang `this` an das Objekt gebunden ist, für das Sie `apply` . Auf diese Weise können Sie Code sparen, wenn Sie mehrere Methoden für ein Objekt aufrufen müssen, die Sie später zurückgeben. Beispiel:

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

Dies ist das gleiche wie das Schreiben:

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

Redewendungen online lesen: https://riptutorial.com/de/kotlin/topic/2273/redewendungen

Kapitel 29: Reflexion

Einführung

Reflection ist die Fähigkeit einer Sprache, zur Laufzeit anstelle der Kompilierzeit Code zu prüfen.

Bemerkungen

Reflection ist ein Mechanismus, um Sprachkonstrukte (Klassen und Funktionen) zur Laufzeit zu untersuchen.

Beim Targeting auf die JVM-Plattform werden Laufzeitreflexionsfunktionen in separaten JAR `kotlin-reflect.jar` verteilt: `kotlin-reflect.jar` . Dies geschieht, um die Laufzeitgröße zu reduzieren, nicht benötigte Funktionen zu reduzieren und es möglich zu machen, auf andere Plattformen (wie JS) zu zielen.

Examples

Eine Klasse referenzieren

Um einen Verweis auf ein `KClass` Objekt zu erhalten, das eine Klasse darstellt, verwenden Sie Doppelpunkte:

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

Funktion referenzieren

Funktionen sind erstklassige Bürger in Kotlin. Sie können einen Verweis darauf mit Doppelpunkten erhalten und dann an eine andere Funktion übergeben:

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

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

Interaktion mit Java Reflection

Um ein Java- `Class` Objekt von `KClass` verwenden Sie die Erweiterungseigenschaft `.java` :

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

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

Das letztere Beispiel wird vom Compiler `KClass` optimiert, dass keine `KClass` Zwischeninstanz `KClass` .

Werte aller Eigenschaften einer Klasse abrufen

`BaseExample Example` , die die `BaseExample` Klasse mit einigen Eigenschaften erweitert:

```
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"
}
```

Man kann alle Eigenschaften einer Klasse erhalten:

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

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

Wenn Sie diesen Code ausführen, wird eine Ausnahme ausgelöst. Die Eigenschaft `private val privateField` wird als privat deklariert, und der Aufruf von `member.get(example)` wird nicht erfolgreich durchgeführt. Eine Möglichkeit, damit umzugehen, um private Eigenschaften herauszufiltern. Dazu müssen wir den Sichtbarkeits-Modifizierer des Java-Getters einer Eigenschaft überprüfen. Im Falle von `private val` der Getter nicht, so dass wir einen privaten Zugriff annehmen können.

Die Hilfsfunktion und ihre Verwendung könnte folgendermaßen aussehen:

```
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)}")
}
```

Ein anderer Ansatz besteht darin, private Immobilien über Reflexion zugänglich zu machen:

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

Einstellungswerte aller Eigenschaften einer Klasse

Als Beispiel möchten wir alle String-Eigenschaften einer Beispielklasse festlegen

```
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
        }
    }
}
```

Das Abrufen von veränderbaren Eigenschaften baut auf dem Abrufen aller Eigenschaften auf, wobei die veränderbaren Eigenschaften nach Typ gefiltert werden. Wir müssen auch die Sichtbarkeit prüfen, da das Lesen privater Eigenschaften zu Laufzeitausnahmen führt.

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

Um alle `String` Eigenschaften auf `"Our Value"` , können wir zusätzlich nach dem Rückgabetyp filtern. Da Kotlin auf Java VM basiert, ist Type Erasure in Kraft und daher sind Eigenschaften, die generische Typen wie `List<String>` , mit `List<Any>` identisch. Leider ist das Nachdenken keine goldene Kugel, und es gibt keinen vernünftigen Weg, dies zu vermeiden. Daher müssen Sie in Ihren Anwendungsfällen aufpassen.

```
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")
        }
```

Reflexion online lesen: https://riptutorial.com/de/kotlin/topic/2402/reflexion

Kapitel 30: Regex

Examples

Idiome für Regex-Abgleich in When-Ausdruck

Verwenden unveränderlicher Einheimischer:

Verwendet weniger horizontalen Raum, aber mehr vertikalen Raum als die Vorlage "Anonyme temporäre". Vorzuziehen gegenüber der Vorlage "anonyme temporäre", wenn sich der `when` Ausdruck in einer Schleife befindet. In diesem Fall sollten Regex-Definitionen außerhalb der Schleife platziert werden.

```
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 */
}
```

Verwendung anonymer Provisorien:

Verwendet weniger vertikalen Raum, aber mehr horizontalen Raum als die Vorlage "unveränderliche Einheimische". Sollte nicht verwendet werden, `when` Ausdruck in einer Schleife befindet.

```
import kotlin.text.regex

var string = /* some string */

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

Verwenden des Besuchermusters:

Hat den Vorteil, die "argument-ful" `when` Syntax genau zu emulieren. Dies ist vorteilhaft, da es

klarer das Argument des `when` Ausdrucks anzeigt und bestimmte Programmiererfehler ausschließt, die daraus `whenEntry` könnten, dass das `when` Argument in jedem `whenEntry` wiederholt werden `whenEntry` . Bei dieser Implementierung kann entweder das "unveränderliche Einheimische" oder das "anonyme temporäre" Template verwendet werden.

```
import kotlin.text.regex

var string = /* some string */

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

Und die minimale Definition der Wrapper-Klasse für das Argument " `when` :

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

Einführung in reguläre Ausdrücke in Kotlin

In diesem Beitrag wird gezeigt, wie die meisten Funktionen in der `Regex` Klasse verwendet werden, wie mit Nullen gearbeitet wird, die sicher mit den `Regex` Funktionen zusammenhängen, und wie rohe Zeichenfolgen das Schreiben und Lesen von Regex-Mustern erleichtern.

Die RegEx-Klasse

Um mit regulären Ausdrücken in Kotlin zu arbeiten, müssen Sie die Klasse `Regex(pattern: String)` und Funktionen wie `find(..)` oder `replace(..)` für dieses Regex-Objekt aufrufen.

Ein Beispiel zur Verwendung der `Regex` Klasse, die true zurückgibt, wenn die `input` c oder d enthält:

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

Für alle `Regex` Funktionen ist es wichtig zu verstehen, dass das Ergebnis auf dem Abgleich des Regex- `pattern` und der `input` basiert. Einige Funktionen erfordern eine vollständige Übereinstimmung, während die restlichen Funktionen nur eine teilweise Übereinstimmung erfordern. Die in diesem Beispiel verwendete Funktion `containsMatchIn(..)` erfordert eine teilweise Übereinstimmung und wird später in diesem Beitrag erläutert.

Null Sicherheit mit regulären Ausdrücken

`find(..)` und `matchEntire(..) MatchResult?` ein `MatchResult?` Objekt. Die `?` Ein Zeichen nach

`MatchResult` ist für Kotlin erforderlich, um Null sicher zu behandeln.

Ein Beispiel, das zeigt, wie Kotlin mit einer `Regex` Funktion sicher mit Null `Regex` , wenn die `find(..)` Funktion Null zurückgibt:

```
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. => ""
```

Mit der Funktion `orEmpty()` kann `b` nicht null sein und das `?` Ein Zeichen ist nicht erforderlich, wenn Sie Funktionen über `b` aufrufen.

Wenn Sie sich nicht für den sicheren Umgang mit Nullwerten interessieren, können Sie mit Kotlin mit Nullwerten wie in Java mit dem `!!` Zeichen:

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

Raw-Strings in Regex-Mustern

Kotlin bietet eine Verbesserung gegenüber Java mit einem Raw-String , der das Schreiben reiner Regex-Muster ohne doppelte Backslashes ermöglicht, die für einen Java-String erforderlich sind. Eine rohe Zeichenfolge wird mit einem dreifachen Anführungszeichen dargestellt:

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

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

Die `input` wird mit dem `pattern` im `Regex` Objekt abgeglichen. `Matchresult?` ein `Matchresult?` Objekt mit dem ersten übereinstimmenden Text nach dem `startIndex` oder `null` wenn das Muster nicht mit der `input` übereinstimmt. Die Ergebniszeichenfolge wird vom `MatchResult?` abgerufen `MatchResult? value` Eigenschaft des Objekts. Der Parameter `startIndex` ist optional mit dem Standardwert 0.

So extrahieren Sie die erste gültige Telefonnummer aus einer Zeichenfolge mit Kontaktdaten:

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

Ohne gültige Telefonnummer in der `input` ist die Variable `phoneNumber null` .

findAll (Eingabe: CharSequence, StartIndex: Int): Sequenz

Gibt alle Übereinstimmungen aus der `input` , die dem Regex- `pattern` .

So drucken Sie alle mit Leerzeichen getrennten Zahlen aus einem Text mit Buchstaben und Ziffern:

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

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

Die `matchedResults` Variable ist eine Sequenz mit `MatchResult` Objekten. Bei einer `input` ohne Ziffern gibt die `findAll(..)` Funktion eine leere Sequenz zurück.

matchEntire (Eingabe: CharSequence): MatchResult?

Wenn alle Zeichen in der `input` dem Regex- `pattern` , wird eine Zeichenfolge zurückgegeben, die der `input` . Andernfalls wird `null` zurückgegeben.

Gibt die Eingabezeichenfolge zurück, wenn die gesamte Eingabezeichenfolge eine Zahl ist:

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

match (Eingabe: CharSequence): Boolean

Gibt "true" zurück, wenn die gesamte Eingabezeichenfolge dem Regex-Muster entspricht. Sonst falsch.

Testet, ob zwei Zeichenfolgen nur Ziffern enthalten:

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

containsMatchIn (Eingabe: CharSequence):

Boolean

Gibt "true" zurück, wenn ein Teil der Eingabezeichenfolge dem Regex-Muster entspricht. Sonst falsch.

Testen Sie, ob zwei Zeichenfolgen mindestens eine Ziffer enthalten:

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

split (Eingabe: CharSequence, Limit: Int): Liste

Gibt eine neue Liste ohne Übereinstimmungen mit den regulären Ausdrücken zurück.

Listen ohne Ziffern zurückgeben:

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

Für jeden Split gibt es ein Element in der Liste. Die erste `input` hat drei Zahlen. Daraus ergibt sich eine Liste mit drei Elementen.

replace (Eingabe: CharSequence, Ersatz: String): String

Ersetzt alle Übereinstimmungen des Regex- `pattern` in der `input` durch die Ersetzungszeichenfolge.

So ersetzen Sie alle Ziffern in einer Zeichenfolge durch ein x:

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

Regex online lesen: https://riptutorial.com/de/kotlin/topic/8364/regex

Kapitel 31: Sammlungen

Einführung

Im Gegensatz zu vielen Sprachen unterscheidet Kotlin zwischen veränderlichen und unveränderlichen Sammlungen (Listen, Mengen, Karten usw.). Die genaue Kontrolle, wann genau Sammlungen bearbeitet werden können, ist hilfreich, um Fehler zu beseitigen und gute APIs zu entwerfen.

Syntax

- listOf, mapOf und setOf gibt schreibgeschützte Objekte zurück, die Sie nicht hinzufügen oder entfernen können.
- Wenn Sie Elemente hinzufügen oder entfernen möchten, müssen Sie arrayListOf, hashMapOf, hashSetOf, linkedMapOf (LinkedHashMap), linkedSetOf (LinkedHashSet), mutableListOf (Sammlung Kotlin MultableList), mutableMapOf (The Kotlin MultableMap) verwenden), sortiertMapOf oder sortiertSetOf
- Jede Sammlung verfügt über Methoden wie first (), last (), get () und Lambda-Funktionen wie Filtern, Zuordnen, Verknüpfen, Reduzieren und viele andere.

Examples

Liste verwenden

```
// 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]"
```

Karte verwenden

```
// 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}"
```

Set verwenden

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

Sammlungen online lesen: https://riptutorial.com/de/kotlin/topic/8846/sammlungen

Kapitel 32: Schleifen in Kotlin

Bemerkungen

In Kotlin werden Loops, wo immer möglich, zu optimierten Loops zusammengestellt. Wenn Sie beispielsweise einen Zahlenbereich durchlaufen, wird der Bytecode basierend auf einfachen int-Werten in eine entsprechende Schleife herunterkompiliert, um den Aufwand für die Objekterstellung zu vermeiden.

Examples

Wiederholen Sie eine Aktion x-mal

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

Schleifen über iterables

Sie können jedes iterable durchlaufen, indem Sie die standardmäßige for-Schleife verwenden:

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

Viele Dinge in Kotlin sind wiederholbar, wie Zahlenbereiche:

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

Wenn Sie während der Iteration einen Index benötigen:

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

Es gibt auch einen funktionalen Ansatz zum Durchlaufen der Standardbibliothek ohne ersichtliche Sprachkonstrukte mit der Funktion forEach:

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

`it` in diesem Beispiel implizit das aktuelle Element, siehe Lambda-Funktionen

Während Schleifen

While- und Do-While-Loops funktionieren wie in anderen Sprachen:

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

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

In der do-while-Schleife hat der Bedingungsblock Zugriff auf Werte und Variablen, die im Schleifenkörper deklariert sind.

Pause und weiter

Break- und Continue-Keywords funktionieren wie in anderen Sprachen.

```
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
    }
}
```

Wenn Sie geschachtelte Schleifen haben, können Sie die Schleifenanweisungen benennen und die Anweisungen break und continue qualifizieren, um anzugeben, welche Schleife Sie fortsetzen oder unterbrechen möchten:

```
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
    }
}
```

Dieser Ansatz funktioniert jedoch nicht für das funktionale `forEach` Konstrukt.

Iteration über eine Karte in Kotlin

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

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

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

Rekursion

In Kotlin ist wie in den meisten Programmiersprachen auch eine Schleife durch Rekursion möglich.

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

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

Im obigen Beispiel wird die `factorial` wiederholt von selbst aufgerufen, bis die angegebene Bedingung erfüllt ist.

Funktionale Konstrukte für die Iteration

Die Kotlin Standard Library bietet außerdem zahlreiche nützliche Funktionen für das iterative Arbeiten mit Sammlungen.

Mit der `map` kann beispielsweise eine Liste von Elementen umgewandelt werden.

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

Einer der vielen Vorteile dieses Stils ist es, Operationen auf ähnliche Weise zu verketten. Es wäre nur eine geringfügige Änderung erforderlich, wenn beispielsweise die Liste nach geraden Zahlen gefiltert werden müsste. Die `filter` kann verwendet werden.

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

Schleifen in Kotlin online lesen: https://riptutorial.com/de/kotlin/topic/2727/schleifen-in-kotlin

Kapitel 33: Schnittstellen

Bemerkungen

Siehe auch: Kotlin-Referenzdokumentation für Interfaces: Interfaces

Examples

Grundlegende Schnittstelle

Eine Kotlin-Schnittstelle enthält Deklarationen abstrakter Methoden und Standardmethodenimplementierungen, obwohl sie keinen Status speichern können.

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

Diese Schnittstelle kann nun wie folgt von einer Klasse implementiert werden:

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

Schnittstelle mit Standardimplementierungen

Eine Schnittstelle in Kotlin kann Standardimplementierungen für Funktionen haben:

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

Klassen, die solche Schnittstellen implementieren, können diese Funktionen ohne erneute Implementierung verwenden

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

Eigenschaften

Standardimplementierungen funktionieren auch für Property-Getter und -Setter:

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

Implementierungen von Schnittstellen-Accessoren können keine Sicherungsfelder verwenden

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

Mehrere Implementierungen

Wenn mehrere Schnittstellen dieselbe Funktion implementieren oder alle mit einer oder mehreren implementieren, muss die abgeleitete Klasse den richtigen Aufruf manuell auflösen

```
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")
    }
}
```

Eigenschaften in Schnittstellen

Sie können Eigenschaften in Schnittstellen deklarieren. Da eine Schnittstelle keinen Status haben kann, können Sie eine Eigenschaft nur als abstrakt deklarieren oder eine Standardimplementierung für die Zugriffsmethoden bereitstellen.

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

    val propertyWithImplementation: String
        get() = "foo"

    fun foo() {
        print(property)
    }
}

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

Konflikte bei der Implementierung mehrerer Schnittstellen mit Standardimplementierungen

Wenn Sie mehr als eine Schnittstelle mit gleichnamigen Methoden implementieren, die Standardimplementierungen enthalten, ist für den Compiler nicht eindeutig, welche Implementierung verwendet werden soll. Im Falle eines Konflikts muss der Entwickler die in Konflikt stehende Methode überschreiben und eine benutzerdefinierte Implementierung bereitstellen. Diese Implementierung kann sich dafür entscheiden, an die Standardimplementierungen zu delegieren oder nicht.

```
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 Keyword

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

Wenn die Methode in der Schnittstelle über eine eigene Standardimplementierung verfügt, können

Sie mit dem Schlüsselwort super auf sie zugreifen.

```
super.funcOne()
```

Schnittstellen online lesen: https://riptutorial.com/de/kotlin/topic/900/schnittstellen

Kapitel 34: Sichtbarkeitsmodifikatoren

Einführung

In Kotlin stehen 4 verschiedene Sichtbarkeitsmodifizierer zur Verfügung.

Öffentlich: Auf diesen kann von überall aus zugegriffen werden.

Privat: Auf diesen kann nur vom Modulcode aus zugegriffen werden.

Geschützt: Der Zugriff ist nur über die definierende Klasse und abgeleitete Klassen möglich.

Intern: Auf diesen kann nur aus dem Geltungsbereich der Klasse zugegriffen werden, die sie definiert.

Syntax

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

Examples

Code-Beispiel

Öffentlich: `public val name = "Avijit"`

Privat: `private val name = "Avijit"`

Geschützt: `protected val name = "Avijit"`

Intern: `internal val name = "Avijit"`

Sichtbarkeitsmodifikatoren online lesen:
https://riptutorial.com/de/kotlin/topic/10019/sichtbarkeitsmodifikatoren

Kapitel 35: Singleton-Objekte

Einführung

Ein *Objekt* ist eine spezielle Klasse, die mit `object` deklariert werden kann. Objekte ähneln Singletons (einem Designmuster) in Java. Es funktioniert auch als statischer Teil von Java. Anfänger, die von Java zu Kotlin wechseln, können diese Funktion anstelle von statischen oder Singletons verwenden.

Examples

Verwendung als Repalcement von statischen Methoden / Java-Feldern

```
object CommonUtils {

    var anyname: String ="Hello"

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

Rufen Sie aus jeder anderen Klasse die Variable und die Funktionen auf folgende Weise auf:

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

Verwenden Sie als Singleton

Kotlin-Objekte sind eigentlich nur Singletons. Sein Hauptvorteil besteht darin, dass Sie `SomeSingleton.INSTANCE` nicht verwenden `SomeSingleton.INSTANCE` , um die Instanz des Singleton zu erhalten.

In Java sieht Ihr Singleton so aus:

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

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

In Kotlin lautet der entsprechende Code

```
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")
}
```

Es ist obvoiusly weniger ausführlich zu verwenden.

Singleton-Objekte online lesen: https://riptutorial.com/de/kotlin/topic/10152/singleton-objekte

Kapitel 36: Typensichere Builder

Bemerkungen

Ein *typsicherer Builder* ist ein Konzept und keine Sprachfunktion. Daher ist er nicht streng formalisiert.

Eine typische Struktur eines typsicheren Builders

Eine einzelne Builder-Funktion besteht normalerweise aus 3 Schritten:

1. Erstellen Sie ein Objekt.
2. Führen Sie Lambda aus, um das Objekt zu initialisieren.
3. Fügen Sie das Objekt zur Struktur hinzu oder geben Sie es zurück.

Typensichere Builder in Kotlin-Bibliotheken

Das Konzept von typsicheren Buildern ist in einigen Kotlin-Bibliotheken und -Frameworks weit verbreitet, z. B .:

- Anko
- Wasabi
- Ktor
- Spec

Examples

Builder für typsichere Baumstruktur

Builder können als eine Reihe von Erweiterungsfunktionen definiert werden, die Lambda-Ausdrücke mit Empfängern als Argumente verwenden. In diesem Beispiel wird ein Menü eines `JFrame` :

```
import javax.swing.*

fun JFrame.menuBar(init: JMenuBar.() -> Unit) {
    val menuBar = JMenuBar()
    menuBar.init()
    setJMenuBar(menuBar)
}

fun JMenuBar.menu(caption: String, init: JMenu.() -> Unit) {
    val menu = JMenu(caption)
    menu.init()
    add(menu)
}
```

```
fun JMenu.menuItem(caption: String, init: JMenuItem.() -> Unit) {
    val menuItem = JMenuItem(caption)
    menuItem.init()
    add(menuItem)
}
```

Mit diesen Funktionen können Sie auf einfache Weise eine Baumstruktur von Objekten erstellen:

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

Typensichere Builder online lesen: https://riptutorial.com/de/kotlin/topic/6010/typensichere-builder

Kapitel 37: Vararg-Parameter in Funktionen

Syntax

- **Vararg-Schlüsselwort** : `vararg` wird in einer Methodendeklaration verwendet, um anzuzeigen, dass eine variable Anzahl von Parametern akzeptiert wird.
- **Spread-Operator** : Ein Sternchen (`*`) vor einem Array, das in Funktionsaufrufen verwendet wird, um den Inhalt in einzelne Parameter zu "entfalten".

Examples

Grundlagen: Verwenden des Schlüsselworts vararg

Definieren Sie die Funktion mit dem Schlüsselwort `vararg` .

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

Jetzt können Sie beliebig viele Parameter (vom richtigen Typ) in die Funktion eingeben.

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

> **Hinweise:** Vararg-Parameter *müssen* der letzte Parameter in der Parameterliste sein.

Spread Operator: Übergabe von Arrays an Vararg-Funktionen

Arrays können mit dem **Spread-Operator** `*` in Vararg-Funktionen übergeben werden.

Angenommen, die folgende Funktion existiert ...

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

Sie können **ein Array** an die Funktion übergeben ...

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

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

Der Spread-Operator kann auch **in der Mitte** der Parameter verwendet werden ...

```
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)
```

Vararg-Parameter in Funktionen online lesen: https://riptutorial.com/de/kotlin/topic/5835/vararg-parameter-in-funktionen

Kapitel 38: Zeichenketten

Examples

Elemente von String

Elemente von String sind Zeichen, auf die mit der Indexierungsoperationszeichenfolge `string[index]` zugegriffen werden kann.

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

String-Elemente können mit einer for-Schleife durchlaufen werden.

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

String Literals

Kotlin hat zwei Arten von String-Literalen:

- Fluchtzeichenfolge
- Roher String

Bei einer Zeichenfolge mit Escapezeichen werden Sonderzeichen durch Escapezeichen behandelt. Die Flucht erfolgt mit einem Backslash. Die folgenden Escape-Sequenzen werden unterstützt: `\t` , `\b` , `\n` , `\r` , `\'` , `\"` , `\\` und `\$` . Verwenden Sie zur Kodierung anderer Zeichen die Unicode-Escape-Sequenz-Syntax: `\uFF00` .

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

Unformatierte Zeichenfolge , die durch ein dreifaches Anführungszeichen `"""` , enthält keine Escapezeichen und kann Zeilenumbrüche und andere Zeichen enthalten

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

Führende Leerzeichen können mit der trimMargin () - Funktion entfernt werden.

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

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

Das Standard-Randpräfix ist Pipe-Zeichen `|` kann dies als Parameter für trimMargin festgelegt werden; zB `trimMargin(">")` .

String-Vorlagen

Sowohl mit Escapezeichen versehene Zeichenfolgen als auch unformatierte Zeichenfolgen können Vorlagenausdrücke enthalten. Der Template-Ausdruck ist ein Stück Code, das ausgewertet wird und das Ergebnis in einer Zeichenfolge verkettet wird. Es beginnt mit einem Dollarzeichen `$` und besteht entweder aus einem Variablennamen:

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

Oder ein beliebiger Ausdruck in geschweiften Klammern:

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

Um ein buchstäbliches Dollarzeichen in eine Zeichenfolge einzufügen, können Sie es mit einem Backslash umgehen

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

Die Ausnahme sind rohe Zeichenfolgen, die kein Escaping unterstützen. In rohen Zeichenfolgen können Sie die folgende Syntax verwenden, um ein Dollarzeichen darzustellen.

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

String Gleichheit

In Kotlin werden Strings mit dem Operator `==` verglichen, der die strukturelle Gleichheit berücksichtigt.

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

Die referenzielle Gleichheit wird mit `===` Operator `===` geprüft.

```
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
```

Zeichenketten online lesen: https://riptutorial.com/de/kotlin/topic/8285/zeichenketten

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit 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	Anmerkungen	Brad Larson, Caelum, Héctor, Mood, piotrek1543, Sapan Zaveri
3	Arrays	egor.zhdan, Sam, UnKnown
4	Ausnahmen	Brad Larson, jereksel, Sapan Zaveri
5	Bedingte Anweisungen	Abdullah, Alex Facciorusso, jpmcosta, Kirill Rakhman, Robin, Spidfire
6	Bereiche	Nihal Saxena
7	Coroutinen	Jemo Mgebrishvili
8	Delegierte Eigenschaften	Sam, Seaskyways
9	DSL-Gebäude	Dmitriy L, ice1000
10	Enum	David Soroko, Kirill Rakhman, SerCe
11	Erweiterungsmethoden	Dávid Tímár, Jayson Minard, Kevin Robatel, Konrad Jamrozik, olivierlemasle, Parker Hoyes, razzledazzle
12	Funktionen	Aaron Christiansen, baha, Caelum, glee8e, Jayson Minard, KeksArmee, madhead, Spidfire
13	Geben Sie Aliase ein	Kevin Robatel
14	Generics	hotkey, Jayson Minard, KeksArmee
15	Grundlagen von Kotlin	Shinoo Goyal
16	Grundlegende Lambdas	memoizr, Rich Kuzsma
17	Java 8-Stream-Entsprechungen	Brad, Gerson, Jayson Minard, Piero Divasto, Sam
18	JUnit	jenglert

19	Klassendelegation	Sam
20	Klassenvererbung	byxor, KeksArmee, piotrek1543, Slav
21	Kotlin Android Extensions	Jemo Mgebrishvili, Ritave
22	Kotlin für Java-Entwickler	Thorsten Schleinzer
23	Kotlin Vorsichtsmaßnahmen	Grigory Konushev, Spidfire
24	Kotlin-Build konfigurieren	Aaron Christiansen, elect, madhead
25	Null Sicherheit	KeksArmee, Kirill Rakhman, piotrek1543, razzledazzle, Robin, SerCe, Spidfire, technerd, Thorsten Schleinzer
26	Protokollierung in Kotlin	Konrad Jamrozik, olivierlemasle, oshai
27	RecyclerView in Kotlin	Mohit Suthar
28	Redewendungen	Aaron Christiansen, Adam Arold, Brad Larson, Héctor, Jayson Minard, Konrad Jamrozik, madhead, mayojava, razzledazzle, Sapan Zaveri, Serge Nikitin, yole
29	Reflexion	atok, Kirill Rakhman, madhead, Ritave, Sup
30	Regex	Espen, Travis
31	Sammlungen	Ascension
32	Schleifen in Kotlin	Ben Leggiero, JaseAnderson, mayojava, razzledazzle, Robin
33	Schnittstellen	Divya, Jan Vladimir Mostert, Jayson Minard, Ritave, Robin
34	Sichtbarkeitsmodifikatoren	Avijit Karmakar
35	Singleton-Objekte	Divya, glee8e
36	Typensichere Builder	Slav
37	Vararg-Parameter in Funktionen	byxor, piotrek1543, Sam
38	Zeichenketten	Januson, Sam