Kotlin – Std. libs

or how one-liners can change the world

Standart.kt

Standart.kt ist Teil de Kotlin Standard Bibliothek und definiert dabei eine eigene wesentliche Funktion. Wirklich bemerkenswert ist die Tatsache, dass die Bibliothek gerade mal **134** Zeilen lang ist, und die meisten enthaltenen Funktionen Ein-Zeiler ist sind. Es gibt insgesamt **10**.

TODO

```
public inline fun TODO(): Nothing = throw NotImplementedError()
```

Jeder kennt folgendes:

```
// TODO blablaba
```

und dann ein paar Stunden oder Tage später..... Oh xxxxx da war ja noch ein TODO, keine Wunder das mein Test nicht funktioniert.

Deshalb gibt es in der Kotlin Std. Lib zwei Funktionen TODO() und TODO(reason: String) welche immer eine NotYetImplemented Exception zurückgegben.

Let

```
fun <T, R> T.let(f: (T) -> R): R = f(this)
```

Let() ist eine scoping Funktion.: Verwendet wird sie dann, wann immer eine Variable für einen bestimmten Scope generiert werden soll, aber nicht darüber hinaus.

Beispiel:

```
DbConnection.getConnection().let { connection ->
}
// connection is no longer visible here
```

Let - Anhang

Let() kann auch als Alternative gegen einen Null-Check verwendet werden

```
val map : Map<String, String> = ...
val config = map[key]
// config is a "Config?"
config?.let {
    //  Dieser ganze Block wird nicht ausgeführt wenn „config“ null ist.
    // Zusätzlich wird die Variable automatisch zu einem „Config“ Objekt gecasted.
}
```

Apply

```
fun <T> T.apply(f: T.() -> Unit): T { f(); return this }
```

Apply() definert eine Extension Funktion auf allen Typen. Wenn sie ausgeführt wird, wird der Closure welcher als Parameter übergeben wurde ausgeführt und anschließend wird das Empfänger (Receiver) Objekt auf welchem der Closure ausgeführt wurde zurückgegeben.

```
val recyclerView: ReyclerView = RecyclerView().apply{
    setHasFixedSize = true
    layoutManager = LinearLayoutManager(context)
    adapter = MyAdapter(context)
    clearOnScrollListener()
}
```

With

```
fun <T, R> with(receiver: T, f: T.() -> R): R = receiver.f()
```

With() ist hilfreich wenn man mehrere Methoden auf dem gleichen Objekt ausführen möchte. Der Unterschied zu apply() besteht darin das bei Apply das Receiver (Also das Objekt auf dem apply() ausgeführt worden ist) zurückgegeben wird und bei with() wird das Resultat des letzten Ausdrucks innerhalb des With-Blocks zurückgegeben. Zudem ist with() keine Extension, d.h. das Objekt auf dem gearbeitet werden soll muss mitgegeben werden.

```
val w = Window()
with(w) {
    setWidth(100)
    setHeight(200)
    setBackground(RED)
}
```

Run

```
fun <T, R> T.run(f: T.() -> R): R = f()
```

Run sieht auf den ersten Blick extrem simpel aus, ist jedoch eine Kombination von with() und let(). Sie ist wie apply() ein Funktions Literal welches keinen Parameter entgegen nimm, aber das Objekt als **this** übernimmt. Sie sollte mit Lambdas verwendet werden welche keinen Wert zurückgeben sondern nur Nebeneffekte generieren.

```
webview.settings?.run {
    javaScriptEnabled = true
    databaseEnabled = true
}
```

Use

```
fun <T : Closeable, R> T.use(block: (T) -> R): R
```

Use() ist das Äqivalent zu Java‘s Try-With-Resources und C#‘s using Ausdrucks. Diese Funktion kann auf allen Objekten des Typ „Closable“ ausgeführt werden und schließt den Receiver beim verlassen des Ausdrucks.

```
// Java 1.7 and above
Properties prop = new Properties();
try (FileInputStream fis = new FileInputStream("config.properties")) {
        prop.load(fis);
}
// Kotlin
val prop = Properties()
FileInputStream("config.properties").use {
        prop.load(it)
} // FileInputStream automatically closed
```

Use - Anhang

Beispiel: Erzeuge eine Property Objekt und lade direkt eine Konfiguration einer Datei hinein:

```
// Kotlin
fun readProperties() = Properties().apply {
        FileInputStream("config.properties").use { fis ->
            load(fis) // this is of the type Properties
        }
}
```

Also

fun <T> T.also(block: (T) -> Unit): T

Also ist eine Extension Funktion welche auf allen Objekten ausgeführt werden kann. Sie führt die gegeben Funktion mit **this** also das Objekt auf welchen Also() aufgerufen worden ist auf und gibt **this zurück**.

```
Val person = Person("Chris",36)
Val result = person.also{person -> person.age += 50}

Println(person)
Println(result)
```

Output:

```
Person(name=Chris,age=86)
Person(name=Chris,age=86)
```

Cheat Sheet - Introduction

```
class MyClass {
    fun test() {
        val str: String = "..."

        val result = str.let {
            print(this) // Receiver
            print(it) // Argument
            42 // Block return value
        }
    }
}
```

This (Receiver):
This ist eine Instanz von MyClass (this@MyClass) weil test() eine Methode von MyClass ist.

It (Argument):
It ist der String "..." auf dem wir let ausgeführt haben.

42 (Rückgabe Wert):
42 ist das Resultat war wir aus dem Block zurückgeben.

Funktion	Receiver (this)	Argument (It)	Result
Let	This@MyClass	String("...")	Int(42)

Cheat Sheet – for the Std.kt

Funktion	Receiver (this)	Argument (It)	Result
Let	This@MyClass	String(“...“)	Int(42)
run	String(“...“)	n/a	Int(42)
run*	this@MyClass	n/a	Int(42)
with*	String(„...“)	n/a	Int(42)
apply	String(„...“)	n/a	String(„...“)
also	this@MyClass	String(„...“)	String(„...“)

* = Keine Extension Funktionen. Diese Methoden müssen auf die „alte“ Art und Weise ausgeführt werden.