Kotlin - Extensions

or the not so ugly thruth about extensions

Introduction

Ähnlich wie in Swift, C# und Gosu ermöglichen Extensions die Funktionalität von Standart Klassen zu erweitern. **Kotlin** unterstützt sowohl **Extension Funktionen** als auf **Extension Properties.**

Extensions modifizieren die Klassen im Hintergrund nicht wirklich, es werden keine neuen Objekte oder Properties Hinzugefügt, stattdessen ermöglichen Extension Funktionen lediglich Funktionen mittels der Dot-Notation auf Variablen des Extensions Typs aufzurufen.

Members always win

Wenn für ein Klasse ein Member Funktion und eine Extension Funktion mit dem gleichen Receiver Typ existiert, so gewinnt immer die Member Funktion.

```
class C {
    fun foo() {
        println("member")
    }
}

fun C.foo() { println("extension") }
```

Es wird immer „member“ ausgegeben.

Nullable Receiver

Extension Funktionen können auch auf einem Nullable Objekt definiert werden. Jedoch muss in diesem Fall in der Extension Funktion **explizit auf Null geprüft werden.**

```
fun Any?.toString(): String {
    if (this == null) return "null"
    return toString()
}
```

Companion

Es ist ebenfalls möglich Extension Funktionen auf Klassen Ebene und nicht Instanz Eben zu definieren. Einzige Bedingung hierbei ist, das die entsprechende **Klasse** ein **public companion Objekt besitzt**.

```
class Foo{

   companion object

   fun sayHello() = "Hello"

}

fun Foo.Companion.sayBye() = "Bye"
```

Nicht jede Klasse besitzt ein companion Objekt. Daher ist es nicht immer möglich statische Funktionen zu definieren.

Extension Properties

Wie auch Extension Funktionen geschrieben werden könne, so können auch Extension Properties definiert werden. Wichtig zu beachten ist jedoch die Tatsache das Extension Properties nicht direkt instanziiert werden können. Sie Können nur mittels explizitem Deklarieren von Getter und Setter verwendet werden.

```
val Foo.bar = 1 // error: initializers are not allowed for extension properties
```

```
val Foo.bar: Int
   get() = 1
```

Extension Dispatching

Es ist sogar möglich Extension zu definieren welche auf einem anderen Receiver Typ zeigen als die Klasse in der die Extension definiert wird.

Die Klasse in der die Extension definiert wird nennt man „Dispatch Receiver“ und die Instanz des Receiver Typs der Extension wird „Extension Receiver“ genannt.

```
class  D {
      fun  bar()  {  ...  }
}
```

```
class  C {
      fun  baz()  {...}
      fun  D.foo()  {
            bar()  // calls  D.bar
            baz()  // calls  C.baz
      }
      fun  caller(d: D) {
            d.foo()  // call the extension  function
      }
}
```

Extension – Reified and Inline

Inline Funktionen können den „reified“ Type verwenden. Dies ermöglicht es uns innerhalb der Extension Funktion Den Klassentyp zurückzugewinnen, anstatt ihn als Parameter zu übergeben.

Inline Funktionen werden zur Kompilierzeit substituiert. D.h. der Inhalt (Code-Block) der Inline Funktion wird an die Stelle platziert wo die ausführende Methode die Inline Funktion aufruft.

```
inline fun<reified T: Activity> Activity.navigate(id:String){
    valintent=Intent(this,T::class.java)
    intent.putExtra("id",id)
    startActivity(intent)
}
```

```
navigate<DetailActivity>("2")
```

Extension – Some useful examples I

ViewHolder and Infalte

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

Extension

```
fun ViewGroup.inflate(layoutRes: Int): View {
    return LayoutInflater.from(context).inflate(layoutRes, this,
false)
}
```

Aufruf mit Extension

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val v = parent.inflate(R.layout.view_item)
    return ViewHolder(v)
}
```

Extension – Some useful examples II

Image Loading

```
Picasso.with(imageView.context).load(url).into(imageView) // Selber Aufruf mit Glide...
```

Extension

```
fun ImageView.loadUrl(url: String) {
    Picasso.with(context).load(url).into(this)
}
```

Aufruf mit Extension

```
imageView.loadUrl(url)
```

Extension – Some useful examples III

Extension Property

```
val ViewGroup.children: List
    get() = (0..childCount -1).map { getChildAt(it) }
```

Aufruf mit Extension

```
parent.children.forEach { it.visible() }
```

Extension – Some useful examples III

Extension Property

```
val ViewGroup.children: List
    get() = (0..childCount -1).map { getChildAt(it) }
```

Aufruf mit Extension

```
parent.children.forEach { it.visible() }
```

Exercise 01

- Baue die Autonotify (Kapitel Delegates – Kotlin Introducion) Methode in den Adapter des Overview Screens ein.
- Betroffene Klassen sind: (Die betroffenen Stellen sind jeweil mit *„Exercise 01“ markiert*
  - OverviewAdapter
  - OverviewFragmentKodeinModule
  - Es kann ein neues Interface für die Extension erstellt werden, muss aber nicht