

Extension Functions

Extensions



The ability to extend a class
with new functionality
without having to inherit
from the class

Extensions

Kotlin, similar to C# and Gosu, provides the ability to extend a class with new functionality without having to inherit from the class or use any type of design pattern such as Decorator. This is done via special declarations called *extensions*. Kotlin supports *extension functions* and *extension properties*.

Extension Functions

To declare an extension function, we need to prefix its name with a *receiver type*, i.e. the type being extended. The following adds a `swap` function to `MutableList<Int>`:

```
fun MutableList<Int>.swap(index1: Int, index2: Int) {  
    val tmp = this[index1] // 'this' corresponds to the list  
    this[index1] = this[index2]  
    this[index2] = tmp  
}
```

The `this` keyword inside an extension function corresponds to the receiver object (the one that is passed before the dot). Now, we can call such a function on any `MutableList<Int>`:

```
val l = mutableListOf(1, 2, 3)  
l.swap(0, 2) // 'this' inside 'swap()' will hold the value of 'l'
```

Extension Properties

Similarly to functions, Kotlin supports extension properties:

```
val <T> List<T>.lastIndex: Int  
    get() = size - 1
```

Note that, since extensions do not actually insert members into classes, there's no efficient way for an extension property to have a [backing field](#). This is why **initializers are not allowed for extension properties**. Their behavior can only be defined by explicitly providing getters/setters.

Scope of Extensions

Most of the time we define extensions on the top level, i.e. directly under packages:

```
package foo.bar  
  
fun Baz.goo() { ... }
```

To use such an extension outside its declaring package, we need to import it at the call site:

```
package com.example.usage  
  
import foo.bar.goo // importing all extensions by name "goo"  
                // or  
import foo.bar.*  // importing everything from "foo.bar"  
  
fun usage(baz: Baz) {  
    baz.goo()  
}
```

Motivation

In Java, we are used to classes named "*Utils": `FileUtils`, `StringUtils` and so on. The famous `java.util.Collections` belongs to the same breed. And the unpleasant part about these Utils-classes is that the code that uses them looks like this:

```
// Java
Collections.swap(list, Collections.binarySearch(list,
    Collections.max(otherList)),
    Collections.max(list));
```

Those class names are always getting in the way. We can use static imports and get this:

```
// Java
swap(list, binarySearch(list, max(otherList)), max(list));
```

This is a little better, but we have no or little help from the powerful code completion of the IDE. It would be so much better if we could say:

```
// Java
list.swap(list.binarySearch(otherList.max()), list.max());
```

But we don't want to implement all the possible methods inside the class `List`, right? This is where extensions help us.