Injecting Using Extension Functions and Properties

We'll cover the following

- Injecting methods using extension functions
- Injecting operators using extension functions
- Injecting properties using extension properties
- Injecting into third-party classes
- Injecting static methods
- Injecting from within a class

From the readability and ease points of view, having application-specific convenience methods on third-party classes will make the lives of your fellow programmers better. And, that's possible because Kotlin permits you to inject methods and properties into any class, including classes written in other JVM languages. Unlike other languages that offer metaprogramming, Kotlin performs injection without patching the runtime or class loading. In Kotlin, classes are open for extension, even though they may not be available for inheritance. Extension functions and extension properties are techniques to add methods and properties, but without altering the bytecode of the targeted classes. When you create an extension function for a class, it gives an illusion that you've implemented an instance method for that class. Don't create an extension function for a method if that method already exists in the class. Members of a class always win over extension functions if there's a conflict. When the Kotlin compiler sees a method call, it checks to see if an instance method is available and uses it if found. If an instance method isn't found, Kotlin looks for an extension function for the targeted class.

You can inject methods and properties into existing classes, including final classes, and even those classes that you didn't write—that's the spirit of a free society.

Let's explore injecting methods into existing classes first, then look into injecting an operator, followed by injecting a property.

Injecting methods using extension functions

Suppose our large enterprise application has two classes `Point` and `Circle`, defined like so:

```
// circle.kts
data class Point(val x: Int, val y: Int)
data class Circle(val cx: Int, val cy: Int, val radius: Int)
```

These classes don't have any methods at this time, and the two classes are independent of each other.

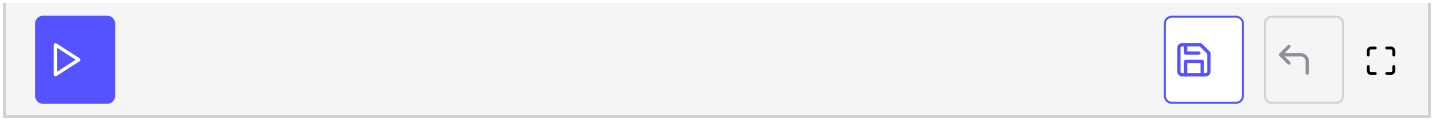
Suppose we want to find if a point is located within a circle. It would be nice to have a convenience method in either of the classes for that. But we don't have to bribe the creator of the `Circle` class or the `Point` class to introduce that method. We can add methods, right from outside, to any of these classes. Let's inject an extension function named `contains()` into the `Circle` class, like so:

```
// circle.kts
fun Circle.contains(point: Point) =
  (point.x - cx) * (point.x - cx) + (point.y - cy) * (point.y - cy) <
    radius * radius
```

This code sits outside any of the classes, at the top level of a package—a default package in this example. Within the `contains()` extension function, we access the member of the implicit `Circle` instance exactly like we'd access it if this extension function were written as an instance method within the class. If we'd written this method within the `Circle` class, the only difference is that we would have written `fun contains(point: Point)` instead of `fun Circle.contains(point: Point)`.

As long as this method is visible—that is, it's either in the same file or we've imported it from the package where it resides—we can use it, like so:

```
val circle = Circle(100, 100, 25)
val point1 = Point(110, 110)
val point2 = Point(10, 100)

println(circle.contains(point1)) //true
println(circle.contains(point2)) //false
```



circle.kts

Even though the class `Circle` doesn't have the `contains()` method, we're able to call it on an instance of that class. When Kotlin sees the extension function, it creates a static method in the package where the extension function exists and passes the context object—`Circle` in the example—as the first argument to the function, and the actual parameters as the remaining parameters. When the compiler sees the call to a method, it figures we're calling the extension function and routes the context object `circle` as the first argument to the method. In short, what appears to be a method call is really a call to a static method when extension functions are involved.

Extension functions have a few limitations. When there's a conflict between an extension function and an instance method of the same name, the instance method always wins. And unlike instance methods, which can reach into the encapsulation boundaries of an instance, extension functions can access only the parts of an object visible from within the package they are defined in.

Injecting operators using extension functions

The extension function may be an operator as well. In the list of operators mentioned in Overloading Operators, we saw that `in` is an operator and it maps to the `contains()` method. A circle contains a point, possibly, but we'd ask if a point is in a circle. It turns out the `in` operator when called like `aPoint in aCircle` will compile down to `aCircle.contains(aPoint)`. But for that to work, we have to annotate the `contains()` method in `Circle` with `operator`. Let's do that:

```
operator fun Circle.contains(point: Point) =
  (point.x - cx) * (point.x - cx) + (point.y - cy) * (point.y - cy) <
    radius * radius
```

We simply added the keyword operator to the front of the extension function—that's it.

Now, we may use the contains() method like before or use the in operator, like this:

```
println(circle.contains(point1)) //true
println(point1 in circle) //true
println(point2 in circle) //false
```

Injecting properties using extension properties

Taking a quick detour from extension functions, we may add extension properties as well. Since these are also not part of the internals of the class, extension properties can't use backing fields—that is, they can't access `field` like actual properties can. They may use other properties or methods on the class to get their work done. Let's add a `area` property to the `Circle` class.

```
val Circle.area: Double
  get() = kotlin.math.PI * radius * radius
```

From the object user's perspective, we can use extension properties much like we use real properties:

```
val circle = Circle(100, 100, 25)
println("Area is ${circle.area}") //1963.49...
```

Where it makes sense, we may also write setters for var extension properties. The setter will have to rely upon other methods of the class to accomplish its goals; just like the getters for extension properties, the setters for extension properties can't use backing fields.

Injecting into third-party classes

You may add extension functions to third-party classes and also route the extension functions to existing methods. Here's an extension function for the good old `java.lang.String` class:

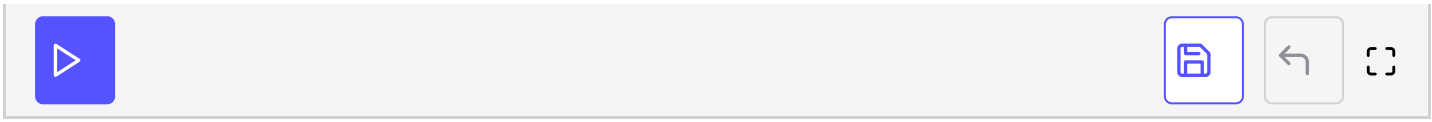
```
// stringext.kts
fun String.isPalindrome(): Boolean {
  return reversed() == this
}
```

The `isPalindrome()` method uses the Kotlin extension function `reversed()` to determine if a given string is a palindrome. Instead of defining a code block, we may route the call to an existing method, using a single line expression, like so:

```
// stringext.kts
fun String.shout() = toUpperCase()
```

Here's an example to use these two methods:

```
val str = "dad"
println(str.isPalindrome()) //true
println(str.shout()) //DAD
```



stringext.kts

The first line shows that the extension function `isPalindrome()` works for `String`. The second line shows that `shout()` is merely using the `toUpperCase()` method to evoke a response I normally hear from my children after I tell a joke.

> **Don't Change Behavior of Existing Methods**
>
> Once, in a Kotlin class I was teaching, a developer with unsurmountable enthusiasm toward extension functions asked if we could replace an existing method with an extension function. In the spirit of live coding, I cranked up the following in class:
>
> ```
> // shadow.kts
> fun String.toLowerCase() = toUpperCase() >//BAD CODE
> ```
>
> Not only did I replace the implementation of an existing method, but I gave it exactly the opposite behavior than what the name indicates. We can see the effect of this by calling the method—the result of calling the `toLowerCase()` is the string in all uppercase:
>
> ```
> // shadow.kts
> val str = "Please Don't" println(str.toLowerCase()) //PLEASE DON'T
> ```
>
> What bothered me most is not that Kotlin permits this, but how pleased the developer was upon seeing the ridiculous result.
>
> While instance methods always win, it's possible to replace an extension function from another package, like `toLowerCase()`, which is defined in the Kotlin standard library, with an extension function in your own package. As the message says, please don't. File that under "Just because we can doesn't mean we should." Changing behavior of well-known methods will cause hard-to-maintain code, loss of hair for your friends, and loss of their

> hard-to-maintain code, loss of hair for your friends, and loss of their friendship when they find out who caused it.

Before we leave the topic of extension functions, we have to complete a long-pending task. Back in Forward Iteration, we tried to iterate over a range of `Strings` from “hell” to “help”, which failed. Here’s the code we tried:

```
for (word in "hell".."help") { print("$word, ") } //ERROR
//for-loop range must have an 'iterator()' method
```

Now we’re ready to fix that error, with an extension function.

The error says that the compiler didn’t find the `iterator()` method on the `ClosedRange<String>` class. Let’s review a few things that can help us to inject that method into the class:

- We can create an iterator as an anonymous object, as you learned in Anonymous Objects with Object Expressions.
- We can access the first element of the range using the start property and the last element using `endInclusive` of the `ClosedRange<T>` class.
- Kotlin will invoke the `compareTo()` method when the `>=` operator is used, as we discussed in Overloading Operators.
- We can use the StringBuilder class from the JDK to hold a mutable `String`.
- The `+` operator of `kotlin.Char` can be used to get the next character, and we can use that to increment the last character in the `StringBuilder`.

Let’s apply these ideas to create the extension function for the `iterator()`:

```
operator fun ClosedRange<String>.iterator() =
  object: Iterator<String> {
    private val next = StringBuilder(start)
    private val last = endInclusive

    override fun hasNext() =
      last >= next.toString() && last.length >= next.length

    override fun next(): String {
      val result = next.toString()

      val lastCharacter = next.last()

      if (lastCharacter < Char.MAX_VALUE) {
        next.setCharAt(next.length - 1, lastCharacter + 1)
      } else {
```

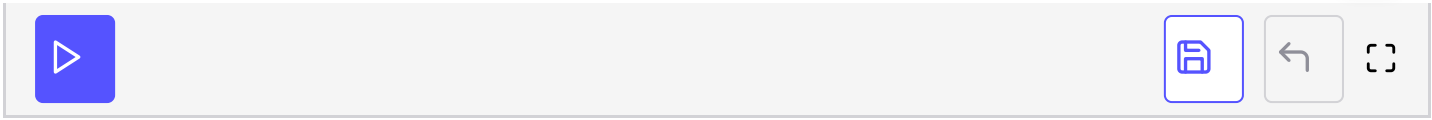
```
        next.append(Char.MIN_VALUE)
      }

      return result
    }
  }
```

forstringrange.kts

Kotlin expects the `iterator()` method to be annotated as operator. In the extension function, we create an implementation of `Iterator<String>`. In that, we store the start value of the range into the next property and the `endInclusive` value into the last property. The `hasNext()` method returns `true` if we've not gone past the last element in the iteration and if the length of the last element isn't less than the string generated in the iteration. Finally, the `next()` method returns the current `String` value in next and increments the last character in that variable. If the last character reaches the `Char.MAX_VALUE`, then a new character is appended to the generated string. This implementation of `next()` was inspired by a similar implementation in the Groovy library.

Let's call the loop again and see the effect of this injected method:

```
for (word in "hell".."help") { print("$word, ") }
```



forstringrange.kts

No error this time, and here's the output:

```
hell, helm, heln, helo, help,
```

We can simplify the implementation of the iterator further using `yield` from Kotlin coroutines—we'll see this in Interleaving Calls with Suspension Points.

Injecting static methods

You can inject static methods into classes by extending their companion object—that is, injecting method into the companion instead of the class. As a result, you may inject static methods only if the class has a companion object.

For example, Kotlin has extended and added a companion to `String`, so you can add a `static` method to `String`, like so:

add a `static` method to `String`, like so:

```
fun String.Companion.toURL(link: String) = java.net.URL(link)
```

Once you add that extension function, you may call the `toURL()` on the `String` class, to take a `String` and return an instance of URL, like in this example:

```
val url: java.net.URL = String.toURL("https://pragprog.com")
```

Adding a `static` or class-level method to `String` took little effort. However, you can't add a `static` method to all third-party classes from Kotlin. For example, you can't add a class-level method to the `java.net.URL` class of the JDK since Kotlin hasn't added a companion class to this class.

Injecting from within a class

All the extension functions we injected so far were at the top level—that is, we added them from outside of any class. Those extension functions are visible from any code that imports the package in which they're located at the top level. Extension functions may also be injected from within classes.

As a designer of a class, you may occasionally come to realize that having some convenience methods on third-party classes may ease the implementation of your own class. For instance, when implementing an `InsurancePolicy` class you may find it useful to have methods on Date that tell you the number of days until the policy expires, if a policy is valid on a given date, and so on. Thus, you may want to create extension functions on third-party classes right within your classes.

If you create an extension function within a class, then that extension function is visible only within the class and its inner classes. Also, within the extension function there are two receivers; that is, in a sense there are two `this` context objects. Let's create an example to grasp these concepts.

Earlier we created a `Point` data class that had two properties, `x` and `y`. Let's create a variation of that class here, but this time we'll store the values in a `Pair` instead of keeping them directly within the `Point`. That will give us an excuse to create a convenience method on `Pair` to be used within our new `Point` class.

The `Point` class's primary constructor will take two parameters `x` and `y`. We create them as parameters instead of as properties and then store those two

parameters in a `Pair<Int, Int>`. We'll create two private properties within `Point`,

to return the sign of the `x` and `y` values, now stored within the `Pair`. We'll call these `firstsign` and `secondsign`. Then, we'll override the `toString()` method and in the implementation we'll call the method on `Pair<Int, Int>` that we'll extend from within the `Point` class. As the last step in the example, we'll implement the extension function `point2String()`, which is called from within the `toString()` method. Let's take a look at the code for the `Point` class before discussing further about the extension function.

```
class Point(x: Int, y: Int) {
  private val pair = Pair(x, y)

  private val firstsign = if (pair.first < 0) "" else "+"
  private val secondsign = if (pair.second < 0) "" else "+"

  override fun toString() = pair.point2String()

  fun Pair<Int, Int>.point2String() =
    "(${firstsign}${first}, ${this@Point.secondsign}${this.second})"
}

println(Point(1, -3)) //(+1, -3)
println(Point(-3, 4)) //(-3, +4)
```

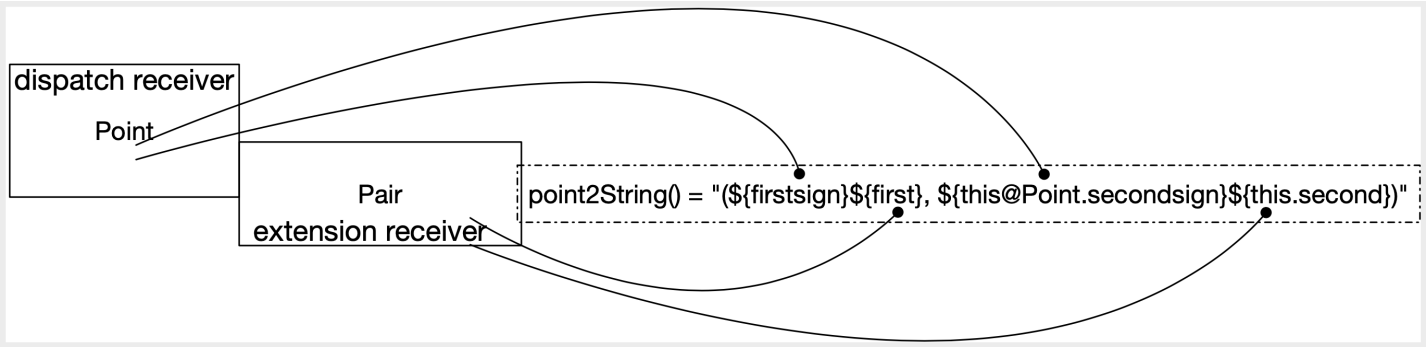
innerextension.kts

The extension function is injected into the `Pair<Int, Int>` from within the `Point` class. So any effort to use that extension function on an instance of `Pair<Int, Int>` from outside the class will result in compilation error. Let's take a closer look at the extension function alone.

```
fun Pair<Int, Int>.point2String() =
  "(${firstsign}${first}, ${this@Point.secondsign}${this.second})"
```

Since the extension function is created within a class, it has two receivers: this and `this@Point`. These two receivers have distinct names in Kotlin, *extension receiver* and *dispatch receiver*, respectively. An extension receiver is the object that the extension function is executing on—that is, the object that received the extension function. A dispatch receiver is the instance of the class from within which we added the extension function—that is, the class from within which we did the injection of the method.

The following figure shows a clear view of the two receivers in the extension function.



A property or method mentioned within the extension function refers to the extension receiver if it exists on that instance. Otherwise, it binds to the corresponding property or method, if present, on the dispatch receiver. The extension receiver takes precedence for binding to properties and methods.

Within the method, `first` binds to the property of `Pair<Int, Int>` the extension receiver. Likewise, `this.second` binds to the `second` property on the same instance. Since the extension receiver, `Pair<Int, Int>`, doesn't have a property named `firstsign` but the dispatch receiver, `Point`, does, the reference to `firstsign` binds to the dispatch receiver. If there's a conflict, and we want to bypass the extension receiver and reference the dispatch receiver, we can use the `this@Outer` syntax for that—we saw this in Nested and Inner Classes. That's the same syntax used to refer to the outer class from within the inner class. The reference `this@Point.secondsign` illustrates this explicit access to a property on dispatch receiver.

Extension functions defined within a class, and even a method of a class, are useful to narrow the scope of the extension functions for use within the particular class or particular method.

So far, we've seen adding methods to classes. In the next lesson, we'll see how to add a method to a function—that'll make the interesting feature of extension functions even more intriguing.