Extension Functions

Understand Kotlin's extension functions, how to define them, and why they are a fundamental feature of Kotlin.

WE'LL COVER THE FOLLOWING

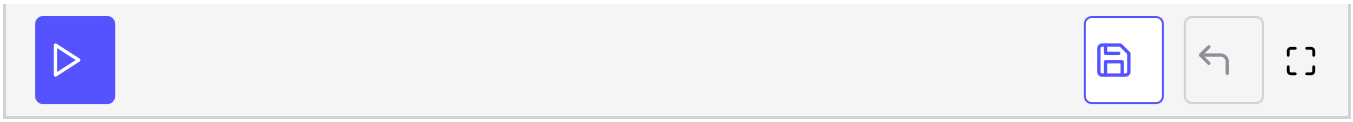
Extension functions allow you to attach new functions to existing types. *This includes ones that you don't own*, such as `Int`, `String`, any predefined Java class such as `Date`, `Instant`, and any type imported from a library.

Typically, this is done using utility classes that encapsulate a series of utility functions. You may be familiar with class names such as `StringUtils` or `DateUtils`. In Kotlin, missing capabilities of a class you want to use can be patched using extension functions instead.

Defining an Extension Function

Let's say you want to use Java's old `Date` class (instead of the `java.time` package). In your application, you want to add a number of days to a given `Date`. Ideally, you'd want to write the following:

```
// Goal
val now = Date()
val tomorrow = now.plusDays(1)  // No such method yet
```

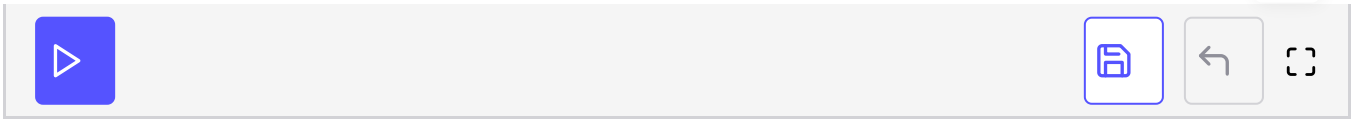


However, Java's `Date` class has no such `plusDays` method. Traditionally, you would create a helper function that allows you to write `val tomorrow = plusDays(now, 1)`, but this call syntax has several drawbacks:

- It doesn't read as naturally
- The function is not scoped to `Date` objects
  - Thus, autocomplete is a lot less helpful

Fortunately, Kotlin's extensions can achieve what we want. A `plusDays` function that fulfills the requirements above can be written as follows:

```
fun Date.plusDays(count: Int) = Date(this.time + count * 86400000L)
```

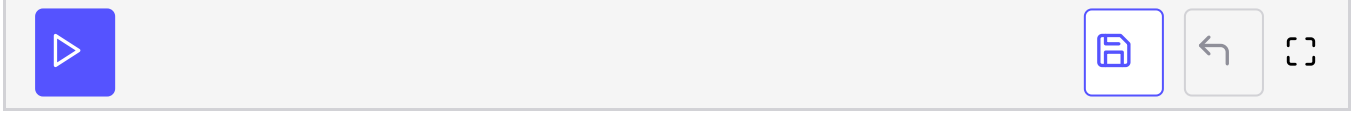


> **Note:** The function signature contains the `Date` class in front of the function name. This implicitly makes `Date` the first parameter of the function, and you can access it inside the function body as `this`.

The actual implementation simply adds the correct number of milliseconds to the original date to add `count` days.

With this function defined as above, Kotlin now allows you to call this method as if it were defined in the class itself:

```
val now = Date()
val tomorrow = now.plusDays(1)
```



> Under the hood, Kotlin transforms this to a regular utility function. This is necessary because the generated Java bytecode doesn't allow for adding a new method to an existing class.

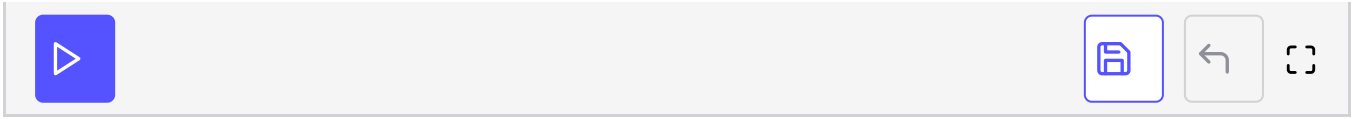
Extensions are Resolved Statically

> These following few paragraphs require knowledge on object orientation.

Although extensions look like regular methods being called on objects, they are resolved statically, not dynamically. What does this mean? Consider the following example:

```
fun Number.print() = println("$this is a Number")
fun Int.print() = println("$this is an Int")

val num: Number = 17
num.print()  // Prints "17 is a Number"
```



`Int` is a subclass of `Number`. If `print` were a regular method in the `Number class` overwritten in the `Int` class, then to method call in the last line would be resolved dynamically at runtime. Since `num` is an `Int` at runtime, it would call `Int.print`.

This is not the case with extension functions. The call in the last line is resolved statically at compile time, and since `num` is declared to have type `Number`, the compiler will transform the `print` call to calling `Number.print`.

If you ever have a bug where the wrong extension is called, knowing these details can be essential.

> **Note:** If the type you extend already has a regular member function with the same name as your extension, the member function takes precedence.

Real-World Examples of Extension Functions

As mentioned, extension functions are a fantastic way to improve clunky parts of any APIs that you're using. Since Kotlin has become the de-facto standard for Android development, let's look at a simple example from the Android domain.

Android domain.

On Android, when you're inside an `Activity` class (imagine a "screen" in an Android app), and you want to get its `ViewModel` (imagine "data provider" for that activity), you usually write this:

```
val viewModel = ViewModelProviders.of(this).get(MyViewModel::class.java)
```

This is an Android API. However, let's make it so that the same can be achieved using this piece of code:

```
val viewModel = getViewModel(MyViewModel::class)
```

To do so, all you need is an extension function on the `Activity` which handles getting a `ViewModel` from the context of the activity:

```
fun <T : ViewModel> FragmentActivity.getViewModel(modelClass: KClass<T>): T {
  return ViewModelProviders.of(this).get(modelClass.java)
}
```

For now, don't worry about the generic types in angle brackets. But on a high level, you can see the extension is more Kotlin-friendly by expecting a `KClass` as parameter instead of a Java `Class`. This way, you can pass in `MyViewModel::class` instead of `MyViewModel::class.java`.

> **Takeaway:** If you ever feel like the API you're using is unnecessarily verbose, think about **encapsulating the boilerplate in an extension function**. For instance, if you always call a function with certain values for some parameters, you could create an extension function with default values for these to reduce the number of arguments.

Quiz

Extension Functions

1

What can extension functions be used for? You can select multiple answers.

COMPLETED 0%

1 of 3

Exercises

An Extension on `Date`

Implement an extension function on the `java.time.Date` which adds a given number of minutes to the original date. You should be able to call this extension as follows:

```
val now = Date()
val later = now.plusMinutes(5)
```

Hint: You'll have to import `java.util.Date` by adding the following at the top of your code:

```
import java.util.Date
```

Problem | Solution

```
// Add your code here
```

An Extension on `String`

An Extension on `String`

In this exercise, implement an extension on `kotlin.String` which removes all occurrences of a given string from the original string. It should provide the following API:

```
val original = "Extension functions allow you to attach new functions to existing types *including ones that you don't own*"

val edited1 = original.removeAll("*")  // Should remove both *
val edited2 = original.removeAll("you")  // Should remove all "you"s
```

Hint: First, figure out how to remove a substring once. After that, figure out how to repeat the removal step for as long as necessary.

Hint: If you're using an IDE with autocomplete, take a look at all the methods Kotlin offers on `String`s. If you're doing this in the browser: the `replace` method may be useful.

Summary

Extension functions in Kotlin are a prevalent part of Kotlin applications in practice and in Kotlin's standard library.

- Extensions allow you to add new functionality to a base API.
- The declaration of extensions is not bound in location, meaning extensions can be defined anywhere, not just inside the receiver class' body.
- Extensions can be called directly on the corresponding types' objects.
    - This enables code completion, improves code structure, and limits the scope of extensions.
    - Under the hood, Kotlin generates a good old utility class. In the generated Java bytecode, you cannot actually add new functions to

types from outside.

- In the following lessons, the concept of extension functions will be explored further with infix functions and operators.

In the next lesson, you'll learn how to implement and use a special class of functions called *infix functions*.