Objects and Singletons

We'll cover the following

- Anonymous objects with object expressions
- Singletons with object declaration
- Top-level functions vs. singletons

The Singleton Design Pattern, discussed in Design Patterns, is one of the easiest to understand and yet one of the most difficult to implement. It turns out that controlling the number of instances of a class isn't a trivial task—you need to prevent reflective access to constructors, ensure thread safety, and, at the same time, not introduce any overhead to check if an instance exists. By providing support for singletons directly, Kotlin removes the burden of implementing the pattern and the risks of getting it wrong. When you need, you can directly create an object without being forced to first define a class. For simple situations you can use objects, and for more complex cases, where you want to define an abstraction, you may create classes. In this section we'll focus on objects, how to create them, and how to make use of singletons.

Anonymous objects with object expressions

If you want an object, then you should have an object—no fluff, no ceremony, no tax. Kotlin object expressions are like JavaScript objects and Anonymous Types in C#, although they're also useful to create instances of anonymous classes, like in Java.

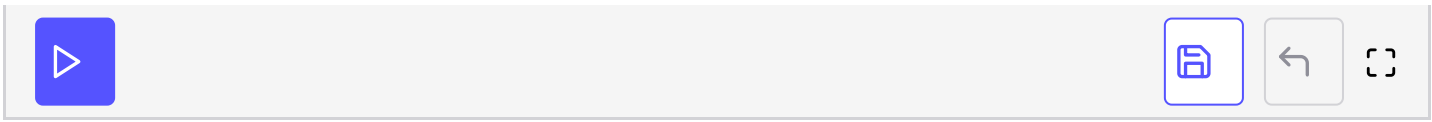
In its most basic form, an object expression is the keyword `object` followed by a block `{}`. Such a trivial object expression is limited in use but not entirely purposeless. Suppose we want to represent a few pieces of data related to a circle. One option would be to create a `Circle` class, but that may be overkill. Another option is to keep around multiple local variables, but that doesn't give us a sense these are closely related values. Object expression can help here, as in this example:

```
fun drawCircle() {
  val circle = object { //an expression
    val x = 10
    val y = 20
    val radius = 30
  }

  //Pass circle.x, circle.y, circle.radius to a draw function here

  println("Circle x: ${circle.x} y: ${circle.y} radius: ${circle.radius}")
  //Circle x: 10 y: 20 radius: 30
}

drawCircle()
```



object.kts

The `circle` object simply provides a grouping of the three values: `x`, `y`, and `radius`—it's just that, a grouping of a few local variables. Likewise, you may create an `alien` object, in a game, with different properties that belong to a dreadful creature that doesn't deserve the status of a full-blown class.

Instead of defining the properties as `val`, you may make them mutable with `var`. You may add methods to the object as well. But if you're going that far, you might as well create a class instead of defining an anonymous object. That's because classes don't have the limitations that anonymous objects have—and there are a few:

- The internal type of anonymous objects can't stand as return types to functions or methods.
- The internal type of anonymous objects can't be used as types of parameters to functions or methods.
- If they're stored as properties in classes, they'll be considered `Any` type and none of their properties or methods will then be available for direct access.

In short, the most rudimentary object expression is useful only to group a few local variables together.

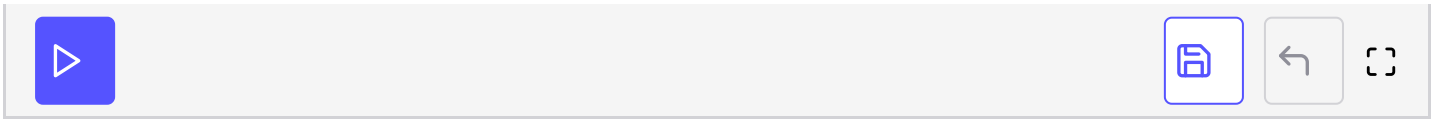
With a slight change, anonymous objects can be useful as implementors of

interfaces—that is, as anonymous inner classes like in Java. An anonymous inner class typically implements an interface on the fly. Between the `object` keyword and the block `{}`, mention the names of the interfaces you'd like to implement, comma separated. Oh, and follow the keyword with a `:` in this case. Let's implement the popular `Runnable` interface from the JDK, here using Kotlin's object expression.

```
fun createRunnable(): Runnable {
  val runnable = object: Runnable {
    override fun run() { println("You called...") }
  }

  return runnable
}

val aRunnable = createRunnable()
aRunnable.run() //You called...
```



anonymous.kts

The variable `runnable` holds a reference to an instance of the anonymous inner class. The type of this variable is `Runnable`. You may return this instance from the function in which it's created, and the return type will take the type of the interface the anonymous inner class implements. If the instance is created by extending an existing class instead of an interface, then that class will serve as the type.

If the interface implemented by the anonymous inner class is a single abstract method interface (what Java 8 calls a functional interface), then we can directly provide the implementation without the need to specify the method name, like so:

```
fun createRunnable(): Runnable = Runnable { println("You called...") }
```

In addition to using the single abstract method implementation syntax, in the above example, we turned the `createRunnable()` function into a concise single-expression function by removing the block body and the `return` keyword.

If the anonymous inner class implements more than one interface, you have to specify the type the instance should represent upon return. Let's take a look at how to do that by changing the previous example to implement both `Runnable` and `AutoCloseable` interface, even though the method still chooses `Runnable` as the

return type.

```
fun createRunnable(): Runnable = object: Runnable, AutoCloseable {
    override fun run() { println("You called...") }
    override fun close() { println("closing...") }
}
```

Outside of the function, we're able to access the instance of the anonymous inner class through an interface it implements—`Runnable` in this case.

In Kotlin, use object expressions anywhere you'd use anonymous inner classes in Java.

Singletons with object declaration

If you place a name between the `object` keyword and the block `{}`, then Kotlin considers the definition a statement or declaration instead of an expression. Use an object expression to create an instance of an anonymous inner class and an object declaration to create a singleton—a class with a single instance. `Unit` is an example of a singleton in Kotlin, but you can create your own singletons using the `object` keyword.

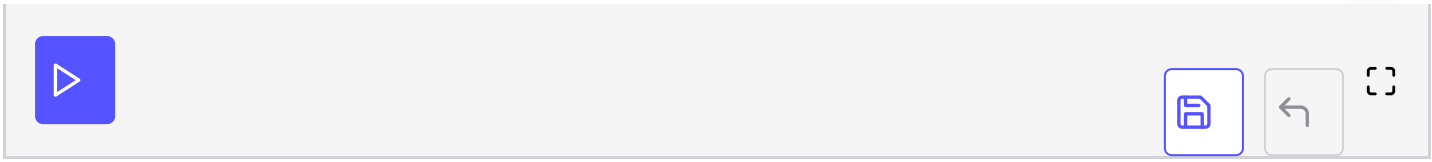
Let's create a singleton utility object to get some details about the system on which the code is run:

```
// util.kts
object Util {
  fun numberOfProcessors() = Runtime.getRuntime().availableProcessors()
}
```

The `Util` object we created using the object declaration is a singleton. We can't create objects of `Util`—it's not considered to be a class by the Kotlin compiler; it's already an object. Think of it like a Java class with a private constructor and only `static` methods. Internally, however, Kotlin represents the singleton object as a `static` instance of a `Util` class. The method itself isn't `static` in the bytecode unless the `@JvmStatic` annotation is used (which we'll see soon in Creating static Methods. You can call the methods of a singleton like you'd call static methods on classes in Java.

Let's call the `numberOfProcessors()` method of the singleton:

```
println(Util.numberOfProcessors()) //8
```



util.kts

The call to `Util.numberOfProcessors()` reports `8` on my machine. What does it say on yours? More? Showoff :).

Singletons aren't limited to having methods. They may have properties as well, both `val` and `var`. Object declarations may implement interfaces or may extend from existing classes, like object expressions do. If a singleton has a base class or interface, then that single instance may be assigned to a reference or passed to a parameter of the base type. Let's explore these with an example:

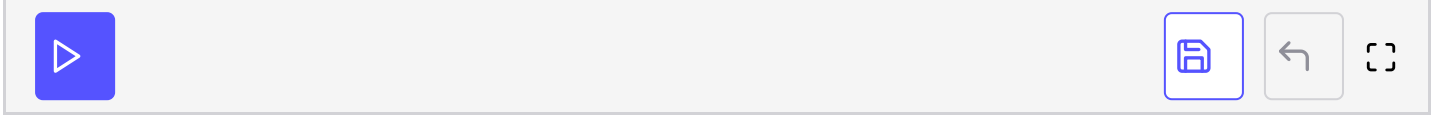
```
object Sun : Runnable {
  val radiusInKM = 696000
  var coreTemperatureInC = 15000000

  override fun run() { println("spin...") }
}

fun moveIt(runnable: Runnable) {
  runnable.run()
}

println(Sun.radiusInKM) //696000

moveIt(Sun)  //spin...
```



singleton.kts

We can pass the singleton Sun to functions like `moveIt()`, since it's expecting an instance of `Runnable`. We can also access properties of the singleton, like `radiusInKM`, much like the way we'd access properties on instance of classes. A word of caution: even though this example illustrates the capability of the language, placing mutable state in a singleton isn't a good idea, especially in a multithreaded application.

Kotlin is highly flexible and unopinionated, in a good way, and lets us choose between top-level functions and singletons. Before we get deeper into the object-oriented concepts, let's discuss when to choose singletons instead of top-level functions.

functions.

Top-level functions vs. singletons

Modularization is key to maintain sanity in any nontrivial application. But we don't want to also take things to the extreme of death-by-modularity. You can maintain a sensible balance with Kotlin based on the needs of your applications.

Should we choose top-level functions—that is, functions placed directly within a package—or singletons? Before we discuss which is better, let's take a look at an example of a package with both top-level functions and singletons.

```
package com.agiledeveloper.util

fun unitsSupported() = listOf("Metric", "Imperial")

fun precision(): Int = throw RuntimeException("Not implemented yet")

object Temperature {
  fun c2f(c: Double) = c * 9.0/5 + 32
  fun f2c(f: Double) = (f - 32) * 5.0/ 9
}

object Distance {
  fun milesToKm(miles: Double) = miles * 1.609344
  fun kmToMiles(km: Double) = km / 1.609344
}
```

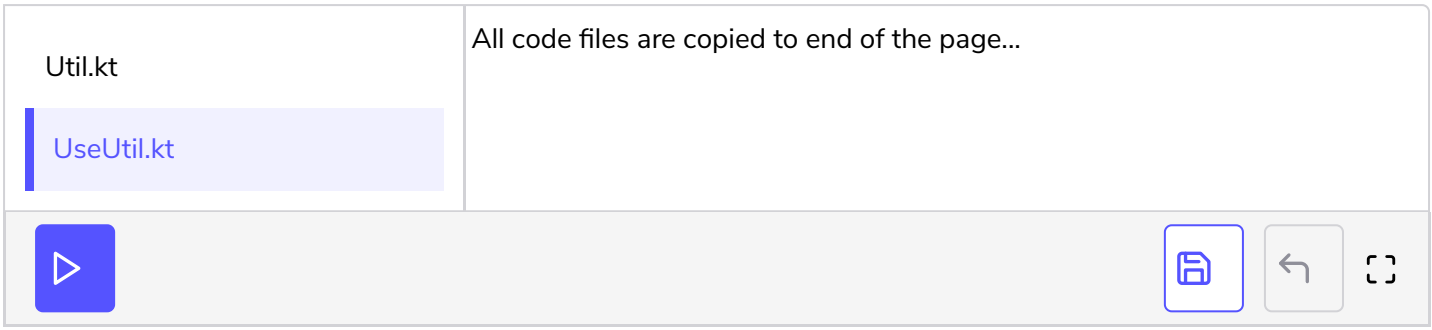
Util.kt

In the file `Util.kt`, we've defined a package with the `package` keyword—the syntax for this in Kotlin is just like in Java, minus the semicolon, which is optional. Next we have a few top-level functions placed directly in the file, within the package. Then we have a few singletons, each of which groups methods that closely belong together.

To use a top-level function, like `unitsSupported()`, we may refer to its fully qualified name, like `com.agiledeveloper.util.unitsSupported()`, or just the name `unitsSupported()`, after importing either `com.agiledeveloper.util.unitsSupported` or `com.agiledeveloper.util.*`. To access the methods within a singleton we can use similar techniques.

If there's a conflict in the names imported, we can either use the fully qualified name or define an alias, like import `somepackage.SingletonName` as `ALocalName` and then use `ALocalName` to refer to `somepackage.SingletonName`.

Let's use some of the methods from the `com.agiledeveloper.util` package.



The code to use the functions in `com.agiledeveloper.util` is placed in a new package, `com.agiledeveloper.use` in this example. It imports all members of the package `com.agiledeveloper.util` and the `c2f` method of the `Temperature` singleton. Within the `main()` method, we're calling the top-level function `unitsSupported()` and the `c2f()` method of the `Temperature` singleton without any qualifiers since they're visible from the imports. But to call the method `f2c()` of `Temperature`, we're qualifying it with the name of the singleton `Temperature`, which has been imported using the broader import `com.agiledeveloper.util.*`.

To compile the above two files locally on your own system and execute the code in `UseUtil.Kt`, use the following commands (use backslash instead of forward slash on Windows):

```
kotlinc-jvm ./com/agiledeveloper/util/Util.kt \ ./com/agiledeveloper/use/UseUt
il.kt \
-d Util.jar

kotlin -classpath Util.jar com.agiledeveloper.use.UseUtilKt
```

In addition to showing how to use packages, functions, and singletons, this example shows why we may choose a top-level function or a singleton. If a group of functions are high level, general, and widely useful, then place them directly within a package. If on the other hand, a few functions are more closely related to each other than the other functions, like `f2c()` and `c2f()` are more closely related to each other than to `milesToKm()`, then place them within a singleton. Also, if a group of functions needs to rely on state, you can place that state along with those related functions in a singleton, although a class may be a better option for this purpose. Use caution: placing mutable state in a singleton may cause issues in multithreaded applications. In short, place functions at the top-level and use singletons to group and modularize functions further, based on application needs.

QUIZ

1 What is the correct syntax to declare a singleton object in Kotlin?

2 Objects can be used to implement interfaces.

3 If functions are closely related to each other, how should they be grouped?

Retake Quiz

Functions and singletons make good sense when we're focused on behaviors, computations, and actions. But if we want to deal with state, then classes are a better choice, and Kotlin offers great support for creating those as well, which we'll cover in the next lesson.

cover in the next lesson.

Code Files Content !!!

⤋ ⤋

```
--------------------------------------------------------------------------
|  Util.kt [1]
--------------------------------------------------------------------------


package com.agiledeveloper.util

fun unitsSupported() = listOf("Metric", "Imperial")

fun precision(): Int = throw RuntimeException("Not implemented yet")

object Temperature {
  fun c2f(c: Double) = c * 9.0/5 + 32
  fun f2c(f: Double) = (f - 32) * 5.0/ 9
}

object Distance {
  fun milesToKm(miles: Double) = miles * 1.609344
  fun kmToMiles(km: Double) = km / 1.609344
}


--------------------------------------------------------------------------
|  UseUtil.kt [1]
--------------------------------------------------------------------------


package com.agiledeveloper.use

import com.agiledeveloper.util.*
import com.agiledeveloper.util.Temperature.c2f
fun main() {
  println(unitsSupported())
  println(Temperature.f2c(75.253))
  println(c2f(24.305))
}




********************************************************************************
```