Companion Objects and Class Members

We'll cover the following

The classes we created so far had properties and instance methods. If a property or a method is needed at the class level and not on a specific instance of the class, we can't drop them into the class. Instead, place them in a companion object. In Singleton with Object Declaration, we created singletons. Companion objects are singletons defined within a class—they're singleton companions of classes. In addition, companion objects may implement interfaces and may extend from base classes, and thus are useful with code reuse as well.

Class-level members

In the next example, a `MachineOperator` class needs a property and a method at the class level; we achieve that using a companion object:

```
class MachineOperator(val name: String) {
  fun checkin() = checkedIn++
  fun checkout() = checkedIn--

  companion object {
    var checkedIn = 0

    fun minimumBreak() = "15 minutes every 2 hours"
  }
}
```

companion.kts

Within the class, the `companion` object, literally defined using those keywords, is nested. The property `checkedIn` within the companion object becomes the class-level property of `MachineOperator`. Likewise, the method `minimumBreak` doesn't

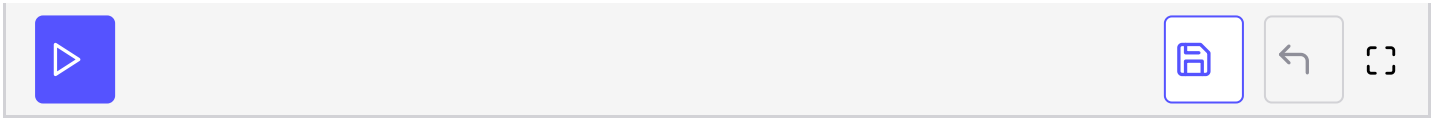
belong to any instance; it's part of the class.

The members of the companion object of a class can be accessed using the class name as reference, like so:

```
MachineOperator("Mater").checkin()

println(MachineOperator.minimumBreak()) //15 minutes every 2 hours

println(MachineOperator.checkedIn) //1
```



companion.kts

Use caution: placing mutable properties within companion objects may lead to thread-safety issues in multithreaded scenarios.

Instance methods like `checkin()` need an instance as a target, but methods like `minimumBreak()`, which are part of the companion object, can't be accessed using an instance. They can only be accessed using the class name as a reference.

That's true for properties, like `checkedIn`, that are defined within the companion objects as well.

Accessing companion objects

Sometimes we need a reference to the companion object instead of one of its members. This is especially true when the companion object implements an interface, for example, and we want to pass that singleton instance to a function or method that expects an implementor of the interface. You may access the companion of a class using. `Companion` on the class—that's an uppercase C. For example, here's the way to get the companion of the `MachineOperator` class:

```
// companion.kts
val ref = MachineOperator.Companion
```

If you don't expect the users of your class to directly need a reference to the companion too often, then the above is fine. But for frequent use, you may want to give a nicer name than `Companion`. The name `Companion` is used only if the companion object doesn't have an explicit name. Here's a way to give an explicit name for the companion object of `MachineOperator`:

name for the companion object of `MachineOperator`.

```
//companion object {
companion object MachineOperatorFactory {
  var checkedIn = 0
  //...
```

The name `Companion` is no longer available on `MachineOperator`. To refer to the companion use the explicit name:

```
val ref = MachineOperator.MachineOperatorFactory
```

The explicit name for the companion object may be any legal name in Kotlin, but the `Factory` suffix alludes to an intent of the companion object to serve as a factory, as we'll explore next.

Companion objects factories

Whether you give an explicit name or not, the companion object can serve as a factory to create instances of the class they are part of.

The constructors initialize an object to a valid state, but there are times when we may have to perform some extra steps before an object becomes available for use. For example, we may want to register an object as an event handler or register with a timer to execute a method. Registering within the constructor isn't a good idea; you don't want a reference to the object to become available anywhere before the construction is completed. Providing a method to be called after construction isn't good either: the user of the class may forget to register or may invoke other methods before registering. The companion object for the class, acting as a factory, can help with this design concern.

To use a companion as a factory, provide a private constructor for the class. Then, provide one or more methods in the companion object that creates the instance and carries out the desired steps on the object before returning to the caller.

Revisiting the previous example, we can change the `MachineOperator` class to use a factory.

```
class MachineOperator private constructor(val name: String) {
    //...
    companion object {
```

```
        //...
        fun create(name: String): MachineOperator {
            val instance = MachineOperator(name)

            instance.checkin()
            return instance
        }
    }
}
```

The constructor of the `MachineOperator` class is marked as private. Outside the class, we can't directly create any instances. Within the companion object, we have a new `create()` method that takes name as a parameter, creates an instance of the `MachineOperator`, invokes the `checkin()` method on it, and returns the instance to the caller. The `create()` method is the only way to create an instance of the class, thus the companion object acts as a factory for the class.

To create an instance, we have to use the create() method:

```
val operator = MachineOperator.create("Mater")
println(MachineOperator.checkedIn) //1
```

The `create()` method is invoked directly on the `MachineOperator` class and it is routed to the companion object. We may have more than one method in a companion object, taking different parameters to create instances of a class.

Not quite static

Looking at how we access the members of the companion objects, through a reference to the class name, we may get the impression that members of companion objects turn into static members, but that's not true. Don't assume that the methods in a companion are `static` methods of a class.

When you reference a member of a companion object, the Kotlin compiler takes care of routing the call to the appropriate singleton instance. But this will pose a problem from the Java interoperability point of view, especially if a programmer is looking for `static` methods. The `@JvmStatic` annotation is useful to resolve this issue. You'll learn about this facility, its purpose, and the benefits it offers soon in Creating static Methods.

The classes we created so far have properties of specific types, but their types may be parameterized as well.

In the next lesson, we'll see how to create generic classes.