Data Classes

We'll cover the following

What is a data class

Much like the case classes of Scala, the *data classes* of Kotlin are specialized classes that are intended to carry mostly data rather than behavior. The primary constructor is required to define at least one property, using `val` or `var`. Non-`val` or `var` parameters aren't allowed here. You may add other properties or methods to the class, within the body `{}`, if you desire.

For each data class Kotlin will automatically create the `equals()`, `hashCode()`, and `toString()` methods. In addition, it provides a `copy()` method to make a copy of an instance while providing updated values for select properties. It also creates special methods that start with the word `component`—`component1()`, `component2()`, and so on—to access each property defined through the primary constructor. We'll refer to these methods as `componentN()` methods for convenience.

Here's an example data class that represents a task or a to-do item, annotated with the data keyword to convey the intent:

```kotlin
// taskdataclass.kts
data class Task(val id: Int, val name: String,
  val completed: Boolean, val assigned: Boolean)
```

Any property defined within the class body `{}`, if present, will not be used in the generated `equals()`, `hashCode()`, and `toString()` methods. Also, no `componentN()` method will be generated for those.

Creating an object of a data class

Continuing with the same example, let's create an object of the data class and exercise one of the generated methods, `toString()`.

```
val task1 = Task(1, "Create Project", false, true)

println(task1)
//Task(id=1, name=Create Project, completed=false, assigned=true)
println("Name: ${task1.name}") //Name: Create Project
```

taskdataclass.kts

The `String` returned by `toString()` has properties and their values listed in the same order in which they appeared in the primary constructor's parameter list. You may access any of the properties by their name, like the `name` property is accessed in our example.

For data classes, Kotlin generates a `copy()` method that creates a new object with all the properties of the receiver object copied into the result object. Unlike the `equals()`, `hashCode()`, and the `toString()` methods, the `copy()` method includes any property defined within the class, not just those presented in the primary constructor. Each parameter to the method receives a default argument, and we may pass an alternative value to any property using named arguments. Let's use this technique to copy task1 but provide a different value for a couple of properties:

```
// taskdataclass.kts
val task1Completed = task1.copy(completed = true, assigned = false)
println(task1Completed)
  //Task(id=1, name=Create Project, completed=true, assigned=false)
```

The newly created instance has copies of all properties from the original, but it has new values assigned to the `completed` and `assigned` properties. The `copy()` function only performs a shallow copy of primitives and references. The objects referenced internally are not deep copied by the method. This isn't an issue if the entire hierarchy of nested objects is immutable.

Destructuring a data class

The main purpose of the `componentN()` methods is for destructuring—see Destructuring. Any class, including Java classes, that has `componentN()` methods can participate in destructuring.

Here's an example to extract the `id` and `assigned` properties from an instance of `Task`:

```
val id = task1.id
val isAssigned = task1.assigned
println("Id: $id Assigned: $isAssigned") //Id: 1 Assigned: true
```

But that's boring and takes as many lines as the number of properties we want to extract. Instead, we can use the destructuring capability of data classes.

To destructure, we have to extract the properties in the same order as they appear in the primary constructor. However, we're not required to extract each and every property. If you don't want a property, simply leave it out from the request. If you need the value of a property that comes after a property you want to ignore, then use underscore—which is the international symbol for "I don't care." An example will help clarify this:

```
// taskdataclass.kts
val (id, _, _, isAssigned) = task1
println("Id: $id Assigned: $isAssigned") //Id: 1 Assigned: true
```

The local variables into which the properties should be extracted may be defined as `val`, as in the code we just used, or `var`. The `id` local variable is assigned the value of the `id` property from `task1`—for this Kotlin invokes the `component1()` method. We ignore the `name` and `completed` properties. Then we assign the `assigned` property's value to the local variable named `isAssigned`—here again, Kotlin uses the `component4()` method. If there were any more properties—which there aren't in this example—they'd be ignored and `_` wouldn't be required in the trailing positions.

The destructuring of data classes in Kotlin comes with a significant limitation. In JavaScript, object destructuring is based on property names, but, sadly, Kotlin relies on the order of properties passed to the primary constructor. If a developer inserts a new parameter in between current parameters, then the result may be catastrophic.

For example, suppose we add a new parameter in between the `name` and the `completed` parameter of `Task`'s primary constructor. Any use of the constructor will now have to be modified and recompiled. But, the effect on destructuring is severe. For example, the previous destructuring code won't cause any compilation error, but the `isAssigned` variable will now be assigned the value of the `completed` property instead of the `assigned` property, due to the change in the ordering of the parameters.

Instead of using type inference, if we specify the type during destructuring, we might get some relief in some situations, but specifying types won't entirely solve the issue in all cases. Whereas we can agree that changing the order of parameters is a poor programming practice, this issue reinforces the need to perform good automated testing, even for statically typed languages like Kotlin.

When to use data classes?

With the choices offered by Kotlin we have to decide when to use a data class vs. a regular class. Use a data class in these situations:

- You're modeling data more than behavior.
- You want `equals()`, `hashCode()`, `toString()`, and/or `copy()` to be generated, knowing that you may override any of these methods if you like.
- It makes sense for the primary constructor to take at least one property— no-argument constructors are not allowed for data classes.
- It makes sense for the primary constructor to take only properties.
- You want to easily extract data from the object using the destructuring facility (be aware that the extraction is based on the order of properties and not their names).

The next lesson concludes the discussion for this chapter.