Delegating Variables and Properties

We'll cover the following

- Delegating variables
- Delegating properties

In the examples so far, we focused on delegation at the class level. You may delegate get and set access to properties of objects and local variables too.

When you read a property or a local variable, internally Kotlin calls a `getValue()` function. Likewise, when you update a property or a variable, it calls a `setValue()` function. By providing as delegate an object with these two methods, you may intercept calls to read and write local variables and objects' properties.

Delegating variables

You can intercept access, both read and write, to local variables and alter what is returned and where and how the data is stored. To illustrate this facility, let's create a custom delegate to intercept access of `String` variables.

Suppose we're creating an application that takes users' comments. The text they enter may be displayed to other users, and we definitely want to be polite. So let's write a delegate that filters out an offensive word, like "stupid".

Let's look at a small script with no filtering:

```
var comment: String = "Some nice message"
println(comment)
comment = "This is stupid"
println(comment)
println("comment is of length: ${comment.length}")
```

Running this will produce the rude output, no surprise:

```
Some nice message
This is stupid
comment is of length: 14
```

Our objective is to replace the word “stupid” so when the string is printed it’s not as rude. For that, let’s create a class named `PoliteString` that has `getValue()` and `setValue()` methods with special signatures:

```
package com.agiledeveloper.delegates

import kotlin.reflect.KProperty

class PoliteString(var content: String) {
  operator fun getValue(thisRef: Any?, property: KProperty<*>) =
    content.replace("stupid", "s*****")

  operator fun setValue(thisRef: Any, property: KProperty<*>, value: String) {
    content = value
  }
}
```

PoliteString.kt

The class `PoliteString` is all set to act as a delegate. Kotlin doesn’t require any interface to be implemented, no ceremony—all it wants is the get method. If the delegate will target a mutable property or variable, then it demands the set method also. It’s that simple. If you’re unsure of the signature of these methods, refer to the interfaces `kotlin.properties.ReadOnlyProperty` and `kotlin.properties.ReadWriteProperty`. Though you don’t have to implement these interfaces, the `getValue()` and `setValue()` methods are the same as the ones in these symbolic interfaces.

The `PoliteString` class receives a mutable property named `content`. From the `getValue()` function we return the value in the `contents` string after cleansing any offending words in it. In the `setValue()` function we merely store the given value into the `content` property. The methods are marked with the annotation operator since they stand for the assignment operator `=` used for get and set.

We’ll have to compile this code into a jar file since it’s in a separate package. We’ll see the command for that soon. Let’s make use of this delegate with the code that contains the impolite comment.

PoliteString.kt

All code files are copied to end of the page...

politecomment.kts




We imported `PoliteString` and changed the `comment` variable to use the `PoliteString` in the declaration. It's a `String` that will delegate access to `PoliteString`. Here are the steps to compile and execute this example locally on your own system:

```
kotlinc-jvm com/agiledeveloper/delegates/PoliteString.kt -d polite.jar
kotlinc-jvm -classpath polite.jar -script politecomment.kts
```

The output from the code shows the offending word replaced:

```
Some nice message
This is s*****
comment is of length: 14
```

In the example, we're passing an instance of `PoliteString` as delegate. That's fine, but if you'd rather use a function that returns a delegate instance, instead of calling the constructor of a class after by, you may do so easily. Let's introduce a top-level function in the file `PoliteString.kt` within the package `com.agiledeveloper.delegates`:

```
//This function goes at the end of class PoliteString
fun beingpolite(content: String) = PoliteString(content)
```

We can now use this function instead of the `PoliteString` class:

```
import com.agiledeveloper.delegates.beingpolite

var comment: String by beingpolite("Some nice message")
```

We can improve on the `PoliteString` delegate to filter out many rude words. Then anywhere we want to keep things polite, we can pass the variable to the delegate to achieve that goal.

Delegating properties

Using the previous approach, we can not only delegate access to local variables but

also to properties of objects. When defining a property, instead of assigning a value, specify by and follow it with a delegate. Again here, delegate may be any object that implements `getValue()` for a val or read-only property, and both `getValue()` and `setValue()` for a read-write property.

In the next example, we'll use a variation of `PoliteString` delegate that we created earlier. Instead of storing the comment within the instance of `PoliteString`, we'll store it in a data source.

By design, the Kotlin standard libraries, `Map` and `MutableMap` (that we discussed in Using Map), can serve as delegates—the first for `val` properties and the second for `var` properties. That's because, in addition to providing the `get()` method, `Map` also has `getValue()`. Likewise, in addition to `set`, `MutableMap` also has the `setValue()` method. In the example, we'll use these as delegates to handle property access.

We'll first create the variation of `PoliteString` to store the comment value in a `MutableMap` that will serve as a data source:

```
import kotlin.reflect.KProperty
import kotlin.collections.MutableMap

class PoliteString(val dataSource: MutableMap<String, Any>) {
  operator fun getValue(thisRef: Any?, property: KProperty<*>) =
    (dataSource[property.name] as? String)?.replace("stupid", "s*****") ?: ""

  operator fun setValue(thisRef: Any, property: KProperty<*>, value: String) {
    dataSource[property.name] = value
  }
}
```

postcomment.kts

Instead of receiving a `String` parameter, here we receive a reference to a `MutableMap<String, Any>` that will hold the comment value. In the `getValue()` method, we return the value from the map for the property's name as key. If the value exists, we safely cast to `String` and cleanse it; otherwise, return an empty string. In the `setValue()` we merely save the given value into the map.

Next, we'll create a `PostComment` class that represents a blog post comment. Instead of storing the fields locally, its properties will delegate the get/set operations to a map. Let's take a look at the code, and then we'll discuss it further.

```
class PostComment(dataSource: MutableMap<String, Any>) {
```

```
  val title: String by dataSource
  var likes: Int by dataSource
  val comment: String by PoliteString(dataSource)


  override fun toString() = "Title: $title Likes: $likes Comment: $comment"
}
```

postcomment.kts

The primary constructor receives a parameter `dataSource` of type `MutableMap<String, Any>`, which will serve as a delegate to the properties of this class. The title is a read-only property of type `String` and is delegated to `dataSource`. Likewise, `likes`, which is of type `Int` but is a read-write property, is delegated to the same object, `dataSource`. The `comment` property, however, is delegated to `PoliteString`, which in turn will store and retrieve data from the same `dataSource`.

When the `title` property of an instance of `PostComment` is read, Kotlin will invoke the `getValue()` method of the delegate `dataSource` by passing the property name `title` to it. Thus, the map will return the value for the key `title`, if present.

The behavior for reading the `likes` property is similar to that of reading the `title` property. Unlike `title`, `likes` is mutable. When the `likes` property is written or set, Kotlin will invoke the `setValue()` method of the delegate passing the property name `likes` and the value. This will result in the value being stored for the key `likes` within the `dataSource` that is the `MutableMap<String, Any>`.

Read and write of the `comment` property will result in calls to `getValue()` and `setValue()`, respectively, on the `PoliteString` delegate. Via this delegate, the `comment` value will be fetched from or stored into the `dataSource`.

Let's create some sample data for a couple of blog post comments. We'll store them in a list of `MutableMap` instances.

```
val data = listOf(
    mutableMapOf(
      "title" to "Using Delegation",
      "likes" to 2,
      "comment" to "Keep it simple, stupid"),
    mutableMapOf(
      "title" to "Using Inheritance",
      "likes" to 1,
      "comment" to "Prefer Delegation where possible"))
```

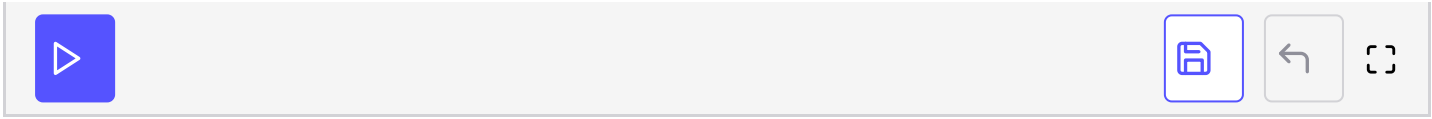
postcomment.kts

Now we can create an instance of PostComment using the data in the list.

```
val forPost1 = PostComment(data[0])
val forPost2 = PostComment(data[1])

forPost1.likes++

println(forPost1)
println(forPost2)
```



postcomment.kts

The instances of `PostComment` act as a façade around the `MutableMaps`—they delegate any access to their properties to their `dataSource`. Here's the output from the above code:

```
Title: Using Delegation Likes: 3 Comment: Keep it simple, s*****
Title: Using Inheritance Likes: 1 Comment: Prefer Delegation where possible
```

An object doesn't have to delegate all its properties. As we saw here, it may delegate properties to different delegates and may also internally store a few in its own fields.

QUIZ

Q Access of which two functions is delegated to the properties and variables?

Retake Quiz

We've seen how to create our own delegates. In the next lesson, we'll see some delegates that are built in to the Kotlin standard library.

Code Files Content !!!

⤋ ⤋

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


package com.agiledeveloper.delegates

import kotlin.reflect.KProperty

class PoliteString(var content: String) {
  operator fun getValue(thisRef: Any?, property: KProperty<*>) =
    content.replace("stupid", "s*****")

  operator fun setValue(thisRef: Any, property: KProperty<*>, value: String) {
```

```
    content = value
  }
}



--------------------------------------------------------------------------
|  politecomment.kts [1]
--------------------------------------------------------------------------


import com.agiledeveloper.delegates.PoliteString

var comment: String by PoliteString("Some nice message")
println(comment)

comment = "This is stupid"
println(comment)

println("comment is of length: ${comment.length}")




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