Chapter 3

Mixing Static and Non-Static Features

3.1 Introduction

In Chapter 1, we focused on static features. Non-static features were the topic of Chapter 2. In this chapter, we mix the two.

In order to focus on the general ideas without being distracted by the details of a particular class, we will illustrate most using a very simple class, the `Rectangle` class that we implemented in the previous chapter.

3.2 Incorporating an Invocation Counter

There are a variety of situations in which we may want to keep track of the number of times a particular method in a class is invoked. Such an invocation count provides a measure of how "popular" a method is, and may for example be used to determine if the efficiency of the method needs to be improved. The count can also be used in settings in which a cost (in dollars, time, or system resources) is associated with using the method and we need to profile the utilization of this cost.

As an example, suppose we were told to keep track of the number of times the `getArea` method of the `Rectangle` class is used and to report this count via a new method named `getCount`. How can we refactor the `Rectangle` class to incorporate this change? As an implementer, we first have to decide how to represent this additional information. Introducing an attribute, named `count`, of type `int` is a natural choice. If we expect an enormous number of invocations, we may choose an attribute of type `long` instead. As always, it should be a `private` attribute.

Should we make the attribute static or non-static? If we make it non-static, then it will be associated with instances of the class. Hence, we will have a separate counter for each instance of the class. This counter will record the number of invocations made on that particular instance. If, however, we make it static, then it will be associated with the class. Hence, there will be a single counter. This counter will keep track of the number of invocations made on all instances of the

class. In other words, the static count would be the sum of all the non-static counts. What we are after is an overall count so we make the attribute static.

We add the following declaration to the attribute section.

```
1 private static int count = 0;
```

Since the attribute is static, in accordance with our attribute initialization guidelines, we initialize the attribute together with its declaration, *not* in the constructor. This implies that no changes need to be made to the constructor section.

We need to increment this counter whenever `getArea` is invoked so we add the following line to its body.

```
1 Rectangle.count++;
```

Note how the attribute was referenced: since it is static we prefix it with the class name.

Finally, we add a new method that acts as an accessor for the new attribute.

```
1 public static int getCount()
2 {
3    return Rectangle.count;
4 }
```

Note that this method is static as it only involves static attributes.

3.3 Stamping a Serial Number on Objects

Assume that we want to associate unique serial number with every instance of a class. This mimics the situation in many factories where each product has a serial number. This pattern can be thought of as made up of two parts, one of which is similar to the invocation count that we discussed in the previous section, and the other is new.

The similarity with the invocation count becomes clear if we consider counting the number of times a constructor is called. In a multi-constructor class, it is common to have all constructors chained so they all ultimately call one of them. If we apply the invocation count idea to that constructor, we will be able to generate a serial number for each instance.

The second part of this pattern requires that we stamp the generated serial number on the instance, i.e. store it in the instance, and this can be done by adding a non-static attribute to hold it.

We therefore conclude that implementing this pattern calls for *two* attributes, one static to generate numbers serially and one non-static to hold the serial number per instance.

Let us apply this pattern by requiring that every rectangle we create from our `Rectangle` class carries a unique serial number that starts at 1 and increments serially. Based on the above discussion, this means we need to add the following declarations to the attribute section.

```
1 private static int count = 0;
2 private int number;
```

For the attribute `number`, we introduce an accessor and mutator.

Next, we need to increment `count`. Recall that after re-factoring the `Rectangle` class to avoid code duplication, we chained its three constructors so that two of them delegate to the two-parameter constructor which, in turn, delegates to the mutators. It is in this two-parameter constructor where we should increment the `count`.

Furthermore, since the attribute `number` was added to the state, it needs to be initialized and we do that also in the two-parameter constructor. Hence, we add two lines to the body of this constructor, as follows.

```
1 public Rectangle(int width, int height)
2 {
3    this.setWidth(width);
4    this.setHeight(height);
5    Rectangle.count++;
6    this.setNumber(Rectangle.count);
7 }
```

The serial number is available to the client via the accessor. Note, however, that we probably do not want to allow the client to change this number. This can be accomplished by making the mutator private.

3.4 Maintaining a Singleton

Imagine having a large, multi-class application in which several classes require the services of one particular class that represents a resource, e.g. a database connection. If every class were allowed to instantiate this resource class then we would end up with several instances of it and this could be wasteful (e.g. too many connections to the same database). What we need is the ability to ensure that only a single instance of a class can be created (per virtual machine) and to provide global access to this instance. The lone instance in this scenario is known as a *singleton* and in this section we discuss how to implement the so-called *singleton design pattern*.

First of all, how do we prevent clients of our class from instantiating it more than once? Anytime a client uses the `new` operator in conjunction with the class name, a new instance is created. Hence, it is clear from the outset that the client should not be able to use `new` and this, in turn, implies that our class must not have any public constructor. We therefore make all the class constructors private.

Now that we have prevented the client from instantiating our class, we need to find a way to deliver the lone instance to the client, and we do that through the return of a method. The method has got to be `static` or else the client cannot invoke it! Hence, this line of reasoning leads us to create a method with a header similar to the following.

```
1 public static ClassName getInstance()
```

How do we store the single instance so that the method can return it? We need to have an attribute that stores this instance thereby persisting it at the class level. This implies the attribute must be static.

```
private static ClassName instance = new ClassName(...);
```

Note that, as always for static attributes, we combine the declaration and the initialization. To initialize the instance, we invoke one of the private constructors. In the body of the `getInstance` method, we simply return the attribute as follows.

```
public static ClassName getInstance()
{
   return ClassName.instance;
}
```

Let us apply this by requiring that our `Rectangle` class be a singleton. We start with making all constructors of the `Rectangle` class private. Based on the above discussion, we need to introduce a static attribute to hold the singleton. We therefore add the following to the attribute section.

```
private static Rectangle instance = new Rectangle();
```

Furthermore, we add the following method to our `Rectangle` class.

```
public static Rectangle getInstance()
{
   return Rectangle.instance;
}
```

Note that the singleton returned by the `getInstance` method represents an empty rectangle, i.e. a rectangle with width zero and height zero. This is not a limitation because the client can use the public mutators to mutate this instance to any desired width or height. The key point is that, regardless of width and height, there is at most one `Rectangle` object in memory at all times.

The `Rectangle` object is created when the `Rectangle` class is loaded into memory. If the invocation of the constructor is expensive (in terms of time or memory), we may want to delay the creation of the `Rectangle` object until the first invocation of the `getInstance` method. In that case, we initialize the attribute `instance` to `null`. We will create the singleton upon receiving the very first request from the client. The `null` value acts as a flag to enable us to determine if the singleton has or has not been created yet.

```
private static Rectangle instance = null;
```

In this case, the `getInstance` method must determine whether an instance has been created yet. If it has, then a reference to it should be returned; otherwise, it should be created by means of the private constructor, stored, and then returned.

```
public static Rectangle getInstance()
{
   if (Rectangle.instance == null)
   {
      Rectangle.instance = new Rectangle();
   }
```

```
7     return Rectangle.instance;
8 }
```

3.5 Enforcing One Instance Per State

Rather than allowing just a single instance of the class, as we did in the previous section, in this section we want to ensure that there is only one instance of our class for every possible combination attribute values. In other words, we need to create a "singleton for every state."

Consider the following code fragment in a client app and try to predict its output.

```
1 String s1 = "York";
2 String s2 = "York";
3 output.println(s1.equals(s2) + " - " + (s1 == s2));
```

The output is `true - true`. While it is not surprising that `s1.equals(s2)` returns `true` (after all, the two strings are made up of the same character sequence), it may be unexpected to see that `s1 == s2` returns `true` as well. After all, the `String` objects `s1` and `s2` may be thought to be different objects residing at different addresses in memory. In an effort to save memory, the Java compiler creates only one instance for each string literal such as `"York"` and makes all references point to it. The compiler is therefore enforcing a single instance per state, where state in this case refers to the characters that make up the string.[1]

How can we enforce a single instance per state? In order to prevent the client from controlling instantiation, we must make all constructors private, as was done for the singleton. Moreover, we better make all mutators private (and, hence, the class becomes immutable as we have seen in Section 2.3.2). Otherwise, the client can change the state and thus create two instances having the same state. And for the singleton pattern, we should provide a static method named, for example, `getInstance`. This time, the client should be able to pass arguments to it to specify the desired state for the requested instance. The body of this method must determine whether an instance of the desired state has been already created. If it has, then return it; otherwise, create the instance using a private constructor, store it, and then return it. The attribute to store the instances has to be static and has to be able to hold many instances, corresponding to many states. It must therefore be a collection. One can, for example, use a `Map` whose key represents the state and whose value represents the lone instance that has that state. The facts that mutators must be private and instance storage must be in a collection make this scenario slightly different from the singleton design pattern.

Let us apply this by requiring that our `Rectangle` class allows only one instance for a given width and height. Based on the above discussion, we need to add a static `Map` attribute to hold the created instances. We add the following declaration to the attribute section:

```
1 private static Map<String, Rectangle> instances = new TreeMap<String,
      Rectangle>();
```

[1] It should be noted that the compiler does this for string literals only. If the character sequence is constructed using concatenation or any other mechanism, different instances may be created even if they share the same character sequence.

Note that we initialized the attribute to an empty map. Next, we make the three constructors and the two mutators of the class private. Finally, we add the following method.

```
1  public static Rectangle getInstance(int width, int height)
2  {
3     String key = width + "-" + height;
4     Rectangle instance = Rectangle.instances.get(key);
5     if (instance == null)
6     {
7        instance = new Rectangle(width, height);
8        Rectangle.instances.put(key, instance);
9     }
10    return instance;
11 }
```

Note that in order to facilitate storage in the map, we need to come up with a unique key that identifies the state. In the above implementation we simply concatenated the width and the height after delimiting them by a hyphen. Alternatives will be discussed in the next section.

3.6 Check you Understanding

Refactor the `Rectangle` class developed in previous section in the following ways.

- Instead of using a `String` as key, use the class `java.awt.Point` for the keys of the `Map`.
- Instead of using a `String` as key, develop a class `Pair` with two attributes of type `int` and use this class for the keys of the `Map`.
- Make the class `Pair` generic and use this class for the keys of the `Map`.
- Instead of using a `Map`, use a `List` to store the instances.
- Instead of using a `Map`, use a `Set` to store the instances.