Immutable Objects

Venkat Subramaniam
venkats@agiledeveloper.com
http://www.agiledeveloper.com/download.aspx

Abstract

An object is said to be immutable if its state can not be modified once the object is created. Examples of immutable objects are String class in both Java and .NET. Making an object immutable has its advantages: resource sharing, simplicity and thread safety. This article presents the advantages of immutable objects, compares it with Singleton and shows two ways to implement immutable objects.

State of an object

The state of an object is determined by the values of its fields. When an object is created, it is said to be in an initial state. This state is largely determined and the validity of this state is ascertained by the use of constructors. Typically a class has two kinds of methods: **query methods** and **modifiers (or mutators)**. A query method simply returns either values of certain fields or computes some property based on the fields and returns to the caller. These methods do not modify the state of an object. That is, invoking query methods does not affect the state of an object. On the other hand, modifiers may change the state of an object. Figure 1 shows the states and possible state transitions of a hypothetical elevator object.

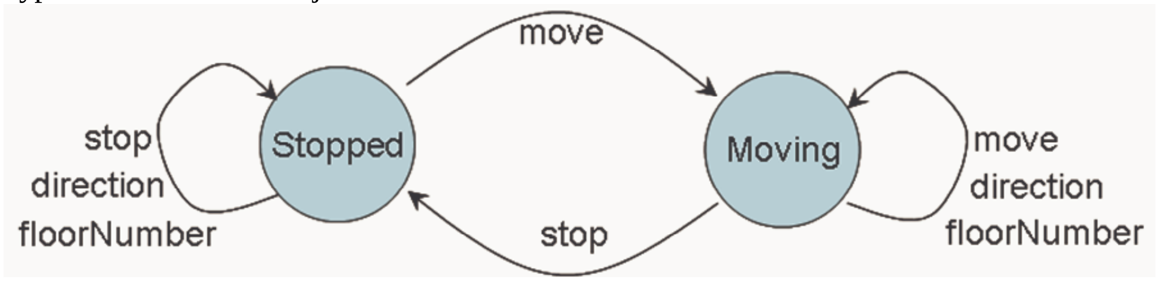

Figure 1 State change diagram for a hypothetical elevator

Stopped is the elevator's initial state. It has two modifiers Stop and move. The other two methods direction and floorNumber are query methods. Invoking the move results in a state change if the current state is "Stopped." Similarly, invoking the Stop results in a state change if the current state is "Moving." Calling the direction method to find the direction in which the elevator is moving (or would be moving) and also calling the floorNumber to obtain the floor number where the elevator is located (or just left) has no effect on the state.

Controlling the instances

Typically a class may have any number of instances. Of course, if a class is abstract, there will be zero direct instances for it. Some times you may want to control the number of instances of a class. For instance, consider a DriverManager which manages the database drivers in an application. There is no need to have more than one instance of this object. One may ask, why not make all the methods of the class static. Doing so would not require creation of any instances at all and isn't that more efficient. The problem is static methods are not polymorphic and if we need to extend the behavior of the DriverManager, it would not be possible without modifying the class. Restricting the number of instances of the class to one will allows us to extend (inherit) from the class to get a modified behavior. The singularity may be achieved by using either the **Singleton Pattern**[2] or the **Monostate pattern**[3]. The two approaches take entirely different approaches. While singleton maintains the singularity extrinsically, the Monostate maintains it intrinsically. Note that a single class is likely to have both modifiers and query methods. However, limiting the number of instances to one allows significant saving on resource usage.

Resource sharing

Limiting the number of instances to one allows better utilization of resources. It eliminates the possibility of creation of several instances of this object, all of which do the same thing. However, there are times when our abstraction requires that multiple instances of a class be created. Each instance may truly be independent of other instances (as far as state is concerned that is).

Let's consider a special case, however. Let us assume that we have a class called Brush. A brush represents a resource that may be used to paint on a window. Let's keep it simple and assume that we would create brushes in different colors. If the color is an argument we give to the constructor of the Brush, we would create an object as shown in Figure 2.

```
Brush blueBrush = new Brush("blue");
Brush redBrush = new Brush("red");
```

Figure 2 Creating Red and Blue Brushes

In a large application, we may have a need for a blue brush in several methods. In each of those methods we would create an object of brush as shown above. This may result in of blue brushes (and red brushes) being created in our application. Comparing the state of blue brushes, one may find that, it is equal. Similarly, the state of red brushes is equal as well. Creating multiple instances of brushes for each color is wastage of resource. How do we share the resources, and do so in a way that it is easier for the programmer? Clearly we are not striving for singularity (as provided by a singleton or monostate) will not help here since we do want multiple instances of the Brush, but one per each color through out the application.

Immutable object

Resource sharing applies to objects whose state does not change once created. An object whose state does not change is called an **immutable object**[1]. All the methods of this class are query methods; it has no modifiers (this is not entirely true, as we will see later, in C++). Consider the example of String in Java (and .NET)[4]. The String class has a number of methods (over 50 in Java). Looking closely at methods like toLower() shows that these methods do not actually modify the object on which the method is invoked. Instead they return a different object with the modified string value. This allows for the memory used by a String to be optimized. Let us say we create String s1 = new String("hello"); and String s2 = new String("hello"); in two different methods of an application. These two string objects will end up sharing the memory where the characters of "hello" are stored.

Of course, if these two string objects internally share the memory where the string literal is stored, allowing that to be modified using either one of the string objects will result in undesirable change to the other. In order to promote better memory utilization and optimization, String objects are implemented as immutable objects. Immutable objects also provide thread safely.

Enforcing Immutability

In C++[5], one could create an object as a **const object**. For instance, I may create an object as "*const X obj;*" While the class X may have both modifiers and query methods, the compiler will allow only query methods to be called on the object obj. Calling a modifier on the object results in compilation error. While the usage of const gives a number of benefits, its usage was also some what difficult for new programmers to follow. Java decided not to have this particular capability (the final in const is quite not as powerful as the const in C++). In fact, in order to safe guard from any C++ code ported to Java, the keyword const in Java is a forbidden keyword. The only way to make an object immutable in Java is by not providing any modifiers.

Enforcing Resource Sharing

Asking the user of our Brush to keep an array or list of references globally in an application and sharing the different color brushes is impractical. At the very least there is no guarantee that every user of our class follows the recommendation and optimizes the resource usage. How do we then share the resource in a way that it can be consistently and easily enforced? There are two ways to do this. One is to force the user to create an object using a static method (similar to the singleton approach). The other is to share the internals of the objects like how the String in Java is implemented. We will discuss these two approaches in the next two sections.

Extrinsic sharing

In extrinsic sharing the user is not allowed to directly create the immutable object. A call to *new Brush("blue");* will result in a compilation error. We will force the user to use a static method to create the object, such as *Brush.get("blue");*. Within our class, we will maintain a collection of references. Based on the input parameter, we will return a reference to an object from the collection. An object is created only if one does not exist. We call it extrinsic sharing because the user of the class is aware of resource sharing. Comparing the references obtained from two calls to the creator method with the same argument will show that they are the identical. For instance, if we have two statements *Brush b1 = Brush.get("blue"); Brush b2 = Brush.get("blue");* then, *b1 == b2* is true.

Let's take a look at how we can implement our Brush class. The code for the Brush class is shown below:

```
import java.util.Hashtable;

public class Brush
{
	private String theColor;
	private static Hashtable brushes = new Hashtable();
```

```
        private Brush(String color)
        {
                theColor = color;
        }

        public static Brush get(String color)
        {
                Brush result = null;

                if (brushes.containsKey(color))
                {
                        result = (Brush) brushes.get(color);
                }
                else
                {
                        result = new Brush(color);
                        brushes.put(color, result);
                }

                return result;
        }

        public String getColor()
        {
                return theColor;
        }
}
```

Notice how the get method checks to see if a Brush for the given color is in the Hashtable. If it is found, that is returned. If not, a new one is created, first it is added to the Hashtable and then a reference to it is returned. A test code that utilizes this is shown below:

```
public class TestCode
{
        public static void main(String[] args)
        {
                Brush b1 = Brush.get("blue");
                Brush b2 = Brush.get("blue");
                Brush b3 = Brush.get("red");

                System.out.println("b1's color is " + b1.getColor());
                System.out.println("b2's color is " + b2.getColor());
                System.out.println("b3's color is " + b3.getColor());

                System.out.println("is b1 == b2? " + (b1 == b2));
                System.out.println("is b1 == b3? " + (b1 == b3));
        }
}
```

The output from running java TestCode is shown below:

```
b1's color is blue
b2's color is blue
b3's color is red
```

```
is b1 == b2? true
is b1 == b3? False
```

As can be seen from the above output, the user receives one object for each color.

Intrinsic sharing

In intrinsic sharing the user is allowed to directly create the immutable object. A call to *new Brush("blue");* will result in a new object of Brush being created and returned. Within our class, however, we will maintain a collection for the internals of the object. In this case, instead of the object of Brush being shared, the internals of it are shared. We call it intrinsic sharing because the user is not aware of the sharing of resources. Comparing the references obtained from two calls to the constructor with the same argument will show that they are the different. For instance, if we have two statements *Brush b1 = new Brush("blue"); Brush b2 = new Brush("blue");* then, *b1 == b2* is false.

Let's take a look at how we can implement our Brush class. The code for the Brush class is shown below:

```
import java.util.Hashtable;

public class Brush
{
        private static class BrushInternals
        {
                private String theColor;

                public BrushInternals(String color)
                {
                        theColor = color;
                }
                public String getColor()
                {
                        return theColor;
                }
                public String toString()
                {
                        return "internals " + hashCode();
                }
        }

        private static Hashtable brusheInternalsCollection = new Hashtable();
        private final BrushInternals internals;

        public Brush(String color)
        {
                internals = get(color);
        }

        private static BrushInternals get(String color)
        {
                BrushInternals result = null;

                if (brusheInternalsCollection.containsKey(color))
                {
```

```
                result = (BrushInternals) brusheInternalsCollection.get(color);
            }
            else
            {
                result = new BrushInternals(color);
                brusheInternalsCollection.put(color, result);
            }

            return result;
    }

    public String getColor()
    {
            return internals.getColor();
    }

    public String printInternals()
    {
            return internals.toString();
    }
}
```

Notice how the get method checks to see if a Brush for the given color is in the Hashtable. If it is found, that is returned. If not, a new one is created, first it is added to the Hashtable and then a reference to it is returned. A test code that utilizes this is shown below:

```
public class TestCode
{
    public static void main(String[] args)
    {
            Brush b1 = new Brush("blue");
            Brush b2 = new Brush("blue");
            Brush b3 = new Brush("red");

            System.out.println("b1's color is " + b1.getColor());
            System.out.println("b2's color is " + b2.getColor());
            System.out.println("b3's color is " + b3.getColor());

            System.out.println("is b1 == b2? " + (b1 == b2));
            System.out.println("is b1 == b3? " + (b1 == b3));

            System.out.println("b1's color is " + b1.printInternals());
            System.out.println("b2's color is " + b2.printInternals());
            System.out.println("b3's color is " + b3.printInternals());
    }
}
```

The output from running java TestCode is shown below:

```
b1's color is blue
b2's color is blue
b3's color is red
is b1 == b2? false
is b1 == b3? false
b1's color is internals 28168925
```

```
b2's color is internals 28168925
b3's color is internals 15655788
```

As can be seen from the above output, the user receives different object. However, it should be noted that the internals are shared as seen for the output as well. In this sense, the Brush is more of a Façade[2].

Pros and Cons of Extrinsic vs. Intrinsic approach

The extrinsic approach is simple to understand and makes the user realize what's going on. However, inheriting from this class is problematic. Generally, such classes are made final (not inheritable from). In the case of the intrinsic approach, the derived class benefits from resource sharing of the base as well. However, the resource sharing is transparent to the user.

Conclusion

Resource sharing is critical in large applications. Immutability is a property of an object which does not change its state once created. Immutable objects are great candidates for resource sharing and provide thread safely. It is important to provide an easy and consistent mechanism to shared resources in an application. We discussed the benefits of these and related concepts. Then we looked at two ways to implement immutable objects and resource sharing.

References

1. Joshua Bloch, *Effective Java Programming Language Guid*, Addison-Wesley, 2001.
2. Gamma, et. al., *Design Patterns,* Addison-Wesley, 1994.
3. Ball, et. al., *Monostate Classes: The power of One – in More C++ Gems*, compiled by Robert Martin.
4. Venkat Subramaniam, *Strings in Java and .NET*, http://www.agiledeveloper.com/download.aspx, March 2003.
5. Bjarne Stroustrup, *The C++ Programming Language*.