De La Salle University
Manila

Immutable Object

John P. Combalicer

Objective

To learn, understand and apply the concept of immutable object in Object Oriented Programming application…

Topic

Immutable Objects

- A Synchronized Class Example
- A Strategy for Defining Immutable Objects

What is an immutable object?

Immutable objects whose state (i.e. the object's data) does not change once it is instantiated (i.e. it becomes a read-only object after instantiation).

Immutable classes are ideal for representing numbers (e.g. java.lang.Integer, java.lang.Float, java.lang.BigDecimal etc are immutable objects), enumerated types, colors (e.g. java.awt.Color is an immutable object), short lived objects like events, messages etc.

The immutable class are written in following guidelines:

A class is declared as final. (The final class cannot be extended)

All its fields are final (final fields cannot be mutated once assigned).

Do not provide any methods that can change the state of the immutable object in any way – not just setXXX methods, but any methods which can change the state.

The “this” reference is not allowed to escape during construction from the immutable class and the immutable class should have exclusive access to fields that contain references to mutable objects like arrays, collections and mutable classes like Date etc by:

- Declaring the mutable references as private.
- Not returning or exposing the mutable references to the caller

Benefits:

Immutable classes can greatly simplify programming by freely allowing you to cache and share the references to the immutable objects without having to defensively copy them or without having to worry about their values becoming stale or corrupted.

Immutable classes are inherently thread-safe and you do not have to synchronize access to them to be used in a multi-threaded environment. So there is no chance of negative performance consequences.

Eliminates the possibility of data becoming inaccessible when used as keys in HashMaps or as elements in Sets

```
public class SynchronizedRGB {
  //Values must be between 0 and 255.
  private int red;
  private int green;
  private int blue;
  private String name;
  private void check(int red, int green, int blue) {
    if (red < 0 || red > 255
        || green < 0 || green > 255
        || blue < 0 || blue > 255) {
      throw new IllegalArgumentException();
    }
  }
  public SynchronizedRGB(int red, int green, int blue, String name) {
    check(red, green, blue);
    this.red = red;
    this.green = green;
    this.blue = blue;
    this.name = name;
  }
  public void set(int red, int green, int blue, String name) {
    check(red, green, blue);
    synchronized (this) {
      this.red = red;
      this.green = green;
      this.blue = blue;
      this.name = name;
    }
  }
  public synchronized int getRGB() {
    return ((red << 16) | (green << 8) | blue);
  }
  public synchronized String getName() {
    return name;
  }
  public synchronized void invert() {
    red = 255 - red;
    green = 255 - green;
    blue = 255 - blue;
    name = "Inverse of " + name;
  }
}
```

A Synchronized Class Example

The class, SynchronizedRGB, defines objects that represent colors. Each object represents the color as three integers that stand for primary color values and a string that gives the name of the color.

Sample Program Interpretation

SynchronizedRGB must be used carefully to avoid being seen in an inconsistent state. Suppose, for example, a thread executes the following code:

```
SynchronizedRGB color = new SynchronizedRGB(0, 0, 0, "Pitch Black");
...
int myColorInt = color.getRGB();      //Statement 1
String myColorName = color.getName(); //Statement 2
```

If another thread invokes color.set after Statement 1 but before Statement 2, the value of myColorInt won't match the value of myColorName. To avoid this outcome, the two statements must be bound together:

```
synchronized (color) {
    int myColorInt = color.getRGB();
    String myColorName = color.getName();
}
```

This kind of inconsistency is only possible for mutable objects — it will not be an issue for the immutable version of SynchronizedRGB.

A Strategy for Defining Immutable Objects

The following rules define a simple strategy for creating immutable objects. Not all classes documented as "immutable" follow these rules. This does not necessarily mean the creators of these classes were sloppy — they may have good reason for believing that instances of their classes never change after construction. However, such strategies require sophisticated analysis and are not for beginners.

1. Don't provide "setter" methods — methods that modify fields or objects referred to by fields.
2. Make all fields final and private.
3. Don't allow subclasses to override methods. The simplest way to do this is to declare the class as final. A more sophisticated approach is to make the constructor private and construct instances in factory methods.
4. If the instance fields include references to mutable objects, don't allow those objects to be changed:
    1. Don't provide methods that modify the mutable objects.
5. Don't share references to the mutable objects. Never store references to external, mutable objects passed to the constructor; if necessary, create copies, and store references to the copies. Similarly, create copies of your internal mutable objects when necessary to avoid returning the originals in your methods.

JPC

Applying this strategy to SynchronizedRGB results in the following steps:

1. There are two setter methods in this class. The first one, set, arbitrarily transforms the object, and has no place in an immutable version of the class. The second one, invert, can be adapted by having it create a new object instead of modifying the existing one.
2. All fields are already private; they are further qualified as final.
3. The class itself is declared final.
4. Only one field refers to an object, and that object is itself immutable. Therefore, no safeguards against changing the state of "contained" mutable objects are necessary.

After these changes, we have ImmutableRGB:

```
final public class ImmutableRGB {
//Values must be between 0 and 255.
final private int red;
final private int green;
final private int blue;
final private String name;
private void check(int red, int green, int blue) {
if (red < 0 || red > 255
|| green < 0 || green > 255
|| blue < 0 || blue > 255) {
throw new IllegalArgumentException();
}
}
public ImmutableRGB(int red, int green, int blue, String name) {
check(red, green, blue);
this.red = red;
this.green = green;
this.blue = blue;
this.name = name;
}
public int getRGB() {
return ((red << 16) | (green << 8) | blue);
}
public String getName() {
return name;
}
public ImmutableRGB invert() {
return new ImmutableRGB(255 - red, 255 - green, 255 - blue,
"Inverse of " + name);
}
}
```

Summary

Immutable objects are much easier to work with than mutable objects. They can only be in one state and so are always consistent, they are inherently thread-safe, and they can be shared freely. There are a whole host of easy-to-commit and hard-to-detect programming errors that are entirely eliminated by using immutable objects, such as failure to synchronize access across threads or failing to clone an array or object before storing a reference to it. When writing a class, it is always worthwhile to ask yourself whether this class could be effectively implemented as an immutable class. You might be surprised at how often the answer is yes.

Reference

Brian Goetz. To mutate or not to mutate?. *Java theory and practice.* Retrieved November 18, 2008, from http://www.ibm.com/developerworks/java/library/j-jtp02183.html

JAVA Essential. Retrieved November 18, 2008, from http://java.sun.com/docs/books/tutorial/essential/concurrency/immutable.html

Achaffan, Stevetuff. Immutable object in Java. Retrieve November 18, 2008, from http://it.toolbox.com/wiki/index.php/Immutable_object_in_Java