# Critiquing Java Source Code

K.Rangaraajan
Man Machine Systems
mms@giasmd01.vsnl.net.in

Code review by peers is known to significantly enhance the quality of software, and is therefore an integral part of the development process adopted by many software organizations all over the world. Some of the problems that a code review can reveal are

1. missing or incorrect functionality
2. use of incorrect or inferior alogorithms
3. code documentation errors
4. coding guideline violations
5. coding errors
6. code design deficiencies, and
7. efficiency concerns

In this article, my intention is to document certain code design deficiencies that I have come across in the process of studying several class libraries. A word of caution is appropriate here. When I talk about "deficiencies", I do not necessarily mean that they stop the code from working, but just that a "good" programmer might not be accused of these! I do agree that these guidelines are sometimes debatable, but I will present them anyway. The examples I present here are contrived, but let me assure you that they are inspired by real, professional (possibly commercial) code. For various reasons, I cannot name the actual sources, but I am sure you will stumble upon such code real soon.

## Inner Class Relationships

There are two aspects to this:

1. Making a class an inner class of another class
2. Making a nonstatic class static.

**Case 1:** Consider the following two Java classes (in the same package).

```
class A {
   int val;
   public void ff() { /* ... */ }

   public static void main(String[] args) {
      A a1 = new A();
      B b1 = new B(a1, 3.2);
      b1.gg();
      // etc
   }
}

class B {
   B( A a, double s) {
      aobj = a; s1 = s;
   }

   void gg() {
      aobj.val += 6;
   }
```

```
    private A aobj;
    private double s1;
    // ...
}
```

A look at the two classes reveals that whenever a B object is created, it requires an A object, suggesting a tight coupling of B to A. In such a case, it may be good idea to define B as an inner class of A as shown:

```
class A {
int val;
public void ff() { /* ... */ }
    class B {
        B(double s) {
            s1 = s;
        }
        void gg() {
            val += 6;
        }
        private double s1;
        // ...
    }
    public static void main(String[] args) {
        A a1 = new A();
        B b1 = a1.new B(3.2);
        b1.gg();
        // etc
    }
}
```

**Case 2**: Sometimes, an inner class does not access any element of the enclosing class, but is defined inside the other class to reflect a logical nesting, something like the following:

```
class Tree {
Node root;
    class Node {
        Node left, right;
        Object data;
        // etc.
    }
    void addElement(Object el) {
        //create a Node object
        //insert it at the correct place
    }
    // etc
}
```

In this case, since Node is a logical component of a Tree, and other classes may not need it, it makes sense to define Node inside of Tree. However, it must be more appropriately a static inner class (sometimes called "nested" class).

**Confusion Between Local Variable and Instance Variable**

I have come across a surprising number of examples where the programmer uses instance variables of a class like local variables. Look at the class given here.

```
class A {
   private int i;
   private int j;
   public void f() {
      i = 9;
      // ...
   }
   private void g() {
      j += 23;
      i = 432;
      // ...
   }
}
```

As the structure itself reveals, the instance variable "i" is always assigned to in a method before it is read. This implies that previous value of the variable is never used. Instance variables are intended, in general, to remember state, not as simple value holders. I would prefer to rewrite the above example as follows:

```
class A {
   private int j;
   public void f() {
      int i = 9;
      // ...
   }
   private void g() {
      j += 23;
      int i = 432;
      // ...
   }
}
```

**Static vs. Instance Variable**

Some programmers do not take the extra care to check whether a variable defined in a class should be instance specific, or it is enough if it is "static". What do you think of the following class?

```
class MyArray {
   private final int MAXSIZE = 100;
   private Object[] obarray;
   public MyArray() {
      obarray = new Object[MAXSIZE];
   }
   public void addElement(Object o) {
      // ..
   }
   public Object getElement(int indx) {
      // ...
   }
}
```

The instance variable "MAXSIZE" is final, and initialized directly. Can we judiciously make this a "static" member? I belive we should. Something like this does not break the code, but tends to differentiate a good Java programmer from the rest of the breed.

**Public Constructor in a Nonpublic Class**

If a class is not public, does it make sense to have a public constructor in the class? Look at the following class structure:

```
class A {
   public A() { /* ... */ }
   // ...
}
```

A member is declared public to be made available to classes in other packages. In particular, a constructor is made public to allow instantiation of that class objects from other packages. But if the class itself is not public, how can such an instantiation occur (the name of the class itself is not available to other packages)? So, I believe that the constructor of a nonpublic class need not be public. What about other instance methods of the class? This suggestion does not cover other methods. The following is a perfectly valid thing to do:

```
public class A {
   public A createObject() {
      return new B();
   }
   public void ff() {
      //...
   }
}

class B extends A{
   B() { /* ... */ }
   public void ff() {
      // override base method
   }
}
```

In this case, the overriding method B.ff() must be public, though the constructors need not be!

**Having A Return Expression in Try as well as Finally Block**

The *return* statement in a *finally* block nullifies the effect of *return* found within the corresponding *try* block. The runtime system always executes the statements within the finally block regardless of what happens within the try block. So a return statement in a try block will have no effect if there is one in the finally block.

```
class MyOutStream {
   static final int good = 1;
   static final int bad = 0;
   private Stream out = null;
   int Write(byte b) {
      try {
         out.Open();
         out.Write(b);
         return MyOutStream.good;
      }
      catch(OpenError e) {
         System.out.println("Open Error");
         return MyOutStream.bad;
      }
      catch(WriteError e) {
         System.out.println("Write Error");
         return MyOutStream.bad;
```

```
        }
        finally {
            // This is what is actually returned!
            return MyOutStream.bad;
        }
    }
    // ......
}
```

I would have expected a compiler to flag the above code as error, or at least generate a warning, but I get nothing from Javac.

**Class with Only Static Elements**

If a class contains only static elements, is it a good idea to prevent instantiation by defining private or protected constructor? Consider the following class structure:

```
class DeviceStatus {
    private static int DONE = 1;
    private static int BUSY = 2;
    private static int FREE = 3;
    public static boolean isFree(int flag) {
        return flag == FREE;
    }
    public static boolean isBusy(int flag) {
        return flag == BUSY;
    }
    public static boolean Done(int flag) {
        return flag == DONE;
    }
}
```

There does not appear to be any advantage in being able to instantiate objects of this class since there is no "instance-specific" behaviour. In this case, my preference is to define a private constructor that disallows instantiation (however, if derivation is to be enabled, I might consider making the constructor protected). The modified class is

```
class DeviceStatus {
    private static int DONE = 1;
    private static int BUSY = 2;
    private static int FREE = 3;
    private DeviceStatus() { } // Empty. To prevent instantiation!
    public static boolean isFree(int flag) {
        return flag == FREE;
    }
    public static boolean isBusy(int flag) {
        return flag == BUSY;
    }
    public static boolean Done(int flag) {
        return flag == DONE;
    }
}

class TestDevice {
    public static void main(String[] args) {
```

```
        int devstat;
        // .. Get device status somehow
        // Check the status
        // DeviceStatus d = new DeviceStatus(); // This line will not compile
        if( DeviceStatus.DONE(devstat) )
            System.out.println("Done!");
        // etc.
    }
}
```

**Automated Code Analysis**

One interesting issue is whether the code review process can be automated by a tool, and if so, what kinds of problems can the tool identify. I believe it is safe to say (with current technology) that no tool, however good it may be, can replace an intelligent programmer. The benefit of automated review is that it can bring reasonably good quality to everyone's desktop, by removing individuality from the picture. I know of three tools in this category as of today, and we are the developers of one of them. These tools are JStyle™ (www.mmsindia.com/jstyle.html), Metamata Code Audit™ (www.metamata.com) and Code Wizard for Java™ (www.parasoft.com). Two of these compute useful metrics from the Java code, in addition to code critiquing.