

Early Binding in Java Considered Harmful

Tim Howe
Sun Microsystems
500 Eldorado Blvd.
Broomfield CO 80021
vsync@central.sun.com

20 March 2000

1 Introduction

1.1 Object orientation and type checking

One of the strengths of Java is its strong object orientation, implemented at runtime as well as compilation. This protects the programmer from a number of errors which may inadvertently (or maliciously) occur. For example, the following code will not run:

```
SimpleBad.java
```

```
class SimpleBad {
    long addAndReturn(Integer i) {
        return i.longValue() * 5;
    }

    public static void main(String arg[]) {
        System.out.println(addAndReturn("foo"));
    }
}
```

Most compilers will not even produce a class file for this program, because of its attempt to pass a `String` to a method expecting an `Integer`. However, should it somehow get past the compiler, due to the methods being in separate source files, or for some other reason, the virtual machine will still catch this and halt execution. This makes Java programs virtually immune to such problems as pointer errors and buffer overflows.

The other major aspect of Java's object orientation is that it allows the creation of "black boxes"—self-contained classes, which can be combined with other classes and modified at will, as long as their external behavior does not change.

For example, Sun's JavaMail API defines a standard set of classes for sending and receiving mail messages. If Sun's developers decided that these classes were implemented in an inefficient manner, they could rewrite them completely, and programs utilizing them would automatically follow the correct behavior. In addition, other developers could write their own versions of the API, as long as they followed the same interface.

1.2 The static and final keywords

The `static` keyword is used when no instance of a class need be created to access a particular variable or method. For example, the `java.awt.Color` class defines a number of colors as static methods, because one should not need to possess one color in order to access another. Of course, one can also access them as instance methods, a feature which can be good or bad, depending on the situation.

The `final` keyword is used for two purposes: to state that a class or method may never be extended or overloaded, or to specify that the value of a variable may not be modified. This can be used to ensure the integrity of a class, or to protect against security flaws. For example, the `String` class is `final`, so any method expecting a `String` can be assured of getting only that.

2 A Real-Life Experience

While implementing the communications protocol for a client-server application, I defined a class called `SolarWind` (the name of the protocol). This was to be utilized by both the client and server applications, and would define the tokens acceptable to the protocol. An excerpt is included here:

SolarWind.java (version 1)

```
public class SolarWind {
    public static class Core {
        public static final String          // Core commands
            BEGIN = "begin",
            DATA = "data",
            END = "end",
            CANCEL = "cancel",
            QUIT = "quit",
            OK = "ok",
            ERROR = "error",
            WARN = "warn",
            PING = "ping",
            PONG = "pong",
            AUTH = "auth",
            PASS = "pass";
    }
}
```

```
}  
}
```

The tokens were declared as `static` because the protocol itself is an abstract entity, and `final` because I did not want any extensions to the protocol mangling the tokens which already existed.

As development progressed, I decided I wanted to make it harder to misuse the protocol. As it stood, the method that sent data over the stream took a single `String` argument. If a programmer hardcoded strings into his or her code, it would then be possible for misspellings and other hard-to-find errors to occur. Therefore, I added another inner class and modified the definitions as follows:

SolarWind.java (version 2)

```
public class SolarWind {  
    public static class ProtocolString {  
        String data;  
  
        ProtocolString(String data) {  
            this.data = data;  
        }  
  
        public boolean matches(String st) {  
            return (data.equalsIgnoreCase(st));  
        }  
  
        public boolean matchesWithCase(String st) {  
            return (data.equals(st));  
        }  
  
        public String toString() {  
            return data;  
        }  
    }  
  
    public static class Core {  
        public static final ProtocolString // Core commands  
            BEGIN = new ProtocolString("begin"),  
            DATA = new ProtocolString("data"),  
            END = new ProtocolString("end"),  
            CANCEL = new ProtocolString("cancel"),  
            QUIT = new ProtocolString("quit"),  
            OK = new ProtocolString("ok"),  
            ERROR = new ProtocolString("error"),  
            WARN = new ProtocolString("warn"),
```

```

        PING = new ProtocolString("ping"),
        PONG = new ProtocolString("pong"),
        AUTH = new ProtocolString("auth"),
        PASS = new ProtocolString("pass");
    }
}

```

This had several benefits. First, as previously mentioned, it prevented misspellings and made it harder to incorrectly extend the protocol. Secondly, it forced matches to be done through the `SolarWind.ProtocolString` class, with the emphasis on case-insensitive comparisons.

I then recompiled the portions of code that utilized this class, changing the references to `String` into references to `SolarWind.ProtocolString`. Once the errors disappeared, I moved on to another portion of the code.

Several days later, I was comparing the compilation speeds of the `jikes` and `javac` compilers. I deleted all the class files and recompiled the main class. I was greatly surprised when it failed, reporting illegal attempts to cast a `SolarWind.ProtocolString` into a `String`.

It turned out that I had neglected to modify several references to constants in the `SolarWind` class, but the application had continued running correctly. My first reaction was that this was some sort of hole in the virtual machine's security code, which is supposed to block miscasts. I sent a somewhat cryptic email to the Java security mailing list (java-security@sun.com), and received a call back from Gary Ellison within 15 minutes. He asked me to isolate the suspected problem and email it to him, which I promised to do.

Over the weekend, I realized that my initial suspicions were probably incorrect. Due to the fact that the code still worked perfectly and the strings being compared were the same as before, it appeared to be a casting error (perhaps the actual data in a `String` was called `data`?).

The next possibility was that the constants in question were actually being compiled into the code. This seemed unlikely, as one of the key features of Java is its modularity. A quick disassembly of the class files confirmed it, however.

Having discovered that it was not a security hole, a fact which I relayed to Mr. Ellison when I returned to work the next week, I decided that the implications of this behavior nevertheless create an inelegant and dangerous inconsistency in the way Java behaves.

3 The Problem

3.1 A Simple Case

The problem, in its simplest case, is as follows: Assume two classes, `Foo` and `Bar`. `Foo` references a `public static final` variable in `Bar`. However, unlike all other variable or method references, the data held in the variable is compiled directly into the class file, as shown in the following files:

Foo.java (version 1)

```
class Foo {
    public static void main(String arg[]) {
        System.out.println(Bar.HAPPY);
        System.out.println(Bar.ONE);
    }
}
```

Bar.java (version 1)

```
class Bar {
    public static final String HAPPY = "Happy!";
    public static final int ONE = 1;
}
```

When run, this code produces the expected output:

```
$ java Foo
Happy!
1
```

However, if the definitions of `Bar.HAPPY` and `Bar.ONE` are changed as follows, without recompiling `Foo.java`, the output will remain exactly the same.

Bar.java (version 2)

```
class Bar {
    public static final String HAPPY = "Sad...";
    public static final int ONE = 2;
}
```

A disassembly of `Foo.class` reveals why:

Disassembly of Foo.class

```
Compiled from Foo.java
class Foo extends java.lang.Object {
    Foo();
    public static void main(java.lang.String[]);
}

Method Foo()
  0 aload_0
  1 invokespecial #7 <Method java.lang.Object()>
```

```
4 return
```

```
Method void main(java.lang.String[])
  0 getstatic #8 <Field java.io.PrintStream out>
  3 ldc #1 <String "Happy!">
  5 invokevirtual #10 <Method void println(java.lang.String)>
  8 getstatic #8 <Field java.io.PrintStream out>
 11 iconst_1
 12 invokevirtual #9 <Method void println(int)>
 15 return
```

3.2 Implications

3.2.1 Development

Developers may run into the same situation I did, in which they make changes to one class, and these changes are not properly reflected throughout the rest of the source tree.

3.2.2 Security

While this is not specifically a security problem, it does have certain implications in that arena. For example, if a developer makes changes that are not propagated to all classes, this may lead to a vulnerability that would not be apparent from examining the source. While the probability of security management code depending on such constants is slight, it should be noted.

3.2.3 Package Versioning

If classes in third-party packages include `static final` variables (the most obvious example being `Strings` or `ints` used as version identifiers) the code utilizing these packages will behave based on the version available at compilation, not at runtime.

4 Suggestions

4.1 Change the Compiler Behavior

The current behavior of the compiler is confusing and inconsistent with the way the rest of the Java language is implemented. One of Java's strengths is the fact that each value and method is loaded from the appropriate class at runtime. This behavior defies that standard.

Inlining constants provides a small speed advantage, and this is probably why it was implemented in the first place. However, with the advent of advanced runtime optimization environments such as HotSpot, this is less valuable than in the past.

The other advantage is to provide functionality similar to the C preprocessor's `#ifdef`. However, if this functionality is needed (which it almost certainly is), it should be provided by a similar mechanism, not by mutilating the object-oriented foundation that is Java's most compelling feature.

Unfortunately, existing code may already take advantage of this behavior. If the compiler's behavior were to be changed, it would need to be widely publicized. An option to emulate the old behavior would not be inadvisable.

4.2 Wrap Constants in Method Calls

Developers wanting to include constants in their code should, for the time being, wrap the values in method calls. While this is an inelegant and somewhat inefficient method of implementing such values, it is the only way to bypass this behavior with total certainty. Also, as mentioned before, runtime optimizations are often able to automatically inline such method calls.

4.3 Recompile Projects Before Release

It should be noted that I did not notice this problem until I tried recompiling my project from scratch. Before releasing a product, Java developers should always recompile the root class fresh, to ensure that all dependencies are based on the current source.

5 Acknowledgements

- The Java security group deserves a round of applause for their attentiveness and prompt response. Even though in this case it was a false alarm.
- The SoftDist group at Sun, where I currently work, is great. Ron, Rachelle, Roger, Dennis, Mike, Willy... You're a great team!
- I'd like to thank the students and staff at Standley Lake High School for their consistently thoughtful and reasonable approach to different issues. Yeah.
- I'd like to thank my parents, my siblings, my agent, and my potted plants. I couldn't have done it without you guys!