Java Programmer Perils

Roger T. Alexander  
Software Productivity Consortium  
alexande@software.org

James M. Bieman  
Colorado State University  
bieman@cs.colostate.edu

John Viega  
Reliable Software Technologies Corp.  
jviega@rstcorp.com

Personal use of this material is permitted. However, permission to reprint/republish this  
material for advertising or promotional purposes or for creating new collective works for  
resale or redistribution to servers or lists, or to reuse any copyrighted component of this work  
in other works must be obtained from the IEEE.

16 August 1999

Abstract

Java is the latest programming language technology to be touted as the programming lan-
guage solution to the most difficult software engineering problems. Developers expect Java to  
help them to write programs that are more reliable, secure, and easier to debug. Unfortunately,  
some features of Java are destined to increase programmer stress by providing obscure places  
for bugs to hide. We discuss seven such Java features and show that these features can lead to  
bugs that are difficult to fix.

Keywords: Java, programming practices, software engineering, software reliability, debugging,  
object-oriented programming, programming language design.

Every few years a new software development technology catches the attention of the software  
development community. Java is now the latest such technology; it is touted as a solution to  
problems of software reliability, portability, security, etc. The use of Java has grown as fast or  
faster than any prior software development technology [5].

Java has many attributes that will promote the development of more reliable and bug free  
software including memory management to prevent memory leaks, strong type checking to prevent  
the misuse of entities, and built-in support for exception handling. The virtual machine model  
increases portability, and the security model provides a degree of safety when importing externally  
developed code. These features are all improvements over C++, Java's nominal predecessor. Initial  
experimental results show greater programmer productivity and fewer program bugs for Java  
development versus C++ [10].

We are strong supporters of Java within our own organizations, which have adopted Java as a  
primary programming language. However, Java is far from an ideal programming language, and  
some features of the language must be used with care.

Address correspondence to: J. Bieman, Computer Science Dept., Colorado State Univ., bieman@cs.colostate.edu,  
Fort Collins, CO 80523, USA. (970) 491-7096, Fax: (970) 491-2466.
We examine Java from a software engineer’s perspective seeking to understand some of the problems inherent in the design of Java. In this paper, we survey seven major flaws in the Java design. These flaws, which are known within the Java community, allow obscure bugs to persist. Software engineers can limit the effect of these flaws by applying recommended coping strategies.

Our objective is not to critique Java. Others have critically examined Java from a language design perspective [1]. Rather, we want to make developers aware of Java design problems, so that these language problems do not contribute to implementation problems.

1 Protected Access is Unprotected

The term “protected” implies that encapsulation is provided. When a program component, such as a variable or method, is modified by the word “protected,” you might naturally assume that visibility is limited to a restricted set of components. In Java, protected access has a visibility hole so large that any protection provided is merely an illusion.

Like C++, protected in Java allows access to other members of the same enclosing class and to members of its descendants via inheritance. Such access does increase the coupling between class definitions, but at least a reference by an object to a field in a superclass is really a reference to part of the object’s own state. However, Java also grants the same access to members of any class in the same package as the protected member. Thus, any class with the same package designator can read and write to protected fields in any other class with the same package designator. This creates undesirable coupling: common coupling between all objects in the same package that reference a protected instance, and content coupling when objects reference a protected method that implements representation dependent behavior.

Java’s form of protected does not effectively support encapsulation. Encapsulation is meant to limit the dependence of external entities on the “secrets” of the implementation of a data abstraction. Once components that are external to the protected member can access the protected member, the protection is lost. Java’s access rules allow any change to a protected member to ripple across to an unlimited (and possibly expanding) number of classes with the same package designator. And any component with the same package designator can modify a protected field and force objects into invalid states.

Figure 1 shows how a data representation with protected access can be corrupted by a new class. In class Vehicle of Figure 1, the protected instance variable VIN represents the vehicle identification number of a Vehicle instance or object. An implied invariant for VIN is that it should be unique for each Vehicle object, and it should not change during the life time of the object. Yet, because class MungeVINS in the “Rogue Class File” and is declared to be a member of package autos, it can access the protected variable VIN. Modifications of VIN within MungeVINS are likely to violate any implied invariants in the Vehicle object (v1) The future behavior of this Vehicle object is now quite unpredictable.

Certainly, if used with care, a package can define a collection of closely related abstractions that honor each other’s semantics and rules for consistency. However, Java cannot enforce such practices. Developers must depend on local honored convention, and these conventions may fail to prevent inappropriate access.

A developer can add a new class into a package at any time simply by using the package designator in the class code. This can be done even by an arbitrary third-party who is unaware of any established convention or policy, or worse yet, by a person with malicious intentions. Objects in the new class gain complete access to all protected members of the named package. Members of the new class can inadvertently, or deliberately, violate any conventions or policies.
/* A class with a protected VIN Field */
package autos;

public class Vehicle{
    private double speed;
    private double direction;
    private String ownerName;
    protected int VIN;
    private static int highestVIN = 0;

    public Vehicle(){highestVIN++ ; VIN = highestVIN;}
    public Vehicle(String name) {this(); ownerName = name;}

    public void setSpeed(double s) {speed = s;}
    public double getSpeed() { return speed;}

    public void setDirection(double d) {direction = d;}
    public double getDirection() { return direction;}
}

package autos; /********* gains access to VIN fields by declaring itself in the targeted package */
import autos.Vehicle;

public class MungeVINS {

    static public void main(String[] args) {
        Vehicle v1 = new Vehicle("George");
        v1.setSpeed(49.5);
        v1.setDirection(45.0);
        v1.VIN = v1.VIN * 10;  //**** We multiply and change a VIN ****/
    }
}

Figure 1: Weak protection with protected access. class MungeVINS breaks the encapsulation of the protected VIN field in class Vehicle. This example is derived from one in [1].
To protect a member from undesired access, developers must avoid using protected access. Protected access does allow descendant classes to access protected members, which is often desirable. However, in Java, there is no way to provide access only to descendant classes. Protection against undesirable access might be possible through a lint-like preprocessor tool.

Until the nature of protected access in Java is changed, we suggest that programmers treat "protected" access as if it reads "unprotected". Better yet, avoid using protected data members.

A similar "protection" problem also occurs for class members that do not specify either public, protected, or private access.

**Default or Package-Level Visibility is Unprotected.** Java class members can be declared without an access modifier. By default, such members exhibit package-level access. Only other members of the class enclosing the member declaration and members of other classes in the same package can access the member directly. Package-level access is similar to protected access, except that package-level access does not provide access to the members in descendant classes, unless they are in the same package. All of the visibility problems of protected access apply to default access. However, with default access, members are made overly visible without any action by the programmer.

With the current default access policy, a programmer unfamiliar with the subtlety of the language rules can unwittingly grant wide access to the internal representation of an abstraction, losing the benefits of encapsulation. A programmer who forgets an access modifier also grants package-level visibility. Such errors will not be found by the compiler.

Programmers should always include access modifiers for every class member declaration, and should also avoid the use of protected access. Unfortunately, such a policy leaves only a choice of either public or private. Further, this eliminates the possibility of using protected access to provide safe interfaces for descendant classes, and it forces complete exposure of the underlying state representation. Java just does not provide safe access flexibility.

**Reflection is Too Revealing.** Version 1.1 of the Java Development Kit gives programmers indirect access, through a Java Library API, to information about classes, fields, methods, and constructors. This facility can reveal dynamic relationships between classes at run time, and provides complete access to the interface and state representation of a class. A programmer can use this support for reflection to subvert encapsulation. The security manager can limit who has this level of access, but a sufficiently strong security policy must be explicitly defined.

2 Constructor Confusion

One of Java's advertised strengths is that it initializes all variables before they are used. Thus, in principle, a class's methods will be invoked only after all class instance variables have been initialized. However, the semantics of initialization and construction in Java are not simple. There are situations when instance variables can be used before the construction of the objects that own them.

The confusion results, in part, from the distinction between initialization and construction and the order of initializations and construction of a class and its superclass(es). Variables are initialized to default values first, then superclass constructors are executed, followed by non-default initialization. A constructor can call methods, and these methods can be overridden in a descendant class. When such an overriding method is called by the superclass constructor while a descendant class
class Super {
    Super() { printThree(); }
    void printThree() { System.out.println( "three" ); }
}

class Test extends Super {
    int indiana = (int)Math.PI;  // That is, pi=3 in Indiana.
    public static void main( String[] args ) {
        Test t = new Test();
        t.printThree();
        void printThree() { System.out.println( indiana ); }
    }
}

Produces the following output:
0
3

Figure 2: A constructor, Super(), causes an uninitialized variable, indiana, to be used, when a subclass, Test, is initialized (from [2] p. 231).

object is being constructed, the overriding method will execute before initialization and construction of the derived class instance is complete. Any local variables used by the overriding method have not been set by the construction process. Strange behavior can result.

The example code and its execution in Figure 2 demonstrates the complex order of initialization and construction, and the potential confusion. The first statement in the main method of class Test in Figure 2 creates a new class Test object. The instance variable indiana is initialized to the default value 0; the non-default assignment initialization is deferred. The constructor of the superclass, Super(), is then invoked, which in turn invokes the PrintThree() method, but not the PrintThree() method within Super. Rather, the PrintThree() method in class Test is invoked, even though the initialization of variable indiana has not been completed. Thus, PrintThree() prints a 0, the current value of indiana. Control returns from Super’s constructor and initialization of variable indiana is completed — the integer 3 (the floating point value of π is cast into an integer). If there was a constructor for class Test it would run now. Construction of the Test object t is now complete. The next statement invokes printThree(), which prints out the current value of indiana which is 3.

Methods that execute prior to initialization or construction are dangerous at best. Their behavior is likely to invalidate assumptions made by the authors of both the parent and descendant classes. When a base class constructor calls a method, unless the constructor invokes only final methods, the method defined in the base class may not be the one to actually execute. When this happens, do not expect any assumptions concerning the called method to hold.

A restriction that requires all method calls in constructors invoke only local methods designated as final does not solve the problem, unless these methods, in turn, only call other final methods, and so on. In other words, all local method calls made from a constructor must only result in the execution of methods that are defined to be final.

Ensuring correctness is a difficult challenge for designers of descendant classes. A designer must have a detailed understanding about the semantics of the implementation of all ancestor classes. The descendant class designer must particularly have a precise understanding of the effects of an overriding method on the parent class’s state-space to avoid consistency problems.
This “constructor confusion” is likely to be the source of many faults, particularly for those programmer with a C++ background, since the behavior of C++ constructors with respect to local method calls is nearly the opposite of that for Java. For example, when constructing an instance of a derived class in C++, a call from a base class constructor to a polymorphic (i.e. virtual) method defined in the base class always results in the execution of the base class method, even when the derived class has an overriding definition of the called method. This C++ construction behavior is in stark contrast to that in Java.

3 Finalization Follies

Because of Java’s mandated garbage collection, programmers can ignore the details of memory management. Unfortunately, they must still manage the ownership of other resources. Thus, Java programmers must still deal with many of the complex issues that C++ programmers address using destructors. Although memory leaks will not occur, the scheduling of the execution of Java class finalizers [12], Java’s form of destructor, can cause other resource leaks.

Java finalizer methods run when an object is being deallocated. However, unless explicitly invoked, a finalizer runs only during garbage collection, rather than when the object loses its last reference. Thus, finalizers run at unpredictable times, just like garbage collection. The uncertainty about the time that the finalizer runs can lead to trouble. Consider a class whose constructor allocates a network connection, and whose finalizer closes down the connection. On many systems, each network connection is mapped to a file pointer in the operating system. Generally only a relatively small number of file pointers can be open at once. If a program instantiates and then discards a large number of these objects before the garbage collector calls any finalizers, then any attempt to create a new file or network connection would fail.

Programmers should not count on finalizers executing in a timely manner. In fact, there is no guarantee that finalizers will ever run at all. For example, at program exit time no finalizers will be run for any objects that have become garbage since the last collection, unless the programmer explicitly ensured that System.runFinalizersOnExit(true) was called by the program. Even so, that is no guarantee that the finalizer will run at all. For example, the current version of Sun’s JVM will not run finalizers if the virtual machine is terminated by a signal from the outside.

Also, programmers should not depend upon finalizers executing in a deterministic order. For example, programmers should not assume that finalizers will run in the order that the objects became garbage; the actual order is not specified [12].

We recommend that finalization be avoided if at all possible. If you must use finalization, and your finalizers need to be called in a timely manner, then you should explicitly call the garbage collector which will invoke the finalizers. However, for this to work, you must know beforehand that a given object will be available for finalization, which means that you must keep track of all references to that object. An explicit call to the garbage collector will not invoke an object’s finalizer if there are any remaining references to the object.

An alternative approach is to add public methods that can be called to release resources held by an object when they are no longer needed (perhaps just prior to orphaning the object). However, for this to work, programmers must keep track all references to the object holding the resources, and they must assign the responsibility for calling the methods to release these resources. An error in this complex task can cause a resource to be deleted even though it is still in use by other clients.
4 Inheritance Without Specialization

As we know, the term subclass refers to classes that are descendants of other defined classes. Java and other object-oriented languages allow subclass objects to be used in place of a superclass object. Certain properties must be satisfied to guarantee that substituting a subclass object for a superclass object can be done safely and that the subclass object inherits only safe operations [4].

In general, use a subclass when the derived class is a specialization of the superclass, called an “is-a” relationship. That is, subclass objects behave similarly to superclass objects, but have additional features and/or operations. Then you can safely substitute a subclass object for superclass objects. For example, a Cartesian point with color attributes can be a specialization of a Cartesian point without color. A colored point can be substituted for a plain point, because any behavior of plain points also applies to colored points.

Problems can occur when a subclass is not a true specialization of its superclass. Consider the java.util.Stack class, which is part of the java.util package. Class java.util.Stack is a subclass of java.util.Vector. The methods defined by Stack include common stack methods such as push(), pop() and peek(). However, because Stack is a subclass of Vector, it inherits all of the methods defined by Vector. A Stack object can be supplied wherever a Vector object is specified. Thus, a program can insert or delete elements at specified locations in a Stack object using the insertElementAt or removeElementAt methods of Vector. It can even remove a specified element from a Stack object without regard to the element’s position in the stack using Vector’s removeElement method. Thus, the java.util.Stack can exhibit behavior that is not consistent with the notion of a stack as a last in first out entity. In addition, a program can access all of the Vector operations on Stack objects directly, when the Stack objects are not being substituted for Vector operations. The integrity of a java.util.Stack object can be easily compromised.

A stack is not a specialized vector, and it should not inherit vector operations. Instead, use a vector as a hidden, private representation of a stack. Then inappropriate vector operations cannot be exported by stack objects. This preferred design uses aggregation, which makes it possible to use inheritance and polymorphism to replace the vector representation with alternative implementations. Thus, greater flexibility and economy of design are possible when inheritance is used properly.

In general, a subclass object should still be an instance of its superclass. For a taxonomy that classifies both proper and improper uses of inheritance, see a recent article by Meyer [8].

Improper use of subclassing in Java can be especially troublesome and a source of difficult bugs to diagnose and correct. Java does not provide the mechanisms that C++ does to make “improper” subclasses a bit safer. In particular, it does not provide a mechanism to hide inherited members or to break the type relationship with its parent. Thus, there is no way to prevent a client from seeing a descendant as an instance of its base class. One “solution” is to provide overriding methods for each inherited method, and implement them by throwing invalid method exceptions. Unfortunately, this solution might be impossible: you cannot override any methods that are declared as final in a parent class; and the exception must be declared in the a parent class, unless you throw an unchecked exception, such as those derived from RuntimeException. Another possible solution would be to use some sort of assertion mechanism to restrict the use of inappropriate inherited methods (see Section 7).
5 No Support for Homogeneous Containers

Java provides little flexibility in creating specialized, homogeneous container classes. You must either use containers that can hold anything, or write special purpose classes defining containers for each kind of element. This is because Java currently does not support type-safe parameterized classes, such as templates in C++ or generics in Ada. Instead, Java provides the universal base class Object, a superclass to every class.

Consider creating an object from the Java Collections Framework class LinkedList, which is meant to be a homogeneous list of String elements. C++ programmers can simply instantiate an object of type LinkedList<String> and the compiler will ensure that only String objects are inserted into the LinkedList<String> object. Java programmers have three options: they may (1) instantiate a LinkedList object that accepts any object whatsoever (objects of class Object), and only place objects of class String into it as shown in Figure 3, (2) write a special purpose adapter class with the functionality of a LinkedList class that operates only on String objects as shown in Figure 4, or (3) write a LinkedList class that, at run time, sets the type of inserted objects based on the class of the first object that is inserted, as shown in Figure 5.

Generally, programmers use the first approach. However, static type safety is impossible. Programmers must ensure that only objects of the desired class are inserted into a list, and they must explicitly cast objects retrieved from the list back into the desired class. Any type checking will be performed at run time by these dynamic casts. As a result, run time type errors may occur — some method may insert non-String objects into the list. To prevent such run time type errors, programmers must track type information to ensure correctness.

In contrast, the second approach ensures that calls to container operations can be type checked properly at compile time. In addition, the special purpose class can perform all casting. Of the three approaches, this offers the most type safety, but at the expense of a proliferation of nearly identical classes. However, code replication can be minimized by having the adapter provide the necessary interface, but implement it in terms of LinkedList. This approach has the advantage of imposing the required level of type safety while reusing the existing available implementation.

The third approach can be implemented using only a single class, a class that captures the type of the first object to be inserted into the list. However, type checking that insures the insertion of specified objects will only occur at run time. As in the first approach, programmers must explicitly cast elements retrieved from the list back into the desired class.

Unfortunately, there is no satisfactory work-around to this problem. Use any of the three suggested coping strategies. Weigh the risks of each approach, and then proceed with caution.

6 Final Parameters are Not Final

Java does not allow a programmer to declare a method to be state-preserving. That is, that the method does not change the state of the object that it is a member of. Thus, to be safe, a client must assume that a method invocation, such as o.m(), may modify the state of the object, o, that invoked the method. To really know if the state of o is changed, the client programmer must examine the implementation of the called method o.m(). Often, programmers do not have access to the implementation, so they must simply trust the documentation. Of course, this documentation imposes no guarantees or constraints on the called method’s implementation.

The only solution is programmer discipline to establish convention and policy to document all side-effects, and ensure that method implementations are kept consistent with the documentation. But this solution cannot be applied to Java components obtained externally.
import java.util.List;
import java.util.LinkedList;

/**
 * From the Java Collections Framework:
 * interface List {
 *    public void add(Object element);
 *    public Object get(int index);
 *    ...
 * }
 */

class StringListExample{
    public static void main(String[] args){
        List l = new LinkedList();
        for(int i=0;i<args.length;i++){
            l.add(args[i]);
            System.out.println((String)l.get(i));
        }
    }
}

Figure 3: Using a universal list, a list of Object elements, to hold Strings.

class StringList{
    private List my_list;
    public StringList() {my_list = new LinkedList();}
    public void add(String elem){my_list.add(elem);}
    public String get(int index) {return (String)my_list.get(index);}
    ...
}

Figure 4: A special purpose StringList that can only hold String objects

class RunTimeList{
    private List my_list;
    private Class thisClass;
    RunTimeList() {my_list = new LinkedList();}
    void add(Object elem) throws BadElement{
        if (my_list.isEmpty()) thisClass = elem.getClass();
        if (thisClass != elem.getClass())
            throw new BadElement(thisClass, elem.getClass());
        my_list.add(elem);
    } 
    Object get(int index){
        return my_list.get(index); // Java won't let us cast back
        // to "thisClass" here.
    }
}

Figure 5: A list that sets the class of its contents when the first element is inserted at run time.
Java 1.1 allows programmers to declare the the formal arguments of a method to be final, ensuring that the state of the argument cannot be modified. Unfortunately, the guarantee only applies to the state of the parameter variable itself, not the state of any class that it references. A formal argument cannot appear on the left-hand-side of an assignment expression, but Java allows calls to any methods through a final parameter variable, including methods that change the state of the object that the final variable references. Thus, when a program supplies an object reference as an argument to a method call, the state of the object argument may change, even if the associated formal parameter is designated as final. The programmer must trust the called method, inspect the method if possible, or add code to verify that no state changes have occurred and add error handling code.

The protection provided by the final parameter designator has severe limitations, limitations that could cause a system to enter an inconsistent state. Testing alone cannot guarantee that all methods behave as expected.

Explicit documentation is the only protection. Methods should contain correct documentation that explicitly describes the effects of a particular method call, both the effects on a given instance and the effects on any instances passed via object references as actual arguments.

7 Initialization Diffusion

The JDK 1.1 Java Language Specification includes instance initialization blocks, blocks of code that initialize the state of object instances [2]. These are similar to the static initialization blocks that initialize class state. An instance initialization block is written simply as an unlabeled block of code appearing at any location in a class definition. There may be multiple instance initialization blocks, which may be distributed at various locations within a class. These initialization blocks are executed in the relative order of their appearance in a class.

Figure 6 demonstrates how initialization can be diffused across a class through the use of initialization blocks. Look at the initialization of the static variable highestVIN. When a new Vehicle (or VehicleDiffusion) object is created, we want to increment highestVIN so that we use a unique VIN for each new Vehicle object. Class Vehicle updates highestVIN in the "expected" place, within the constructor method Vehicle(). Class VehicleDiffusion updates highestVIN in an initialization block rather than within a constructor. To further confuse code readers, this update is placed in a location that is separated from the constructors. The semantics of creating a new Vehicle or VehicleDiffusion object are essentially identical.

Instance initialization blocks, introduced with Java 1.1 syntax and semantics [1], add two new sources for program errors. First, initialization code is distributed between constructors and initialization blocks, which may be distributed throughout a class. To fully understand the full instance initialization and construction process, a programmer must understand the semantics of constructors and instance initialization blocks. He or she must must scan an entire class definition looking for instance initializers, analyze the semantics of each initializer and their order of execution, and then analyze the semantics of the class construction methods. This process can, potentially, be very difficult and error prone, depending on the number and distribution of instance initializer blocks.

Second, the syntax of instance initializer blocks can lead to errors. The only syntactic difference between an instance initializer block and a static initializer is a single keyword: static. If static appears immediately before a class-level block, then the block defines a static initializer. If the static keyword is missing, and no other lexical element appears in its place (e.g. a method signature), then the block defines an instance initializer. In addition, an instance initializer has a structure that is identical to method definition that is missing its class body declarator. Un-
/ "Normal" initialization in class Vehicle */
class Vehicle{
    private double speed;
    private double direction;
    private String ownerName;
    private int VIN;
    private static int highestVIN = 0;

    public Vehicle() {highestVIN++;       // increment highestVIN
                     highestVIN;}
    public Vehicle(String name) {this(); ownerName = name;}

    public void setSpeed(double s) {speed = s;}
    public double getSpeed() { return speed;}

    public void setDirection(double d) {direction = d;}
    public double getDirection() { return direction;}
}

---------------------------

/* Diffused initialization in class VehicleDiffusion */
class VehicleDiffusion{
    private double speed;
    private double direction;
    private String ownerName;
    private int VIN;
    private static int highestVIN = 0;

    public VehicleDiffusion() {VIN = highestVIN;}
    public VehicleDiffusion(String name) {this(); ownerName = name;}

    public void setSpeed(double s) {speed = s;}
    public double getSpeed() { return speed;}

    {highestVIN++;}                // increment command moved here

    public void setDirection(double d) {direction = d;}
    public double getDirection() { return direction;}
}

Figure 6: Initialization Diffusion caused by instance initialization blocks: Vehicle and VehicleDiffusion objects exhibit identical behavior although the command highestVIN++ in the Vehicle constructor is moved to an initialization block in VehicleDiffusion. This example is derived from one in [1].
fortunately, it very easy to accidentally delete a single word, for example “static” or a line that defines a method interface. These simple editing errors turn a static initializer or a method into an instance initializer, which might compile without any warnings. Debugging these errors is difficult.

We urge programmers to avoid the use of instance initializer blocks, and put initialization code in constructor bodies. If for some reason you must use an initializer block, use only one per class.

Other Worries

Syntax Troubles. Java has a number of syntax quirks and inherits many of the syntax problems of C++ and C. These problems are described in detail by Thimbleby [11].

No Separate Class Specification Components. Java does not allow a separation between a class specification and its implementation, since both the public class interface and method bodies must be included in one file. Thus, anyone accessing the public interface can view its implementation. Knowledge of the implementation makes it possible to write software components that depend on this implementation, a violation of encapsulation principles.

Our primary concern is ease of understanding, which is of key importance during software maintenance and reuse. Programmers writing a client must wade through an entire class body just to view the public interface of a potential server class. A correction this problem would not require having a separate interface file and implementation file for each class, as the ‘.h’ and ‘.cc’ files are used in C++. Rather, Java could have a separate syntactic mechanism within a class for specifying its interface. The corresponding implementation might still appear inline within the class body.

The Javadoc facility can extract interface information. However, Javadoc depends upon the author of a class writing appropriate Javadoc comments, and keeping them up to date. Over time comments and code usually diverge, and thus the Javadoc interface information will not be accurate.

No Support For Assertions. Java does not have a built-in assertion mechanism. Properly used, an assertion mechanism can increase the quality and correctness of a software component. Programmers can specify pre- and post-conditions for methods, and data invariants for class state variables. Other object-oriented languages, such as Eiffel and Chu, provide assertion support [6, 7]. Tools are available to support the use of assertions in Java, for example AssertMateTM from Reliable Software Technologies [9] and iContract from Reto Kramer of Reliable Systems [3]. Developers can construct a rudimentary assertion mechanism, as described in the Java FAQ [12]. However, we would prefer an assertion mechanism supported directly by the language. Meanwhile, our advice to the Java practitioner is to either acquire and use an assertion package, or to take the roll-your-own approach described in the Java FAQ.

Array Type Checking Failures. Limitations of static type checking can allow programs with obscure array type errors to compile. The generally type safe and flexible use of dynamic binding of subclass objects in combination with arrays can lead to a type clash or covariance problem. Generally, an instance of a subclass to a formal parameter can be supplied as an argument in a method call. However, sometimes dynamic binding conventions and associated type checking rules cannot detect type errors. Figure 7, shows example Java code where all actual parameters are instances of subclasses of the formal parameters. The program compiles even though it contains a serious type error. ArrayConfusion.main calls ArrayConfusion.proc with two arguments, anArrayOfB, and a, an instance of class A. ArrayConfusion.proc has two formal parameters, x, an array of
class A { . . . }

class B extends A { . . . }

class ArrayConfusion {
    static void proc(A[] x, A y) {
        x[0] = y;
    }
    public static void main(String args[]) {
        B[] anArrayOfB = new B[5];
        A a = new A(5);
        proc(anArrayOfB, a);
    }
}

Figure 7: Java static type checking does not catch the type error when the first argument to ArrayConfusion.proc is an array of class B and the second argument is an object of class A. The assignment x[0] = y raises a run time exception.

A is a class, and x, an A object. No type errors are detected during compilation because argument anArrayOfB is an instance of a subclass of formal parameter x’s class, A[], and the type of argument a exactly matches the type of formal parameter y. Yet, the program fails at run time. The assignment x[0] = y raises a java.lang.ArrayStoreException because an object of class A cannot be stored in an array of class B objects. A is not a subclass of B. This program, which has an illegal assignment of a supertype to a subtype variable and fails at run time, compiled with no detected errors.

Conclusions

A goal of any industrial programming language should be to help programmers ensure the quality of software by preventing them from making dangerous mistakes. Java is successful to some extent, but it falls short in at least seven ways. These shortcomings are particularly worrisome, especially since Java is intended to be used to develop concurrent, distributed and critical systems.

Programmers can remedy or soften the impact of these problems by following a few suggestions:

1. Avoid using protected or package-level (default) access; declare all members as either private or public.

2. Take extra care in understanding the construction of new objects that override superclass methods and instance variables.

3. Use container classes with caution. Java type checking is not effective here.

4. Use the subclassing mechanism only to define specializations of a superclass.

5. Explicitly force finalizers to run when you want them to.

6. Use inheritance only to model “is-a” relationships.

7. Document all side effects, and make sure that the documentation is consistent with the code.
8. Avoid using instance initializer blocks.

A few changes to the language itself would more effectively solve these problems:

1. Replace the package-level component of `protected` access with a mechanism that allows a class to specify what other specific classes, or group of classes with a particular characteristic, may access it.

2. Make default access `private`.

3. Change the semantics so that constructors cannot invoke any methods, directly or indirectly, in subclasses.

4. Support templates or generics.

5. Support delegation explicitly — provide a mechanism that allows a class to delegate a set of method calls to a specified delegate object.

6. Make the `final` parameter designator ensure that a method with a `final` parameter cannot modify the state of objects referenced by the parameter. Add the notion of a `constant` method designator that insures that the state of the object containing the method does not change.

7. Require a keyword to specify an instance initializer, and allow only one such initializer block per class.

In reality, future versions of the Java Language will not include most of our suggested changes. A language, used by many, must evolve in a manner that satisfies many competing interests. New features must be evaluated for linguistic simplicity and expressiveness, compiler and application performance, portability, etc. Existing programs must work under new language releases, and existing components must work with components developed under a new version. However, our suggested changes would make Java programs more dependable; at least the problems that we recognize will no longer be waiting to haunt us. Some of these changes could be made with little effort. Others, in particular, suggestions 4 and 6, will require more major efforts and probably some research.

Testing and static program analysis can also potentially identify programs with the described problems. Static analysis can easily identify programs that use instance initializer blocks and/or use `protected` and package (default) access. Research can be directed towards developing tools and techniques, either through static analysis or through testing, to identify programs with the remaining problems.

**Acknowledgements**

We thank Reliable Software Technologies Corp. in Sterling, Virginia, and the University of Maryland at College Park for their generous support during Jim Bieman’s sabbatical while this paper was prepared. We also thank Jeff Offutt and Gary McGraw whose comments on earlier drafts of this paper greatly improved the presentation. Finally, we thank the anonymous reviewers for their comments, which greatly improved both the content and presentation.
References


