Java is not type-safe

Vijay Saraswat
AT&T Research,180 Park Avenue, Florham Park NJ 07932

Table of Contents

Last modified date: `Fri Aug 15 19:05:33 1997`

- Added comment on method-table generation.
- Added commentary on consequences of fix for method-table generation.

Abstract

A language is type-safe if the only operations that can be performed on data in the language are those sanctioned by the type of the data.

Java is not type-safe, though it was intended to be.

A Java object may read and modify fields (and invoke methods) private to another object. It may read and modify internal Java Virtual Machine (JVM) data-structures. It may invoke operations not even defined for that object, causing completely unpredictable results, including JVM crashes (core dumps). Thus Java security, which depends strongly on type-safety, is completely compromised.

Java is not type-safe because it allows a very powerful way of organizing the type-space at run-time (through user-extensible *class loaders*). This power can be utilized to write a program that exposes some flawed design decisions in the Java Virtual Machine. Specifically, one can produce a class A and an associated ersatz class A' which can "spoof" A: its name N is the same as A, but it defines members (fields and methods) arbitrarily differently from A. A "bridge " class B can be defined which delivers to a class D (for which the name N is associated with A') an instance of A. D can then operate on this instance as if it is an instance of A', thus violating type-safety.

There are two ways in which the violation of type-safety may be addressed. I identify a necessary and

sufficient conditions on class loaders such that if all the classloaders definable in a Java program satisfy this condition, then the program will not have any "bridge" classes at run-time, and hence will not exhibit this kind of type-spoofing. Thus one may still informally argue that a *particular* Java program may not exhibit this kind of type-spoofing, and one may design Java programs in the future to satisfy this condition. A reading of the informal description of class loaders given in HotJava indicate that it may satisfy this condition.

For the *language* to be type-safe however --- a far more desirable alternative --- either the classloader interface must be redesigned or the JVM must be fixed. I argue that arbitrary user-definable classloaders represent a significant conceptual advance in Java, and should not be limited in any way. On the other hand, I show that the JVM design can be fixed (without any run-time penalites) by fixing the (link-time) constant pool resolution process to take into account information available at link-time and not just compile-time. Interestingly, this also points out that Java is actually a rather impoverished language for programming the Java Virtual Machine -- programs cannot be written in Java which exploit (in a type-safe way) some of the capabilities of the JVM to manipulate classes loaded in different class-loaders.

Further study is needed to determine if there are any other ways in which type-safety can be compromised in Java.

This is a revised version of an earlier note, of the same name, that was informally circulated Monday Jul 21, 1997. That version had (mistakenly, it now turns out) argued that some run-time type-checks were unavoidable for unrestricted class-loader functionality. Thanks to Gilad Bracha, Drew Dean, Kathleen Fisher, Nevin Heintze, Tim Lindholm, Martin Odersky and Fernando Pereira for useful feedback and discussion. I remain responsible for the actual contents of this note.

Section 1. The problem

Let *A* and *A'* be two different classfiles defining a Java class with the same fully qualified name (FQN) *N*. In a running Java Virtual Machine (JVM) J, let *A* be loaded by a class loader *L* (possibly the "null" loader) producing a *Class* object *C* and *A'* by a class loader *L'* producing a *Class* object C'. Let *v'* be a variable of "type" (we will have more to say later about what is a "type" in Java) *N* in a class *D* loaded in *L'*.

Proposition: Any instance *t* of *C* can be stored in *v'*.

Proposition: J will (attempt to) execute any operation defined in *A'* on *t.* J will (attempt to) read/write any field defined in *A'* as if it existed in *t.*

This behavior is unexpected. It contradicts the assertion [Lindholm, P 10]:

> Compatibility of the value of a variable with its type is guaranteed by the design of the Java language ...

This behavior can be exploited to place the Java Virtual Machine in an "undefined" state in which its behavior is unpredictable, potentially compromising the Virtual Machine as well as the computer on which it is running.

As I show below (Section 2), this behavior is a consequence of the design of the constant pool resolution process in the Java Virtual Machine. Empirically, I have verified that this behavior is exhibited by Sun's JDK 1.1.3 system, on both Solaris and Windows.

But first let us examine some concrete examples and see what can go wrong.

A concrete example.

Let *R* be a base class that is desired to be spoofed. For simplicity, let it contain just a *private* field:

```
public class R {
 private int r = 1;
}
```

Assume that *R* has been loaded into J through some class loader, L in J. (For simplicity, take L to be the "system loader". So then it must be the case that this file for *R* is stored in a directory on CLASSPATH.)

> **Footnote:** After this note was written, I learnt (Lindholm, private communication) that, for some essentially obscure reasons, the null classloader behaves slightly differently from other classloaders in ways not publically documented. However, I have since verified that the problems discussed in this note arise when L is taken to be some non-null loader.

Assume that it is possible to obtain instances of *R* through another class, *RR*, also loaded into L (thus *RR* also exists in a directory on CLASSPATH). *RR* is the crucial "bridge" class --- accessible from within two different classloaders, it will allow "crossover". For simplicity, *RR* may be thought of as being defined as:

```
public class RR {
  public R getR() {
    return new R();
  }
}
```

Arrange now to load an ersatz class *R* in another classloader L' in J. It is important that this class have the same fully qualified name (FQN) as the target class *R*. However, the signature of this class (its fields and methods, and their associated types) may be completely arbitrary, and designed to suit the requirements for spoofing. For simplicity, assume that it is just desired to be able to read/modify the value of the private variable. Then *R* can be defined simply as:

```
public class R {
  public int r;
  }
}
```

Arrange now for your code (say in a class *RT*, loaded into L') to receive in a variable, say *r* (of type *R*) an instance of the *R* class loaded in L.

This can be accomplished, for instance, by arranging for L' to "share" the use of the *Class* object for *RR* loaded into L, as follows. The code for *loadClass* in L' forwards a request to load *RR* to L:

```
/** A classloader that delegates some loads to the system loader,
```

```
 * and serves other requests by reading in from a given directory.
 */
public class DelegatingLoader extends LocalClassLoader {
  public DelegatingLoader (String dir) {
    super(dir);
  }

  public synchronized Class loadClass(String name, boolean resolve)
  throws ClassNotFoundException {
    Class c;
    try {
      if (name.equals("RR") || name.startsWith("java.")) {
        System.out.println("[Loaded " + name + " from system]");
        return this.findSystemClass(name);
      } else
        return this.loadClassFromFile(name, resolve);
    } catch (Exception d) {
      System.out.println("Exception " + d.toString() + " while loading " + name + "
      throw new ClassNotFoundException();
    };
  }
}
```

Here, *LocalClassLoader* is an abstract Class loader that knows how to load (through the method *loadClassFromFile*) a class file from a local directory. This local directory should not be on the system path (CLASSFILES). Thus an instance of *DelegatingLoader* will load all classes other than those named *RR* or in *java.** packages from the local directory.

Now, loading *RT* into L' will eventually trigger the loading of *RR* by L'. This request is met by returning the *Class* object created by the system loader when it loaded *RR*. Loading *RT* will also eventually trigger the loading of *R* in L' --- however, this will cause the ersatz *R* file to be loaded into L'.

Thus the stage is set for type confusion. *RT* is set to receive an object from *RR* which it believes to be an instance of the class described by ersatz *R*. *RR* is prepared to send an object to *RT* which is an instance of the class described by *R*.

Here is a simple definition of *RT*:

```
/** The user class, referencing and using the ersatz class R.
 */
public class RT {
  public  static void main() {
    try {
      System.out.println("Hello...");
      RR rr = new RR();
      R r  = rr.getR();
      System.out.println("  r.r is " + r.r + ".");
      r.r = 300960;
      System.out.println("  r.r is set to " + r.r + ".");
      System.out.println("...bye.");
    } catch (Exception e) {
      System.out.println("Exception " + e.toString() + " in RT.main.");
    }
  }

}
```

Now all that remains is to ensure that *RT* is loaded into L'. This can be accomplished through the helper class *Test* .

We may now get the trace:

```
chit.saraswat.org% java Test RT
[Loaded java.lang.Object from system]
[Loaded java.lang.Exception from system]
[Loaded RT from ersatz/RT.class (996 bytes)]
[Loaded java.lang.System from system]
[Loaded java.io.PrintStream from system]
Hello...
[Loaded RR from system]
[Loaded java.lang.StringBuffer from system]
[Loaded R from ersatz/R.class (238 bytes)]
  r.r is 1.
  r.r is set to 300960.
...bye.
chit.saraswat.org%
```

Consequences.

Intuitively, the JVM is using the information associated with ersatz *R* to operate on an instance of *R*. The ersatz *R* class specifies the field *r* to be public, so the JVM allows access and update.

But the structure of the ersatz *R* need not be related to *R* at all. Suppose for instance, ersatz *R* is defined as:

```
public class R {
  public int r0;
  public String s = "This represents s.";
  public int r;
  }
}
```

Now the JVM believes that the field *r* lies at a specific offset in the memory representing an instance of ersatz *R* --- and this offset may well be different from that representing the actual field *r* in *R*. Indeed, given that the size of an instance of *R* is smaller than the size of an instance of ersatz *R*, references through fields of ersatz *R* are going to access memory outside the region set aside to represent the instance of *R*. We get:

```
chit.saraswat.org% java Test RT
[Loaded java.lang.Object from system]
[Loaded java.lang.Exception from system]
[Loaded RT from ersatz/RT.class (996 bytes)]
[Loaded java.lang.System from system]
[Loaded java.io.PrintStream from system]
Hello...
[Loaded RR from system]
[Loaded java.lang.StringBuffer from system]
[Loaded R from ersatz/R.class (544 bytes)]
  r.r is 6946913.
  r.r is set to 300960.
...bye.
chit.saraswat.org%
```

Similarly, ersatz *R* may define methods that do not exist in *R*, or are in a different position in the method list, or take a different number of arguments, or take arguments of different types ... causing complete havoc. For instance, suppose ersatz *R* is defined as:

```
public class R {
  public int r0;
  public String s = "This represents s.";
  public int r;
  public void speakUp() {
    System.out.println("I have spoken!");
  }
}
```

and *RT2* is defined as:

```
/** Call a method defined on the ersatz class, but not the spoofed class.
 */
public class RT2 {
  public  static void main() {
    try {
      System.out.println("Hello...");
      RR rr = new RR();
      R r  = rr.getR();
      System.out.println("Now checking to see if a method defined on this loader's r
      r.speakUp();
      System.out.println("...bye.");
    } catch (Exception e) {
      System.out.println("Exception " + e.toString() + " in RT2.main.");
    }
  }
}
```

We get the very interesting looking:

```
chit.saraswat.org% java Test RT2
[Loaded java.lang.Object from system]
[Loaded java.lang.Exception from system]
[Loaded RT2 from ersatz/RT2.class (934 bytes)]
[Loaded java.lang.System from system]
[Loaded java.io.PrintStream from system]
Hello...
[Loaded RR from system]
Now checking to see if a method defined on this loader's r can be invoked.
[Loaded R from ersatz/R.class (544 bytes)]
SIGBUS    10*  bus error
    si_signo [10]: SIGBUS    10*  bus error
    si_errno [0]: Error 0
    si_pre [1]: BUS_ADRERR [addr: 0x443a7]

        stackbase=EFFFF180, stackpointer=EFFFEEC0

Full thread dump:
    "Finalizer thread" (TID:0xee300220, sys_thread_t:0xef320de0, state:R) prio=1
    "Async Garbage Collector" (TID:0xee3001d8, sys_thread_t:0xef350de0, state:R) pri
    "Idle thread" (TID:0xee300190, sys_thread_t:0xef380de0, state:R) prio=0
    "Clock" (TID:0xee3000d0, sys_thread_t:0xef3b0de0, state:CW) prio=12
    "main" (TID:0xee3000a8, sys_thread_t:0x40e08, state:R) prio=5 *current thread*
        RT2.main(RT2.java:9)
        Test.doIt(Test.java:17)
```

```
        Test.main(Test.java:24)
Monitor Cache Dump:
Registered Monitor Dump:
    Verifier lock: " "
    Thread queue lock: ""
    Name and type hash table lock:
    String intern lock:
    JNI pinning lock:
    JNI global reference lock:
    BinClass lock:
    Class loading lock:
    Java stack lock:
    Pre rewrite lock:
    Heap lock:
    Has finalization queue lock:
    Finalize me queue lock:
    Monitor IO lock:
    Child death monitor:
    Event monitor:
    I/O monitor:
    Alarm monitor:
        Waiting to be notified:
            "Clock"
    Sbrk lock:
    Monitor cache expansion lock:
    Monitor registry: owner "main" (0x40e08, 1 entry)
Thread Alarm Q:
Abort (core dumped)
chit.saraswat.org%
```

A more insidious example

Here is a more "natural" example of how such a problem may be triggered. Suppose that a loader `L` "exports" the service offered by a class `RR` to other loaders, including `L'`. `RR` provides a public method that needs an instance of `R`. But it so happens that `L'`, unaware of the dependency of `RR` on `R`, also loads `R` (the `ersatz R`). Now any other class `RT` in `L'` that wants to use `RR` will end up sending an instance of its `R`, thereby triggering the incompatibility.

Can an applet exploit type-spoofing?

To answer this question, let us develop some terminology. Let `J` be some running JVM, initialized with some program `P`, and accepting inputs and delivering outputs to its environment. In the following, we consider class objects in `J` (i.e., instances of `java.lang.Class`) to represent types in `J`. For any such object `o`, `cl(o)` stands for the loader object that created `o` (i.e. who's invocation of `defineClass` created `o`). We say that `cl(o)` defines `o`. The constant pool of `o`, `cp(o)`, is the constant pool of the class file that was used by `cl(o)` to create `o`. `n(o)` is the fully qualified name of the class whose classfile was read by `cl(o)` to create `o`.

Over the course of execution of `J`, a loader `l` may be presented with requests by the JVM to load a class, emanating from its desire to do constant resolution. The JVM guarantees that, as part of constant resolution, for any name `n`, it will call `l` at most once to load a class with name `n`. Thus at any given instant in the execution of `J`, `l` will have responded to some finite set of requests, by either returning a valid class object, or refusing to define a class object. (A loader `l` may also have refused to terminate on some request, but since we are only concerned with safety properties, we shall ignore that possibility.)

We shall model this by associating with `l` a mapping m: m(l) from the set `dom(l)` of names in the domain of `l` to class objects.

A name `n` is said to be foreign for `l` if `n` is in `dom(l)` and `cl(m(l)(n))` is different from `l`.

Definition[a refers to b] Let `a` and `b` be two class objects in `J`. Say that `a` refers to `b` if `n(b)` occurs in `a's` constant pool, and `m(cl(a))(n(b))` is defined and equals `b`. That is, `a` refers to `b` if the code for `a` refers to the name of `b`, and the name of `b` is resolved by the loader for `a` into `b`.

Definition[Bridge] Let `J` be a running JVM. A bridge in `J` is a set of four class objects `(r, a', s, a)` such that: (1) `cl(s) = cl(a) =/= cl(r) = cl(a')` (2) `r` refers to `s` (3) `r` refers to `a'` (4) `s` refers to `a` and (5) `n(a) = n(a')`. `r` is said to be the receiver of the bridge, `a'` the spoofer, `s` the sender, and `a` the spoofee.

Definition[Bridge-safe] A JVM `J` is bridge-safe if at no time during its execution (and for any input during its execution) may a bridge come into existence.

Let us develop some general conditions on (class) loaders that will be necessary and sufficient to prevent such bridges from coming into existence.

Definition[Isolating foreigners] A loader `l` isolates foreigners if for every name `n` foreign for `l` every class name `q` in the constantpool of `m(l)(n)` (and in the domain of `l` and `cl(m(l)(n)))` is foreign for `l`.

In the example discussed earlier, no instance of `DelegationLoader` isolates foreigners, since the name `R` occurring in the constantpool of a foreign name, `RR`, is not foreign.

Proposition. Let `J` be a JVM. `J` is bridge-safe iff every class loader that can come into existence during its execution isolates foreigners.

Informal proof. Suppose a bridge `(r, a', s, a)` exists. Then, `n(s)` is a foreigner for `cl(r)`. Assume `cl(r)` isolates foreigners. Then `n(a)` is foreign for `cl(r)` . But `n(a) = n(a')` and `n(a')` is not foreign for `cl(r)` (it is mapped to `a'`). In the other direction assume there is a loader `l` that does not isolate foreigners. Let `n` be a name foreign to `l`, and name `q` be in the constant pool of `m(l)(n)`, and `q` be not foreign to `l`. Construct `a` class `r` in `l` that refers to `n` and `q`. Then each of `r`, `m(l)(q), m(l)(n), m(cl(m(l)(n)))(q)` exists, and taken together constitute a bridge. End of proof.

In general, proving for any arbitrary class loader that it is bridge-safe may be very difficult -- there may not be enough data available, e.g. about the constantpools of the foreign classes. However, some general strategies can be followed for *designing* loaders that isolate foreigners.

Applet classloader

For instance, a loader constructed as follows will always isolate foreigners: It divides its domain into two disjoint parts, the "core" domain, `cdom(l)` and the "user" domain, `udom(l)`. All and only the names in the core domain are foreign. Now any such `l` will isolate foreigners provided that it is the case that for every `n` in the core domain of `l`, `cp(m(l)(n))` is a subset of `cdom(l)`. Again, in general there may not be enough data available to make this decision --- but in practice, one would write the "core classes" (the union, across all `l`, of the sets obtained by mapping `m(l)` across `cdom(l)`) in such a way that they only reference core classes. Under such a design practice, the loaders would isolate foreigners. Note however, that each time a new class was added to the core, one would have to verify that it references only core classes.

From the informal description of the classloaders given in HotJava, it appears that they are written using this methodology. Thus, a user may never be unconditionally certain that a particular HotJava browser running on his desktop is bridge-safe --- but he may be certain under the (reasonable) assumption that the core classes already on his disk (and any other core class to be added later) satisfy the property that they only reference core classes.

Indirect bridges are already ruled out.

Before leaving this topic, I want to point out that another way of causing type-spoofing, apparently described earlier by David Hopgood, does not work. (I should say "does not work anymore".) Given that a "direct" bridge is not possible for loaders that isolate foreigners, one may try instead to construct an indirect bridge as follows. Consider `s` and `r`, such that `cl(s)` is distinct from `cl(r)`. Find an intermediary class `i`, such that `cl(i)` is distinct from `cl(s)` and `cl(r)`. Thus `i` is foreign to both `s` and `r`. Pick a name `q` in the domain of `cl(s)` and `cl(r)`. Define `a` in `cl(s)` to inherit from (the type associated with) `i`, and `a'` in `cl(r)` to inherit from `i`. Now communicate from `s` an instance of `q` typecast to `i`, receive it at `r` at type `i`, coerce it to type `q`, and use it to spoof.

For instance, concretely, two applets `S` and `R` may work in tandem to launch this attack. Both will be loaded into their own loaders. Both define a type, say `RStream` to extend `java.lang.InputStream`, intending to use `java.lang.System.in` as an unwitting conduit between them: `S` creates an instance of its own `RStream`, and stores it in `System.in`. When the user visits the page containing the applet `R`, `R` reads `System.in`, casts the result to (ersatz) RStream, and proceeds to wreak havoc.

The attack fails because the explicit cast at the receiving end generates a `ClassCastException` [Lindholm P. 175]: it checks that the class that the message is an instance of identical to the class being typecast to, or inherits from it. So the `checkcast` JVM instruction checks the "run-time type" as it should.

Section 2. How does this happen?

Why does type-spoofing work? What is happening in the JVM?

On an abstract note, the heart of the problem lies in the somewhat different views of "types" taken by the Java compiler and the Java Virtual Machine. The reality in the JVM is that multiple class files with the same name and arbitrarily different fields and methods can be simultaneously loaded into different classloaders. Therefore, a type should be a **pair** `(FQN, CL)` of a name and the classloader in which the corresponding class was loaded. (Primitive types can be considered to be identical across all classloaders.) Thus two classes have the type iff they have the same `FQN` and the same `CL`. Though this is stated explicitly in [Lindholm P. 24, Sec 2.8.1], very surprisingly neither the Java compiler, nor the JVM build this more refined notion of types fully into their operation.

Current scope or base scope?

If a type is to be thought of as the pair `(FQN, CL)`, then the huge problem arises of how to make sense of [Gosling 96] ! Throughout the book, a type is talked of as if it is an `FQN`. There are clearly two ways of obtaining an `(FQN, CL)` pair from an `FQN` --- one may either assume that an `FQN` stands for `(FQN, CL)` where `CL` is the "current" classloader (I will call this *current scope*), or one may assume `FQN` stands for `(FQN, null)`, where `null` is the "null" or the system classloader (I will call this *base scope*).

> ***It appears that [Gosling 96] intends different interpretations in different places.***

For instance, [Gosling 96, p 40] says:

> The standard class `Object` is a superclass (Sec 8.1) of all other classes. A variable of type `Object` can hold a reference to any object, whether it is an instance of a class or an array (Sec 10).

Which type `Object`? The one associated with the class `(java.lang.)Object` loaded in the *current* classloader (and hence in every class loader) (current scope), or the one loaded in the `null` classloader (base scope)? Experimentally I have verified (in JDK 1.1.3) that an array object can be assigned to a variable with typename `java.lang.Object`, even though at runtime the class `java.lang.Object` loaded into the current classloader is different from the class `java.lang.Object` loaded in the `null` classloader. So it seems that current scope was intended. However, we have on [Gosling 96, p 466]:

> There is no public constructor for the class `Class`. The Java Virtual Machine automatically constructs `Class` objects as classes are loaded; such objects cannot be created by user programs.

Which type `Class`? Current scope or base? Experimentally I have verified that (in JDK 1.1.3) the *base* interpretation is intended in this case: always the `Class` objects created are instances of the `Class` class loaded in the `null` classloader.

It seems to me that the designers intended current scope for "user-defined" classes (and this is how the JVM is designed). Clearly, the notion of multiple classloaders does not make sense otherwise. (You want the classnames in the applet code loaded to refer to the classes loaded into the same loader.) However it seems that base scope is intended for some predefined "system" classes (this notion is not explicitly defined in the book, but implicitly referred to) such as `java.lang.Class` and `java.lang.String`.

In the interests of cleanliness of system design it seems to me that current scope should be adopted uniformly. One very attractive property of such a proposal would be that it would allow different object systems, with very different behaviors to be implemented very easily within Java --- merely by changing the basic classes loaded into a given loader! --- increasing its attraction as a language in which to experiment with different OO language designs.

This confusion in thinking leads to the problem highlighted in this paper. To see how, let us turn to an analysis of how Java links and runs code.

Dynamic linking in Java.

To support dynamic linking, the class file corresponding to a Java source file retains (in its constant pool) the symbolic FQNs of the classes, interfaces, fields, methods (and their typenames) in its byte-code. For instance, a method invocation on an object is associated in the class file with the name of the method being invoked, the name of the class containing the declaration of the method, and the *descriptor* of the method, which captures the *name* (and sequence) of the argument typenames and the return typenames of the method.

At run-time operations involving these symbolic references are converted into operations involving actual offsets into field and method-tables through a process known as constant pool resolution (CPR) [Lindholm Chapter 5].

> **Footnote:** The opcodes which can initiate CPR activity are: `getfield`, `getstatic` (getting the value of instance and static fields); `putfield`, `putstatic` and `aastore` (setting the value of instance and static fields, and entries in an array); and `invokeinterface`, `invokespecial`, `invokestatic` (invoking constructors and methods).

Now consider what happens at run-time when a method `m` on class `c`, with descriptor d `is` invoked on object `o` (using the `invokevirtual` instruction, [Lindholm P267-8]. (Similar considerations apply to other instructions concerned with reading/writing fields, or invoking methods.) The associated class loader is asked to load the class `c`. (This may involve, recursively, the loading of other classes, e.g. the superclass of `c`.)

> **Footnote:** Note that the code loaded by the class loader in response to this request may have *no* relationship with the code used by the compiler to compile this class. No "compiled-with" information is stored by the compiler in the class file for use at link-time.

Once that is accomplished, `d` is matched against the descriptors of methods defined in the just loaded class (this is called resolving the method) [Lindholm 97, p148].

> If the referenced method does not exist in the specified class or interface, field resolution throws a `NoSuchMethodError`.

From the description it is not completely clear how it is determined that the referenced method does not exist in the specified class or interface. The most natural assumption seems to be that two methods are considered "equal" if they are name-equivalent , i.e., they have the same name and the same method descriptor, which records the sequence of FQNs for the arguments and the FQN for the result.

An exception is thrown if there is no such method. The result of resolving is an index `i` into a method table.

Note that this entire process involves classes loaded by the current class loader. These classes are supposed to be the runtime equivalents of the classes used by the compiler when creating the class file, so this process is analogous to what a compiler would have done in a statically-linked language: identify the layout of the class on which a method is being invoked, and determine the offset of the method in it. Note that:

> ***No "run-time" information (e.g., the actual `Class` object corresponding to `o`) is used in this process.***

Now that this offset has been determined, it is *assumed* that this is a valid offset in the method table of the actual (run-time) class of the object. The method description *assumed* to be at that offset is then executed [Lindholm 97, p267-8]

> The constant pool entry representing the resolved method includes an unsigned *index* into the method table of the resolved class and an unsigned byte *nargs* that must not be zero.
>
> The *objectref* must be of type *reference*. The *index* is used as an index into the method table of the class of the type of *objectref*.

Footnote: On first glance, it may seem that the index `k` could equally have been used to index into the method table `M[B]` of the resolved class `B` . However, that would be incorrect. The object being operated upon may actually be an instance of a subtype `C` of the "compile-time" type `B` . `M[C].k` , the entry at index `k` in the method table for `C` may thus contain a pointer to a piece of code that overrides `M[B].k` , and it is `M[C].k` that should execute, per the language rules detailed in [Gosling 96].

For this technique to work, it is crucial that the index `k` computed at link-time from the compile-time typename `B` point to the "same" method in `M[C]` . Therefore, the method lookup operation --- which determines from a method signature and an object the piece of code of that signature that should run on the object --- can be optimized away at compile-time, as is standard for statically-typed OO languages. However, the notion of "method tables" --- and how they might be computed, and how method lookup might be optimized away, and the constraints that it imposes on method and field layout --- is not discussed anywhere in [Lindholm 97], a most regrettable oversight, particularly so because it will turn out to be quite related to a proposed fix below.

Here is where the problem becomes manifest: The above scheme is a correct implementation strategy **exactly** under the assumption that the class of the type of *objectref* is `C`, the just-resolved class. As we have seen, this assumption is not always true.

Therefore, type-spoofing arises as a consequence of the particular way in which JVM instructions have been defined. Hence it should arise in any valid implementation of the JVM spec. In addition, Sun's JVM implementation uses certain "quick" instructions to rewrite the opcode corresponding to the invocation with the information obtained from method resolution. This is crucial to avoid the cost of symbolic lookup on every member access, and it makes sense under the assumption above, since the information obtained from member resolution is invariant under any operations on the JVM (e.g. the JVM does not allow classes to be reloaded). But if the assumption is invalid for a particular call, then the "quick" instructions merely speed up an erroneous process.

However, it is clear that any reasonable implementation must work to avoid incurring the constant pool resolution cost on every member access. Therefore, an important consideration in evaluating schemes to fix the type-spoofing problem has to be its support for "quick" schemes.

Section 3. How can it be fixed?

This particular failure of type-safety may be fixed in various ways. One may consider enriching the notion of types that the compiler works with to include also some static representation of class loader identity. However, rather than modifying the Java language, in the following I consider three ideas that tackle the problem of repairing type-safety for Java at the level of the JVM.

Allow only one class per FQN to be loaded in.

Type-spoofing cannot happen as long as every class loader L responds to a `loadClass` request by performing a `defineClass` on some appropriately obtained bytes. Consequently, L will be asked to resolve any type references within the class just loaded, and so on --- thus there can be no possibility of an instance coming into L's world (that is, into the state of an object that is an instance of a class loaded by L) which is not an instance of a class loaded in L. And since L, like every other class loader, guarantees that there is at most one loaded classfile for every FQN, there can be no spoofing.

In a related vein, one may mandate a global consistency condition across all classloaders: *for any given FQN, at most one class file can be loaded into a JVM*. This can be achieved, for instance, by generating an exception if any class loader attempts to call a *defineClass* for a FQN for which a *defineClass* has already been called (regardless of the loader involved).

This proposal has some merits. The notion of class loaders still makes sense --- a particular class loader can still be used to enforce "name space access" policies. Constant pool resolution can still be used to trigger a request to the class loader to load a class --- which a class loader is free to deny or service.

However, it will also make impossible some rather interesting uses of class loaders that are currently permitted. Currently, it turns out to be possible to define a class loader which can redefine system objects, e.g. `java.lang.Object`, for the classes loaded into it. This is of great use in cases (e.g. in the design of Matrix) where it is desired to run arbitrary Java code unchanged, while guaranteeing some additional properties (e.g. that the number of objects created by the code is bounded). However this can only be accomplished if there are two classes with the FQN `java.lang.Object` loaded into the JVM: one is used in the name-space for the application to provide the "controlled" version of the type, and the other is used in some other loader to provide the primordial class from which all other classes are constructed.

> **Footnote:** Some care has to be taken in compiling these classes since the Java compiler --- unaware that these two classes with the same FQN are going to be loaded into different class loaders ... it has no conception even of different class loaders!! --- may erroneously claim type circularity. A simple solution is to transform the classfile generated and splice in the correct superclass manually.)

Schemes for security in a similar vein are also suggested in [Wallach 97].

Modifying the semantics of the JVM.

We consider now two proposals to fix this problem by fixing the JVM.

Check for type-spoofing at run-time.

Java is often said to have a "static" type-system. A more accurate term would be "link-time" type system, since many type checks are delayed till link-time (and almost no type-checks are performed at run-time; here by run-time I mean the second or subsequent invocation of an instruction). For instance, as discussed above, symbolic references to methods are resolved into concrete offsets into the method table only at link-time, after constant pool resolution. If the method does not exist, an exception is thrown.

One way of fixing the type-spoofing problem is to perform the check for type-safety at runtime. Thus instructions such as `invokevirtual` should check that the `Class` of the object being operated upon is in fact the object generated by loading the classfile obtained by resolving the type. If not, then an `IllegalReferenceException` should be thrown.In essence it should not be possible for a class like *RT* to use static types to operate on an instance of *R* --- in some sense the type corresponding to *R* should be considered *hidden* in L' by ersatz *R*. (However, it should continue to be possible for *RT* to operate on an instance of *R* through reflection (that is, using the class object corresponding to *R*). Such a use is type-safe since only the methods defined in *R* can be used to operate on the instances of *R*.)

A natural question arises whether this run-time check can be reduced to a link-time check. That is, would it work to just check the use of the particular invokevirtual instruction first time it is executed? The intuition would be that if the first time around the `Class` of the object being operated upon is identical to the class obtained by resolving the type, the the instruction could be rewritten to the quick form of the instruction. Subsequently the quick version would not need to perform the runtime check.

This scheme cannot work, however, for there may be more than one sources for the spoofed type. Using earlier terminology, there may be multiple bridges, sharing the same receiving endpoint. Put the expression in a method call, so now there is no link-time way of knowing whether or not all or none of the executions of `invokevirtual` will generate errors:

```
public callSpeakUp(R r) {
  r.speakUp();
}
```

quick instructions may still be of some use however: Check if the runtime class is the expected class, if so use the offset stored in the quick instruction, else throw an exception.

Check for type-equivalence not name-equivalence

Run-time performance is a big drawback of the scheme given above, though the additional flexibility of run-time typing is considerable.

However, let us ask ourselves the question: why did the need for run-time type-checking arise in the first place? Let us go back and examine the canonical program:

```
public class RT {
  public  static void main() {
    try {
      System.out.println("Hello...");
      RR rr = new RR();
      R r  = rr.getR();
      System.out.println("  r.r is " + r.r + ".");
      r.r = 300960;
      System.out.println("  r.r is set to " + r.r + ".");
      System.out.println("...bye.");
    } catch (Exception e) {
      System.out.println("Exception " + e.toString() + " in RT.main.");
    }
  }

}
```

If we assume that this program text is to be understood with FQNs resolved using current scope, then it is clear that the `rr.get(R)` should return something of type `(R,L')`, where `L'` is the classloader in which `RT` is loaded. However, the method `getR` defined in `RR` (which is of type `(RR,L)` actually returns something of type `(R,L)`, a different type! Therefore the method that is being looked for here, namely a method named `getR` of type `() -> (R,L')` does not actually exist in class `(RR,L')` (which is the same as `(RR,L)`). Therefore method resolution should *fail* , and a `NoSuchMethodError` should be thrown.

This therefore is a general fix for this problem: use *type-equivalence* instead of name-equivalence when resolving methods and fields. Instead of comparing equality of method descriptors, resolve the names that occur in the descriptors, and consider the descriptors to be the same only if the resolved names are identical. Thus, in this example, compare the signatures `() -> (R, L)` and `() -> (R,L')` instead of the descriptors `() -> R` and `() -> R` .

We do not yet have a formal semantics for the JVM (though a simple constraint-based typing scheme for Java and the JVM is being developed for which it should be easy to establish soundness). Here we can only argue informally for correctness. Intuitively, with this fix, we will have the property that any location `l` with typename `N` (e.g. local variable) created from a class `C` can only store objects whose type is `(N, L)` where `L` is the classloader that `C` was loaded in. Thus it is as if the code executing at run-time is obtained from the code at compile-time by uniformly replacing the names `N` by the types `(N,L)`. If at compile-time the constraints on types generated from a class were consistent when type-names were substituted for types, then at run-time these constraints should be consistent with `(name, CL)` pairs being substituted for the types --- or else a linkage error would occur. (One can think of these errors being discovered through propagation of equality constraints between types; when a classloader `L` forwards a request to load a class `C` to a loader `L'`, it is as if it is publishing the constraint `(C,L) = (C,L')`. Link time type-checking is merely propagating the consequences of a conjunction of such constraints.) Thus compile-time type-consistency should "parametrically" translate to run-time type-consistency. This argument needs to be made precise.

An attractive property of this fix is that there is no run-time cost, since constant pool resolution is a link-time activity. Thus this appears to be the appropriate fix for this problem.

Implications for method table computation.

An important implication of uniformly using type-equivalence rather than name-equivalence is worth describing explicitly, since it highlights some subtle interactions.

Consider the code:

```
class B {
  void m(T a) {..code1...}
}
class C extends B {
  void m(T a) {...code2...}
}
class D {
 void r(B b) {
   b.m(new T());
  }
 void s() {
  r(new C());
 }
}
```

Now consider two class loaders `L`and `L'` such that (in our earlier terminology) `cl(m(L)(B))=L'`, `cl(m(L)(C))=L` and further `cl(m(L)(T)) =/= cl(m(L')(T))`. That is, `B` and `C` are loaded into different class loaders, and the two classloaders differ on how they interpret `T`. Suppose `D` is loaded in `L'`. In this case, the call to `b.m` will resolve at type `(T, L') -> void` and will obtain the offset corresponding to `code1`. Suppose `D` is loaded in `L`. In this case, the resolution of `b.m` at type `(T, L) -> void` will yield a `MethodNotFoundError`. In neither case will `code2` be considered to have overridden `code1` . This is the case even if at runtime, as in the case of the call from within `s`, the actual argument passed into `r` is an instance of a class with typename `C`.

An implication of this example is that type information, rather than just typename information, must be taken into account at the time that the method table for a class is built. In detail, when a loader `L` is ready to create an instance of class `C` that inherits from `B`, `L` must determine the method table of `C` given that of `B`. In order to do so, the typenames that occur in the arguments of methods defined in `C` must be resolved, so that it can be determined whether `L` and the classloader for `B` agree on their interpretation. (Two classloaders `L` and `L'` agree on the interpretations of a name `N` if they both map `N` to the same `Class` object.)

The requirement to resolve method typenames when a method table is to be constructed for a class may be considered somewhat onerous. It requires "preloading" some classes (the classes corresponding to argument types of methods). Three points are to be made here.

First, preloading is needed only if the class is not already loaded --- as more and more classes get loaded over time, the number of classes that would need to be preloaded should decrease.

Second, preloading is necessary only if the parent class has been loaded into a different class loader. If it is loaded into the same loader, then by definition the types associated with the same name in the constant pools of both classes will be the same. For most (perhaps even almost all) classes, this will be the case.

Third, it is not necessary to perform any of the operations with the preloaded code (e.g. preparation, initializaton, verification etc). Rather, an even weaker notion than interpretation-equivalence --- *definer-equivalence* --- can be used. For a classloader `L` and name `N` , define `D(L, N)` to be the classloader whose `defineClass` operation will yield the `Class` object that `L` will return when asked to resolve `N`. Then, all that is needed is to determine, for each relevant `N`, whether `D(L, N) = D(L', N)`.

As an aside, it is not very difficult to design a protocol between the JVM and the classloader which allows the JVM to deduce definer-equivalence information in a reliable way. When the JVM needs to obtain `D(L, N)` information, it calls a (user-definable)

```
java.lang.ClassLoader definingLoader(String n)
```

method on `java.lang.ClassLoader` object `L` with argument `N`, recording the result in an internal table. This table may now be used to resolve definer-equivalence questions. Subsequently, when the JVM has need (e.g. through the constant pool resolution process), to resolve `N`, it will call

```
Byte[] loadBytes(String)
```

operation on the object previously recorded, and perform an internal `defineClass` operation to obtain the class object.

Acknowledgement: Thanks to Gilad Bracha for clarifying discussions on this point, and for suggesting that an

explicit discussion here would be appropriate.

Implications for name space coordination.

A consequence of this fix is that the responsibility for avoiding link-time type-errors now falls on the class-loaders. If they are to share a type (e.g. `RR`), then they must arrange to share all types referenced in that type (e.g., `R`), otherwise link-time errors will be generated. Crucially, the JVM will not crash --- only link-time errors will be generated, which, in some sense, is the best that can be expected. It is not too difficult to devise "type-publication" schemes (as we have done for Matrix by which class-loaders can cooperate (by dynamically sharing appropriate parts of their name spaces) so that link-time type-errors can also be avoided. More details will be developed in a fuller version of this paper.

A final remark. If this solution were to be taken to be the one that Java designers had in mind, it is rather surprising that there is such a big gap between the type-expressiveness of Java and what is possible with classloaders. In essence, the notion of classloaders has not been reified in the type-structure of the language --- it remains strictly under the hood. The only programs that can be written in Java (the language as it now stands) are those that are "uniformly parametric" over classloaders (that is they work the same way in all classloaders), and that cannot statically refer to types other than in the current classloader. I do not view either of these conditions as necessary for what I take to be a real technical contribution of Java designers, namely, link-time type-checking. For instance it should be possible to define link-time type-checkable schemes which allow a class to impose certain constraints on "foreign types", e.g. requiring that they be mutually consistent (i.e., come from the same classloader). This view of a classloader as imposing a certain consistency condition on type-resolution needs to be developed more fully.

Section 4. Conclusion

Java is a big, paradigm-forming leap forward for the C/C++ family of languages. It is clean enough that formal analyses of the language (and its type system) can be contemplated, and rich and powerful enough that large (distributed) systems development can be supported. Even more importantly, its elegance makes it a pleasure to program in.

Nevertheless, it is a new language being developed with breakneck speed, sometimes in areas which are not yet clearly understood by researchers. A rigorous, perhaps even formal, analysis of the language, focusing on its security properties seems urgently called for. Otherwise we will continue to have the spectre of subtle, but potentially fatal, design flaws hovering over our heads.

Related work. Some brief comments about related work. Recently there has been much interest in security for Java. Javasoft's security FAQ contains information their status on security-related bugs they know of currently. The Kimera project has developed their own bytecode verifier and is using some weak methods to probe for flaws in Javasoft's bytecode verifier. In addition, they are working on a security architecture for Java. The Secure Internet Programming group at Princeton has explored a variety of security-related issues. The Java Security: Hostile Applets, Holes, and antidotes contains a very readable account of recent work on security bugs in Java.

Classloaders have come under some scrutiny recently. The so-called Princeton class-loader attack involves a hostile class-loader that responds with different class objects to queries for the same name. This has been neutralized by keeping the table mapping names to classes internal to the JVM --- the

JVM now guarantees that it will call a loader at most once for any given name. The Hopwood tag-team applet attack is described above (building an indirect bridge). The attack apparently used to work because classcasting of exceptions and interfaces was not implemented correctly in earlier versions of Java. The technique for subverting the type system described above is more insidious in that it does not rely on any classcasting.

After I circulated this note, some earlier related work was brought to my attention. Drew Dean remarked that he had made this realization in January 97, after someone posted a program on a news group, which implied this problem. He has since developed some ideas for fixing this problem.

In his ECOOP ’96 tutorial, Martin Odersky noted that multiple classes with the same name can be loaded at once in different classloaders and are treated as "the same type". This is not strictly true, since as we have seen above, different instructions behave differently, some (such as checkcast) are sensitive to (typename, loader) information, and some only to typename information. He also noted that private variables can be accessed by declaring them public in a clone class. Indeed, there seems to be a "related" bug in JDK in which public access (from classes loaded from the null classloader) to private methods is not checked by the Javasoft verifier. (This privacy violation was also pointed out to me by Nevin Heintze.)

Bibliography

[Lindholm 97] Tim Lindholm and Frank Yellin "The Java Virtual Machine Specification", Addison-Wesley, 1997.

[Gosling 96] James Gosling and Bill Joy and Guy Steele "The Java Language Specification", Addison-Wesley, 1996.

[Saraswat 97] Vijay Saraswat "The Matrix of Virtual Worlds", AT&T Research, manuscript, July 1997.

[Wallach 97] Dan S. Wallach, Dirk Balfanz, Drew Dean, Edward W. Felten "Extensible security architectures for Java", Technical Report 546-97, Department of Computer Science, Princeton University, April 1997. Online version.

Appendix: Code listing

Place in the current directory the (.class) files for: Test, (the real) R, RR, DelegatingLoader, LocalClassLoader. Make sure the current directory is on CLASSPATH.

Place in ./ersatz the (.class) files for: (ersatz) R, RT, RT2, RT3.

```
// LocalClassLoader.java
import java.lang.*;
import java.util.*;

import java.lang.reflect.*;
import java.io.*;

/** Defines a Class Loader that knows how to read a class
 *  from the local file system.
```

```
 */

public abstract class LocalClassLoader extends java.lang.ClassLoader {
  private String directory;
  public LocalClassLoader (String dir) {
   directory = dir;
  }

  protected Class loadClassFromFile(String name, boolean resolve)
       throws    ClassNotFoundException, FileNotFoundException {
    File target = new File(directory + name.replace('.', '/') + ".class");
    if (! target.exists()) throw new java.io.FileNotFoundException();
    long bytecount = target.length();
    byte [] buffer = new byte[(int) bytecount];
    try {
      FileInputStream f = new FileInputStream(target);
      int readCount = f.read(buffer);
      f.close();
      Class c = defineClass(name, buffer, 0, (int) bytecount);
      if (resolve) resolveClass(c);
      System.out.println("[Loaded " + name + " from " + target + " ("+ bytecount + "
      return c;
    }
    catch (java.lang.Exception e) {
      System.out.println("Aborting read: " + e.toString() + " in LocalClassLoader.")
      throw new ClassNotFoundException();
    };
  }
}

// Test
import java.lang.reflect.*;

/** Test harness for classloader examples. Loads the user class into
 * a newly constructed DelegatingLoader.
 */
public class Test {
  DelegatingLoader loader;

  public void doIt(String argv[]) {
    try {
      if (argv.length < 1) {
        System.out.println("Usage: java Test ");
        return;
      }
      String target = argv[0];
      this.loader = new DelegatingLoader("ersatz/");
      Class c = this.loader.loadClass(target, true);
      Object [] arg = {};
      Class [] argClass = {};
      c.getMethod("main", argClass).invoke(null, arg);
    } catch (Exception e) {
    System.out.println("Error " + e.toString() + " in Test.doIt.");
    }
  }
  public static void main(String argv[]) {
    Test t = new Test();
    t.doIt(argv);
  }
}
```

```
// RT3
public class RT3 {
  public  static void main() {
    try {
      System.out.println("Hello...");
      System.out.println("Going to attempt to read a field that exists in the ersatz
      RR rr = new RR();
      R r  = rr.getR();
      System.out.println("  r.s is " + r.s + ".");
      System.out.println("...bye.");
    } catch (Exception e) {
      System.out.println("Exception " + e.toString() + " in RT3.main.");
    }
  }

}
```