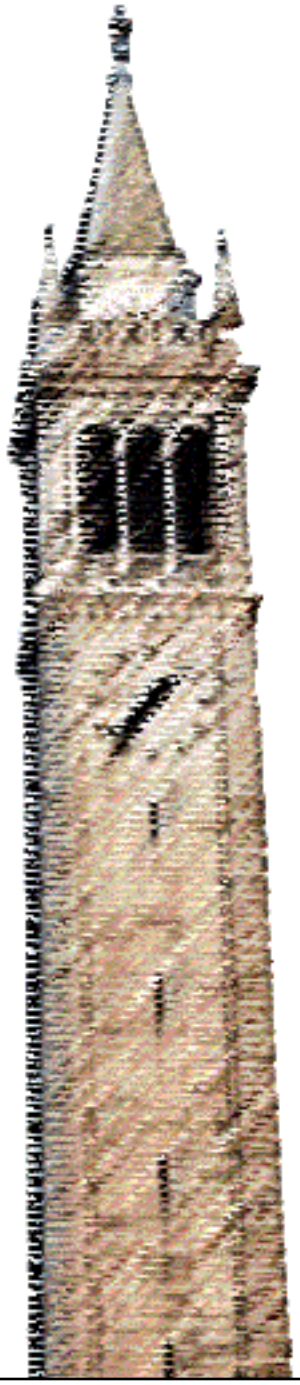
# The Joe-E Language Specification (draft)



*Adrian Matthew Mettler*
*David Wagner*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2006-26
http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-26.html

March 17, 2006

Acknowledgement

This work was supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422), and by Hewlett-Packard.

# The Joe-E Language Specification (draft)

Adrian Mettler David Wagner
{amettler,daw}@cs.berkeley.edu

March 17, 2006

**Disclaimer**: This is a draft version of the Joe-E specification, and is subject to change. Sections 6 - 8 mention a some (but not all) of the aspects of the Joe-E language that are future work or current works in progress.

## 1 Introduction

We describe the Joe-E language, a capability-based subset of Java intended to make it easier to build secure systems. The goal of object capability languages is to support the Principle of Least Authority (POLA), so that each object naturally receives the least privilege (i.e., least authority) needed to do its job. Thus, we hope that Joe-E will support secure programming while remaining familiar to Java programmers everywhere.

## 2 Goals

We have several goals for the Joe-E language:

- **Be familiar to Java programmers.** To minimize the barriers to adoption of Joe-E, the syntax and semantics of Joe-E should be familiar to Java programmers. We also want Joe-E programmers to be able to use all of their existing tools for editing, compiling, executing, debugging, profiling, and reasoning about Java code.

  We accomplish this by defining Joe-E as a subset of Java. In general:

  > *Subsetting Principle:* Any valid Joe-E program should also be a valid Java program, with identical semantics.

  This preserves the semantics Java programmers will expect, which are critical to keeping the adoption costs manageable. Also, it means all of today's Java tools (IDEs, debuggers, profilers, static analyzers, theorem provers, etc.) will apply to Joe-E code.

  In this document, we define the Joe-E language by specifying constraints that will be verified by a Joe-E verifier. These checks may be performed at the source level, or possibly upon bytecodes produced by a compliant compiler.

- **Include as much of Java as possible.** Some Java constructs must be omitted from Joe-E, because they are incompatible with capability programming. However, we ideally want to retain as much of Java's expressiveness as possible.

  > *Maximal Subset Principle:* Choose the largest subset of Java that is compatible with secure capability programming. Forbid only language constructs that render capability-style programming or reasoning impossible or error-prone.

  Thus, Joe-E will permit construction of secure programs, but it will not guarantee that programs in this subset will be secure.

- **Enable capability-style reasoning about Joe-E code.** To ensure that objects receive the least authority needed, the language must allow capability-style reasoning. Capability-style reasoning involves thinking about the directed graph of object references: if object $O$ has a reference to $O'$, then we draw an edge $O \rightarrow O'$ in the graph. To characterize how $O$ can causally affect the outside world (its authority), we examine the set of capabilities that $O$ might have. An upper bound for this set is the bidirectional transitive closure of this graph, i.e., the set of objects reachable from $O$ via backward and forward edges in the reference graph. If the code is constructed appropriately, it is often possible to improve upon this approximation by verifying that the program semantics for some methods prevent the transitive closure worst-case from occurring; it should be easy to write code like this (e.g., facets), and it should be easy to verify the correctness of this code through purely local reasoning.

  Note that this style of reasoning assumes that references are unforgeable and that references are the only thing that convey authority—in particular, that there is no ambient authority.

- **Unforgeable references.** It must be impossible to manufacture a reference to an arbitrary object. Java's memory-safety ensures that objects can only obtain references through specific controlled mechanisms. Specifically, references held by an object $O$ can originate only in the following ways.

  1. **Endowment:** When $O$ is instantiated, it receives as its birthright any references passed to its constructor and (if it is a non-static inner class) anything visible as part of its lexical scope (including a reference to itself, via `this`);
  2. **Parenthood:** When $O$ creates a new object, it receives a reference to that object.
  3. **Introduction:** References can be introduced to an object in a variety of ways, namely:
     (a) By field access: If $O$ has a reference to $O'$, then $O$ can obtain any references stored in the accessible fields of $O'$.
     (b) By field mutation: If $O$ has a field accessible to another object, $O$ receives any references stored into this field by that object.
     (c) By invoking: When $O$ calls a method on some other object, $O$ receives any reference returned or thrown by that method.
     (d) By being invoked: When a method is called on $O$, $O$ receives any references passed as the arguments to this method call.

- **No ambient authority.** A precondition for capability-style reasoning is that the only way for an object $O$ to affect the outside world is through the references it possesses. To ensure least authority and support capability-style reasoning, the references that $O$ possesses should be limited by lexical scoping rules: e.g., limited to its fields, with no "global variables" allowed. For instance, `java.lang.System.out` violates this principle, because it is available to every object and allows causally affecting the outside world.

  Avoiding ambient authority is important for POLA. It should be easy to limit the amount of authority given to an object to only those capabilities necessary to perform its function, and to do so in a way that is foolproof and easy for an human auditor to recognize. In particular, the rule should be that no object receives a capability unless it has been explicitly granted that capability in one of the ways described above. Variables that are defined in a location distant from any code that can use them should be avoided, because they violate this rule. In addition to this static (code-based) constraint, sharing of state between objects associated with different flows of control or protection domains should be avoided as much as possible, even if they share some source code. In general, since ambient authority is available to all objects, ambient authority is incompatible with POLA and must be avoided at all costs.

- **Support local reasoning about Joe-E programs.** A closely related goal is that it should be possible to reason about Joe-E programs through only local analysis. Suppose we are given a Joe-E program, consisting of many objects. If we are given the code to one object $O$, we would like to be able to reason about the capabilities that $O$ might have access to and might pass on to others. It

should be possible to reason about this just by looking at the code of $O$ and the objects it interacts with (possibly continuing transitively as far as is needed).

For instance, suppose method `C.m()` creates a new object $T$. If $T$ never escapes from the method, then we would like to be able to conclude that no other object can affect $T$. Moreover, we would like to be able to verify, just by looking at the code of the class `C`, that $T$ cannot escape. This is an example of local reasoning, a kind of reasoning about composition, and the language should support this kind of reasoning. Such reasoning might assume nothing about the rest of the program, other than that it is valid Joe-E code; the benefit is that a programmer does not have to keep the entire program or system in his head in order to reason about local properties of his code.

As another important example of this, it must be possible (preferably with very little effort) to bound the capabilities granted to any object $O$. This should be possible even when the code of object $O$ has not been analyzed; typically, it is done by reasoning about the capabilities that are passed to $O$ when it is created and thereafter. This task allows for the creation of systems that are secure in the face of unknown, possibly adversarial code.

- **Unforgeable types.** If we invoke a method on object $O$, we may be relying on it to behave as we expect. Therefore, we may need some way to verify that $O$ is the correct entity before we invoke method calls on it. In the E language, this is accomplished with guards, auditors, and introspection; in Joe-E, we use types. Some classes are part of the base system and hence trustworthy; other classes are provided by the programmer and thus trusted. Joe-E programs should be able to check that the type of $O$ is the expected one before invoking method calls on $O$. This ability is provided by Java's type system.

  Without this feature, masquerading attacks are a serious risk. Suppose that method `m()` accepts an object $O$ and uses it to perform various operations. If `m()` does not validate the type of $O$, then an attacker might be able to call `m()` and pass a malicious object $O_M$. For instance, $O_M$ might undetectably emulate the expected behavior while simultaneously observing information that `m()` passes to its argument, leaking these secrets back to the attacker. Or, $O_M$ might behave in unexpected ways: e.g., if `m()` expects its argument to behave like an `Integer`, then `m()` might be surprised if two attempts to read its contents return different results (this can lead to TOCTTOU attacks, for instance). It is essential for there to be a way to avoid this kind of masquerade attack.

- **Permit use of capability discipline.** Capability discipline refers to a set of guidelines for writing programs in a way that maximizes the chances that the program will be secure and respect POLA. Suppose I want to give Alice access to a file on the filesystem. I can give her a reference to a `File` object $F$. Of course, in doing so, I have given Alice all of the authority that can be invoked through the interface of this object. If every `File` object has a `formatHardDrive()` method that erases the entire hard disk, then in giving Alice access to $F$, I have also given her the ability to re-format the hard disk, a violation of POLA. In this case, we say that $F$ fails to respect capability discipline.

  It is a goal of Joe-E that it should be natural and easy to build classes that respect capability discipline. It is not a goal of Joe-E to somehow guarantee that every Joe-E class will respect capability discipline; capability discipline requires knowledge of application semantics and the desired security policy, and hence cannot be enforced at the language level.

  Our general stance in defining Joe-E is include everything that is not outright incompatible with capability-style reasoning (see the Maximal Subsetting Principle). One can identify many syntactic source code patterns that are suspicious and often correspond to violation of capability discipline (e.g., `public` fields), but that are not inherently incompatible with security. We do not attempt to forbid such syntax in the Joe-E language. One might build a separate "capability-style lint" to check for suspect language constructs that are risky but not necessarily incompatible with capability reasoning; however, such considerations are out of scope for this document.

- **Provide a set of capability-friendly base classes.** Joe-E should provide a set of library classes that enable programming in the capability style. At a minimum, this collection should contain the minimum necessary to build useful Joe-E programs. These base classes should respect capability discipline and

should be constructed to maximize the likelihood that programs built using the base classes will respect POLA and will be secure.

# 3 Definitions

The Joe-E Language is a subset of the Java source language as defined in the Java Language Specification, 3rd Edition.

## 3.1 The Overlay Type System

Joe-E defines marker interfaces that are used to indicate properties of certain classes. In many cases, it would be useful if we could have system classes in the standard Java libraries be marked as implementing these interfaces. However, we cannot modify existing Java class libraries. Instead, Joe-E defines an extended type system overlayed on top of the Java type system.

The *base type system* is the type system defined by the Java language; it defines certain subtyping relationships. In addition, Joe-E implementations may provide a way to declare some classes to *honorarily implement* a specified interface, and this defines some additional subtyping relationships. The *overlay type system* is obtained by taking the union of the subtyping relationships from these two sources, along with their transitive closure. Note that since honorary components of the type system only add interfaces to classes, the overlay type system will be a consistent, "legal" typing relation (no circular subtyping, etc).

All static typing uses the base type system, just like Java. However, some of Joe-E's additional restrictions are defined in terms of the overlay type system.

## 3.2 Compliance and Deeming

Each marker interface is associated with a set of restrictions that must hold for all classes implementing that interface. This allows one to use the type system to leverage certain types of capability reasoning, for example that a specific class of objects conveys no authority and thus can be freely distributed to other components of a software system. These restrictions are automatically checked as described in Section 4; the decision procedure used is sound but not complete. Because there will be classes that satisfy the contracts associated with a marker interface, but which cannot be automatically verified to do so, a Joe-E implementation may *deem* certain classes to satisfy these requirements, thus exempting them from automated verification. Such deemings must be made with care, and are restricted to classes in the Java and Joe-E libraries that have been manually verified to satisfy the deemed interfaces.

Example: Suppose class `C` is deemed to honorarily implement interface `I`. Suppose also that class `D` is declared to extend `C` in the base type system. Then `D` is considered to implement `I` in the overlay type system, even though it might not implement `I` in the base type system. `D` is not considered to have been deemed to honorarily implement `I` merely by virtue of extending a class that was deemed to honorarily implement `I`. While the marker interface (and thus its associated obligations) are inherited, the deeming status is not; this is because `D` may violate obligations that `C` maintained.

## 3.3 Deep Frozen Types

A type $T$ is deep frozen if and only if it implements the marker interface `org.joe_e.DeepFrozen` according to the overlay type system. The `org.joe_e.DeepFrozen` interface must be provided by the Joe-E implementation.

A Joe-E implementation may deem certain types from the standard Java libraries and from the Joe-E library to implement `org.joe_e.DeepFrozen`, provided their behavior can be manually verified to be that of a transitively immutable object.

### 3.4 Incapable Types

A type $T$ is incapable if and only if it implements the marker interface `org.joe_e.Incapable` according to the overlay type system. The `org.joe_e.Incapable` interface must be provided by the Joe-E implementation, and it must be declared to extend the `org.joe_e.DeepFrozen` interface. This ensures that every incapable type is also deep frozen (but not necessarily vice versa).

A Joe-E implementation must ensure that the following types honorarily implement `org.joe_e.Incapable`:

- the null type;
- the primitive scalar types (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, or `double`); and
- `java.lang.Throwable`
- `java.lang.Enum`.

These types are deemed incapable using the honorary mechanism, since they cannot be declared to implement this interface due to linguistic limitations.

A Joe-E implementation may deem specific additional types from the standard Java libraries and from the Joe-E library to implement `org.joe_e.Incapable`, so long as their behavior can be verified to be that of a transitively immutable object that does not reveal its object identity (its "address").

**Rationale:** An incapable object conveys no inherent or identity-based authority and thus can be excluded from the object reference graph without loss of soundness. Any authority granted to the holder of the object is solely a product of the data it contains; this authority could be "forged" by anyone with knowledge of this data and thus does not reflect a type of capability that can be guarded by our system. (Note that cryptographic keys fall into this category; our system is not able to reason about cryptography.) Incapable objects are both deep frozen and selfless. Such objects can be considered to be no more than the data (transitively) contained in their fields. Two references to a single such object are indistinguishable from two references to two copies of the object. The `Incapable` interface can be used to assert that all instances of a user class are incapable. This assertion is checked at compile time by the Joe-E verifier, as described in Section 4.3.

A deep frozen object contains no mutable state and has no references to mutable state. A reference to a deep frozen object cannot be used to enact any state change visible to another reference-holder of the same object. A deep frozen object conveys no authority except for its own unforgeable identity. This unforgeable identity that distinguishes an incapable object from one that is merely deep frozen. (By definition, all incapable objects are also deep frozen.) As above, the `DeepFrozen` marker interface can be used by a programmer to assert that all instances of a user class are deep frozen; this assertion is verified at compile time (see Section 4.3).

In practice, most deep frozen objects are likely to also be incapable. The exceptions are objects explicitly used as unforgeable tokens whose identity can be verified using the `==` operator.

Note that array types are always mutable and thus are never deep frozen or incapable, regardless of their element type.

## 4 Restrictions on Joe-E classes

### 4.1 Threads

For now, we are restricting Joe-E to the single-threaded subset of Java.

**Rationale:** Unrestricted synchronization could allow a set of threads otherwise effectively isolated from the outside world to communicate sensitive information outside its sphere of isolation with relative ease using lock/unlock operations. In the absence of such primitives, one can achieve a (perhaps imperfect, but better) solution to the confinement problem for such a module.

## 4.2 Static Fields

All static fields must be declared `final` and be of an incapable type. (This is automatically the case for enumerations, since java.lang.Enum is honorarily Incapable).

**Rationale:** A reference to an object of any capable type may convey authority. A mutable field conveys the authority to change its value. A public static variable in either of these categories provides ambient authority. While less obvious, this reasoning also applies to private static fields. Since any piece of code can create an object of the type in question, the object thus created has privileged access to the private static state. For the static state to have a reason for being declared non-final, there must be a way for it to be modified. Similarly, if it is of a capable type, in nearly all cases, a newly-created object of this type thus grants authority to utilize this capability. Since anyone can instantiate the object, this is a likely source of ambient authority which could be difficult to spot if static fields were allowed. It should also be mentioned that simple alternatives to the use of static fields nearly always exist.

## 4.3 Incapable and DeepFrozen

If a class `C` implements `org.joe_e.Incapable` in the overlay type system, at least one of the following must be true:

- every instance field `f` of `C` is both declared `final` and of an incapable type.
- `C` is a library class explicitly deemed to implement `org.joe_e.Incapable`.

In addition, `C` cannot be a subclass of `org.joe_e.Token`. Any violation of either of these constraints is a verification-time error. The `org.joe_e.Token` class must be provided by the Joe-E implementation, and neither `org.joe_e.Token` itself nor its sole superclass `java.lang.Object` may implement `org.joe_e.Incapable`.

If a class `C` implements `org.joe_e.DeepFrozen` in the overlay type system, at least one of the following must be true:

- every instance field `f` of `C` is both declared `final` and of a deep frozen type.
- `C` is a library class explicitly deemed to implement `org.joe_e.DeepFrozen`.

"Every instance field of `C`" includes fields of any superclasses (whether accessible to `C` or not; this includes, for instance, all private fields of all superclasses). If `C` is an inner class, then this includes all fields of its lexically enclosing classes. Any violation of the above constraints is a verification-time error.

**Rationale:** The above requirements allow code to be certifiably incapable or deep frozen while allowing such classes to be extended, as long as their subclasses maintain these properties. The verifier will flag an error if a subclass does not fulfill its obligation according to the marker interface. This technique also prevents a circular dependency in determining whether classes that contain mutually-recursive types in their fields uphold the requirements imposed by `Incapable` and `DeepFrozen`.

## 4.4 Throwables

Note that making `java.lang.Throwable` an incapable type ensures that no `Throwable` can contain an object of type `org.joe_e.Token` or any of its sub- or superclasses.

**Rationale:** Exceptions can implicitly communicate capabilities across security boundaries. This propagation can be hard to reason about, because the exceptional flow might not be immediately apparent in the source code. To see how this can cause unpleasant surprises, suppose Alice calls Bob. Bob has some special capability that she lacks, and Bob wants to avoid leaking this to her. At some point, Bob might need to invoke Chuck to perform some operation, passing this capability to Chuck. If (unbeknownst to Bob) Chuck can throw an exception that Bob doesn't catch, this exception might propagate to Alice. If this exception contains Bob's precious capability, this might cause Bob's capability to leak to Alice, against Bob's wishes. Example:

```
class E extends RuntimeException {
    public Object obj;
    public E(Object o) { obj = o; }
}
class Bob {
    // a capability, intended to be closely held
    private Capability cap;
    ...
    void m() {
        new Chuck().f(cap);
    }
}
class Chuck {
    void f(Capability cap) {
        ... do some work ...
        throw new E(cap);
    }
}
class Attacker {
    void attack() {
        Bob bob = ...;
        try {
            bob.m();
        } catch (E e) {
            Capability stolencap = (Capability) e.obj;
            doSomethingEvil(stolencap);
        }
    }
}
```

The problem is that it is hard to tell, just by looking at the code of `Bob`, that Bob's private capability can leak to the caller of `m()`. This is a barrier to local reasoning about the flow of capabilities. By requiring that all throwables be incapable, we ensure that exceptions cannot convey authority or communicate capabilities across security boundaries.

## 4.5 Object Identity

The `==` and `!=` operators can only be applied to:

- two values of primitive type (this includes the case where one of these values is a boxed type that will be auto-unboxed);
- any object being compared with `null`; or
- two references, one or more of which is declared to be of type `org.joe_e.Token` or one of its subtypes.
- two references, one or more of which is declared to be of type `java.lang.Enum` or one of its subtypes.

Any other use of `==` or `!=` is a compile-time error.

**Rationale:** The ability to uniquely identify a deep frozen object independent of the value that it contains can imbue an object that otherwise contains "just data" with a form of authority. For example, a locked box class can recognize whether it has been supplied the right key by keeping a private reference to the key object used only to test if a supplied key is the same one. An object used for this purpose must be an instance of `org.joe_e.Token` or one of its subclasses. A reference to an enumeration type does not convey any authority, not even by its identity, since all such objects are global, universally exported via static fields. For such values, `==` and it's "safer" selfless alternative, `equals()`, are equivalent.

### 4.6 Parameterized Types

At present, a parameterized class is valid Joe-E code if every possible instantiation of the type variables would result in a class that satsifies the Joe-E subset. For example, a class with a static final field of a parameterized type is only allowed if an upper bound of the type is a subtype of Incapable. As another example, a parameterized class implementing DeepFrozen with an instance field of a parameterized type, the associated type variable must be upper bounded by DeepFrozen.

### 4.7 Library Protection

Optional: A Joe-E implementation may impose restrictions to ensure that Joe-E user code does not declare itself to be a member of a library package. These checks are required if any user code is loaded using the same classloader as any library code not protected by the classloader (e.g. by package sealing).

These checks are not required for a standard Java environment, at least not for Java classpath code. The standard classpath libraries are loaded by the native primordial class loader, while all user code is loaded with a Java-language class loader that extends `java.lang.ClassLoader`. These checks may be necessitated if a Joe-E implementation uses the same ClassLoader for Joe-E user code and the Joe-E library.

### 4.8 Native Methods

Native methods are forbidden.

**Rationale:** Native methods bypass the memory- and type-safety checks of the Java language that are necessary to ensure the unforgeability of capabilities. Therefore, they are forbidden.

## 5 Malicious bytecode

This specification requires that Joe-E programs be written in source form, compiled using a Java compiler, passed through the Joe-E verifier, and then executed with a JVM. In particular, we require that every classfile be produced by a correctly operating Java compiler; no part of a Joe-E program is permitted to contain bytecode obtained from untrusted sources or generated by hand.

**Rationale:** Java compilers perform many checks that are not duplicated by the JVM. These checks include[1]: exception safety; access control for inner classes; isolation of user code from classpath code; in-range checks for short types. Since these checks are performed only by the compiler, malicious bytecode can evade these checks. This means that if any part of the program includes bytecode not produced by a legitimate Java compiler, Joe-E programmers will be unable to rely upon exception safety, access control for inner classes, etc.

In principle, one could duplicate all these checks in the Joe-E verifier, and then raw bytecode would not need to be forbidden since programmers could rely upon these language features even in the presence of malicious bytecode. However, the benefit of adding this into the verifier seems to be questionable given the costs, so we omit this (for now, anyway).

## 6 Taming

This is left for later.

Reminder: We will need to suppress `java.lang.Object.hashCode()` and any other `hashCode()` implementations that reveal object identity. We will need to tame `java.lang.Throwable` to ensure it meets the semantics of an incapable class, since Section 3.4 requires it to be deemed incapable. We may also need to tame some existing exception classes defined in the standard Java libraries and then explicitly deem them to implement `org.joe_e.Incapable`, so that their behavior will indeed be consistent with what one would exist of an incapable type (since all throwables are required to be incapable).

[1] See Appendix A for further discussion.

# 7 Work in Progress

A useful property for guarding against time-of-check-to-time-of-use (TOCTOU) is that all reads of final fields be deterministic and consistent. This requires that one cannot read the value of such a field before it is initialized; otherwise the field's value can change between reads.

If any source of nondeterminism is considered a capability, one can also reason that any method called on a deep frozen object (provided it doesn't include an authority-bearing argument) will deterministically return the same result every time it is invoked. Unfortunately, the unmodified Java language provides ambient acces to nondeterminism via virtual machine errors which can be caught by any class. In particular, a class can behave nondeterministically based on the amount of memory available by keeping track of how much memory it must allocate before it receives and recovers from an out-of-memory error. In order to consider nondeterminism a capability, virtual machine nondeterminism must be hidden from objects not granted the authority to see it.

This draft does not ensure thatt these two properties hold. We are currently investigating additional restrictions to impose in order to guarantee them.

# 8 Open Questions

Additional issues under consideration:

- Synchronized on public static DF/Selfless objects: the state of the lock is part of the state of an object. It can be used to communicate information between two otherwise disconnected classes, an overt channel. Not an issue now, since we're in the single-threaded subset of Java, but remember to deal with this, when we come to threaded code.
- Do we need to forbid expressions like `e.class`, where `e` is any expr? I think the answer depends on what taming decisions we make for class `Class`, but let's remember this for later when we look at taming.

# A Issues with malicious bytecode

If we were to accept (possibly maliciously constructed) bytecode from an unknown source and link it into our program (after checking that it is accepted by the Joe-E verifier), there would be many pitfalls to worry about. These include:

- Java's visibility modifiers (private, protected, etc.) can be subverted in the presence of inner classes. The Java compiler inserts synthetic getter/setter methods; while the compiler checks that Java source cannot use these synthetic methods, the JVM does not, so malicious bytecode could exploit them to to gain access to private state. Consequently, Java programmers cannot count on visibility modifiers on inner classes, in the presence of malicious bytecode.

  Joe-E code compiled from source can ensure that these protections are enforced. A more sophisticated implementation of the Joe-E verifier would be able to make the same guarantees for arbitrary bytecode. Extension of the verifier to handle arbitrary bytecode is nontrivial but quite feasible, and may be considered for a version subsequent to the first prototype.

- Exception-safety can be subverted in the presence of malicious code. The Java compiler checks a property that one might call *exception-safety*: if the program calls a method `m()`, and if `m()` terminates abruptly, it can only do so with an exception that is (1) (a subclass of) an class declared in `m()`'s clause; (2) (a subclass of) a `RuntimeException`; or, (3) (a subclass of) an `Error`. This property is enforced only by the Java compiler, and not by the JVM. Consequently, maliciously constructed Java bytecode can throw checked exceptions not declared in its `throws` clause.

  Just as in Java, Joe-E programmers cannot rely on exception-safety in the presence of malicious bytecode. While it would be possible to re-implement the exception-safety checks in the Joe-E verifier, it seems this would require more effort than it is worth.

- Final fields can be assigned to multiple times, in the presence of malicious bytecode. The Java compiler performs so-called "definite assignment" checks to ensure that every final field is assigned to exactly once by the time the constructor exits, and is never assigned after that point. This check is not necessarily duplicated by the JVM; consequently, malicious bytecode might be able to modify `final` fields, subvert our deep frozen/immutability analysis, and potentially violate other security properties.

- Since generic types are implemented by erasure, malicious bytecode can partially subvert type-checking for type parameters. The correct use of type parameters is only checked at compile time, and then is erased. Consequently, though the JVM does check basic type system, the JVM does not and cannot check that type parameters are used correctly. Further details: `http://portal.acm.org/citation.cfm?id=997144` (§5).

- It seems the intuitive range limits for short types can possibly be violated by malicious bytecode. Consider the following method:

  ```
  boolean isByte(byte b) {
      return (-128 <= bb) && (bb <= 127);
  }
  ```

  One might naively expect that `isByte()` can only return true. However, it's not clear that this is actually guaranteed, in the presence of malicious bytecode. The JVM stores `byte` values in 32-bit registers, and handles them almost exactly the same way as `int` values. No range checks are performed. It may be possible for malicious bytecode to pass a full 32-bit value to `isByte()`, causing the method to return false. Similar risks may apply to `boolean`, `byte`, `short`, and `char`. We have not yet evaluated this risk. References: `http://groups.yahoo.com/group/java-spec-report/message/756?threaded=1` `http://archives.java.sun.com/cgi-bin/wa?A2=ind9802&L=java-security&P=2691`

- It might be possible to subvert some of the access checks for `protected` members. Let `x` be a protected non-static field of class `C`, and let `S` be a subclass of `C` from a different package. The Java compiler enforces a special check that `S` can access `obj.x` only if `obj`'s class is in the inheritance subtree rooted at `S` (and not, for instance, if `obj` is of class `C`). This check is a little bit tricky in the presence of overriding, and there have been reports that malicious bytecode can bypass this check. We haven't investigated this in any detail. `http://www.kestrel.edu/home/people/coglio/ftjp04.pdf`

- There may also be some potential danger spots with monitors. What if malicious bytecode fails to follow a stack discipline with its `monitorenter`/`monitorexit` instructions? What if the bytecode fails to call `monitorexit` before returning? What a security-critical library acquires a lock and calls our malicious bytecode, which then uses `monitorexit` to release the library's lock? It is unclear what may be possible here.

- The `ACC_SUPER` bit may allow to subvert the semantics of overridden methods. Suppose class `C` defines a method `m()`, which is overridden by some subclass `S`. With current Java semantics, holding a reference to `S`, you aren't supposed to be able to invoke the body of `C.m()` on this object. However, malicious bytecode could reset the `ACC_SUPER` bit in its classfile and get access to the hidden method, so that it can get an instance of `S` to execute the code specified in `C.m()`. If `S` was relying on overriding to prevent access to `C`'s implementation of `m()`, then malicious bytecode could falsify this assumption.

We make no claims that this is the complete list of pitfalls associated with malicious bytecode. The simplest way to avoid these pitfalls is to compile everything from source using a trusted Java compiler and refrain from accepting raw bytecode from unknown sources; this is exactly what we require when writing Joe-E programs.