Eager Class Initialization for Java

Dexter Kozen[1] and Matt Stillerman[2]

[1] Computer Science Department, Cornell University, Ithaca, NY 14853-7501, USA
kozen@cs.cornell.edu
[2] ATC-NY, 33 Thornwood Drive, Ithaca, NY 14850-1250, USA
matt@atc-nycorp.com

Abstract. We describe a static analysis method on Java bytecode to determine class initialization dependencies. This method can be used for eager class loading and initialization. It catches many initialization circularities that are missed by the standard lazy implementation. Except for contrived examples, the computed initialization order gives the same results as standard lazy initialization.

1 Introduction

Class initialization refers to the computation and assignment of initial values specified by the programmer to the static fields of a class. It is not to be confused with *preparation*, which refers to the assignment of default values to each static field when the class is created—`null` to reference types, 0 to numeric types, etc.

Initialization is a notoriously thorny issue in Java semantics [3,5,10]. For example, consider the legal Java fragment in Fig. 1. What are the initial values

```
class A {
   static int a = B.b + 1;
}
class B {
   static int b = A.a + 1;
}
```

Fig. 1.

of `A.a` and `B.b`? The answer to this question is not determined by the fragment. They can be either `A.a` = 2 and `B.b` = 1 or vice versa, depending on the order in which the classes are loaded.

The standard Java lazy class loading and initialization method can detect such circular dependencies, but it does not treat them as errors, because in some cases they are useful. For example, the program of Fig. 2 contains a common Java idiom. The circularity in this example is a self-loop on the class `Widget`. The class initializer of `Widget` calls the instance initializer of `Widget`, which in turn accesses the static field `nextSerialNumber`. This is a "good" circularity.

```
class Widget {
   static int nextSerialNumber = 10000;
   int serialNumber;
   static Widget protoWidget = new Widget();

   Widget() {
      serialNumber = nextSerialNumber++;
   }
}
```

Fig. 2.

```
int serialNumber;
static Widget protoWidget = new Widget();
static int nextSerialNumber = 10000;
```

Fig. 3.

However, if we were to permute the declarations as in Fig. 3, it would be erroneous, because `nextSerialNumber` is accessed before it is initialized. The value of `protoWidget.serialNumber` will be 0, the default value of the static field `nextSerialNumber` supplied during class preparation, instead of the intended 10000. Although it is illegal for a static initializer to access a static field of the same class whose declaration occurs lexically later [9], the compiler check for this error is typically limited to direct access only. Indirect access, such as through the instance initializer in this example, escapes notice.

The guiding principle here is that static fields should be initialized before they are used. The fragment of Fig. 1 above violates this principle no matter what the initialization order, and any such circularity arising in practice is almost surely a programming error. Ideally, the initialization process should respect initialization dependencies and catch such erroneous circularities wherever possible. But because this principle is difficult to enforce without ruling out good circularities such as Fig. 2, Java compilers do little to enforce it.

Even in the absence of circular dependencies, lazy initialization may fail to initialize correctly. For example, in the fragment of Fig. 4, if `A` is loaded before `B`,

```
class A {
   static int a = 2;
   static int aa = B.b + 1;
}
class B {
   static int b = A.a + 1;
}
```

Fig. 4.

then the fields are initialized correctly, but not if they are loaded in the opposite order.

Lazy loading and initialization, in which classes are loaded and initialized at the time of their first active use, is the preferred strategy of the Java language designers. Other strategies are allowed in principle, but the Java virtual machine specification insists that any exceptions that would be thrown during loading and initialization are to be thrown at the same time as under the standard lazy implementation [9, p. 42]. Unfortunately, the runtime overhead imposed by this restriction would reduce the performance advantage gained by using an eager initialization strategy, besides being a pain to implement. Thus this restriction effectively rules out other strategies for standard Java implementations.

Nevertheless, an eager approach to class loading and initialization may be more appropriate for certain specialized applications. For example, in applications involving boot firmware, boot drivers for plug-in components, and embedded systems, platform independence and security are issues of major concern. The IEEE Open Firmware standard [7], based on Sun OpenBoot, specifies Forth as the language of choice for firmware implementation for reasons of platform independence. The Forth virtual machine is similar to the JVM in many ways, except that instructions are untyped, there is no support for objects, and there is no bytecode verification. But because security is a growing concern, and because the Open Firmware device tree architecture is naturally object-oriented, Java presents an attractive alternative.

Firmware runs in an extremely primitive environment with little or no operating system support or mediation. Boot device drivers run in privileged mode and have full access to the entire system, including other devices. In addition, embedded systems may be subject to real-time constraints. For these reasons, it is desirable to avoid the runtime overhead of lazy class loading and initialization.

Besides the obvious runtime performance advantages, there are other benefits to eager initialization:

- Errors are identified earlier.
- There is a clean description of class initialization semantics.
- Class initialization can be precompiled in JVM-to-native (just-in-time) compilation.

In this paper we describe an algorithm for determining a class initialization order that can be used for eager class loading and initialization. The algorithm runs at the bytecode level and computes a conservative estimate of the true dependency relation on static fields by static analysis of the call graph. Bad circularities, which are almost surely programming errors, are caught, whereas good circularities are allowed to pass. This distinction is defined formally in Section 2.

The key insight that allows us to distinguish good circularities from bad is that the instantiation of a class `B` in the static initializer of `A` does not automatically create an initialization dependency `A` $\Rightarrow$ `B` ("$\Rightarrow$" = "depends on" = "should be initialized after"). The creation of a new instance of `B` by itself is not the source of any dependencies. The only reason `B` might have to be initialized first is if the constructor `B.<init>`, or some method called by it directly or indirectly, references a static field of `B`. We can discover such a dependency by static analysis of the call graph.

This introduces a rather radical twist to the initialization process: during class initialization, we might actually end up instantiating a class before it is initialized, provided its constructor (or any method called directly or indirectly by the constructor) does not reference any static fields of the class.

Another radical departure from conventional wisdom is that there is no inherent dependency of subclasses on superclasses. The JVM specification requires that superclasses be initialized before their subclasses, but there is really no reason for this unless the static initializer of the subclass references, directly or indirectly, a static field of the superclass. Our static analysis will discover all such potential references.

Our method flags the examples of Figs. 1 and 4 above as errors, but allows Fig. 2 to pass. Currently our implementation allows Fig. 3 to pass, but it could be extended without much difficulty to catch errors of this form as well.

We conjecture that circularities such as Figs. 1, 3, and 4 are rare, and that when they do occur, they are almost surely unintended. Moreover, we conjecture that in virtually all practical instances, any class initialization order respecting the static dependencies computed by our algorithm will give the same initial values as the standard lazy method.

We have tested our first conjecture experimentally by running our algorithm on several publicly available Java class libraries (see Section 3), including the entire COLT distribution from CERN [4] and a portion of the JDK version 1.4 from Sun [8]. In no case did it report a bad circularity. It is possible to concoct pathological examples for which our algorithm erroneously reports a bad circularity where in fact there is none, but these are so contrived that we suspect they would be unlikely to arise in practice.

2 Algorithm

In this section we describe the implementation of our algorithm for determining the class initialization order.

The order is defined in terms of a dependency relation $\Rightarrow$ on classes. Roughly speaking, `A` $\Rightarrow$ `B` if it can be determined by static analysis that the execution of the static initializer of `A` could potentially read or write a static field of `B`. Thus if `A` $\Rightarrow$ `B`, then `B` should be initialized before `A`. Note that this is independent of whether the initialization of `A` can create an instance of `B`.

We assume that all classes are locally available for static analysis and that all methods are available in bytecode form (i.e., no native methods). We distinguish between *system classes* (e.g., `java.util.Hashtable`) and *application classes*. Our algorithm does not analyze system classes, since no system class would normally know about application classes and thus would not reference their static fields. It can be proved formally that without explicit syntactic reference, system class initialization cannot directly or indirectly access any static field of an application class (this is false for general computation).

We describe $\Rightarrow$ as the transitive closure of the edge relation $\rightarrow$ of a particular directed graph whose vertices are classes and methods. The graph will be constructed dynamically. Let LC be the set of application classes, SC the set of system classes, and AM the set of static and instance methods of application classes. The relation $\rightarrow$ is defined to be the edge relation on $(LC \cup SC \cup AM) \times (LC \cup SC \cup AM)$ consisting of the following ordered pairs:

(i) If `A` is an application class and `A` has a class initializer `A.<clinit>`, then `A` $\rightarrow$ `A.<clinit>`.
(ii) If `f` is a static or instance method of an application class, and if `f` calls another static or instance method `g`, then `f` $\rightarrow$ `g`. Such a call must be of the form either `invokestatic g`, `invokespecial g`, `invokeinterface g`, or `invokevirtual g`. In addition, if `g` is an instance method invoked by `invokevirtual g`, and if `g'` is another method with the same name and descriptor in a subclass of the class in which `g` is defined, then `f` $\rightarrow$ `g'`.
(iii) If `f` is a static or instance method of an application class `A`, and if `f` contains an instruction `getstatic B.a` or `putstatic B.a`, which reads or writes the static field `B.a`, then `f` $\rightarrow$ `B`.

We are actually only interested in the restriction of $\Rightarrow$ to classes, since this will determine the class initialization order. Also, for efficiency, we do not construct the entire relation $\rightarrow$, but only the part reachable from the main class.

We start with an initial set of vertices consisting of

(a) all application classes accessible by some chain of references from the main class,
(b) all system classes accessible from the main class, and
(c) all `<clinit>` methods of the application classes in (a).

The classes in (a) are available from the constant pools of all loaded classes. Classes are loaded and prepared eagerly, and we assume that this has already been done. Any class whose name appears in the constant pool of any loaded class is also loaded. The initial set of edges is (i), the edges from the application classes to their own `<clinit>` methods.

We now describe the computation of the call graph. Initially, we push all `<clinit>` methods in (c) on a stack, then repeat the following until the stack is empty.

Pop the next method `f` off the stack. If we have already processed `f`, discard it and go on to the next. If we have not processed `f` yet, scan its code looking for all instructions that would cause an edge to be created. These can be instructions `getstatic B.a` or `putstatic B.a` that access a static field or a method invocation `invoke... g`. In the case of a `getstatic B.a` or `putstatic B.a` instruction, create a new edge `f` $\rightarrow$ `B` if it is not already present. In case of a method invocation `invoke... g`, create a new edge `f` $\rightarrow$ `g` and push `g` on the stack for subsequent processing. It may also be necessary to insert `g` as a new vertex in the graph if it does not already exist. In addition, if `g` is an instance method invoked by `invokevirtual g`, and if `g'` is another method with the same name and descriptor in a subclass of the class in which `g` is defined, create a new edge `f` → `g′` and push `g′` on the stack for subsequent processing. When done, mark `f` as processed.

The reason for the special treatment of instance method invocations `invoke-virtual g` is that `g` is not necessarily the method that is dispatched. It could be `g` or any method that shadows it, i.e., a method with the same name and descriptor as `g` declared in a subclass, depending on the runtime type of the object. In general we may not know the runtime type of the object at the time of initialization, so to be conservative, we insert edges to all such methods.

```
class B {
   static int g(A1 a) {
      return a.f();
   }
   static A3 a = new A3();
   static int b = g(a);
}
class A1 {
   static int e = 1;
   int f() {
      return e;
   }
}
class A2 extends A1 {
   static int e = 2;
   int f() {
      return e;
   }
}
class A3 extends A2 {}
```

Fig. 5.

Fig. 5 illustrates this situation. The correct initial value for `B.b` is 2, because `A2.f` is dispatched in the call to `a.f()` in `B.g`, not `A1.f`. We must insert the edge `B.g` → `A2.f` when we insert the edge `B.g` → `A1.f` to account for the dependency of `B.b` on `A2.e`.

A dependency `A` ⇒ `B` is thus induced by a chain of intermediate method calls starting with `A.<clinit>` and ending with a `getstatic` or `putstatic` instruction, and all computed dependencies `A` ⇒ `B` are of this form. The number of intermediate calls can be arbitrarily long. For example, if `A.<clinit>` calls `f`, which calls `g`, which accesses the field `B.a`, then this entails a dependency `A` ⇒ `B`. However, note that it does not necessarily entail a dependency `A` ⇒ `C`, where `C` is the class of `f` or `g`.

A `new B` bytecode instruction appearing in `A.<clinit>` or any method called directly or indirectly by `A.<clinit>` also does not by itself introduce a dependency A $\Rightarrow$ B. The purpose of the `new B` instruction is to tell the JVM to allocate and prepare a new instance of the class B, and no static fields are accessed in this process. However, a `new B` instruction would normally be followed by an explicit call to an initializer `B.<init>`, which can access static fields. But our algorithm will see the call to `B.<init>` and will push it on the stack for later processing.

Once the graph is created, we perform depth-first search and calculate the strongly connected components. This takes linear time [2]. Each component represents an equivalence class of methods and classes that are all reachable from each other under the dependence relation $\rightarrow$.

In our current implementation, a *bad component* is taken to be a strongly connected component containing at least two classes. If A and B are two classes contained in a bad component, then there must be a cycle

$$\mathtt{A} \rightarrow \mathtt{A.<clinit>} \rightarrow \mathtt{f}_2 \rightarrow \cdots \rightarrow \mathtt{f}_n \rightarrow \mathtt{B} \rightarrow \mathtt{B.<clinit>} \rightarrow \mathtt{g}_2 \rightarrow \cdots \rightarrow \mathtt{g}_m \rightarrow \mathtt{A},$$

indicating that the initialization of A directly or indirectly accesses a static field of B and vice versa. Such bad components are flagged as errors.

If there are no bad components, then the relation $\Rightarrow$ restricted to vertices $LC \cup SC$ is acyclic. In this case, any topological sort of the induced subgraph on $LC \cup SC$ can be used as a class initialization order. In our implementation, we just use the postorder number of the low vertex of each component computed during depth-first search.

In the absence of reported bad circularities, our eager initialization strategy and the standard lazy strategy should normally give the same initial values. This is because we conservatively trace all possible call chains, so if there is a true dependency of a static field `A.a` on another static field `B.b`, where A and B are distinct, both the lazy method and our method will see it and will initialize `B.b` first. Any call chain involving at least two distinct classes that would result in a field receiving its default value instead of its initial value in the lazy method will appear as a bad circularity in our method.

However, it would be difficult to formulate a complete set of conditions under which the eager and lazy strategies could be formally guaranteed to give the same initial values. One would have to rule out all possible ways in which a class initializer could directly or indirectly modify a static field of another class. Without this restriction, each class could identify itself in a common location as it is initialized, thereby recording the actual initialization order. Thus initial values would not be the same unless the initialization order were the same. To avoid this, one would have to rule out a variety of possible indirect channels: exceptions, concurrency, reflection, native methods, and file or console IO, for example.

3 Experimental Results

To provide evidence that bad circularities are rare in practice, we have analyzed several large publicly available Java class libraries in a variety of application areas. We found no bad circularities. Besides portions of the JDK version 1.4 [8], we have analyzed the complete distribution of each of the following libraries.

The COLT distribution from CERN [4] is an extensive toolkit for computational high energy physics. It provides packages for data analysis and display, linear algebra, matrix decomposition, statistical analysis, and Monte Carlo simulation, among others. The distribution consists of 836 class files.

GEO [6] is a class library and environment supporting the creation, manipulation, and display of 3D geometric objects. The distribution consists of 43 class files.

The ACME distribution [1] is a package with several general-purpose utilities, including extensions to the Java Windows Toolkit, PostScript-like graphics, a `printf()` facility, a cryptography package including implementations of DES, Blowfish, and secure hashing, an HTTP server, a multithreaded caching daemon, a netnews database backend, image processing software including PPM, JPEG, and GIF codecs and RGB image filters, and a simple chat system. The distribution consists of 180 class files.

4 Remarks and Conclusions

An interesting open problem is to formulate conditions under which, in the absence of reported bad circularities, the eager and lazy strategies would be guaranteed to give the same initial values. A formal statement and proof of this result might be based on a bytecode or source-level type system in the style of [5].

As illustrated in Fig. 4, the true dependency relation is between static fields, not classes. The relation $\Rightarrow$ between classes is only a coarse approximation. A finer-grained approximation $\Rightarrow$ between static fields could be computed and would give sharper results. But because the `<clinit>` methods are compiled to be executed atomically, we could not take advantage of this extra information without recompilation. Besides, our experimental results indicate that the class-level approximation is sufficient for all practical purposes.

As mentioned, for *class* initialization, there is no inherent dependency of subclasses on superclasses. Such a dependency exists only if a static field of a superclass is referenced directly or indirectly by the static initializer of the subclass. Static initializers are never invoked explicitly from bytecode, but only by the virtual machine. *Instance* initialization is another matter, however. The constructor `B.<init>` always contains an explicit call to the parent constructor `SuperclassOfB.<init>`. Thus if `B` is instantiated during the course of static initialization, our algorithm automatically traces the chain of calls to the parent constructors.

Acknowledgements

We thank Thomas J. Merritt, Greg Morrisett, and Andrew Myers for valuable ideas and discussions and Prashant Palsokar and Reba Schuller for their assistance with the implementation. This work was supported in part by the AFRL/Cornell Information Assurance Institute AFOSR grant F49620-00-1-0209, DARPA contracts DAAH01-01-C-R026 and DAAH01-02-C-R080, NSF grant CCR-0105586, and ONR Grant N00014-01-1-0968. The views and conclusions contained herein are those of the authors and do not necessarily represent the official policies or endorsements, either expressed or implied, of these organizations or the US Government.

References

1. ACME Java class library. `http://www.acme.com/java/software/Package-Acme.html`.
2. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.
3. Egon Börger and Wolfram Schulte. Initialization problems for Java. *Software Concepts and Tools*, 20(4), 1999.
4. COLT Java class library. `http://tilde-hoschek.home.cern.ch/~hoschek/colt/V1.0.1/doc/overview-summary.html`.
5. Stephen N. Freund and John C. Mitchell. A type system for object initialization in the Java bytecode language. *Trans. Programming Languages and Systems*, 21(6):1196–1250, 1999.
6. GEO Java class library. `http://www.kcnet.com/~ameech/geo/`.
7. IEEE Standard for Boot (Initialization Configuration) Firmware: Core Requirements and Practices, 1994. IEEE Standard 1275-1994.
8. Java development kit, version 1.4. `http://www.java.sun.com/`.
9. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.
10. Martijn Warnier. Specification and verification of sequential Java programs. Master's thesis, Utrecht University, January 2002.

Appendix

The following sample run shows the output obtained on the fragment of Fig. 5 in the text above, along with the main class

```
class Pathologies {
   static public void main(String args[]) {
      System.out.println(B.b);
   }
}
```

The depth-first search tree is shown. After each vertex are listed its preorder and postorder numbers and low vertex, followed by a list of edges. The *low vertex* is the eldest reachable ancestor and serves as a canonical representative of the strongly connected component [2]. The nontrivial components (i.e., those with more than one member) are listed, and those with at least two classes are flagged as bad. In this example there are no bad components.

```
pathologies Sun Jul 22 09:49:24 EDT 2001
Loading class info...
Local classes loaded: 5
System classes: 3
Main class: Pathologies
Resolving references...
0 error(s)
Constructing class hierarchy...
Calculating dependencies...
Component analysis:
A1, A1.<clinit>()V
A2, A2.<clinit>()V
B.<clinit>()V, B
3 nontrivial component(s)
0 bad component(s)

pathologies Sun Jul 22 09:49:24 EDT 2001
A3.<init>()V: PRE=0 POST=2 LOW=A3.<init>()V
  A2.<init>()V TREE
A1.<init>()V: PRE=2 POST=0 LOW=A1.<init>()V
B.<clinit>()V: PRE=3 POST=11 LOW=B.<clinit>()V
  B.g(LA1;)I TREE
  B TREE
  A3.<init>()V CROSS
A2.<init>()V: PRE=1 POST=1 LOW=A2.<init>()V
  A1.<init>()V TREE
A1.<clinit>()V: PRE=7 POST=3 LOW=A1
  A1 BACK
A2.<clinit>()V: PRE=10 POST=6 LOW=A2
  A2 BACK
B: PRE=11 POST=10 LOW=B.<clinit>()V
  B.<clinit>()V BACK
A1: PRE=6 POST=4 LOW=A1
  A1.<clinit>()V TREE
A2.f()I: PRE=8 POST=8 LOW=A2.f()I
  A2 TREE
Pathologies: PRE=12 POST=12 LOW=Pathologies
A2: PRE=9 POST=7 LOW=A2
  A2.<clinit>()V TREE
A1.f()I: PRE=5 POST=5 LOW=A1.f()I
  A1 TREE
A3: PRE=13 POST=13 LOW=A3
B.g(LA1;)I: PRE=4 POST=9 LOW=B.g(LA1;)I
  A1.f()I TREE
  A2.f()I TREE
```