

Implementing a Java™ Virtual Machine in the Java Programming Language

Antero Taivalsaari

SMLI TR-98-64

March 1998

Abstract:

JavalnJava is a Java virtual machine written in the Java™ programming language. The system was built at Sun Microsystems Laboratories in order to examine the feasibility of constructing high-quality virtual machines using the Java programming language and to experiment with new virtual machine implementation techniques. In this paper we describe the overall architecture of *JavalnJava* and summarize a number of interesting technical issues that were encountered during its implementation.



M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
antero.taivalsaari@eng.sun.com

© Copyright 1998 Sun Microsystems, Inc. The SML Technical Report Series is published by Sun Microsystems Laboratories, a division of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS

Sun, Sun Microsystems, the Sun logo, Java and Write Once, Run Anywhere are trademarks or registered trademarks of Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. All SPARC trademarks, including the SCD Compliant Logo, are trademarks or registered trademarks of SPARC International, Inc. SPARCstation, SPARCserver, SPARCengine, SPARCworks, and SPARCcompiler are licensed exclusively to Sun Microsystems, Inc. All other product names mentioned herein are the trademarks of their respective owners.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.

Implementing a Java™ Virtual Machine in the Java Programming Language

Antero Taivalsaari

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303
U.S.A

1. Introduction

The *JavaInJava* project was started in mid-October 1997 to study the feasibility of building high-quality virtual machines in the Java programming language. The main goal of the project was to investigate the suitability and scalability of the Java programming language for virtual machine implementation. More specifically, we wanted to understand the possible difficulties in constructing such a virtual machine compared to building virtual machines in C or C++.

In addition to studying virtual machine implementation in the Java programming language, we wanted to build a clean, extensible platform for experimenting with different virtual machine implementation techniques and for building new kinds of debugging and visualization tools for the Java language. Another goal was also to write a clean, “reference” Java™ virtual machine (JVM) implementation that would be easier to understand than virtual machines written in C or C++. At the same time, we wanted to keep the back-end of the system general so that it could potentially be used for implementing back-end support for other programming languages as well.

The implementation of JavaInJava was done in a clean-room fashion without borrowing code or implementation techniques from existing virtual machine implementations for the Java programming language. In order to facilitate the understanding of the system, we tried to document most of the design decisions also at the code level. As a result, the comment/code ratio in JavaInJava source code is high (about 40-50% of the code are comments). Generally, since we expected the system to be substantially slower than normal JVM implementations, we were more concerned with clarity and style than performance.

The implementation of the basic JavaInJava system was completed in mid-December 1997. By this time, most of the design goals had been met, and the system could run simple benchmarks on different platforms. As will be explained later, the implementation of the native function interface turned out to be more challenging than originally anticipated, and this part is still incomplete as of this writing. Consequently, JavaInJava cannot run programs relying on the Abstract Window Toolkit (AWT) or other graphical libraries for the Java programming language.

The JavaInJava system currently consists of 42 classes and approximately 10,000 lines of source code of which about 40-50% are comments. The executable size is about 100 kB (or 130 kB if debugging information such as line numbers and local variable information is included in the compiled classfiles). The system is very slow: the JavaInJava virtual machine running on a standard Java virtual machine executes Java programs roughly three orders of magnitude more slowly than the standard Java virtual machine does alone. For instance, the DeltaBlue benchmark [FMB90] executes approximately 732 times slower when run with the JavaInJava VM than when running the equivalent benchmark on Sun's Java virtual machine without JavaInJava, and in some other benchmarks the ratio is even worse. Also, we have measured that on average the underlying JVM has to execute about 540 Java bytecodes per each executed JavaInJava bytecode. However, quite a few of these bytecodes are used only for JavaInJava system profiling, and the performance of JavaInJava could be improved by removing these extra profiling instructions.

At the moment, the JavaInJava system is available only internally at Sun Microsystems, Inc. No decisions about possible wider distribution have been made yet.

The rest of the paper is organized as follows. We start by describing the overall design of JavaInJava (Section 2), explaining its general architecture and the runtime representation of its internal structures including classes, objects, stack frames and threads. In Section 3, we present some interesting implementation issues that were encountered during the design and implementation of JavaInJava. Section 4 compares our JavaInJava experiences with our previous VM implementation experiences in C/C++ and presents some general comments on the suitability of the current Java™ Virtual Machine Specification for cleanroom JVM implementation. Finally, Section 5 provides a summary of the results of the paper.

2. JavaInJava design

Virtual machines are usually composed of a number of tightly interacting, performance-critical subsystems such as the memory manager, thread scheduler, stack frame manager, interpreter, JIT (Just-In-Time) compiler, and so on. When implementing a virtual machine in C or C++, these subsystems are usually coupled with each other closely by using global variables or machine registers to store the virtual machine registers (such as the instruction pointer and the stack pointers) and to share data efficiently between the different subsystems. Also, a C++ programmer can easily avoid using the object-oriented

features of the language and access the data of different subsystems more directly and rapidly. Unfortunately, this makes the different subsystems highly dependent on each other, meaning that changes in one part of the system can cause substantial changes in the other parts. The close coupling of subsystems lowers the level of abstraction, thus making the virtual machine harder to understand and maintain.

In general, even though it is possible to build well-designed, clearly structured software in C and C++, in virtual machine development performance considerations are usually so central that they take priority over good design and architecture. We did not want this to happen with JavaInJava, so careful thought was given to a clean overall system architecture.

2.1. The overall architecture of JavaInJava

At the very beginning of the JavaInJava project we noticed that some features of the Java programming language impose interesting constraints on how the overall architecture of the JavaInJava virtual machine could be defined. For instance, one of the central features of the Java programming language is that all data has to be defined in classes and that there are no global variables. Thus, the Java language requires the designer to structure his programs in an object-oriented fashion, whereas in C or C++ it is very easy to bypass all design paradigms and abstractions in the name of efficiency.

Further, the fact that the Java programming language does not support data pointers, function pointers, or embedded assembly code obviously prevents the VM designer from using some programming tricks that C and C++ programmers have typically utilized for improving the performance of their virtual machines. Consequently, the overall architecture and design of a virtual machine written in the Java programming language is inherently rather different from a virtual machine built in C or C++.

One of the goals in designing JavaInJava was to keep the low-level implementation structures such as the thread scheduler and stack frame management open so that the virtual machine could potentially be generalized later for other languages as well. Therefore, we wanted to have a distinct interface through which the Java bytecode interpreter and a possible future compiler would access the internal structures, rather than exposing all the details and allowing the interpreter to access and manipulate the data directly.

Architecturally the most important class in JavaInJava is *VM.class* (Figure 1). *VM.class* is a *façade class* (see *Façade* design pattern description in [GHJ95]) that serves as a common interface to the most important internal runtime structures of the JavaInJava virtual machine, hiding the implementation details from the other parts of the system. Class VM provides operations for thread management, stack frame management and exception handling, and implements methods for manipulating local variables and the data stack. In addition, class VM has a method 'Initialize()' that allows all the necessary

internal runtime structures of the virtual machine to be initialized with just one method call.

Organizing a virtual machine around the VM *façade* class turned out to be a nice approach. It reduced the dependencies between different subsystems substantially. For instance, the interpreter has consistent access to all the internal structures of the virtual machine through a single class interface, and thus the interpreter does not have to know how threads or stack frames have been implemented and how their data is represented. In general, all the state information of the virtual machine is hidden under VM.class. However, using the *façade* pattern obviously imposes a performance overhead, since the internal data of the virtual machine is accessed by one or more levels of method calls rather than by referencing the data directly.

Note that at least some of the overhead imposed by the use of the *façade* pattern could have been avoided by using the *import* mechanism of the Java programming language to introduce the interfaces of the internal virtual machine classes directly to the interpreter. However, we decided not to utilize this mechanism, since that would have exposed the internal structures of the virtual machine to the interpreter almost as badly as global variables. For the same reason, we also avoided the use of public data fields, and used accessor methods to wrap access to data instead.

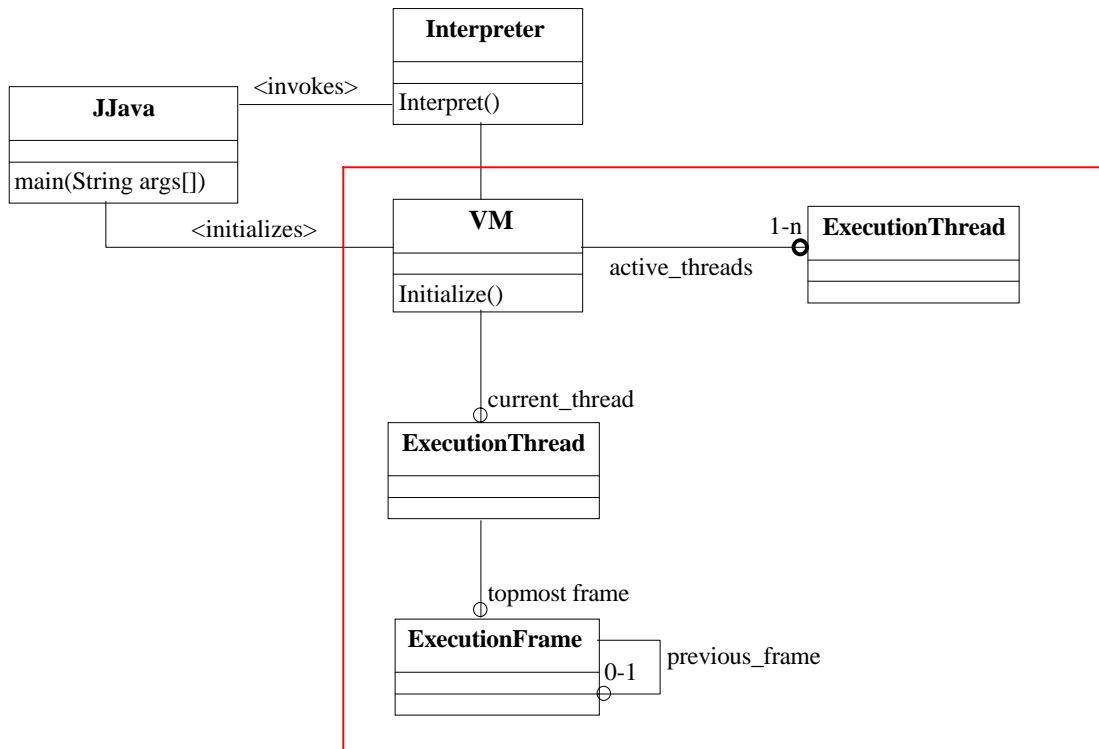


Figure 1: The overall architecture of JavaInJava

An interesting fact about JavaInJava is that there are no traditional virtual machine registers such as the instruction/bytecode pointer, frame pointer, or operand stack pointer.

Rather, all the VM state information is stored inside classes. For instance, in JavaInJava every stack frame object stores information about the current bytecode location within the method for which the stack frame was created, and no global bytecode pointer is used. Similarly, every operand stack object has its own stack pointer, which is part of the stack abstraction, and the data in the operand stack can be accessed only through the interface of the stack object.

The decision to encapsulate all state information inside the corresponding abstractions was partly a deliberate design decision and partly something imposed by the Java programming language. For instance, due to the lack of global variables, it would not have been possible to define global virtual machine registers in the traditional fashion. And due to the lack of pointers it would not have been possible to encode virtual machine registers simply as raw pointers to code and data, or to map the virtual machine registers to physical machine registers using embedded assembly code or other machine-dependent optimizations. Such solutions are common in virtual machines written in C or C++.

The Java programming language does not allow non-portable, platform-specific optimizations. In general, even though we had intentionally planned to make the overall architecture of the JavaInJava object-oriented and had deliberately avoided introducing close couplings and dependencies between the subsystems, it was partly the Java programming language that guided us to make many of the architectural decisions in that way.

2.2. Runtime class representation

A unique feature of the Java programming language is the way compiled Java classes are stored in *classfiles* [JVM96 pp. 83-137]. A Java classfile is a compact, highly portable, easily transferable structure that stores all the relevant information of a Java class and its internals in a symbolic, machine-independent, extensible fashion. A Java™ virtual machine must be able to load and unload classfiles dynamically at runtime, and thus the size and the functionality of a running Java program may actually vary during execution.

When Java classfiles are loaded into a Java virtual machine, the virtual machine creates the corresponding internal runtime structures to represent classes in a more efficient, machine-dependent way to enable rapid access to class information. Obviously, the actual internal runtime representation of classes may vary substantially from one Java virtual machine implementation to another.

Two main goals motivated the design of the structures for storing runtime classes in JavaInJava. The first goal was to make the internal representation of classes object-oriented and modular so that the structures could potentially be used easily for implementing visualization and reverse engineering tools that extract information from Java classfiles. The second goal was to design the runtime structures so that they would

reflect the original classfile structure as closely as reasonable. Obviously, these goals are conflicting since the Java classfile structure is not very object-oriented.

The classes implementing runtime class representation support for JavaInJava are depicted in Figure 2. Class *JavaClassLoader* implements the necessary methods for reading Java classfiles and for generating corresponding *JavaClass* instances. Every *JavaClass* instance is composed of *ConstantPool*, *MethodTable*, *FieldTable* and *InterfaceTable* instances. The instance of class *ConstantPool* maintains a Java constant pool structure for storing the symbolic references inside a Java class. Physically, the *ConstantPool* instance is composed of an array of *ConstantPoolEntries*. Class *ConstantPoolEntry* is an abstract class that has a number of concrete subclasses for implementing the different constant pool entry structures according to the JVM Specification [JVM96 pp. 92-101].

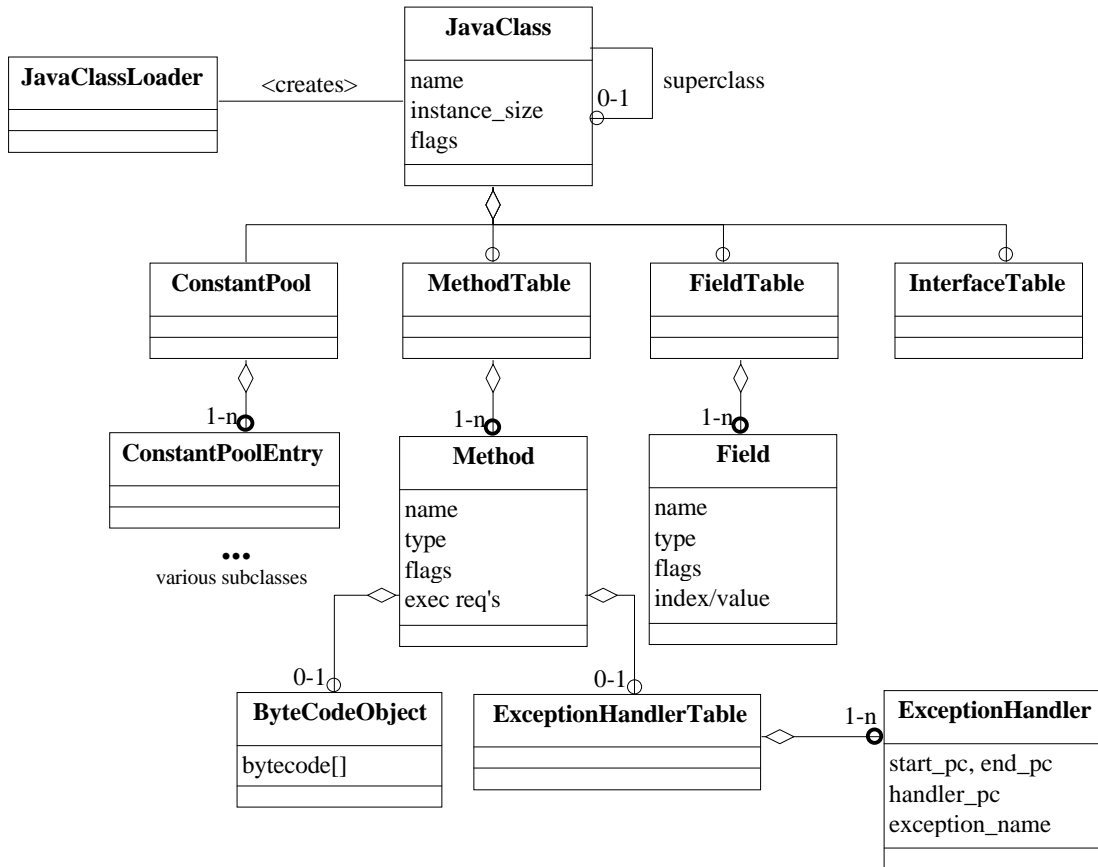


Figure 2: Internal class representation classes in JavaInJava

The *MethodTable* and *FieldTable* instances contain hashtables of *Method* and *Field* objects, respectively. These hashtables are used for rapidly finding the various methods and fields of the class at runtime. Both *Method* and *Field* objects store the name and descriptor (the Java signature) of the method, as well as a reference to an *AccessFlags* object, which stores the relevant access flags such as ‘public’, ‘protected’, ‘private’, ‘static’, ‘native’, ‘synchronized’, and so on (class *AccessFlags* is not shown in Figure 2).

The Method instances stored in MethodTable contain additional information for storing the execution requirements of each method, such as the required amount of operand stack and local variable space, and possibly a pointer to a ByteCodeObject (if the method is not native or abstract). Furthermore, each Method instance may refer to an ExceptionHandlerTable object if the method has associated exception handlers.

The InterfaceTable instance is simply a vector of strings storing the names of the interface classes that the class must implement. The interface classes are loaded into the virtual machine only if the internals of the interface classes are actually needed at runtime.

The current implementation of the JavaInJava class loader has some limitations. For instance, currently the class loader ignores all the optional classfile attributes such as line number and local variable information. Also, currently JavaInJava does not have any classfile verifier; in other words, the system simply assumes that classfiles given to JavaInJava are always valid.

2.3. Runtime representation of objects

In virtual machines there are typically two kinds of objects: those that can be manipulated at the level of the programming language which the virtual machine implements and those that are used only internally by the virtual machine. The internal object structure of JavaInJava reflects this division, and there are two main superclasses: *JavaObject* and *InternalObject*. Class *JavaObject* serves as a common superclass for all the structures that can be treated as full-fledged Java objects at runtime. Class *InternalObject* and its subclasses represent internal virtual machine structures, such as method tables, field tables, exception handlers and so on (see Figure 3). Both *JavaObject* and *InternalObject* subclasses of 'java/lang/Object'.

Class *JavaObject* has two main subclasses: *JavaInstance* and *JavaArray*. Obviously, instances of class *JavaInstance* and *JavaArray* represent runtime Java object and array instances in JavaInJava. However, the representation of the instances of these classes is somewhat unconventional due to some features of the Java programming language. In particular, since the Java language does not allow the programmer to create objects that could arbitrarily mix both primitive and non-primitive data, it is not possible to store the fields (instance variables) of Java objects directly in *JavaInstance* objects. Rather, a separate array object is used for storing the fields. The same solution is used for *JavaArray* instances, which are also composed of two parts. The *JavaArray* object itself only stores the necessary administrative information, such as the class pointer and the possible monitor reference, whereas the actual data of the array is stored in a separate Java array object. The implementation of arrays is discussed in more detail in Section 3.3.

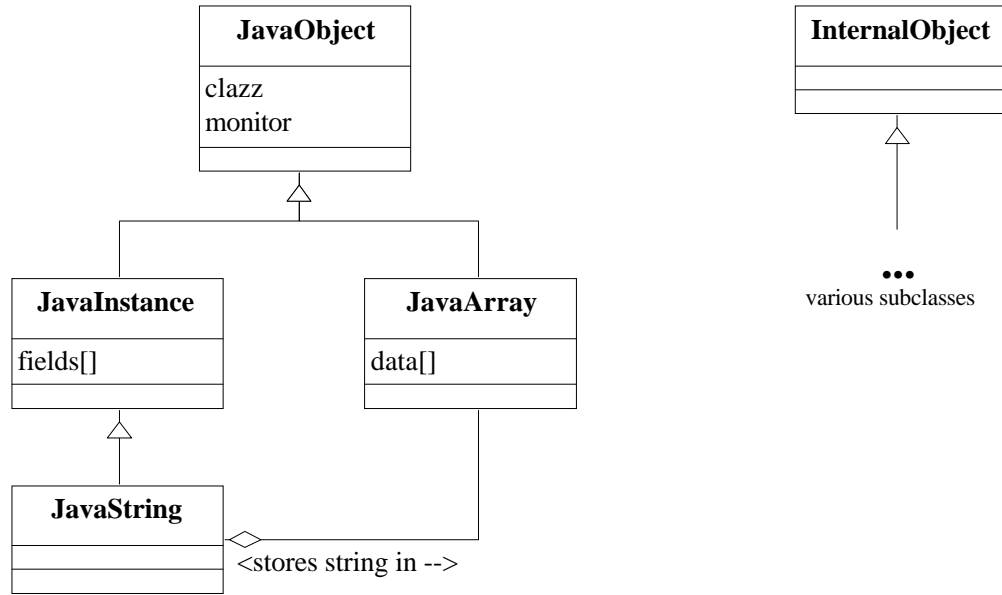


Figure 3: The fundamental object representation classes in JavaInJava

In general, to overcome the fact that the primitive data types (boolean, char, int, long, ...) of the Java programming language are not first-class objects and that the programmer cannot freely typecast between primitive and non-primitive types as in C or C++, we decided to avoid the use of primitive types in JavaInJava altogether. Thus, in JavaInJava primitive values such integers or floats are represented as boxed instances of the corresponding Java classes. For instance, a primitive value integer 3 is physically represented as an 'Integer(3)' object, a floating point number 3.0 as 'Float(3.0)' and so on. The boxing of primitive values turned out to work generally very well, but it has some interesting implications on the other parts of the virtual machine. These implications will be discussed in more detail later.

Except for the primitive types and arrays, the other Java objects in JavaInJava are represented in the expected manner. For instance, Java string objects in JavaInJava are stored simply as JavaInstances whose field structure corresponds to the structure defined by the standard Java class 'java/lang/String'. In practice this means that the first instance field of the string object holds a pointer to a JavaInJava character array object storing the string, while the second and third instance fields are reserved for storing the starting offset and the length of the string. In order to facilitate debugging, a special subclass of JavaInstance called *JavaString* was created (Figure 3); class *JavaString* adds some additional string printing and debugging methods but otherwise its structure and functionality are identical with class *JavaInstance*.

2.4. Stack frame management

An essential feature of any virtual machine is the ability to efficiently manage stack frames, also known as method activation records or function call frames. As the commonly used term “stack frame” implies, these structures are usually allocated from a stack and are often stored contiguously to make the allocation and deallocation of frames as fast as possible. This strategy is also commonly used in Java virtual machines.

A characteristic feature of many Java virtual machine implementations is that stack frames contain not only the local variables and other method execution and synchronization information, but also the space needed for storing the intermediate results of computation (the “operand stack”). The resulting structure is often colloquially referred to as the “Java stack”.

When designing JavaInJava, we decided to make the various distinct elements stored in the Java stack more explicit, and created a separate class to represent each logically distinct concept. Furthermore, to facilitate understanding and debugging of the virtual machine, we decided to allocate frame objects from the heap rather than from the stack, which is the conventional technique. Even though heap-based allocation adds some performance overhead to the virtual machine (in terms of both time and space), we felt that the overhead could be justified with the increased clarity in representing the fundamental VM concepts.

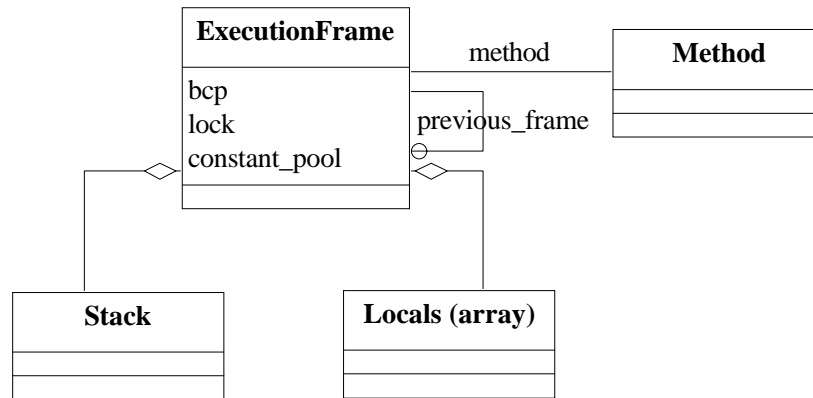


Figure 4: Execution frame representation in JavaInJava

The stack frame architecture of the JavaInJava virtual machine is illustrated in Figure 4 above. As shown in the figure, every stack frame is physically composed of three objects: 1) the actual frame object (class *ExecutionFrame*), 2) the local variable array, and 3) the operand stack. The frame part contains the pointer to the method being executed, the bytecode pointer (offset), the pointer to the previous frame, and optional locking information for synchronized methods. The local variable array object stores the parameters and local variables. The operand stack object serves as the working data area for the method during its execution; this means that Java bytecodes such as POP, DUP, IADD or LCMP always operate in the context of the operand stack of the current frame of the current thread. As explained earlier, the JavaInJava bytecode interpreter is not

allowed to access these structures directly; rather, the methods of class VM are used for manipulating the structures. This approach ensures that the internal execution structures can be changed without affecting the behavior of the interpreter.

Again, since the Java programming language does not support pointers in the traditional sense, it should be noted that the bytecode pointer is actually an offset into the method's bytecode array rather than an actual pointer.

2.5. Multithreading

Any full Java™ virtual machine implementation has to support *multithreading*, that is, be capable of running multiple threads of program control simultaneously. The Java Virtual Machine Specification [JVM96 pp. 371-388] only specifies general rules for multithreading; it does not specify in detail how multithreading is actually to be implemented. For instance, in some virtual machines multithreading may utilize the multiprocessing capabilities of the underlying computer and operating system architecture, whereas in other implementations multithreading may be done purely in software.

Since we wanted the JavaInJava virtual machine to be independent of any particular computing platform, multithreading in JavaInJava is implemented entirely in the Java programming language without support from the native computing environment. Also, we decided not to utilize the multithreading capabilities of the underlying Java virtual machine; rather, the necessary multithreading facilities have been implemented at the JavaInJava level.

In short, JavaInJava features fully portable, preemptive, machine- and language-independent multithreading. Thread scheduling can be either time-sliced or instruction-sliced. In the former approach a thread switch is enforced after the currently executing thread has received a certain amount of wall-clock time multiplied by its Java language-level priority. In the latter approach (the default) each thread may execute a certain number of bytecodes (multiplied by its language-level priority) until a thread switch is enforced.

In order to keep the multithreading implementation independent of the details of the Java™ programming language, the basic multithreading classes in JavaInJava are subclasses of class `InternalObject` rather than `JavaObject`; that is, they can be manipulated only internally by the virtual machine. To provide compatibility with the Java language level thread classes (such as `'java/lang/Thread'` and `'java/lang/ThreadGroup'`), the system automatically maps each `ExecutionThread` instance into the corresponding `'java/lang/Thread'` instance. For example, when the user creates a new `'/java/lang/Thread'` object, that is, creates a new Java thread, an internal `ExecutionThread` object is created transparently. When the user modifies certain attributes of the Java-level thread, such as its priority, the JavaInJava VM intercepts the native method calls and also performs the changes in the underlying `ExecutionThread` object. In general, the virtual

machine is normally interested only in the internal thread objects, and accesses the ‘java/lang/Thread’ objects only when absolutely necessary.

Figure 5 depicts the basic multithreading classes of JavaInJava. As explained earlier, all the essential virtual machine functions can be controlled via the façade class VM, which also provides operations for controlling multithreading. Class VM has a field called ‘active_threads’, which maintains a queue of all the active ExecutionThread instances in the system. Another field, ‘current_thread’, maintains a reference to the ExecutionThread that is currently receiving processor time. Every ExecutionThread has a corresponding Java-level ‘java/lang/Thread’ object and a pointer to the topmost ExecutionFrame of the thread. Note that since JavaInJava has no global VM registers and all the state information is encapsulated inside classes, thread switching in JavaInJava is trivial: it consists simply of changing the ‘current_thread’ field to point to another ExecutionThread object.

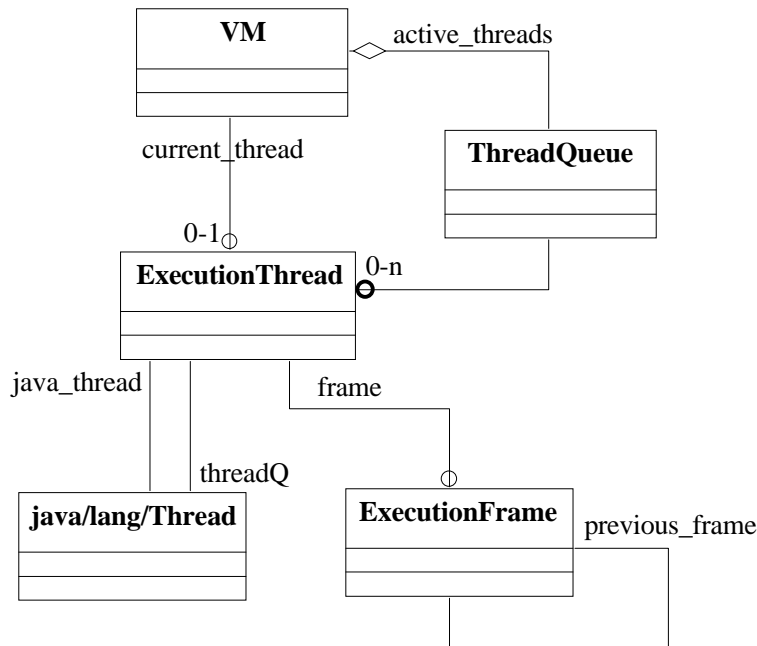


Figure 5: Multithreading classes

3. Implementation issues

During the implementation of JavaInJava, several interesting technical issues were encountered. Many of these issues arose from the lack of certain language mechanisms in the Java™ programming language. Other issues emerged because the Java Virtual Machine Specification did not provide enough information about certain virtual machine implementation details.

3.1. Problems with double-length primitive types

In order to guarantee platform independence and maximum portability of the Java programming language, the Java Language Specification and Java Virtual Machine Specification define the standard Java data types very carefully. For instance, the integral primitive types *byte*, *short*, *int*, and *long* must be represented as 8-bit, 16-bit, 32-bit and 64-bit signed two's-complement integers, respectively, and character type *char* is always a 16-bit unsigned integer representing Unicode characters [UNI92]. Similarly, floating-point types *float* and *double* are represented according to the IEEE 754 standard [IEE85].

When implementing a Java virtual machine in the Java programming language, data type compliance with the Java Language Specification is seemingly trivial to achieve, since the implementation can rely on the data types of the underlying Java virtual machine. However, in JavaInJava this turned out to be a false assumption due to the fact that primitive types had to be represented as objects instead of the standard primitive Java types (Section 2.3). The representation and manipulation of double-word (64-bit) data types such as *double* and *long* in particular turned out to be more challenging than originally anticipated.

In normal Java virtual machine implementations, double and long values are assumed to take two words (64 bits) in stack frames, the operand stack, arrays and the data fields of objects. However, in JavaInJava these data types are represented as explicit objects and are referred to by 32-bit object references, so the routines for reading and storing longs and doubles must operate accordingly. Two alternative approaches are possible. The first approach is to keep the space allocation for long and double values as it is in standard Java virtual machines, and add an extra 32-bit null value as padding to every stack frame, operand stack, data field and array location where a long or double value is needed. The second approach is store long and double values simply as 32-bit references and to alter the virtual machine to operate correctly with 32-bit long and double references.

Both approaches have their benefits and drawbacks. The first solution is more compliant with the original Java Virtual Machine Specification, since it maintains the original stack allocation, local variable indexing and other memory requirements. However, having to add the extra padding does not seem conceptually elegant. The second approach was chosen for JavaInJava. It simplifies Java bytecodes a lot: by eliminating the need for 64-bit primitive data allocation in stack frames, operand stack and data fields, the virtual machine can operate consistently with 32-bit data words only. As a result, many standard load, store, return and array manipulation bytecodes become identical, and a large number of bytecodes could be eliminated from the Java bytecode instruction set. This simplicity does not come without a cost, however. For instance, parameter passing from operand stack to execution frames becomes cumbersome, since the local variable offsets in compiled Java methods assume long and double numbers for take two 32-bit words in the stack frame. These offsets have to be recalculated either at class loading time or at runtime, or otherwise parameter passing fails when long or double parameters are used. Also, double-length stack manipulation bytecodes (such as *pop2*, *dup2*, *dup2_x1* and

dup2_x2) have to be augmented with runtime checks, or these bytecodes may pop or duplicate the wrong operand stack items.

3.2. Problems with object initialization

Making primitive types first-class objects inside a Java virtual machine has some interesting consequences for object initialization. According to the Java Language Specification [JLS96], all the fields of an instance have to be initialized to default values when a new object is instantiated. For instance, by default every integer field has to be initialized to 0, every float field to 0.0, every object reference to *null*, and so on.

Typically Java compilers assume that they don't have to do anything special to initialize primitive fields; simply storing a zero/null value into every new field is enough to initialize fields correctly regardless of the type of the field. However, this assumption fails as soon as primitive types are no longer represented as embedded values and the initial values of primitive types differ from each other.

In JavaInJava the correct initial value of a primitive field is not zero/null, but varies according to the type of the structure. For instance, integer fields should be initialized to `Integer(0)`, float fields to `Float(0.0)`, long fields to `Long(0)` and so on. Correspondingly, when the programmer instantiates a new primitive array (e.g., `int x[] = new int[1000];`), all the data in the array should be initialized to the correct default value. Obviously, when instantiating large arrays the initialization could become rather slow, since every single data slot of the array would have to be explicitly set to refer to the object containing the default value.

In order to overcome the problems of object initialization and to keep array initialization more efficient, we decided to avoid creation time initialization and introduced *read barriers* to all the fundamental field and array access operations. In other words, all the basic field and array read primitives explicitly check whether the data in the current location is still uninitialized. If so, the current value of the location is replaced with the correct default value based on the type of the structure. This solution works fine, but obviously adds performance overhead to the execution of the virtual machine.

3.3. Problems in array implementation

Implementation of Java arrays in JavaInJava turned out to be more challenging than expected. The original design idea was simply to map JavaInJava array objects directly to the underlying Java VM, and use the underlying arrays directly. Unfortunately, the type system of the Java programming language, when combined with the design solutions such as boxed primitives discussed earlier in this paper, is not quite flexible enough to allow this. Most of the problems arise from the fact that in the Java programming language there is a clear distinction between primitive and non-primitive types, and thus it is impossible, for example, to typecast between object arrays and primitive arrays.

Thus, we could not use the same source code for manipulating primitive and non-primitive arrays as we had originally planned.

After spending quite a bit of time thinking about possible solutions, we decided to make array objects similar to regular object instances in JavaInJava. Like JavaInstance objects, each JavaArray object is composed of two parts. The body of the JavaArray object stores the class pointer and the possible monitor reference as in other objects. In addition, there is a reference to a separate Java object array (Object[]) for storing the actual data in the array. Primitive values are stored as objects; therefore, all array data slots are 32 bits wide and the same source code can be used for implementing the various type-specific array manipulation bytecodes. For instance, in JavaInJava all the array load bytecodes (IALOAD, LALOAD, FALOAD, DALOAD, AALOAD, BALOAD, CALOAD and SALOAD) share the same source code.

3.4. Native function interface

A fully functional Java virtual machine is typically composed of three physically distinct components: 1) the actual virtual machine program, 2) the standard Java libraries (java.lang.*, java.io.*, ...) and 3) the shared native libraries implementing the platform-specific native functions needed by the standard Java libraries. It is important to notice that a substantial portion of the behavior of a Java virtual machine is located in the libraries rather than in the actual virtual machine program and that a typical Java virtual machine is heavily dependent on these components. The implementation of the native functions tends to be especially tricky, because the Java Virtual Machine Specification assumes that these functions are platform-dependent, and thus they are not documented in detail anywhere.

When implementing a Java virtual machine in the Java programming language, intuitively one might think that the implementation of the native function interface is almost trivial. After all, the underlying Java virtual machine used by the JavaInJava VM must provide a full implementation of all the necessary library functions, and thus one could easily assume that JavaInJava could simply pass all the native function calls on to the underlying virtual machine and let the underlying VM do the actual work. Unfortunately, this turns out to be an incorrect assumption. For instance, since JavaInJava implements its own multithreading scheme, the native thread manipulation primitives such as 'start()', 'yield()' and 'setPriority0()' cannot simply be passed along to the underlying VM. Rather, they have to be intercepted and carried out by JavaInJava. Similarly, most I/O initialization primitives and reflection primitives have to be handled independently of the underlying JVM.

When passing parameters from JavaInJava to the native functions of the underlying JVM, a lot of care is needed in order to ensure that the relationships between the runtime structures of JavaInJava and the underlying JVM are maintained properly. Even though the Java Reflection API provides some help for this, the maintenance of the relationships is still tricky and error-prone at best. The problems are exacerbated by the fact that there

does not exist any detailed documentation on what native functions a JVM is expected to implement and what the native functions are actually expected to do. Many additional problems are also caused by incompatibilities between the different versions of the Java programming language (not to mention the differences in the products of different vendors).

JavaInJava was designed to run with standard JDK 1.1 libraries; that is, no modifications to the standard JDK 1.1 library classes are required. As of this writing, the implementation of the native function interface is still incomplete because of the above mentioned problems. But the system is complete enough to allow most of the standard non-graphical benchmark programs to run. In addition, the system is probably the first Java virtual machine that can run *itself*; that is, it is possible to run multiple levels of JavaInJava recursively simply by giving the main JavaInJava class `JJava` as a parameter to another JavaInJava instance.

With the introduction of the Java™ Native Interface (JNI) API and JDK 1.2, the implementation of the native function libraries is expected to be substantially easier than in previous versions of the Java programming language. JNI does not help JavaInJava development, though, since currently JNI API's are not available for the Java language; after all, how could the developers of JNI have expected that somebody would want to implement the *native* functions for a Java virtual machine in the Java language? Thus, the development of the native function interface for JavaInJava will probably have to be continued in the traditional fashion.

3.5. Additional comments and observations

In addition to the issues discussed above, many other interesting problems and possible future research areas were encountered while implementing JavaInJava. In this subsection we briefly summarize some of these; detailed discussion is beyond the scope of this paper.

Exception handling. Implementing a Java virtual machine in the Java programming language provides various alternative solutions for implementing exception handling. Since the JavaInJava VM has its own execution thread and stack frame management classes, the straightforward solution would be to implement the exception handling capabilities simply as part of these fundamental JavaInJava classes without utilizing the exception handling capabilities of the Java programming language or the underlying Java virtual machine. However, a considerable amount of work could be saved by taking advantage of the exception handling capabilities of the underlying JVM. For instance, it would be quite possible to encapsulate the whole JavaInJava interpreter inside a *try* block and to simply rely on the ability of the underlying JVM to catch all the exceptions. The *catch* part of the try block would then contain the necessary code for mapping the caught exceptions with JavaInJava-level exception handlers and for setting the JavaInJava interpreter to execute the handler routines.

By utilizing the exception handling capabilities of the underlying JVM, it would be possible to eliminate all the explicit error checks from the JavaInJava virtual machine. For instance, in theory no explicit array range checks or null pointer checks would be needed in the JavaInJava codebase, because the JavaInJava system is assured that these errors are caught automatically by the underlying Java VM. However, in practice this would not be as simple as it sounds, since exceptions in a Java virtual machine may occur for different reasons. Some of the exceptions indicate perfectly normal and frequently occurring conditions, such as, for example, reaching the end of file while reading it, whereas other exceptions indicate errors or other unexpected occurrences in the program that is being executed. Still, other exceptions might be thrown because of errors in the JavaInJava virtual machine itself. The catch block for the JavaInJava interpreter would have to be intelligent enough to analyze each exception and to behave differently depending on the source of the exception. In practice this would be very tricky and error-prone.

When designing JavaInJava, we decided to take advantage of some of the exception handling capabilities of the Java programming language, while keeping the implementation faithful to the Java™ Virtual Machine Specification. For instance, the JavaInJava interpreter indeed *is* encapsulated inside a try block. However, since we wanted to build a virtual machine that could potentially be used as a reference implementation, we did not cheat by omitting the array range or null pointer checks from the codebase. Also, we explicitly check for possible problems that might indicate bugs in the JavaInJava system itself, and thus the catch block of the interpreter can assume that all the exceptions are program-induced and not caused by bugs in the virtual machine itself.

Memory management. An essential feature of a virtual machine for any dynamic programming language is the ability to manage memory efficiently. In JavaInJava, memory management currently relies entirely on the underlying Java virtual machine; that is, no memory management routines have been implemented at the JavaInJava level. Since the object structure of JavaInJava is very fine-grained and all the objects including stack frames are allocated from the heap, the system generates a few megabytes of garbage every second it runs. This makes JavaInJava an interesting non-I/O bound benchmark for measuring the performance of the underlying JVM and its garbage collector. Indeed, some people inside Sun have expressed interest in using JavaInJava as an official benchmark program for Java virtual machines.

Note that in principle there is nothing that prevents a virtual machine designer from implementing the memory management facilities of a Java virtual machine in the Java language. However, since the Java programming language does not support pointers and raw memory access, the overall architecture of a memory manager/garbage collector would be very different from implementations written in C or C++. Instead of accessing memory directly, the memory manager would have to be based on objects and collections written in the Java programming language. This would probably result in a very clean but rather inefficient memory manager.

An entirely different approach for implementing memory management for Java in Java would be to utilize a generative approach for building the whole virtual machine. In other words, rather than writing the virtual machine itself in the Java language, a Java-based virtual machine *generator* could be written that would allow the programmer to generate an optimum, platform-specific Java virtual machine. A related approach is currently used in some state-of-the-art Java virtual machine implementations written in C++, such as the new *HotSpot* virtual machine from JavaSoft. JavaInJava performance issues are discussed in more detail in Section 4.3.

4. Discussion

4.1. The Java programming language vs. C/C++ in virtual machine implementation

As far as we know, JavaInJava is one of the first virtual machines written in the Java programming language, and probably the first Java virtual machine written in the Java language. In contrast, there are thousands of virtual machines for various languages written in C and C++, including a few built by the author of this paper [Tai93, Tai96]. Since the Java programming language is becoming popular and has recently received a lot of attention and feedback from C and C++ programmers, it is interesting to analyze how well the Java programming language fares when compared to virtual machine implementations written in C and C++.

Basically, the JavaInJava project has shown us that building a clean, well-structured Java virtual machine is easier in the Java language than in C++. Especially from the conceptual viewpoint it seems that certain features (or lack thereof) of the Java language guide the programmer to design his virtual machine in a more structured and rigorous manner. In contrast, a C++ programmer is typically so concerned with performance issues that there is a great tendency to bypass all design paradigms and abstractions in the name of efficiency. Also, the Java™ programming language has many features, such as automatic memory management and comprehensive standard libraries, that relieve the programmer from worrying about certain technical details that have traditionally diverted the programmer's attention away from the program itself. For instance, a virtual machine designer working with C or C++ must typically spend a substantial amount of time worrying about memory allocation for various internal buffers and temporary string manipulation areas. Also, the designer has to implement all kinds of helper data structures because the standard libraries provided with the language are so limited.

In general, programming in the Java programming language tends to be more fun than in C or C++ since the programmer can better concentrate on the program and the problem domain. Nevertheless, it is obvious that writing programs in the Java language does not automatically guarantee that the resulting program would be good. In general, no programming language or language mechanism should be used as a substitute for creative thinking, or as an excuse for avoiding software design and architecture.

During the implementation of JavaInJava it became clear that the Java language is more an application development language rather than systems programming language. In systems programming it is commonly necessary to access low-level, platform-specific structures; however, the Java language does not provide such facilities other than extending the native function interface. Generally, all facilities for “shooting oneself in the foot”, familiar from C or C++, are missing from the Java programming language.

In terms of performance, a naïvely written straightforward bytecode interpreter such as JavaInJava is inherently much slower than a virtual machine written in C or C++, especially if we assume that the virtual machine written in the Java language has to run on top of another Java virtual machine. But this does not imply that a virtual machine written in the Java language would *generally* have to be slower than C or C++ based implementations. Techniques for competing with the performance of virtual machines written in C or C++ will be discussed briefly in Section 4.3.

Even though it is fairly obvious to most programmers who have used the Java language for a while that programming in Java™ is a much more pleasurable experience than in C++, this does not mean that Java is the ultimate programming language. For instance, according to our experience, the Java language falls short in supporting advanced object-oriented programming techniques. In particular, the lack of first-class primitive types and first-class methods (for example, Smalltalk-style blocks) makes the life of the programmer more difficult in certain situations than it should be. In this regard the designers of both Java and C++ could learn a lot from more dynamic and more purely object-oriented programming languages such as *Smalltalk* [GoR83] and *Self* [UnS87, SmU95].

When implementing JavaInJava we noted that some commonly needed programming techniques and strategies are very difficult or outright impossible in the Java language. For instance, when one wants to implement lookup tables for calling functions written in the Java language, the programmer has three basic choices. The first, naïve solution is simply to create a hashtable for mapping the names and the signatures of the functions to certain values, and then use a switch statement for mapping these key values to the actual functions. This solution works but is slow and ugly. The second solution is to create an extra class for storing the functions and to let the Java Reflection API do the dynamic mapping from function names and types to functions. This alternative is better but is painful for the programmer since the method lookup and parameter passing conventions in the Reflection API are rather tedious to use. The third solution is to use inner classes for storing the methods. This solution is perhaps the cleanest but is unpleasant from the conceptual modeling viewpoint since it means that a large number of classes will have to be created only because of the lack of a certain language feature — not because the modeling of the problem domain actually demands this. In C or C++ the programmer can easily solve the above problem by using function pointers, not to mention the lucky Smalltalk programmer who can utilize the powerful block mechanism provided by the language.

In summary, it is quite clear that Java is not a perfect programming language, but for most tasks it is still much better than the other widely used languages, or at least a big step in the right direction.

4.2. Comments on Java Virtual Machine Specification and the Java libraries

The JavaInJava project has shown us that building a complete Java virtual machine implementation in a cleanroom fashion based on the Java Virtual Machine Specification book [JVM96] is possible in a reasonably short amount of time. The implementation of the basic JavaInJava virtual machine took only about two man-months, which is not very much by any standard. However, this should not be taken as a representative figure since the author already had a lot of prior experience with the Java programming language and with building virtual machines for other languages. Also, many of the most challenging areas in virtual machine implementation, such as memory management or building a Just-In-Time (JIT) compiler, were ignored in the JavaInJava implementation. Usually the development and debugging of the memory management system alone takes months, as does the tuning of an adaptive Just-In-Time compiler to reach optimum performance.

In general, the JavaInJava project addressed only the easy, previously well-explored areas in virtual machine implementation and left all harder areas to be examined in future projects. But at least part of the rapid progress can be attributed to the Java programming language, which helped us focus on the design of the virtual machine itself rather than diverting the attention to technical details. However, at the same time it should be mentioned that the development of JavaInJava suffered substantially from the low quality of the current Java programming environments; we used several of them but found all of them rather buggy and unfriendly to use.

Apart from the interesting technical problems discussed earlier in this paper, no major Java Virtual Machine Specification related obstacles were encountered during the project. As mentioned earlier, the only major headache was the native function interface for which very little documentation is available. In general, even though the native function interface is an “implementation issue”, it seems like a bad idea not to provide detailed documentation on which functions are generally assumed to be native and what the operational semantics of these functions are. The lack of documentation makes it hard to ensure the compatibility of Java library implementations and to use the same Java library implementations on different platforms. This seems harmful, since generally the majority of the functions of the library classes could be used easily on various platforms without modifications. The problems with native functions could potentially become a major obstacle to the “write once, run anywhere™” principle. Fortunately, JNI and JDK 1.2 are should make things much better in this regard.

Another library-related problem that was noted during JavaInJava implementation is in the Java virtual machine initialization process. The current initialization process is rather ad hoc; for instance, it requires the presence of certain private functions in the class libraries and assumes that the calling of the I/O and thread initialization operations and

the loading of the standard classes are done in a certain order. Again, the JVM Specification assumes the initialization process to be purely an implementation issue, and thus very little documentation is available, making it hard to share standard Java class library implementations across different platforms. Also, the current initialization process seems overly time-consuming. For instance, In JDK 1.1, the virtual machine has to execute 90,000 to 130,000 bytecodes before it does anything useful for the user; in programs that use graphics the overhead is even higher. The majority of the initialization time goes to setting up the complex character encodings and internationalization features, most of which are currently used by only a handful of Java programs.

4.3. Performance issues

As mentioned at the beginning of the paper, the JavaInJava virtual machine running on a standard Java VM executes Java programs roughly three orders of magnitude more slowly than the standard Java virtual machine does alone. It should be emphasized that the current results do not justify the conclusion that a virtual machine written in the Java programming language would generally be this slow. The current implementation of JavaInJava is built around a very straightforward bytecode interpreter that has been optimized only for conceptual clarity and style. Also, the current interpreter contains a lot of code that is used only for profiling purposes. Even with simple optimizations such as adding the quick bytecodes [JVM96 pp. 389-428] to avoid expensive constant pool lookups, using inline caching of method calls, and avoiding the extensive use of string manipulation operations in method and field lookup, the performance of the system could probably be improved by at least one order of magnitude.

It should also be remembered that JavaInJava currently has to be run on top of another Java virtual machine, and this obviously adds a lot of performance overhead because of the two-level interpretation process and replicated safety and sanity checking. Experiments done by Nik Shaylor from SunSoft indicate that by using a native Java-to-C compiler, the performance of the JavaInJava system can be improved easily by an order of magnitude. In general, by utilizing a native compiler to make the comparison of the JavaInJava interpreter with other Java interpreters more realistic, and by adding the simple optimizations mentioned above, the JavaInJava interpreter would probably be only 10-20 times slower than a Java interpreter written in C or C++. It is rather difficult to reach better performance than that without cheating, since the runtime environment for a native Java language compiler will still have to perform certain safety and sanity checks, or otherwise the system could not be called a pure Java environment.

If one wants to match or exceed the performance of current Java virtual machines written in C or C++, then an entirely different approach is necessary. Basically, by writing a *Java virtual machine generator* in the Java language, and by including a fast adaptive Just-In-Time compiler in the system, it is possible to reach the performance of a virtual machine written in any language. In other words, rather than writing the virtual machine itself in the Java programming language, we would write a Java-based generator framework that would allow the programmer to generate an optimum, platform-specific

Java virtual machine for different platforms. A related approach is currently used in some state-of-the-art Java virtual machine implementations written in C++, such as the new *HotSpot* virtual machine from JavaSoft. Writing a virtual machine generator framework and an advanced JIT compiler in Java rather than in C++ would probably make the resulting system easier to understand and maintain. This would be a challenging and exciting future research area.

5. Conclusion

In this paper we have summarized our experiences in building JavaInJava — a Java virtual machine implementation written in the Java programming language. We started by describing the overall design and architecture of the system, and then presented some interesting technical issues that were encountered during its implementation. In spite of the slow execution speed of the system, roughly three orders of magnitude slower than a standard Java VM, the JavaInJava experiment has been very encouraging, since it proves that building clean virtual machines in the Java language is possible. In general, the JavaInJava system is based on a conceptually clean and simple overall design: it has a modular, well-commented codebase, and it provides various options for profiling and debugging Java™ program execution. Many parts of the system, such as the class loader, could potentially be used separately for building different kinds of Java-based visualization and debugging tools for the Java language. Also, the system is probably the first Java virtual machine that can run itself, albeit extremely slowly. In the end of the paper our JavaInJava experiences were compared to building virtual machines in other languages, and some possible future research directions — such as writing a virtual machine generator framework in the Java programming language — were outlined.

References

- FMB90 Freeman-Benson, B., Maloney, J., Borning, A., An Incremental Constraint Solver. *Communications of the ACM* vol 33, nr 1 (Jan) 1990, pp.54-63.
- GHJ95 Gamma, E., Helm, R., Johnson, R., Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- GoR83 Goldberg, A., Robson, D., *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.
- IEE85 *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Std 754-1985, IEEE, New York, 1985.
- JLS96 Gosling, J., Joy, B., Steele, G., *The Java™ Language Specification*. Addison-Wesley, 1996.
- JVM96 Lindholm, T., Yellin, F., *The Java™ Virtual Machine Specification*. Addison-Wesley, 1996.
- SmU95 Smith, R.B., Ungar, D., Programming as an Experience: The Inspiration for Self. In Olthoff, W., (ed): *ECOOP'95 Conference Proceedings* (Aarhus, Denmark, August 7-11), Lecture Notes in Computer Science 952, Springer-Verlag, 1995, pp.303-330.
- Tai93 Taivalsaari, A., Concatenation-based object-oriented programming in Kevo. *Actes de la 2ème Conférence sur la Représentations Par Objets RPO'93* (La Grande Motte, France, June 17-18), EC2, France, 1993, pp.117-130.
- Tai96 Taivalsaari, A., On the Notion of Inheritance. *ACM Computing Surveys* vol 28, nr 3 (Sep) 1996, pp.438-479.
- UNI92 *The Unicode Standard: Worldwide Character Encoding*, version 1.0 (volume 1). Addison-Wesley, 1991.
- UnS87 Ungar, D., Smith, R.B., Self: the power of simplicity. In Meyrowitz, N. (ed): *OOPSLA'87 Conference Proceedings* (Orlando, Florida, October 4-8), ACM SIGPLAN Notices vol 22, nr 12 (Dec) 1987. pp.227-241.

About the Author

Antero Taivalsaari is staff engineer at Sun Microsystems Laboratories in Mountain View, California, where he is studying new implementation techniques for dynamic programming languages. His research interests include object-oriented programming and design, implementation of interactive programming languages, collaborative software engineering environments, programming techniques for small portable devices, and prototype-based programming. He has published a number of research papers in international journals and conferences, given invited lectures on various topics, and organized several international workshops in the field of object-oriented programming.

Before joining Sun Microsystems Laboratories in August 1997, Antero worked four years as a research manager at Nokia Research Center in Helsinki, Finland, leading a research group on collaborative software design environments, and managing some of Nokia's international research projects. Prior to that, he worked several years in the academic world, and completed a doctoral degree in computer science at the University of Jyväskylä, Finland, in 1993, after spending 1 ½ years as a guest researcher at Concordia University and University of Victoria in Canada. His doctoral thesis, entitled "A Critical View of Inheritance and Reusability in Object-Oriented Programming", was awarded as the best doctoral dissertation in computer science in Finland in 1994.