Resurrecting the Applet Paradigm

Eric Roberts
Stanford University
Department of Computer Science
Stanford, CA 94305
+1 650-723-3642
eroberts@cs.stanford.edu

ABSTRACT

Since the introduction of Java in 1995, educators have recognized the potential of Java applets as an educational resource. Sadly, the continuing evolution of Java has made it harder to use applets, largely because it is so difficult to keep those applets compatible with the many different versions of the Java runtime environment supported by existing browsers. Over the past two years, the ACM Java Task Force (JTF) has developed an effective strategy that makes it possible to write applets using up-to-date versions of Java that will nonetheless run on browsers that support only the JDK 1.1 environment. This paper describes the **`acm11.jar`** library, which uses this strategy to achieve the desired backward compatibility. It also describes a more general solution strategy for which we have a prototype, although we are unable to release the prototype until we get permission from Sun Microsystems. The **`acm11.jar`** library can be used with any Java applet and does not depend on adopting the JTF library packages. That library therefore represents a general resource for teachers and students who want to write Java code that runs in web environments.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: computer science education.

General Terms

None.

Keywords

Applet, Java, teaching resources.

1. INTRODUCTION

Since the introduction of Java in 1995, educators—not only in computer science but in a wide variety of other disciplines as well—have recognized the enormous potential of Java applets as an educational resource [4]. Today, there are tens of thousands of applets available on the web that illustrate concepts in sciences including physics, chemistry, biology, and mathematics, along with many other subject areas outside of the sciences altogether. In computer science, Java applets have been used extensively to animate algorithms [7, 11], data structures [14], and various topics in discrete mathematics [8]. Because applets are interactive, they are typically more engaging that web sites that contain only expository material. By giving the student an active role, applets enable a more participatory approach to learning, along the lines recommended by the constructivist philosophy of education [3].

In recent years, however, it has become increasingly difficult to create and maintain applets that work compatibly across the many versions of Java and the range of available browsers. Many of the applets that exist on the web today require particular versions of the Java runtime environment or work correctly only on specific browsers, thereby limiting their utility as teaching tools. As discussed in section 3, Sun Microsystems has largely given up on the problem, recommending instead that application designers migrate to the Java™ Web Start technology [17].

The shift away from applets, however, comes at a significant cost. Although Java Web Start makes it possible to initiate Java applications from inside a web page, it offers almost none of the advantages that the use of applets provides. The great strength of applets is that they make it possible to combine interactive demonstrations with other web content in a smoothly integrated way. Java Web Start offers none of that. Moreover, since anyone wishing to use Java Web Start has to download new plug-ins for their browser, it has little impact on the compatibility problems that led to the decline in the popularity of applets in the first place.

In developing its collection of materials to simplify the use of Java in introductory computing courses, the ACM Java Task Force (JTF) [1, 2, 13] concluded that the applet paradigm was too important to abandon. The Java Task Force web site at **`http://jtf.acm.org/`** uses applets extensively in a variety of contexts. In particular, the JTF materials include an interactive tutorial [14] and a collection of downloadable demo applets that illustrate how to use each of the classes defined in the JTF packages. As far as we know, those applets run on all Java-enabled browsers, thereby fulfilling the "write once, run anywhere" promise that was part of the early Java lore.

The purpose of this paper is to explain the strategy we used to achieve backward compatibility for the JTF applets so that other developers can adopt it as well. The primary tool we offer is a special library archive called **`acm11.jar`** (*a-c-m-one-one-dot-jar*) that gives an applet access to the JTF libraries along with backward-compatible versions of the classes in the subset that the JTF recommended in its *Project Rationale* [1]. That subset appears in Figure 1.

The most significant limitation in the **`acm11.jar`** library is that it is applicable only to applets that limit themselves to classes in the JTF subset. To support greater generality, we have also developed a prototype application called **`jar11`** (*jar-one-one*) that takes an existing JAR file and restructures it so that it will work correctly even in a browser that runs only version 1.1 of the Java Development Kit (JDK). This approach gives an applet access to the complete functionality of J2SE 5.0. Unfortunately, it is not currently possible to distribute the **`jar11`** application under the terms of the Java binary code license agreement [16], which prohibits the modification of the standard libraries on which this approach depends. The JTF is entering into discussions with Sun Microsystems to determine whether we can obtain permission to release this tool.

For the benefit of readers whose primary interest is in gaining the backward compatibility, the following section offers a brief guide to using the **`acm11.jar`** library . The remaining sections outline the evolution of the underlying strategy.

SIGCSE'07, March 7-10, 2007, Covington, Kentucky, USA.

Figure 1. Java classes that form the preferred JTF subset

java.applet:
- Applet
- AudioClip

java.awt:
- BorderLayout
- Color
- Component
- Container
- Dimension
- Event
- FlowLayout
- Font
- FontMetrics
- Frame
- Graphics
- GridLayout
- Image
- LayoutManager
- MediaTracker
- Point
- Rectangle
- Toolkit

java.awt.event:
- ActionEvent
- ActionListener
- AdjustmentEvent
- AdjustmentListener
- ComponentEvent
- ComponentListener
- FocusEvent
- FocusListener
- KeyEvent
- KeyListener
- MouseEvent
- MouseListener
- MouseMotionListener
- WindowEvent
- WindowListener

java.io:
- BufferedReader
- BufferedWriter
- File
- FileReader
- FileWriter
- IOException
- PrintWriter
- Reader
- StreamTokenizer
- StringReader
- StringWriter
- Writer

java.lang:
- Boolean
- Character
- Class
- Cloneable
- Comparable
- Double
- Enum
- Exception
- Float
- Integer
- Long
- Math
- Number
- Object
- Runnable
- String
- System
- Thread

java.math:
- BigInteger

java.net:
- URL
- URLConnection

java.text:
- DateFormat
- DecimalFormat
- NumberFormat

java.util:
- ArrayList
- Arrays
- BitSet
- Collection
- Comparator
- HashMap
- HashSet
- Iterable
- Iterator
- LinkedList
- List
- ListIterator
- Map
- PriorityQueue
- Queue
- Random
- Scanner
- Set
- SortedMap
- SortedSet
- Stack
- StringTokenizer
- TreeMap
- TreeSet

javax.swing:
- Box
- BoxLayout
- ButtonGroup
- JApplet
- JButton
- JCheckBox
- JComboBox
- JComponent
- JDialog
- JFileChooser
- JFrame
- JLabel
- JList
- JOptionPane
- JPanel
- JPopupMenu
- JRadioButton
- JScrollBar
- JScrollPane
- JSlider
- JSpinner
- JTable
- JTextArea
- JTextField
- JTextPane
- JToggleButton
- JWindow
- KeyStroke
- Timer

javax.swing.event:
- ChangeEvent
- ChangeListener
- ListDataEvent
- ListDataListener
- ListSelectionEvent
- ListSelectionListener

Underlined class names are new since JDK 1.1

2. USING THE `acm11.jar` LIBRARY

The use of the **`acm11.jar`** library described in the preceding section is most easily illustrated by example. Suppose that you have just typed in the simple "hello, world" applet called **`HelloGraphics`** described on the first page of the *JTF Tutorial* [2]. Assuming that you are using Sun's traditional command-line tools, you can compile that applet by typing the command

```
javac -classpath acm.jar HelloGraphics.java
```

and then run it as an application by typing

```
java -cp .:acm.jar HelloGraphics
```

The resulting program uses the Java runtime environment that you have installed on your computer. Because the JTF libraries make use of the many features provided by Swing, the **`HelloGraphics`** program will run correctly only if your computer has JDK 1.2 or some more recent version of the Java runtime environment. Since you control your own environment, keeping things up to date is not too hard.

Now suppose that you want to add **`HelloGraphics`** as an applet to a web page so that others can use it as well. The first step in that process is to create a JAR file containing both the code for your applet and the resources from the **`acm.jar`** file. The easiest way to accomplish that goal using the command-line tools is to issue the following commands:

```
cp acm.jar HelloGraphics.jar
jar uf HelloGraphics.jar HelloGraphics.class
```

You also need to create an HTML file containing the following **`applet`** tag:

```
<applet archive=HelloGraphics.jar
        code=HelloGraphics.class
        width=500 height=300>
</applet>
```

When your browser reads this **`applet`** tag, it will reserve a 500 x 300 area and display the **`HelloGraphics`** applet in this space.

This applet, however, works correctly only on browsers that are running with JDK 1.2 or some more recent environment. On a JDK 1.1 browser, the class loader won't be able to resolve the **`HelloGraphics`** class, let alone the classes in the JTF libraries. The **`HelloGraphics`** class extends **`GraphicsProgram`**, which in turn extends **`Program`**, which is itself a subclass of **`JApplet`**. The **`JApplet`** class, unfortunately, is part of Swing, which is not defined in JDK 1.1.

It is at this point that the **`acm11.jar`** library comes to the rescue. If you want to create a backward-compatible applet, the simplest approach is to replay the steps you used to build this applet, replacing the **`acm.jar`** archive with **`acm11.jar`** everywhere it appears. The resulting **`HelloGraphics.jar`** file will contain no references to classes outside of the JDK1.1 environment and will therefore run even on older browsers.

The major limitation in this approach is that it works only for programs that limit their use of classes to those that were available in JDK1.1 and any additional classes shown in Figure 1. This strategy therefore works for **`HelloGraphics`** because it—like all the classes in the JTF collection of downloadable demo programs—limits itself to the acceptable class subset. The good news, however, is that it is considerably harder to move outside

the recommended subset than you might at first imagine. Intuitively, it seems as if the generic classes introduced in J2SE 5.0 would surely fall outside the boundaries of the JTF subset. In fact, they don't. The Java Virtual Machine [10] has not changed in any significant way since 1999. As a result, all source-level references to generic classes are mapped internally into references to the base classes from which they arise, adding in any necessary casts or runtime type checking [6]. In terms of what shows up in the class file, references to `ArrayList<Integer>` turn into simple references to `ArrayList`, which is included in the subset.

3. THE RISE AND FALL OF APPLETS

In the early years of Java, applets were responsible for much of the tremendous excitement associated with the language. The World Wide Web was hot in those days of the dot.com boom, and Java applets offered the enticing prospect of making it possible to run sophisticated programs safely on the web. Applets of all kinds began to appear, growing in that unplanned, anarchic way that typifies distributed development on the web. As an expression of the extraordinary buzz surrounding applets at that time, it is hard to do better than the following quotation from 1997, which appears in the collaboratively authored book *Using CGI:*

> Anyone using a computer who hasn't been in a cave for the last year has heard of Java applets. Many people believe that Java is useful only for applets. [18]

Computer science educators were quick to embrace the applet paradigm. The proceedings of the SIGCSE Symposia from the last years of the 1990s invariably include several papers extolling the virtues of applets as teaching tools. One paper written by the Java-development team at Montana State University argued that applets represented a fundamental paradigm shift in computing education, going so far as to punctuate that assessment with an exclamation point in their title [5].

Sadly, the excitement surrounding that initial vision has faded in recent years. The reason, however, is not that the advantages of applets were oversold in the initial rush of enthusiasm for a new technology. In terms of the potential they offer for interactive web-based teaching and learning, applets are as exciting as they ever were, at least in theory. The problem comes in practice. As Java evolves, web-based resources written in Java—particularly those based on the applet paradigm—have became much more difficult to develop and maintain. The fundamental problem lies in achieving the backward compatibility required to allow those applets to run in an ever-expanding array of browsers that often implement radically different versions of the JDK. As a result, applet-based systems have started to give way to other design strategies such as those based on Java Web Start, the technology that Sun Microsystems offers as a replacement for applets. The reasons behind the decline in the applet paradigm are clearly identified in the following 2001 article from *Java World:*

> Java applets fueled Java's initial growth. The ability to download code over the network and run it on a variety of desktops offering a rich user interaction proved quite compelling. However, Java's Write Once, Run Anywhere (WORA) promise soon became strained as browsers began to bloat and several incompatibilities emerged that were caused by the Java language itself. [15]

At the same time, it is important to emphasize that applets have not gone away, nor are they likely to do so in the foreseeable future. There are simply too many of them in place for a browser vendor to ignore applets altogether. What has happened is that the Java that runs in the applet world has not kept pace with the ongoing evolution of the Java language. Browsers that run long-obsolete versions of Java are commonplace. If, for example, you walk into an Internet storefront such as your local Kinko's or an EasyInternetCafe, you will find that their browsers are running JDK 1.1 because that is the version that comes with the standard Microsoft distribution. To use a more up-to-date release of Java, users must explicitly download a new version of the JDK. Unfortunately, many choose not to do so.

At the same time, the number of Java-enabled browsers that are running JDK versions that *precede* JDK 1.1 seems vanishingly small. That fact suggests at least a potential strategy for solving the compatibility problem. As long as you are content to write applets that limit themselves to JDK 1.1 and avoid doing anything at all complex or tricky, you can count on having your applets run in almost any Java-enabled browser. The best of the existing web applets do precisely that. Thus, those who are in the business of creating web-based tools have an effective, if somewhat inconvenient, strategy for maintaining compatibility.

However useful that strategy may be for applet developers, it would be completely inappropriate to teach that style of coding to beginning students. Java has evolved in many significant ways since the JDK 1.1 release. Teaching students to operate under that paradigm would be the computational equivalent of returning to writing on cuneiform tablets. JDK 1.1, after all, appeared in 1998, which is an eternity ago at the speed of technological evolution.

The designers of the JTF tools therefore faced an uncomfortable dilemma. On the one hand, it is important to maintain backward compatibility so that applets written using the JTF packages can run on the widest selection of existing browsers. On the other, the programming style that offers the easiest strategy for achieving such compatibility is pedagogically indefensible. We needed to find a more creative approach.

4. EVOLUTION OF THE JTF APPROACH

At the time of the first public draft of the JTF design in February 2005, we had concluded that the best way to achieve the goal of backward compatibility was to have the code for the JTF packages check the runtime environment to see what facilities were available. If the applet were running in a browser that supported Swing and other modern features, then the implementation would take advantage of those capabilities using the Java mechanism known as *reflection,* which allows a Java applet to gain access to class definitions, methods, and fields by name. Using reflection was necessary because older browsers would otherwise be unable to load the JTF code. If the code contained explicit references to classes not supported by the browser's implementation of the JDK, the applet would simply fail to load.

Although we were able to implement the reflection-based strategy, it proved to be problematic for several reasons:

1. The code for the library packages—which often had to include two separate implementations for the different JDK versions—became too complicated, making it harder to develop and maintain.
2. The code could no longer serve as a model for students. In an ideal implementation of a package intended to provide scaffolding for novices, students should eventually reach the point at which they can open the black box that they have been using and see what makes it tick. Writing code that depended on reflection for anything outside of the JDK 1.1 universe meant that students would learn little in terms of programming style by looking at the definitions.
3. The fact that we used JDK 1.1 classes in the implementation of our packages was incorrectly interpreted by some members of the community as an endorsement of an obsolete programming style.
4. The reflection-based strategy did not provide a comprehensive solution to the compatibility problem. Implementing the libraries in this fashion made it *possible* for adopters to write compatible

applets but did not really simply the task of doing so. Applets intended to run in older browsers required their authors to code those applets using the same constraints we had used to write the JTF packages.

After recognizing the weaknesses in our original approach, the Task Force adopted a new strategy for the beta release in February 2006. The key to this strategy lies in the fact that introductory students cannot be expected to master the full range of classes available in the JDK but will of necessity focus on a subset. As we describe in Chapter 9 of our *Project Rationale* [1], the Task Force identified a subset of classes that we believed, based on a survey of prominent Java textbooks, would allow us to reign in some of the complexity caused by the sheer size of the JDK but still give individual instructors considerable flexibility to select a range of teaching strategies. The classes that form the JTF preferred subset appear in Figure 1 earlier in this paper The underlined class names are the ones that have been added since the 1.1 release of the JDK. Of the classes shown in Figure 1, just over half (66 out of 130) are new; the rest date from the JDK 1.1 release. If we could find a way to make these new classes work correctly in older browsers, we would be able to solve the compatibility problem for programs that students were likely to write.

5. DEVISING A CONCRETE STRATEGY

Although selecting a reduced subset helps to limit the scope of the problem, it does not in itself constitute a solution. As is usually the case, the devil is in the details. The new classes marked by underlining in Figure 1 are by no means standalone entities. These classes depend on other classes that would need to be included in the subset even though introductory students would be unlikely to use them explicitly. Those classes in turn make reference to other classes. Computing the transitive closure of the dependency graph ends up dragging in well over half of the classes in the Swing library, thereby eliminating most of the complexity savings.

There is, however, a more significant problem. Although it initially seems as if one could simply add the new classes and their extended dependencies to the JAR file for the applet, that strategy runs afoul of the security manager in an applet-based environment. By default, applet security managers disallow loading any new classes into any of the packages in the **`java`** package hierarchy, and in some browsers similarly prevent the definition of new classes rooted in the **`javax`** hierarchy as well. That restriction is clearly justified. If clients could, for example, load new classes into the **`java.lang`** package, those classes would then have access to the package-private classes and methods in **`java.lang`**, creating a massive security loophole. Thus, if the code for the missing classes are to be added to the JAR file, those classes must go into some package for which the security restrictions do not apply. Unfortunately, the applet code the student writes will try to import those classes from the packages in which they are *supposed* to reside rather than from wherever we decided was safe to put them. Thus, the success of the repackaging strategy depends on finding a way to create the correct associate the classes the student assumes are part of the Java package space with the code that is loaded elsewhere. After considerable experimentation, we identified two viable strategies for solving this problem, which give rise to the two tools—the **`acm11.jar`** archive and the **`jar11`** application—described in the introduction.

5.1 The `acm11.jar` Library Archive

The first strategy we devised was to reimplement the missing classes (the ones underlined in Figure 1) and to package those new implementations together with the JTF classes in an archive called **`acm11.jar`**. An applet compiled with **`acm11.jar`** makes no references outside the JDK 1.1 world.

This simple explanation, however, glosses over a great deal of detail. The first question likely to occur to readers is that the **`acm11.jar`** must involve an enormous amount of code. The reality of that assessment depends, of course, on one's definition of *enormous*. While it is true that simulating the missing classes requires some amount of code—just under 7000 lines in the current release—the source code for the standalone implementations for these classes represents only about two percent of the source code for the Swing libraries as a whole. In part, the reduction in size reflects the fact that a relatively small part of the Swing library is required to achieve the basic functionality. Another part of the savings comes from the fact that it is often easy to simulate the new classes using existing classes in the JDK 1.1 world. The collection classes represent a illustrative case in point. Reimplementing **`ArrayList`** and **`HashMap`** in terms of **`Vector`** and **`Hashtable`** means that there is very little new code to write. It is also important to keep in mind that the Task Force had already implemented JDK 1.1 versions of many of the Swing classes as part of its initial approach to maintaining compatibility, which means that there was relatively little new code to write.

The more important question is that it is still unclear where to put these reimplemented classes. As noted at the end of the preceding section, it is not possible to put the new **`ArrayList`** implementation into **`java.util`** without upsetting the security manager. Although assigning it to some other package would be safe, it seems as if doing so would require either changing the source code for the applet or adopting some postprocessing scheme along the lines of the **`jar11`** application described in the following section. That we were able to get away without imposing either of these constraints depends on two common properties of novice programs:

1. *Novice programs tend to reside in the unnamed default package*. To avoid the complexity of introducing Java's package structure too early, most instructors allow students to write their programs without including a **`package`** declaration. As a result, the classes in the student program are placed in the unnamed package that Java assumes by default.
2. *Novice programs often incorporate all the classes in a package in a single import line*. Although some instructors prefer to have students import each class individually, it is far more convenient to use wildcard imports. In an application that uses **`ArrayList`**, **`HashMap`**, **`Iterator`**, and various other classes from **`java.util`**, it is easier to import these classes all at once using the line

```
import java.util.*;
```

than it is to write a separate **`import`** line for each class.

As long as these two properties apply, it is possible to avoid the repackaging problems simply by assigning the reimplemented classes to the unnamed package. When the student uses **`HashMap`**, for example, in a compilation that includes **`acm11.jar`** in its classpath, the definition of **`HashMap`** is taken from the unnamed package rather than from **`java.util`**. Because Java's name-resolution rules—as indeed they must—always prefer classes in the current package to identically named classes in other package, no ambiguity arises. As a result, the **`acm11.jar`** can introduce new implementations of the required classes without forcing the student to take any explicit action.

5.2 The `jar11` Application

Although the **`acm11.jar`** strategy is effective, it suffers from a lack of generality. Some instructors may want to use classes that fall outside the JTF subset, at least in their design of web tools if

not in their teaching. Others may insist on having students put their code in a named package or having them import every class individually. In either of these cases, the simple **acm11.jar** strategy is no longer adequate. Even so, the basic idea of repackaging the missing classes still applies. The only difference is that some explicit action is necessary on the programmer's part to ensure that the references to these missing classes are correctly resolved to the appropriate packages.

The prototype **jar11** application provides more robust compatibility by implementing a more general strategy. To create a backward-compatible applet using **jar11**, all you need to do is invoke the **jar11** application on the JAR file that contains the applet code. For example, if the code for your applet is stored in the file **MyApplet.jar**, all you need to do to make it work with JDK 1.1 browsers is invoke the command

```
jar11 MyApplet.jar
```

Invoking this command **executes** the following actions:

1. It adds the entire Swing library (along with other classes missing from JDK 1.1) to the **MyApplet.jar** file. In so doing, however, it does not place those classes in the **javax** package tree but in an isomorphic **jar11** tree that contains the same class hierarchy, suitably modified so that any internal references remain consistent. Thus, the class **javax.swing.JApplet** becomes **jar11.swing.JApplet**, which gives the security manager no trouble at all. Classes in other packages are similarly renamed.
2. It rewrites the class files in the original **MyApplet.jar** file so that external references to any renamed classes correctly include the updated package name. Doing so requires parsing the class files and making the substitutions only in the strings that refer to class names.

As is clear from the above description, the **jar11** application modifies the binary code for the Java standard libraries as it creates the parallel **jar11** package. Unfortunately, such modification is not permitted under the current binary code license agreement in force for the JDK [16]. Given these restrictions, the JTF will not be able to release the **jar11** application until the licensing issues can be resolved. Fortunately, Sun seems to be embracing more open-source technology; their recent decision to adopt the GNU public license for Java is a very positive sign [9].

The major limitation of the **jar11** application is that the JAR files it creates are much larger than the original applet archive. As a result, applets built using **jar11** take considerably more time to load over the web. It therefore makes sense to adopt the more space-efficient **acm11.jar** strategy wherever possible. That strategy was sufficient for all the demo applications included on the JTF web site and is likely to work for most adopters as well.

6. CONCLUSION

The **acm11.jar** library—and the **jar11** tool when it appears—provides an effective strategy for writing applets that run in browser environments that run any version of the JDK back through version 1.1. Those tools, moreover, offer programmers a choice between convenience and generality that should make them widely applicable. The use of these tools in no way depends on adopting the supplemental packages provided by the JTF library and can therefore be used with any Java applet.

REFERENCES

[1] ACM Java Task Force. *Project Rationale*. August 25, 2006. **http://jtf.acm.org/rationale/**

[2] ACM Java Task Force. *JTF Tutorial*. August 25, 2005. **http://jtf.acm.org/tutorial/**

[3] Mordechai Ben-Ari. Constructivism in computer science education. *Proceedings of the Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education,* Atlanta, GA, February 1998. **http://doi.acm.org/10.1145/273133.274308**

[4] Joseph Bergin, Thomas L. Naps, et al. Java resources for computer science instruction. Report of the ITiCSE'98 Working Group on Curricular Opportunities of Java Based Software Development, SIGCSE Bulletin, September 1997. **http://doi.acm.org/10.1145/316572.358291**

[5] Christopher Boroni, Frances Goosey, Michael Grinder, and Rockford Ross. A paradigm shift! The Internet, the web, browsers, Java and the future of computer science education. *Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education,* Atlanta, GA, February 1998. **http://doi.acm.org/10.1145/273133.273181**

[6] Gilad Bracha. Generics in the Java Programming Language. Sun Microsystems, July 5, 2004. **http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf**

[7] Joshua Cogliati, Frances Goosey, Michael Grinder, Bradley Pascoe, Rockford Ross, and Cheston Williams. Realizing the promise of visualization in the theory of computing. *Journal of Educational Resources in Computing,* June 2005. **http://doi.acm.org/10.1145/1141904.1141909**

[8] Susanna S. Epp. Discrete mathematics applets etc. As revised on March 24, 2006. **http://condor.depaul.edu/~sepp/DMappletsEtc.htm**

[9] Martin LaMonica. Sun picks GPL license for Java code. *CNET News,* November 12, 2006. **http://news.com.com/Sun+picks+GPL+license+for+Java+code/2100-7344_3-6134584.html**

[10] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine*. Second edition. Sun Microsystems, 1999. **http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html**

[11] Thomas Naps. Algorithm visualization on the World Wide Web—the difference Java makes!. *Proceedings of the 2nd Conference on integrating Technology into Computer Science Education,* Uppsala, Sweden, June 1997. **http://doi.acm.org/10.1145/268819.268839**

[12] Willard C. Pierson and Susan H. Rodger. Web-based animation of data structures using JAWAA. Proceedings of the 29th SIGCSE Technical Symposium on Computer Science Education, Atlanta, GA, February 1998. **http://doi.acm.org/10.1145/273133.274310**

[13] Eric Roberts. Resources to support the use of Java in introductory computer science. Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, Norfolk, VA, March 2004. **http://doi.acm.org/10.1145/971300.971384**

[14] Eric Roberts. An interactive tutorial system for Java. *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*. Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, Houston, TX, March 2006. **http://doi.acm.org/10.1145/1121341.1121447**

[15] Raghavan N. Srinivas. Java Web Start to the rescue. Java World, July 2001. **http://ww.javaworld.com/javaworld/jw-07-2001/jw-0706-webstart.html**

[16] Sun Microsystems. Binary code license agreement for the Java 2 Platform Standard Edition Development Kit 5.0. **java.sun.com/j2se/1.5.0/jdk-1_5_0-license.txt**

[17] Sun Microsystems. *Java™ Web Start Overview*. May 2005. **java.sun.com/developer/technicalArticles/WebServices/JWS_2/JWS_White_Paper.pdf**

[18] K. Mitchell Thompson. Tips and techniques for Java. Chapter 23 of *Using CGI* by Jeffry Dwight, Michael Erwin, and Robert Niles. Indianapolis, IN: Que Publishing, 1997.