

Programming HotJava Applets

Java has come a long way, and it has a long way to go.

The Web is humming, and a lot of that noise is coming from the clash of emerging standards. From HTML 2.0 to Netscape extensions to VRML a slew of new technologies have popped up that promise to flesh out a Web that is still more flash than cash. Among these new developments, none is more eagerly anticipated than executable content - Web content that actually executes on the workstation the browser is running on.

HTML is essentially a 'flat' technology - static text with hyperlinks to other lumps of static text. The current crop of Web browsers take a stream of static text and display them on the screen. The only logic embedded in HTML text consists of text formatting and image/sound file loading commands. The combination of HotJava and Java changes all that. Now the Web can start to fulfill some of its potential for action and interaction.

HotJava is NotJava

Java is a language. A Java program (myprogram.java) is compiled into bytecodes (myprogram.class) which are interpreted at run-time by the Java interpreter. HotJava is a Web browser written in the Java language. It supports a new HTML tag <APP> that allows you to load an applet (myprogram.class) located at some arbitrary URL and run it locally. With appropriate limitations, this applet has very broad access to the resources of the local machine - screen (via the browser window), mouse, keyboard, sound and network cards. Applets are written in Java and compiled into Java bytecodes. You could write a complete, stand-alone application in Java without involving the Web or HotJava in any way. As someone on the mailing list said, HotJava is just a novel way of delivering Java applications.

Java's raison d'etre is architecture-neutrality. The language itself contains no platform dependencies. All types have a fixed size (8,16,32,64 byte) that may or may not correspond to the norm on whatever platform you're running. But you pay a price for that neutrality. Java and its packages attempt to supply all the mechanisms of the native GUI APIs for systems such as X, MacOS, and Windows through a common syntax. That search for common ground sometimes means throwing out things, like the middle and right mouse buttons, that aren't supported on all platforms. Thus, the key to Java's eventual usefulness for developers is not so much its level of functionality, which will always lag the native GUIs, but how many platforms it runs on.

The Application

Deep in our nerdy hearts, we all dream of writing that one really cool app. For me that app was a WAN-based, multi-user game, and Java/HotJava seemed the perfect vehicle for delivering that application. To test this hope, I developed a variant of the old board game Battleship. Players on a 40 by 40 grid fire at each other, scoring points for hits on other players and losing points for taking hits from other players.

Battle of the Java Sea has three parts, the HTML source, the Java applet and the game server daemon. The HTML source merely provides entry to the applet via the <APP ...> tag. The server daemon is a C++ program that runs on a remote host. The Java applet provides the interface to the game and most of its logic. Figure 1 shows where the various pieces of the application run.

Listing 1 shows the HTML code for the page the applet appears on. The applet is placed in-line where the <APP ...> tag appears. The applet sizes itself within the init method, and all the HTML following the applet appears below the applet. If the applet fails to load, a placeholder appears where the applet would have been. The HotJava frame, titlebar, menubar, URL edit field, vertical scrollbar, and navigation buttons remain on the screen. The applet scrolls seamlessly with the HTML text. You can place as many applets as you want on the page. Our HTML page has the game applet and another applet for displaying the high scores.

Code Organization

In an attempt to provide a manageable namespace, Java lumps classes into packages. In your Java source, you can import a package, or a single class within a package. The compiler identifies classes as "mypackage.myclass". My application defines a new package, Ship, and four classes within it, GameSrv, Ship, Explosion and PortThread. There is still considerable debate as to the proper use of package and class names, given the distributed nature of the Java/HotJava developer community. The most sensible suggestion I've seen is to incorporate the company and project name into the package name, leaving classes as unique on a per-project basis.

For the most part, the Ship package uses only three of the packages delivered with HotJava: awt, net and browser. awt encompasses all the graphics and drawing functionality. browser contains the applet wrapper class that our applet subclasses. net implements the socket class that provides our connection to the game server.

Taking out the Garbage

One of the great joys of programming in Java is the automatic garbage collection. As a C/C++ programmer, it seems I've spent half my life chasing and trying to prevent memory leaks. Look closely at the source for Ship and you'll see plenty of news, but no deletes. You can call the garbage collector in your program, but there's no need. It runs automatically in a separate thread.

Arrays

Java requires a novel syntax when declaring arrays. Listing 2 shows the declaration of the array of Explosions. The class variable xp is declared as an array of Explosion objects with no dimension. xp exists as a symbol with a type, but its actual Explosion objects are not instantiated until the new statement in the body of the init method. No memory is allocated for an array until it is created with the 'new' statement. Arrays are objects in the Java hierarchy, not simple types, and thus embody more intelligence than C/C++ arrays. All array references are bound checked, for one thing, and the length variable gives the size of the array.

Arrays bring us to another key feature of Java - Exceptions. The Java language package (java.lang) defines a few dozen Exceptions which have default behaviors and can be caught and thrown as necessary. Bad array references throw an `ArrayIndexOutOfBoundsException`, bad arguments to the Integer constructor throw a `NumberFormatException`. You'd do well to understand Exceptions before proceeding with Java coding. I went to considerable trouble in the message parsing routines (`PortThread.java` lines xxx-xxx) to avoid throwing a `NumberFormatException` when a better approach would have been to catch the Exception and deal with the problem then.

Interfaces

Another enjoyable facet of Java is its implementation of interfaces. Interfaces allow you to define a type of object without subclassing. Listing 3 shows the `PortThread` class, an implementation of the `Runnable` interface. Unlike a subclass, which attaches all the baggage of the superclass (class variables, un-overridden methods ...) an interface donly requires the implementation to supply the methods specified by the interface. Unlike classes, interfaces can be multiply-inherited. For many problems, the interface system is a much more natural solution. For instance, for debugging purposes, I'd like to read a stream of game-server messages from a file, rather than opening a network socket. In that case, I'd define two new classes `FileStream` and `SocketStream`, and one new interface, `MyStream`. `MyStream` would define four methods, `open/read/write/close`. `FileStream` and `SocketStream` would implement both `Runnable` (to get their own thread) and `MyStream`.

The application starts with the `GameSrv` object, which subclasses `Applet` and implements the `Runnable` interface. Subclassing `Applet` allows `GameSrv` to be loaded as an `Applet` by HotJava. THIS IS WRONG start and stop are THREAD things The `Runnable` interface and `GameSrv`'s implementation of its three methods - `start`, `stop` and `run` - allow `GameSrv` to run in its own thread. The thread is actually created by allocating a `Thread` object and passing this as the sole parameter. Then, in the `run` method we can do whatever we want, in this case repainting the applet window at 100ms intervals to reflect the changing game state.

Along with implementing the Runnable interface methods, GameSrv overrides four Applet methods - mouseUp, mouseMove, keyDown and update. Of these, the most interesting is update. update (and repaint) will be familiar to Windows coders as the WM_PAINT case in your message processing switch. As with any GUI app, most of the detail work of the application goes into the paint routine. Applets can override either paint or update to do their painting. If you use paint the window is cleared before paint gets called. That may be fine for some applications, but it was no good for us as it gave the window an annoying flickering appearance. update is the solution to that problem, leaving all window management up to us. That means that each time a ship is moved, we have to erase the old ship, and each time a status string is changed, we have to erase the old one. Listing 4 shows the update method and one of the paint methods it calls.

The thorniest problem in implementing the update() method was a by-product of Java's inherent multithreadedness. In Windows 3.1 SDK programming, you can process your WM_PAINT message without worrying that globals or statics outside the paint routine are going to change unexpectedly. Not so in a multi-threaded environment.

The paintShip method needs to erase the old ship and draw the new ship. This requires three steps: erasing the current ship, painting the new ship, and saving the new ship's coordinates as the current ship's. Listing 5 shows the original update method. The keyDown method changes the ship's coordinates by calling Move. The bug in this is that during the X method invocations between clearRect that erases the current ship and the call to Ship.setLastLoc, the keyDown method can be invoked, setting LastXLoc and LastYLoc to values other than those that the ship is currently painted at. This occurs because update and keyDown can be called from separate threads. Figure 2 illustrates the problem.

Veteran painters will also notice that update contains an important cheat - it doesn't repaint the background. Were we to use an image background, or a color other than the default browser background, we'd have to clear all the background sections of the window. As it is, we get a good visual effect for very little code.

The Applet class provides start and stop methods that are called whenever the applet becomes visible, or invisible. Though this version of the game doesn't use them, future versions will skip repainting and stop all network communications when the applet is invisible, in order to minimize CPU load and network traffic.

The PortThread object, which reads input from the game server, uses the run method a little differently. The run method creates an instance of a socket, given an IP address and port number, and sits in a loop doing blocking reads of the Socket's inputStream member. The PortThread object also illustrates another important bit of thread trivia. A Java application doesn't exit until all non-daemon threads are killed. The PortThread thread calls setDaemon so that the app can exit without explicitly killing the PortThread. Listing 2 shows the PortThread use of the Runnable interface.

The fixed message size, fixed field length message protocol I started with was surprisingly painful to implement mostly due to the difficulty of creating the necessary Java object from an array of bytes. I chose that message style because I've implemented it a hundred times in C. Now I've done it once in Java and I promise I won't do it again. There are no pointers in Java and you can't cast between different types, which includes char (16 bit UniCode) and byte (8 bit). Where in C, I'd have written a couple of memmove's, with appropriate casts, in Java I have to create various objects from copied sections of the array. The next version will undoubtedly go with a variable message length, variable field length, character delimited protocol. This will complicate the socket reading routine a little, but will allow me to use the supplied StringTokenizer class to parse the message.

With Java, one of the greatest temptations for someone who's built GUI apps before is to just go wild, creating windows left and right, changing the menubar and so on. Your ability to do so is often limited by the visibility (or lack thereof) of variables. For instance, to change the menubar, you need access to mbar in browser.hotjava. Access to components like mbar are architectural issues within HotJava that are not really settled yet.

Debugging

Debugging Java code is problematic, as there's no debugger. What you're left with is the tried and true 'print to standard out' style. Java encapsulates some of the common system functions in a System object so a module under development will be sprinkled with System.out.println() calls.

Documentation

The language documentation is very good. The class documentation, on the other hand, is very frustrating. I often followed a package-class-method documentation trail that left me at a page that contained only the method's name. You really have to follow the mailing lists (including the archives and the soon-to-be-created HotJava newsgroup) to get the most out of the Java/HotJava class packages. That said, for anyone who's spent time writing C++ code for the current crop of GUIs, awt's classes will seem like a very intuitive and straightforward abstraction of the native GUI capabilities.

I built Ship using HotJava Alpha 2 running under Windows NT 3.51. While Java itself is relatively stable and well-mannered, the browser (HotJava) and awt packages are still moving targets. There is already an Alpha3 running under Sun Solaris. Win95 and Mac versions are supposed to be right around the corner, and the beta release is scheduled for August.

Try It Yourself

If you have HotJava, you can run Battle of the Java Sea by checking into <http://XXXXXXXXX.mfi.com>.

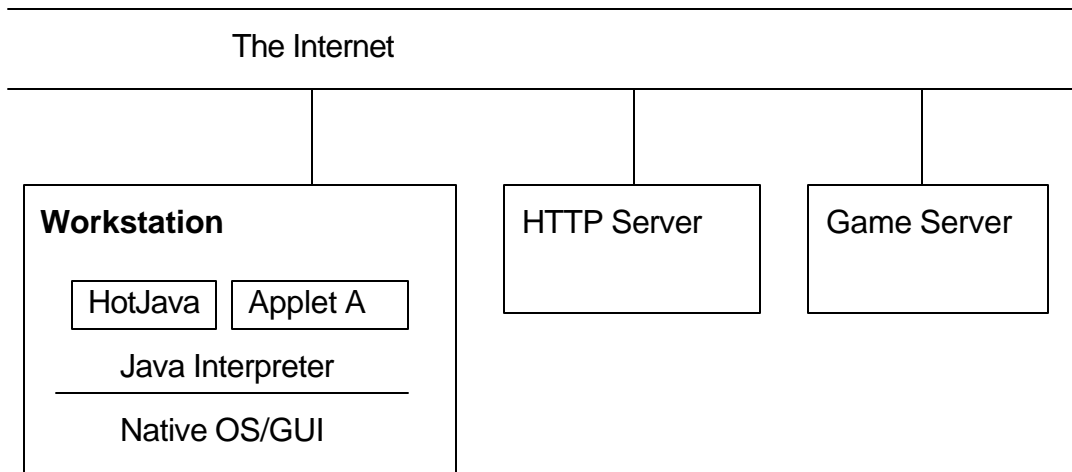


Figure 1: A Workstation running HotJava and one Applet (Applet A)

Time Slice	Thread 1	Thread 2
1	repaint	
2	clearRect(currentShipRect)	
3		keyDown(darrow)
4		newShipRect = currentShipRect;
5		newShipRect.y++;
6	currentShipRect = newShipRect	

Figure 2: Paint update

