NativeJ

Win32 Native Launcher Generator for Java

Version 1.0.5

Table of Contents

1 INTRODUCTION

1.1 OVERVIEW

NativeJ is a new product by CTech Software (http://ctech-software.hypermart.net) that generates Win32 native launchers for your Java applications. No more ugly batch files! The launchers generated by NativeJ look and behave exactly like native Win32 programs. They possess their own custom icons when viewed in Explorer, and they do not appear as another "java.exe" or "javaw.exe" in the process list. You can generate launchers that behave like console programs, or graphical programs, or even Win32 services.

Besides looking and feeling like native apps, these launchers also have another advantage: *speed*. Executing a batch file requires first loading up the command processor, which steps through each line of the batch file before it encounters the line which calls the Java frontend *java.exe* or *javaw.exe*. This frontend imposes further overhead as it parses the command-line arguments before loading the JVM with the proper parameters. The native launchers generated by NativeJ eliminates all these overhead and load the JVM directly, resulting in snappier program loading.

NativeJ is extremely easy to use. Simply specify the launcher type, the program icon, the JVM parameters, the Java application parameters etc. and NativeJ will generate a custom .EXE file for you that will launch your Java application. There is no need to write custom C code, or wrestle with a C compiler. Just click-and-go! It's that easy!

Despite its simplicity and ease-of-use, NativeJ is also extremely flexible. There are many parameters you can tweak and tune. You can choose to use the JRE installed on the target machine, or you can choose to bundle a specific JVM with your application. You can specify the baseline version of the JVM you need to work with. You can pass both the JVM and the Java application custom parameters. In short, you will find NativeJ an extremely capable assistant for your needs.

1.2 FEATURES

Some features of NativeJ are:

- Generates console and graphical launchers, as well as Win32 services.
- Simple click-and-go interface with no need to write custom code.
- Powerful and flexible, with many parameters to tweak and tune.

1.3 SYSTEM REQUIREMENTS

- Windows NT/2000/XP is required for running NativeJ. However, launchers generated by NativeJ will run on Windows 9x/ME as well.
- Supports Sun JRE 1.2/1.3/1.4, as well as IBM JRE 1.2/1.3.

2 USING NATIVEJ

2.1 INSTALLATION

Double-click on the *SETUP.EXE* file provided in the distribution media and follow the instructions on-screen to install NativeJ to a directory of your choice.

2.2 GENERATING A CONSOLE LAUNCHER

Note: The following example assumes you have the Java Runtime Environment (1.2 and above) installed on your machine.

In this example, we are going to learn how to generate a console launcher i.e. a Win32 executable that behaves just like another command-line program.

The Java program that we are going to create a launcher for is a simple "Hello, World" program. This program is called *Console.java*, and can be found in the *examples/* subdirectory.

File: Console.java

```
1:   package examples;
2:
3:   public class Console
4:   {
5:       public static void main(String[] args) throws Exception
6:       {
7:           if (args.length == 0)
8:               System.out.println("Hello World!");
9:           else
10:              System.out.println("Hello " + args[0] + "!");
11:      }
12:  }
```

This simple program prints out "*Hello, World!*" when run without any parameters.

```
C:\Program Files\NativeJ> java examples.Console
Hello, World!
```

However, when supplied with a name as a parameter, it will print out "*Hello, <name>!*".

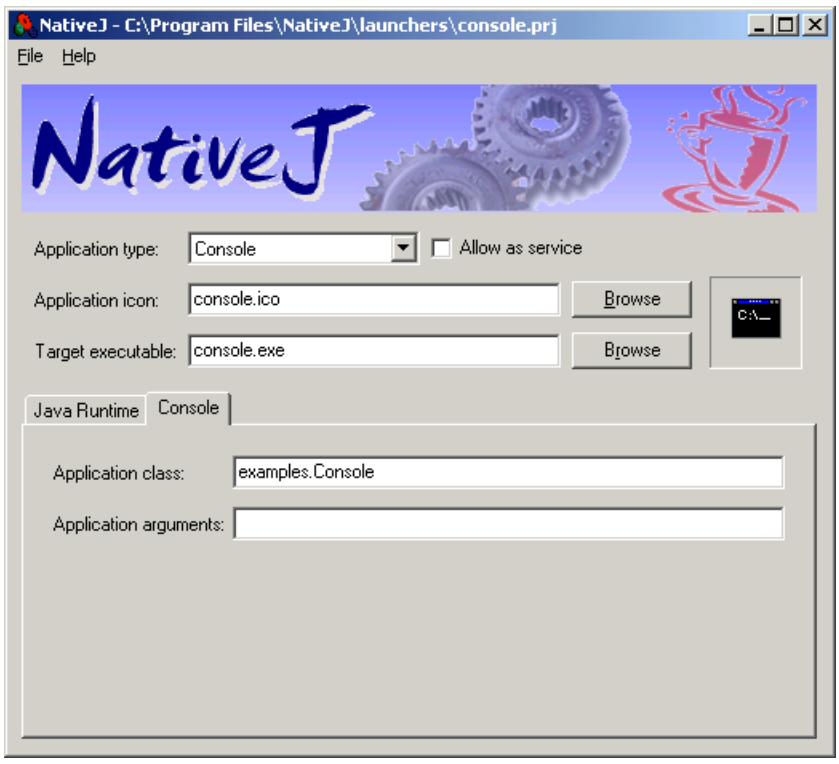
```
C:\Program Files\NativeJ> java examples.Console John
Hello, John!
```

Let's see how we can create a launcher called *console.exe*, which will run the *examples.Console* program.

The *Console.java* file has been compiled and packaged in *launchers/examples.jar*, along with the other examples. In this subdirectory, you will also find *console.prj*, along with other *.prj* files.

Run NativeJ. Then, click on "*File... Open...*", and open "*launchers\console.prj*". This is the project file for *examples.Console*, and it has the following parameters:

Name	Value	Remarks
Main		
Application type	`Console`	The launcher generated will run as a console program.
Allow as service	`<unchecked>`	The launcher generated will not run as a Win32 service.
Application icon	`console.ico`	The application icon.
Target executable	`console.exe`	The filename of the generated launcher.
Java Runtime		
Required version	`1.2`	The minimum required runtime version is 1.2. This argument only applies if the JVM DLL is left blank.
JVM DLL	`<blank>`	When left blank, the launcher will look for a suitable JRE on the target system. If you are bundling a JRE with your application, you can specify the path of your JVM DLL eg. *jre\bin\hotspot\jvm.dll*.
JVM arguments	`<blank>`	The arguments to be passed to the JVM. You should only pass the *-X* arguments to the JVM eg. *-Xms64m -Xmx128m*.
Classpath	`examples.jar`	This contains required class files i.e. *examples.Console*.
Console		
Application class	`examples.Console`	The name of the Java class we wish to run. This class should have a *main()* method.
Application arguments	`<blank>`	The default arguments we want to pass to *main()*.



Now, click on "*File... Generate...*" to generate the launcher. If everything goes well, you will have receive a notification that "*console.exe*" has been generated.

Run *console.exe* without any parameters:

```
C:\Program Files\NativeJ\launchers> console
Hello, World!
```

Now, run *console.exe* with a name:

```
C:\Program Files\NativeJ\launchers> console John
Hello, John!
```

You can also supply some default arguments to *examples.Console*. Enter *"John"* in "*Application arguments*", then generate the launcher again. Now, run *console.exe* without any parameters:

```
C:\Program Files\NativeJ\launchers> console
Hello, John!
```

Run *console.exe* with another name besides *John*:

```
C:\Program Files\NativeJ\launchers> console Mary
Hello, Mary!
```

As you can see, the launcher will use the default arguments if none are available. If the user supplies the arguments, then these are used instead of the default.

In this example, the *JVM DLL* parameter is left blank so that when *console.exe* is run, it will automatically detect and load a suitable JVM (in this case, any JVM version >= 1.2) on the target system. You can also bundle a specific JVM with your application and force the launcher to use that JVM. To find out how, please refer to *3.1 Bundling a Particular Java Runtime*.

2.3 Generating a Graphical Launcher

In this example, we are going to learn how to generate a graphical launcher i.e. a native executable that behaves like a Win32 GUI program.

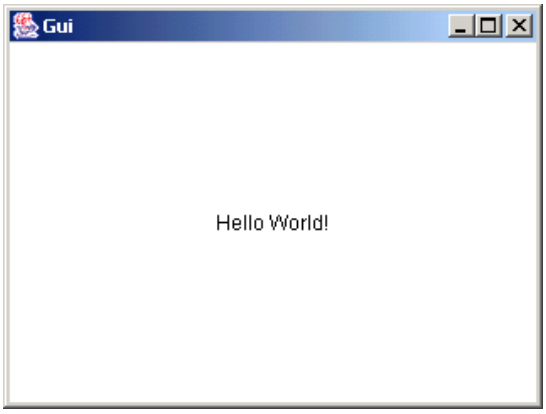
The Java program that we are going to create a launcher for is a simple "Hello, World" type AWT program. This program is called Gui.*java*, and can be found in the *examples/* subdirectory.

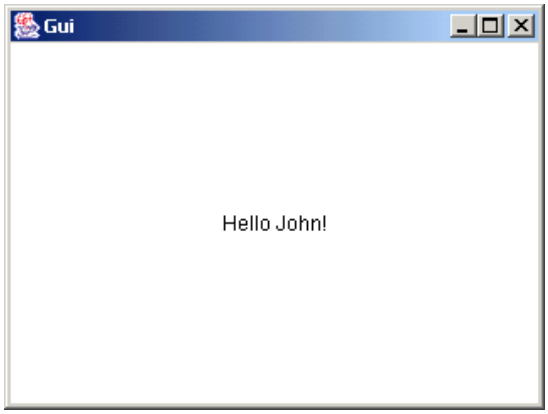
File: Gui.java

```
1:   package examples;
2:
3:   import java.awt.*;
4:   import java.awt.event.*;
5:
6:   public class Gui
7:   {
8:       public static void main(String[] args)
9:       {
10:          // Create the main window and components used by this app
11:          Frame frame = new Frame("Gui");
12:          String msg = "Hello World!";
13:          if (args.length > 0) msg = "Hello " + args[0] + "!";
14:          Label label = new Label(msg, Label.CENTER);
```

```
15:
16:          // Handle the exit event for the main window
17:          frame.addWindowListener(new WindowAdapter()
18:          {
19:              public void windowClosing(WindowEvent e)
20:              {
21:                  System.exit(0);
22:              }
23:          });
24:
25:          // Position the components within the main window
26:          frame.setLayout(new BorderLayout());
27:          frame.add(label, BorderLayout.CENTER);
28:
29:          // Resize and show main window
30:          frame.pack();
31:          frame.setSize(320, 240);
32:          frame.show();
33:      }
34:  }
```

When run without any parameters, this program displays a "*Hello, World!*" message. When run with a name as the parameter eg. *java examples.Gui John*, it will display "*Hello, <name>!*" instead.



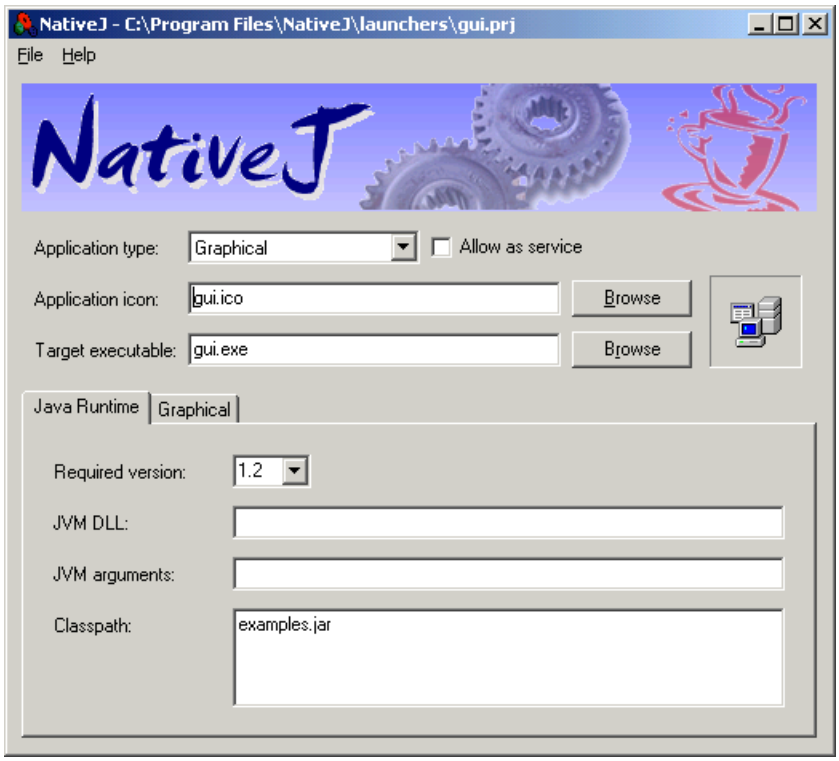


Let's see how we can create a launcher called *gui.exe*, which will run the *examples.Gui* program.

The *Gui.java* file has been compiled and packaged in *launchers/examples.jar*, along with the other examples. In this subdirectory, you will also find *gui.prj*, along with other *.prj* files.

Run NativeJ and open the *gui.prj* project. Confirm that it defines the following parameters:

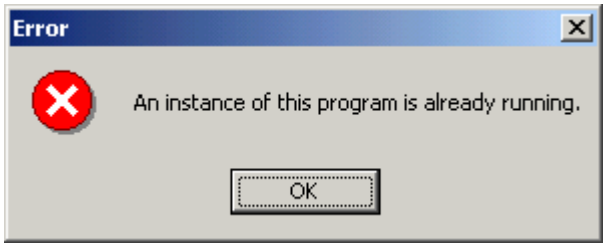
Name	Value	Remarks
Main		
Application type	`Graphical`	The launcher generated will run as a GUI program.
Allow as service	`<unchecked>`	The launcher generated will not run as a Win32 service.
Application icon	`gui.ico`	The application icon.
Target executable	`gui.exe`	The filename of the generated launcher.
Java Runtime		
Required version	`1.2`	The minimum required runtime version is 1.2. This argument only applies if the JVM DLL is left blank.
JVM DLL	`<blank>`	When left blank, the launcher will look for a suitable JRE on the target system. If you are bundling a JRE with your application, you can specify the path of your JVM DLL eg. *jre\bin\hotspot\jvm.dll*.
JVM arguments	`<blank>`	The arguments to be passed to the JVM. You should only pass the *-X* arguments to the JVM eg. *-Xms64m -Xmx128m*.
Classpath	`examples.jar`	This contains required class files i.e. *examples.Gui*.
Graphical		
Application class	`examples.Gui`	The name of the Java class we wish to run. This class should have a *main()* method.
Application arguments	`<blank>`	The default arguments we want to pass to *main()*.
Allow multiple instances	`<checked>`	Whether to allow multiple instances of the launcher. In the default case, yes.



Now, click on "*File... Generate...*" to generate the launcher. If everything goes well, you will have receive a notification that "*gui.exe"* has been generated.

Run *gui.exe* without any parameters, and with the name "*John*". You should observe the same output as when running the program using *java*.

Now, run *gui.exe* twice without quitting. You should see two instances of the program. Uncheck the *Allow multiple instances* option, and generate *gui.exe* again. This time, you should get the following error if you try to run *gui.exe* twice.



Notice that in line 21 of *Gui.java*, the program executes *System.exit(0)* to terminate the program. This practice is typical of AWT or Swing programs, and widely used in Sun's own sample codes. However, this has the effect of terminating the launcher prematurely as well, which might not be desirable in certain cases (for example, when your program is doubling as a Win32 service). For more details on why this happens and how to workaround this behaviour, please refer to *3.4 Avoiding System.Exit()*.

2.4 Generating a Service Launcher

In this example, we are going to learn how to generate a service launcher i.e. an executable that could be installed as a service and started/stopped using the *Services* control panel applet. Noted that services are only available on Windows NT/2K/XP, and not on the Win9x platforms.

The Java program that we are going to use is a simple console program that continually logs the date/time to a file called *service.log* at 5 seconds interval. The *main()* method takes a *-start* argument to start the logging activity, and a *-stop* argument to stop the logging.

File: Service.java

```
1:   package examples;
2:
3:   import java.io.*;
4:   import java.util.*;
5:
6:   public class Service
7:   {
8:       static boolean stop = false;
9:       public static void main(String[] args) throws Exception
10:      {
11:          // Start the service
12:          if (args[0].equals("-start"))
13:          {
14:              while(true)
15:              {
16:                  // Append current date/time to log file
17:                  log("Current date/time is " + new Date());
18:
```

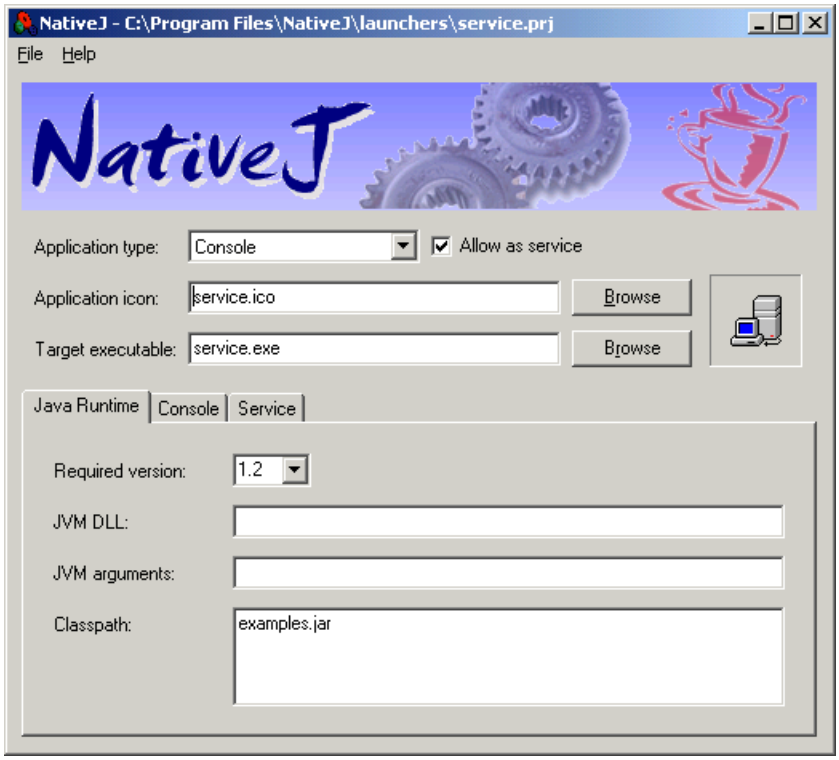
```
19:                   // Sleep for 5 secs
20:                   Thread.currentThread().sleep(5000);
21:
22:                   // Check for termination
23:                   if (stop)
24:                   {
25:                       log("Service stopped.");
26:                       break;
27:                   }
28:               }
29:           }
30:           else
31:           // Stop the service
32:           if (args[0].equals("-stop"))
33:           {
34:               // Set the termination flag
35:               stop = true;
36:           }
37:       }
38:
39:       /**
40:        * Log given string to file "service.log".
41:        */
42:       static void log(String msg) throws IOException
43:       {
44:           PrintWriter pw = new PrintWriter(
45:               new FileWriter("service.log", true));
46:           pw.println(msg);
47:           pw.close();
48:       }
49:   }
```

Let's generate a launcher called *service.exe*, which will run our *examples.Service* program.

The *Service.java* file has been compiled and packaged in *launchers/examples.jar*, along with the other examples. In this subdirectory, you will also find *service.prj*, along with other *.prj* files.

Run NativeJ and open the *service.prj* project. The project defines the following parameters:

Name	Value	Remarks
Main		
Application type	`Console`	The launcher generated will run in the console mode.
Allow as service	`<checked>`	The launcher generated will run as a Win32 service.
Application icon	`service.ico`	The application icon.
Target executable	`service.exe`	The filename of the generated launcher.
Java Runtime		
Required version	`1.2`	The minimum required runtime version is 1.2. This argument only applies if the JVM DLL is left blank.
JVM DLL	`<blank>`	When left blank, the launcher will look for a suitable JRE on the target system. If you are bundling a JRE with your application, you can specify the path of your JVM DLL eg. *jre\bin\hotspot\jvm.dll*.
JVM arguments	`<blank>`	The arguments to be passed to the JVM. You should only pass the *-X* arguments to the JVM eg. *-Xms64m -Xmx128m*.
Classpath	`examples.jar`	This contains required class files i.e. *examples.Service*.
Console		
Application class	`examples.Service`	The name of the Java class we wish to run. This class should have a *main()* method.
Application arguments	`<blank>`	The default arguments we want to pass to *main()*.
Service		
Service Name	`Test Service`	The service name which will appear in *Services* when installed.
Start arguments	`-start`	The argument(s) to be passed to the *main()* method to start the program.
Stop arguments	`-stop`	The argument(s) to be passed to the *main()* method to stop the program.

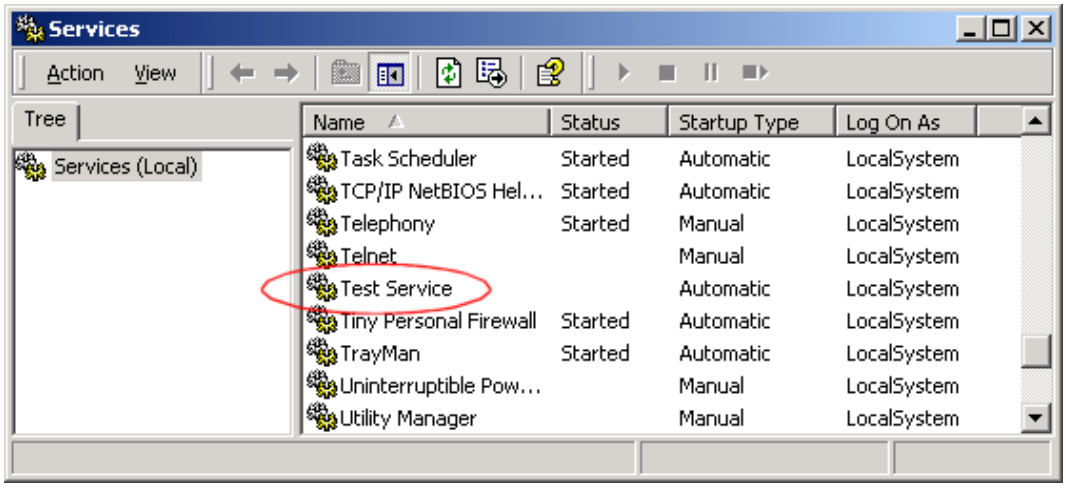


Now, click on "*File... Generate...*" to generate the launcher. If everything goes well, you will have receive a notification that "*service.exe*" has been generated.

Now use "*service -install*" to install the service. You should see the following message:

```
C:\Program Files\NativeJ\launchers>service -install
Service installed: Test Service
```

Check the services list using the *Services* control panel applet. You should also see a new entry called *Test Service*.



Now start *Test Service* using *Services*. You should find a file called *service.log* written to the same directory where *services.exe* resides. The file contains the date/time logged approximately every 5 seconds.

```
Current date/time is Sat Jun 01 22:20:04 GMT+08:00 2002
Current date/time is Sat Jun 01 22:20:09 GMT+08:00 2002
Current date/time is Sat Jun 01 22:20:14 GMT+08:00 2002
Current date/time is Sat Jun 01 22:20:19 GMT+08:00 2002
Current date/time is Sat Jun 01 22:20:24 GMT+08:00 2002
```

Now, stop the service. The last line of service.log should contain the words "*Service stopped*.", logged by line 25 of *Service.java*.

If you encounter any errors while starting or stopping the service, launch *Event Viewer* and look under *Application Log* for messages attributed to *Test Service*.

Use "*service -uninstall*" to uninstall *Test Service*.

If you run *examples.Service* directly using "*java examples.Service -start*", you won't be able to stop it using "*java examples.Service -stop*". That's because they will be running under two different JVM instances, which means the *stop* flags set to *true* in line 35 of *Service.java* will not be reflected in the first JVM instance. The NativeJ-generated launcher has no problems with this approach because it actually calls *main()* with the *-stop* argument through a thread within the same JVM instance, but you won't be able to do so when running using *java* or *javaw*.

If you wish to implement a start/stop mechanism that is generic and works under *java/javaw* (in fact, across all Java-enabled platforms), please refer to *3.5 Implementing a Generic Start/Stop Mechanism*.

2.5 Generating a Launcher for Tomcat

In this example, we are going to generate a launcher for Tomcat. Tomcat is a open-source reference servlet container developed by the Apache Group. It is widely used and is chosen as an example to show how easy it is to generate a launcher for a non-trivial Java application.

Since Tomcat is rather large and is constantly being improved, we chose not to bundle it with NativeJ. Instead, you can download the latest source/binary builds from the following URL: *http://jakarta.apache.org/tomcat/index.html*. This example applies to Tomcat 4.0.x, though it can be easily adopted for other version of Tomcat..

The downloaded binary distribution should be in the form *jakarta-tomcat-4.0.x.zip*. Unzip it into the *launchers/* subdirectory. The base directory is *jakarta-tomcat-4.0.3/*. For simplicity, rename it to *tomcat/*. When you are done, the directory structure should look like this:

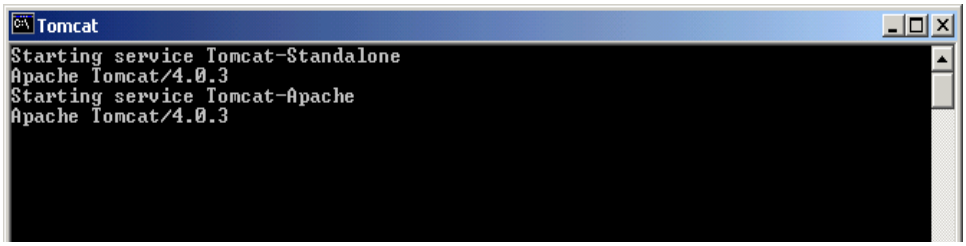
```
C:\Program Files\NativeJ\
  launchers\
    tomcat\
```

Now let's test if Tomcat is working properly.

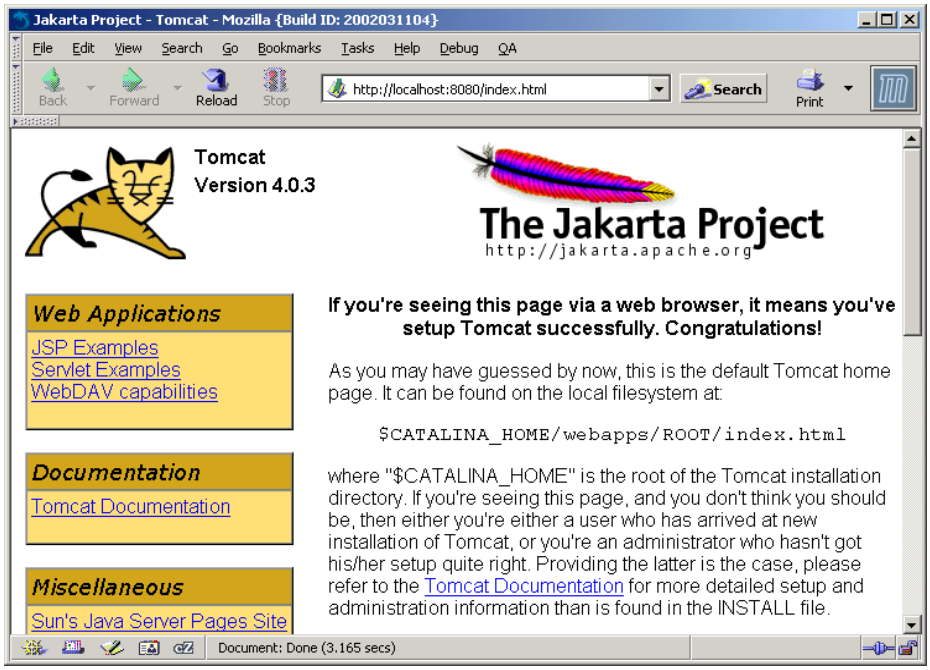
```
> cd C:\Program Files\NativeJ\launchers\tomcat
> set JAVA_HOME=C:JDK
```

```
> set CATALINA_HOME=C:\Program Files\NativeJ\launchers\tomcat
> bin\startup.bat
```

If all goes well, a separate command window will pop up:



You should now be able to access the Tomcat homepage at *http://localhost:8080/*.



Use *bin/shutdown.bat* to stop Tomcat.

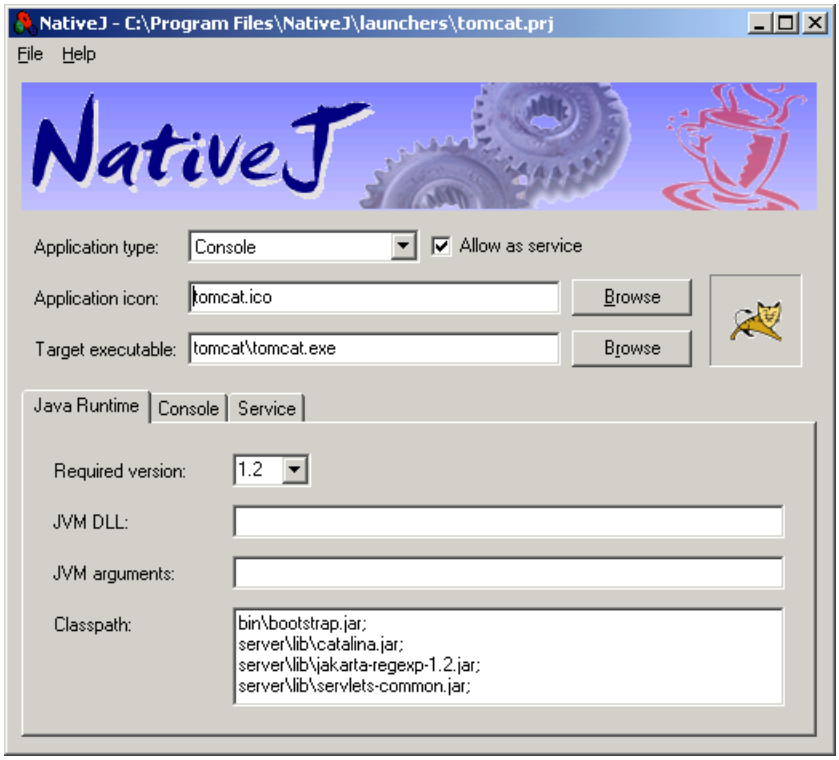
Now, launch NativeJ, and populate the new project with the following parameters:

Name	Value	Remarks
Main		
Application type	`Console`	The launcher generated will run as a console program.
Allow as service	`<checked>`	The launcher generated will run as a Win32 service as well.
Application icon	`tomcat.ico`	The application icon.
Target executable	`tomcat\tomcat.exe`	The filename of the generated launcher.
Java Runtime		

Name	Value	Remarks
Required version	`1.2`	The minimum required runtime version is 1.2. This argument only applies if the JVM DLL is left blank.
JVM DLL	`<blank>`	When left blank, the launcher will look for a suitable JRE on the target system. If you are bundling a JRE with your application, you can specify the path of your JVM DLL eg. *jre\bin\hotspot\jvm.dll*.
JVM arguments	`<blank>`	The arguments to be passed to the JVM. You should only pass the *-X* arguments to the JVM eg. *-Xms64m -Xmx128m*.

Classpath	`bin\bootstrap.jar;`
	`server\lib\catalina.jar;`
	`server\lib\jakarta-regexp-1.2.jar;`
	`server\lib\servlets-common.jar;`
	`server\lib\servlets-default.jar;`
	`server\lib\servlets-invoker.jar;`
	`server\lib\servlets-manager.jar;`
	`server\lib\servlets-snoop.jar;`
	`server\lib\servlets-webdav.jar;`
	`server\lib\tomcat-ajp.jar;`
	`server\lib\tomcat-util.jar;`
	`server\lib\warp.jar;`
	`common\lib\activation.jar;`
	`common\lib\jdbc2_0-stdext.jar;`
	`common\lib\jndi.jar;`
	`common\lib\jta-spec1_0_1.jar;`
	`common\lib\mail.jar;`
	`common\lib\naming-common.jar;`
	`common\lib\naming-resources.jar;`
	`common\lib\servlet.jar;`
	`common\lib\tyrex-0.9.7.0.jar;`
	`common\lib\xerces.jar`
		The required *.jar* files.

Console		
Application class	`org.apache.catalina.startup.Bootstrap`	The name of the Java class we wish to run. This class should have a *main()* method.
Application arguments	`start`	The default arguments we want to pass to *main()*.
Service		
Service name	`Tomcat`	The service name which will appear in *Services* when installed.
Start arguments	`start`	The argument(s) to be passed to the *main()* method to start the program.
Stop arguments	`stop`	The argument(s) to be passed to the *main()* method to stop the program.

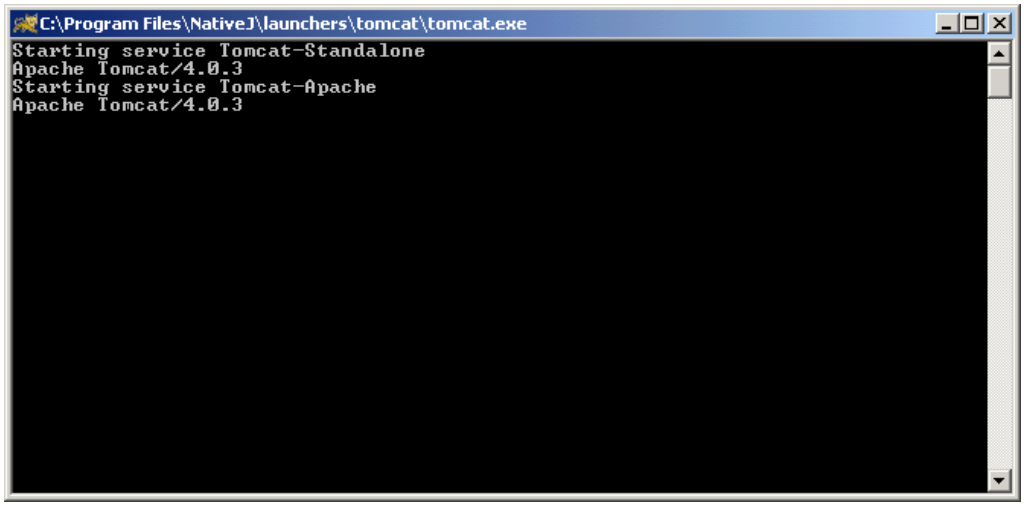


Now, click on "*File... Generate...*" to generate the launcher. If everything goes well, you will have receive a notification that "*tomcat.exe*" has been generated.

The Tomcat launcher can function *both* as a console program, as well as a Win32 service.

To start Tomcat from the command line, type:

```
C:\Program Files\NativeJ\launchers\tomcat> start tomcat
```



To stop Tomcat from the command line, type:

```
C:\Program Files\NativeJ\launchers\tomcat> tomcat stop
```

To install Tomcat as a service, type:

```
C:\Program Files\NativeJ\launchers> tomcat -install
Service installed: Tomcat
```

To uninstall the Tomcat service, type:

```
C:\Program Files\NativeJ\launchers> tomcat -uninstall
Service uninstalled: Tomcat
```

Note that to execute JSP scripts when running Tomcat in this way, make sure you copy the *\jdk\lib\tools.jar* file to *\jdk\jre\lib\ext*. That's because by default the JRE does not contain the compiler class *com.sun.tools.javac.Main* (contained in *\jdk\lib\tools.jar*). But executing JSP scripts require the Java compiler class to be present, so you will need to copy this *.jar* file to the JRE subdirectory *ext*, which is reserved for extension libraries.

3 ADVANCED TOPICS

3.1 BUNDLING A PARTICULAR JAVA RUNTIME

All the examples given in *2 Using NativeJ* generate launchers that detects and use the default JRE installed on the target machine. This is done by leaving the *JVM DLL* blank. The generated launcher will find and use the first JRE installed on the target machine which meets the JVM version requirement.

It is also possible for you to bundle a particular JRE with your application, and generate a launcher that uses the bundled JRE regardless of other JREs are installed on the target machine.

There are a few advantages to bundling a specific JRE with your application.

- Since one can never be certain of which make and version of JRE is installed on the target machine (if indeed one is installed at all), the safest bet is to bundle your own JRE so that you can be sure the application will work flawlessly, since you have ultimate control of which make or version of JRE to bundle with your application.

- If the user does not already have a JRE installed on the target machine, bundling a JRE spares him the inconvenience of downloading a JRE himself. This is especially true if the user might not have broadband access to the Internet.

- You can also choose to include the JRE installer with your distribution. However, this requires the user to run another installer, which might pose an inconvenience to the user. You can of course choose to run the JRE installer automatically with your application installer by detecting if a JRE is already present on the target machine. However this makes the application much more complexinstaller. The installer needs to detect whether a JRE is already available on the target system, whether the make and version is up to the mark, and whether it is alright to overwrite the current JRE etc.

- You might also be forced to bundle a JRE with your application if your application depends on an older JRE (or a particular make/version of a JRE) to work. Since there is always the possibility that the user already has the latest JRE installed on his machine, it will be unacceptable to expect him to replace his JRE with an older version just so that he can run your application.

The primary concern with bundling your own JRE is the footprint of your application. Bundling your own JRE potentially increases your distribution and installation footprint by tens of megabytes. If the application needs to be delivered via the Internet, or the target system has diskspace constraints, then bundling your own JRE might not be acceptable.

The decision of whether to bundle or not to bundle your own JRE depends on the few factors mentioned above. However, if you do choose to bundle your own JRE, it is very simple to do so with NativeJ.

Let's say we decide to bundle a particular JRE for our first example program, *examples.Console*. Originally, the application directory contains only the launcher *.exe* file and

examples.jar.

```
approot\
  console.exe
  examples.jar
```

Let's copy the JRE from an existing JDK installed on your machine. The directory structure of the JDK should look like this:

```
jdk\
  bin\
  help\
  include\
  jre\
  lib\
```

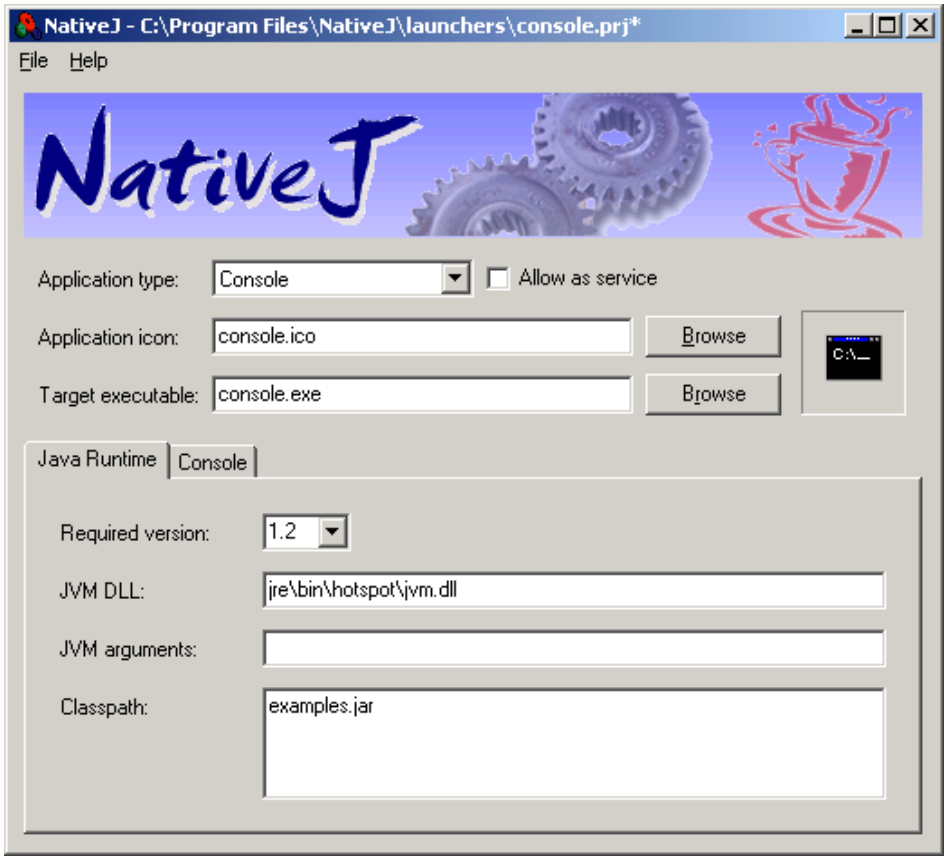
The JRE is found in the *jre/* subdirectory. Copy the *jre* subdirectory to your application directory. The directory structure of your application directory should now look like:

```
approot\
  jre\
    bin\
      classic\
        jvm.dll
      hotspot\
        jvm.dll
      server\
        jvm.dll
    lib\
  console.exe
  examples.jar
```

The *jvm.dll* under *jre\bin\classic*, *jre\bin\hotspot* and *jre\bin\server*) represents the different JVMs which you can use. The *classic* JVM refers to the non-JIT, interpreted JVM. The *hotspot* JVM uses Sun's HotSpot JIT technology, and is optimized for client applications. The *server* JVM is also a HotSpot-JIT JVM, but is optimized for server applications.

Note that the structure above varies according to the make and version of the JRE. For example, in Sun's JDK 1.2, the classic JVM refers to a JIT JVM licensed from Symantec. The Hotspot JVMs were only introduced in in JDK 1.3. Similarly, the structure for IBM's JDK, or JDK from other vendors could be different.

Once you have copied the *jre/* subdirectory from the JDK installation directory to your application directory, you can generate a launcher that will make use of that particular JRE. Just specify the relative path to the desired JVM DLL file in NativeJ, and generate the launcher. For example, if you specify *jre\bin\hotspot\jvm.dll* in the *JVM DLL* parameter box, it means you want to use the client-optmized HotSpot JVM for your application.



3.2 Passing Arguments to the Java Runtime

For certain situations, you may need to pass some arguments to the JVM. For example, to increase the maximum amount of memory available to the JVM, we typically use:

```
java -ms64m -mx128m examples.Console
```

This means the initial memory available to the JVM is 64MB, while the maximum amount of memory available to the JVM is 128MB. This is frequently necessary when running complex server applications.

The typical options available in *java.exe* are as follows:

```
C:\java -h
Usage: java [-options] class [args...]
           (to execute a class)
   or  java -jar [-options] jarfile [args...]
           (to execute a jar file)

where options include:
    -client       to select the "client" VM
    -server       to select the "server" VM
    -hotspot      is a synonym for the "client" VM  [deprecated]
                  The default VM is client.

    -cp -classpath <directories and zip/jar files separated by ;>
                  set search path for application classes and resources
    -D<name>=<value>
```

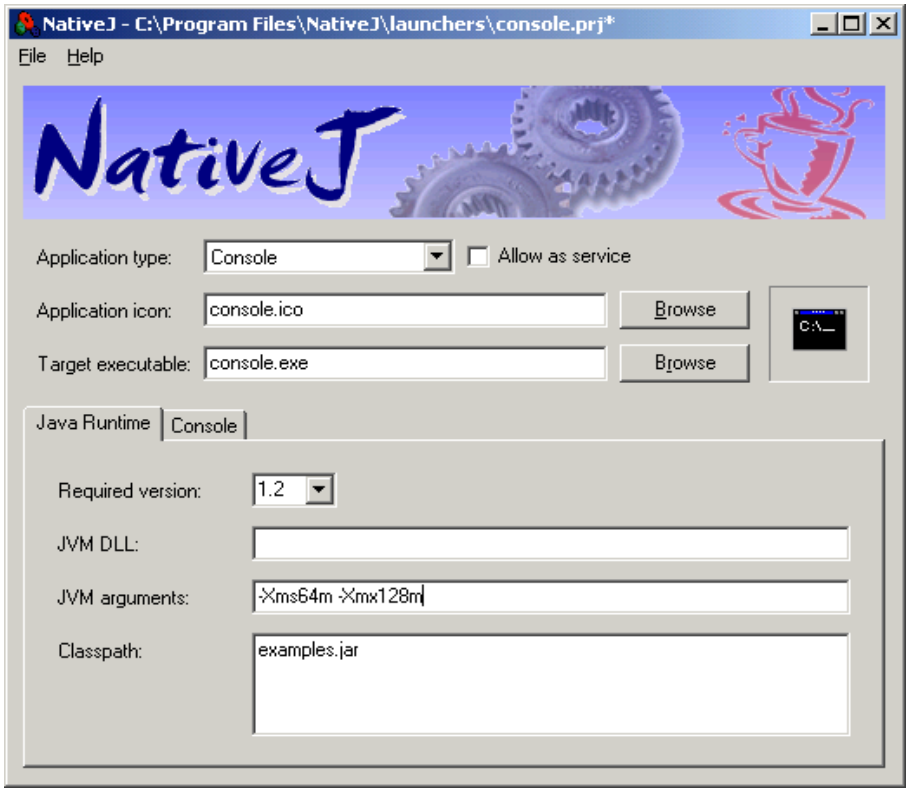
```
                  set a system property
    -verbose[:class|gc|jni]
                  enable verbose output
    -version      print product version and exit
    -showversion  print product version and continue
    -? -help      print this help message
    -X            print help on non-standard options
    -ea[:<packagename>...|:<classname>]
    -enableassertions[:<packagename>...|:<classname>]
                  enable assertions
    -da[:<packagename>...|:<classname>]
    -disableassertions[:<packagename>...|:<classname>]
                  disable assertions
    -esa | -enablesystemassertions
                  enable system assertions
    -dsa | -disablesystemassertions
                  disable system assertions
```

However, because NativeJ's launcher operate as a lower level than *java.exe*, you cannot use the standard options such as *-ms*, *-mx*, *-cp* etc. Instead, you will need to use non-standard options prefix by *-X*.

```
C:\java -X
    -Xmixed           mixed mode execution (default)
    -Xint             interpreted mode execution only
    -Xbootclasspath:<directories and zip/jar files separated by ;>
                      set search path for bootstrap classes and resources
    -Xbootclasspath/a:<directories and zip/jar files separated by ;>
                      append to end of bootstrap class path
    -Xbootclasspath/p:<directories and zip/jar files separated by ;>
                      prepend in front of bootstrap class path
    -Xnoclassgc       disable class garbage collection
    -Xincgc           enable incremental garbage collection
    -Xloggc:<file>    log GC status to a file with time stamps
    -Xbatch           disable background compilation
    -Xms<size>        set initial Java heap size
    -Xmx<size>        set maximum Java heap size
    -Xss<size>        set java thread stack size
    -Xprof            output cpu profiling data
    -Xrunhprof[:help]|[:<option>=<value>, ...]
                      perform JVMPI heap, cpu, or monitor profiling
    -Xdebug           enable remote debugging
    -Xfuture          enable strictest checks, anticipating future default
    -Xrs              reduce use of OS signals by Java/VM
    -Xcheck:jni       perform additional checks for JNI functions

The -X options are non-standard and subject to change without notice.
```

Hence, to increase the maximum amount of memory available to an application under NativeJ, you need to specify *-Xms64m -Xmx128m* under the *JVM arguments* parameter box.



Since the *-X* arguments are *non-standard* arguments and vary across different makes and versions of JVMs, this approach works best if you are bundling a particular JRE so that you can be sure that the *-X* arguments that you supply will work with the bundled JVM.

3.3 An Alternative Approach to Classpath

If you are bundling a particular JRE with your application, it may not be necessary for you to specify the *.jar* files your application uses in the *Classpath* parameter box.The JRE provides a repositoty in *jre\lib\ext* where these files can reside so that they can be automatically recognized and loaded by the runtime.

In the directory structure below, *examples.jar* has been moved to *jre\lib\ext*. Hence there is no longer any need to specify it in the *Classpath* parameter box.

```
approot\
  jre\
    bin\
      classic\
        jvm.dll
      hotspot\
        jvm.dll
      server\
        jvm.dll
    lib\
      ext\
        examples.jar
  console.exe
```

3.4 Avoiding System.Exit()

In a typical Java graphical app where the AWT (Abstract Windowing Toolkit) is involved, the program almost always calls *System.exit()* to terminate the program. This applies also to Swing apps, since Swing is based on the AWT. Why is this so?

Java programs are based heavily on threads. Some threads are user-created, while other threads are system-created. Regardless of who creates these threads, they can be divided into two broad categories: *daemon* and *non-daemon*. The JVM will terminate when all non-deamon threads in the virtual machine has terminated i.e. even if there are threads still running, as long as these threads are all daemon threads, and all non-daemon threads have died, the JVM will terminate.

Let's look at a simple console app such as *examples.Console*.

File: Console.java

```
1:   package examples;
2:
3:   /**
4:    * This is a sample Java program that runs in console mode.
5:    */
6:   public class Console
7:   {
8:       public static void main(String[] args) throws Exception
9:       {
10:          if (args.length == 0)
11:              System.out.println("Hello World!");
12:          else
13:              System.out.println("Hello " + args[0] + "!");
14:      }
15:  }
```

There is only one non-daemon thread in this program, which is the one running the *main()* method. When the *main()* method terminates, that non-daemon thread dies, which means the JVM terminates thereafter, even though there is no explicit *System.exit()* statement.

Let us now examine our sample graphical app: *examples.Gui*.

File: Gui.java

```
1:   package examples;
2:
3:   import java.awt.*;
4:   import java.awt.event.*;
5:
6:   public class Gui
7:   {
8:       public static void main(String[] args)
9:       {
10:          // Create the main window and components used by this app
11:          Frame frame = new Frame("Gui");
12:          String msg = "Hello World!";
13:          if (args.length > 0) msg = "Hello " + args[0] + "!";
14:          Label label = new Label(msg, Label.CENTER);
15:
16:          // Handle the exit event for the main window
17:          frame.addWindowListener(new WindowAdapter()
18:          {
19:              public void windowClosing(WindowEvent e)
20:              {
21:                  System.exit(0);
```

```
22:             }
23:         });
24:
25:         // Position the components within the main window
26:         frame.setLayout(new BorderLayout());
27:         frame.add(label, BorderLayout.CENTER);
28:
29:         // Resize and show main window
30:         frame.pack();
31:         frame.setSize(320, 240);
32:         frame.show();
33:     }
34: }
```

Notice that the *main()* method exits at line 34, which means the non-daemon thread running *main()* also terminates. Why is it that the program is still running and keeping the main window alive until the user hits on the [x] button to close the window?

The reason is because the JVM creates an *AWT thread* for event procesisng whenever AWT components are created in a method. There is only one instance of the AWT thread, so it is instantiated at the first creation of an AWT component. It does not matter if the component is ultimately displayed on-screen or not. As long as an AWT component is created, the AWT thread comes alive if it does not already exist.

The thing to note is that whether the AWT thread is daemon or non-daemon depends on the original thread running the method that creates the AWT component. In the example above, this happens in the *main()* method, which is run by a non-daemon thread. Hence the AWT thread also becomes non-daemon. When the *main()* method terminates, the AWT thread is still alive, and since it is non-daemon, the JVM does not terminate.

What is the problem with using *System.exit()*? Calling this function terminates the program immediately, with no ifs or buts. However, this also terminates the NativeJ-generated launcher there and then, and all the housekeeping logic in the launcher for program termination does not get executed.

If your Java GUI program is running as a standalone application, this is mostly alright. The housekeeping logic will simple unload the JVM DLL and release any allocated memory. Terminating the launcher program achieves the same purpose under modern Win32 OSes.

However, if your Java GUI program is doubling as a Win32 service, this will pose a problem, since the housekeeping logic that interacts with the Service Manager will not get run. As a result, the status of the service is not accurately reflected when the service is stopped.

It is rather easy to modify *Gui.java* to avoid calling *System.exit()*. The trick is to make the AWT thread a daemon thread, and to prevent the *main()* method from terminating until the the exit event is received. These changes are shown in *Gui2.java*.

File: Gui2.java

```
1:   package examples;
2:
3:   import java.awt.*;
4:   import java.awt.event.*;
5:
6:   public class Gui2 extends Thread
7:   {
8:       /**
```

```
9:         * The main() method delegates the configuration and display of
10:        * main window to the "setup" thread. It waits for the "setup"
11:        * thread to terminate by doing a join().
12:        */
13:       public static void main(String[] args)
14:       {
15:           Gui2 setup = new Gui2(args);
16:           setup.start();
17:           try { setup.join(); } catch(InterruptedException e) {}
18:       }
19:
20:       /**
21:        * This is our only chance to set this thread to daemon mode
22:        * i.e. before the thread is started.
23:        */
24:       String msg;
25:       public Gui2(String[] args)
26:       {
27:           setDaemon(true);
28:           msg = "Hello World!";
29:           if (args.length > 0) msg = "Hello " + args[0] + "!";
30:       }
31:
32:       /**
33:        * When the thread is started, configure and display the
34:        * main window. Then it calls join() and waits to be
35:        * interrupted when the exit event for the main window is
36:        * triggeered.
37:        */
38:       Frame frame = null;
39:       public void run()
40:       {
41:           // Create the main window and components used by this app
42:           frame = new Frame("Gui2");
43:           Label label = new Label(msg, Label.CENTER);
44:
45:           // Handle the exit event for the main window
46:           frame.addWindowListener(new WindowAdapter()
47:           {
48:               public void windowClosing(WindowEvent e)
49:               {
50:                   frame.dispose();
51:                   Gui2.this.interrupt();
52:               }
53:           });
54:
55:           // Position the components within the main window
56:           frame.setLayout(new BorderLayout());
57:           frame.add(label, BorderLayout.CENTER);
58:
59:           // Resize and show main window
60:           frame.pack();
61:           frame.setSize(320, 240);
62:           frame.show();
63:
64:           // Wait for thread to be interrupted
65:           try { join(); } catch(InterruptedException e) {}
66:       }
67:   }
```

Notice how the *Gui2* class is now a thread, and all the AWT setup logic is moved to the *Gui2.run()* method (line 39). The first thing that happens in the *Gui2* constructor is to set the thread to daemon mode (line 27). Once a thread starts running, it is no longer possible to change its daemon status.

So *main()* instantiates a *Gui2* thread (line 15), and calls *start()* to run the thread. (line 16).

Then it issues a *join()* on the thread (line 17) to wait for the thread to terminate.

The *run()* method *Gui2* does all the AWT setup and causes the main window to appear in line 62. Since *Gui2* is a daemon thread, the AWT thread also becomes a daemon thread. Once this is done, the *Gui2* thread calls *join()* to suspend itself (line 65) and wait for an *InterruptedException*, which is raised when the *interrupt()* method is called on the thread.

There are now three threads in the system:

- The *main()* thread (non-daemon)
- The *Gui2* thread (daemon)
- The AWT thread (daemon)

The *main()* thread is the only thread preventing the JVM from terminating, and it is waiting for *Gui2* to terminate. On the other hand, *Gui2* is waiting for an *InterruptedException*, which will only be issued in line 51 when the exit event for the main window is triggered.

Now, when the user clicks on the [x] button in the main window to terminate the program, the main frame will be disposed, causing it to disappear from view (line 50). Then an *interrupted()* is issued (line 51), which causes the Gui2 thread to come out of its *join()* in line 65 and terminate. When this happens, the *main()* thread exits from its *join()* at line 17 and terminates too.

This leaves the AWT thread as the only surviving thread in the JVM. Since it is a daemon thread, and all the non-daemon threads have terminated, the JVM will terminate. No *System.exit()* calls!

3.5 Implementing a Generic Start/Stop Mechanism

In *Service.java*, the *main()* method accepts two arguments *-start* and *-stop*. The *-start* argument activates a loop which continually logs the current date/time to a log file called *service.log*. On the other hand, the *-stop* argument simply sets a *stop* flag to *true* at line 35. When the main program loop "sees" the *stop* flag, it will terminate (line 23–27).

File: Service.java

```
1:   package examples;
2:
3:   import java.io.*;
4:   import java.util.*;
5:
6:   public class Service
7:   {
8:       static boolean stop = false;
9:       public static void main(String[] args) throws Exception
10:      {
11:          // Start the service
12:          if (args[0].equals("-start"))
13:          {
14:              while(true)
15:              {
16:                  // Append current date/time to log file
17:                  log("Current date/time is " + new Date());
```

```
18:
19:                 // Sleep for 5 secs
20:                 Thread.currentThread().sleep(5000);
21:
22:                 // Check for termination
23:                 if (stop)
24:                 {
25:                     log("Service stopped.");
26:                     break;
27:                 }
28:             }
29:         }
30:         else
31:         // Stop the service
32:         if (args[0].equals("-stop"))
33:         {
34:             // Set the termination flag
35:             stop = true;
36:         }
37:     }
38:
39:     /**
40:      * Log given string to file "service.log".
41:      */
42:     static void log(String msg) throws IOException
43:     {
44:         PrintWriter pw = new PrintWriter(
45:             new FileWriter("service.log", true));
46:         pw.println(msg);
47:         pw.close();
48:     }
49: }
```

This approach works with NativeJ-generated launchers, but it will not work when moved to another platform such as Solaris, where shell scripts are the norm. This is because the *-start* and *-stop* arguments will be passed to different JVM instances, which means they will be executing in different address space.

A more platform-independent way of terminating a Java program that is meant to run continuously as a service (Win32) or daemon (Unix) is to implement some form of IPC (inter-process communication). An example is given in *Service2.java*, which uses IP datagrams to communicate the intent for program termination.

File: Service2.java

```
1:   package examples;
2:
3:   import java.io.*;
4:   import java.net.*;
5:   import java.util.*;
6:
7:   public class Service2
8:   {
9:       /**
10:      * This is the port over which the termination signal is sent.
11:      */
12:     private final static int port = 5678;
13:
14:     /**
15:      * This is the message to be sent to signal termination.
16:      */
17:     private final static String terminator = "QUIT";
18:
19:
20:     /**
```

```
21:       * The main() function accepts one single parameter:
22:       * "start" or "stop".
23:       * The first parameter starts the date/time logging service,
24:       * while the second parameter stops the service.
25:       */
26:      public static void main(String[] args) throws Exception
27:      {
28:          // Start the service
29:          if (args[0].equals("start"))
30:          {
31:              try
32:              {
33:                  // Run the termination listener thread
34:                  TerminationListener t = new TerminationListener(
35:                      port, terminator, Thread.currentThread());
36:                  t.start();
37:
38:                  // Start the logging service
39:                  while(true)
40:                  {
41:                      // Append current date/time to log file
42:                      log("Current date/time is " + new Date());
43:
44:                      // Sleep for 5 secs
45:                      Thread.currentThread().sleep(5000);
46:                  }
47:              }
48:              catch(InterruptedException e)
49:              {
50:                  // Exit when thread is interrupted.
51:                  log("Service stopped.");
52:              }
53:          }
54:          else
55:          // Stop the service
56:          if (args[0].equals("stop"))
57:          {
58:              try
59:              {
60:                  // Send the termination message
61:                  DatagramSocket socket = new DatagramSocket();
62:                  DatagramPacket packet = new DatagramPacket(
63:                      terminator.getBytes(), terminator.length(),
64:                      InetAddress.getLocalHost(), port);
65:                  socket.send(packet);
66:              }
67:              catch(Exception e)
68:              {
69:                  e.printStackTrace();
70:              }
71:          }
72:      }
73:
74:      /**
75:       * Log given string to file "service.log".
76:       */
77:      static void log(String msg) throws IOException
78:      {
79:          PrintWriter pw = new PrintWriter(
80:              new FileWriter("service.log", true));
81:          pw.println(msg);
82:          pw.close();
83:      }
84:  }
85:
86:  /**
87:   * This is a thread that will listen for the termination message
88:   * over the designated port, then interrupt the parent thread.
89:   */
```

```
90:  class TerminationListener extends Thread
91:  {
92:      Thread parent;
93:      int port;
94:      String terminator;
95:
96:      /**
97:       * Set this thread to daemon mode so that if for some reason
98:       * the main thread exists, this thread will not prevent the
99:       * JVM from terminating.
100:      */
101:     public TerminationListener(
102:         int port, String terminator, Thread parent)
103:     {
104:         setDaemon(true);
105:         this.port = port;
106:         this.terminator = terminator;
107:         this.parent = parent;
108:     }
109:
110:     /**
111:      * Listen for the termination signal over the designated port.
112:      */
113:     public void run()
114:     {
115:         try
116:         {
117:             // Setup datagram socket
118:             DatagramSocket socket = new DatagramSocket(port);
119:             DatagramPacket packet = new DatagramPacket(
120:                 new byte[terminator.length()], terminator.length());
121:
122:             // Stop only when we have received the termination message
123:             while(true)
124:             {
125:                 // Wait for a message
126:                 socket.receive(packet);
127:                 String msg = new String(packet.getData());
128:
129:                 // Make sure the message is coming from the same
130:                 // machine. This is included for additional security
131:                 // so that the program cannot be terminated from an
132:                 // external machine).
133:                 if (!packet.getAddress().equals(
134:                     InetAddress.getLocalHost()))
135:                         continue;
136:
137:                 // Make sure the message is the termination message
138:                 if (!msg.equals(terminator)) continue;
139:
140:                 // Interrupt parent thread
141:                 parent.interrupt();
142:
143:                 // Terminate this thread
144:                 break;
145:             }
146:         }
147:         catch(Exception e)
148:         {
149:         }
150:     }
151: }
```

A *TerminationListener* class is defined in line 90, that runs in the background and waits for an IP datagram from a predefined port (default: 5678) on the local machine. This is instantiated and started in lines 34-36 when the *-start* argument is used. Let's say this runs in JVM instance #1.

When the *-stop* argument is issued, this runs in JVM instance #2. The instructions in lines 61-65 will send out an IP datagram containing the message "QUIT". This will be routed over to the *TerminationListener* in JVM instance #1, logging activities will stop, and JVM instance #1 will teminate. Similarly, once JVM instance #2 has sent out the "QUIT" message, it will also terminate.

This approach to graceful termination of a server app will work under NativeJ, as well as all Java-enabled platforms with TCP/IP capability. Hence, you will have one set of source codes that can work with both NativeJ-generated Win32 launchers, as well as on other platforms using more traditional batch files, shell scripts, or just plain "*java <class>*".