An Introduction to Computer Science with Java

Contents Chapter 14 – Java Software Deployment

Introduction to Computer Science with Java

Chapter 14 – Java Software Deployment

This short chapter discusses the deployment of Java software on modern computer systems, including the use of Java Archive files, Java Applets, Java Web Start technology, and the use of third-party software to deploy Java applications as traditional executable files.

Chapter Learning Outcomes

Upon completion of this chapter students should be able to:

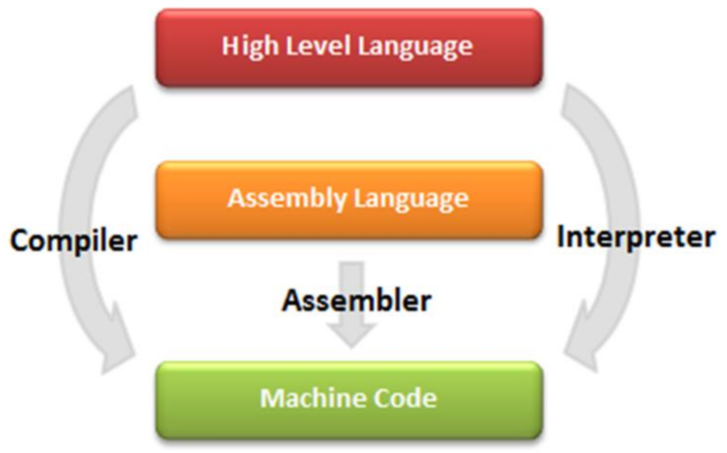
- describe what an executable file is and the traditional approach for deploying software.
- describe Java's approach for compiling and running software, the design goals that gave rise to this approach, advantages of this approach, and evidence that it has been successful.
- describe three *official* ways that finished Java software can be deployed for use on various systems.
- describe the difference between Java client-side and server-side applications, and the nature of enterprise applications and tiered applications.
- describe what a Java archive (JAR) file is and how to create JAR files.
- describe what a batch file (.bat) file is and how to create batch files.
- create JAR files, JAR files that use external resource files, and batch files that run JAR files which use console I/O.
- describe what a Java Applet is, the structure of a Java applet, and several reasons for the decline in the use of Java Applets in recent years.
- describe what Java WebStart technology is and how it differs from server-side software.
- describe what the Java Network Launching Protocol (JNLP) is and how Java Web Start and JNLP can be used deploy software over a network.
- create JNLP files and HTML files that use Java Web Start technology to launch applications and launch local java applications using the technology.
- describe three different levels of Java to EXE software products and examples of each.

14.1 Creating Executable Software

There are several ways to distributer software on modern computer systems. The traditional approach is to distribute an executable file. An **executable file** is a file that will run when the program is opened, either by typing in the name of the program or by clicking an icon for the program in a graphical user interface, such as with Microsoft Windows.

To better understand how this works, we need to review a bit about programming languages, discussed in **section 1.4** in chapter 1 of this book. The set of binary numbers that the CPU understands as its instruction set is called the computer's machine code. Each CPU, or each family of CPUs, has its own machine code. So, there are as just many different machine codes as there are families of processing units.

Eventually, every program that runs on a computer must be translated into a processor's machine code. Assembly languages are translated using assemblers, while high-level languages, such as C++, are translated using compilers or interpreters.



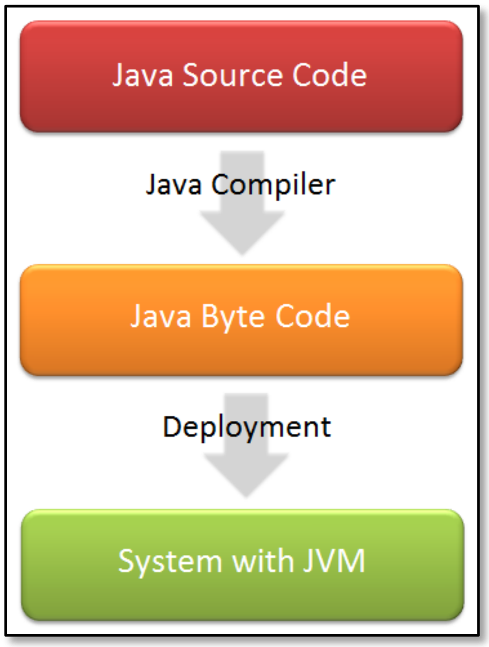
. All programs must be translated into machine code. Compilers, interpreters and assemblers perform this translation. Java adds an intermediate step for portability.

With a compiler, there are two stored copies of the program, the source code in the original language and a machine code copy called the object code. An executable file is built by linking the object code with routines from the operating system so that the software will run on a specific computing platform.

An executable file that runs on one computing platform usually won't run on another platform with a different processor or a different operating system. For example, a C++ program that has been compiled to run on an Intel-based computer under Microsoft Windows won't run on the same hardware under Linux. Usually, programmers need to recompile the source code and make a new executable file for each platform on which the software will run.

Java Software Deployment

Java takes a different approach for compiling and running software. Portability and high performance were two design goals for the Java programming language. These goals are sometimes in conflict because of the differences from one platform to another. Java adds an intermediate step in the process to increase the portability of high performance software — the Java Virtual Machine (JVM). Java programs are compiled into a simplified intermediate language called byte code, which is translated and executed on specific host platforms by a program called a Java Virtual Machine (JVM).



Almost all Web browsers are Java-enabled, meaning that they include a JVM and can run Java byte code. Most personal computers, tablets and smart phones are enabled with a Java Runtime Environment or its equivalent, which has a JVM so the system can run Java. Processors embedded in other electronic systems, such as GPS devices, can use a Java Card, the smallest available JVM.

This approach has proven to be highly successful. According to Oracle, 97 percent of all enterprise desktops run Java and more than 5 billion additional devices worldwide are equipped with a Java Card. There are more copies of the JVM in use than there are people on earth. Of course part of this popularity is because Java Runtime Environments and Java software development systems are available free of charge, but there is little doubt that Java is the world's most widely used high-level programming language because of its portability in addition to its quality and reliability. A Java program that has been compiled into a class file can be stored and moved from one platform to another and can be run on almost any platform equipped with a Java Virtual Machine.

Third party products are available to turn Java source code into traditional executable files, but there are four other widely used *official* ways that finished Java software can be deployed for use on various systems:

1. as a Java Archive file. A **Java Archive file**, known as a **JAR** file from its *.jar* file extension, is a set of files bundled into a single archive file. Typically a JAR file contains the class files and other needed to run a Java application.
2. as a Java applet. A **Java applet** is software that can be downloaded by a web browser to provide functionality within a Web page. Java applets are Java GUI applications using AWT or Swing that will run in a browser, provided the browser is equipped with a Java Virtual Machine (JVM). The browser window acts as the top level frame for the Java GUI application. This allows for the creation of smart Web pages with interactivity and other built-in functions.
3. using Java Web Start technology. **Java Web Start** technology is software built into a JRE that allows the system to download and run Java applications from the Web. The Java application is downloaded and run on the user's computer.

4. as a server-based Java enterprise application. A **Java enterprise application** is a multi-tiered collection of computer programs and data files that work together as a single enterprise-wide system. An enterprise can be anything from a small company to a large government agency or multi-national corporation. Online shopping and payment processing systems, such as the *Amazon.com* system, an automated banking system, and a corporation's human resource management system, are all examples of enterprise-wide systems that can be based on Java based on enterprise applications.

   The Java Enterprise Edition (Java EE) can be used to create multi-tiered enterprise applications. Each **tier** in a software system is a different part of the system based on its own computer, including client systems, web servers, database servers, and so on. Software that resides on and runs on a server is known as a **server-side application**, as opposed to **client-side applications** that run on single-user systems, such as personal computers, tablets, or smart phones.

   Enterprise-wide systems are often distributed software systems, since the software does not need to reside on any single computer, but can be spread out across computers in different locations, sometimes all over the world, including both client-side and server-side software.

   NetBeans and other IDEs can be used to create Web-based Java enterprise applications packaged for deployment as Java **Web Archive files – WAR files**. WAR files are similar to JAR files, but they are generally used to package and distribute server-side Java software. The creation of such software is beyond the scope of this course, but we will see how to create client applications that interface with other systems via the Internet, particularly the use of Java programs to access SQL database systems on an SQL database server.

In the rest of this chapter we will look at four methods for deploying Java software – JAR files, Java Applets, Java Web Start technology, and the use of third-party products to distribute Java executable files.

Beware of Platform-Specific Libraries

A programmer created a JAR file from a Java Netbeans project whose code included the following import statement:

```
    import org.jawin.win32.Ole32;
```

The program ran well on a Dell computer with Windows 7, but would not run on a Mac or even on the same Dell computer under Linux. Isn't Java supposed to be platform independent?

The Java language is platform independent, but a Java program that uses platform-specific classes and methods is not. In this case, the import statement indicates the software is using classes and methods from the ***jawin*** package. *Jawin* is the *Java/Windows 32 Integration Project*, which is open source software that gives Java code access to 32-bit Microsoft Windows system software, such as Microsoft's *Component Object Model (COM)* or *Win32 Dynamic Link Libraries (DLLs).* Any Java software that uses items from the *jawin* library will not run on other operating systems. There are many such platform-specific libraries available.

See: http://jawinproject.sourceforge.net/jawin.html for more information on *jawin.*

Sometimes hardware dependent or operating system dependent classes and methods can increase the efficiency of software by providing access to resources that make the most of a particular machine or operating system, but of course, they limit the range of platforms on which the software will run. In general, we should only use platform-specific classes and methods as a last resort, when there is no other way to reasonably create correct and efficient software. Beware of platform-specific libraries when creating Java code.

14.2 Java Archive Files (JAR files)

The most common and straightforward way to distribute a Java application is by using a Java Archive file (JAR file). As mentioned above, a JAR file is similar to a zip file, which may contain a bundle of files zipped together. In fact, JAR files use compression and bundling techniques based on the zip file format. JAR files usually contain Java *.class* files that run on a Java Virtual Machine, but may contain other resource files needed to support the *.class* files. A JAR file usually has a manifest. A JAR **manifest** is a special file that can contain information about the files packaged in a JAR file, including such things as the name of the class that contains the *main()* method to be used to start the application, version information, and security attributes.

We will not mess with the manifest in the examples in this chapter. Information about more sophisticated uses of JAR files and their manifests can be found in *Oracle's Java Tutorials*, online at: http://docs.oracle.com/javase/tutorial/deployment/jar/index.html

A Java application that has been saved as a JAR file can be run directly under Microsoft Windows and most other operating systems, provided that the JAR file contains a class with a *main()* method and the host platform is equipped with a JVM. However, if the program's output is simple console I/O, such as that created by *System.out.println* statements, then the process becomes a little more complicated.

Some resources that are needed to run a JAR file might be included in the JAR file, but external files, such as image files, sound files, or other data files that are not in the JAR file can be packaged in the same directory with the JAR file. Files that are needed but stored in a separate directory complicate things, as they require the use of full path names that could change, or they require manipulating the system's default *path* setting, which isn't the easiest or safest thing for most users to do. Our goal is to learn to package and deploy Java applications so that almost anyone can run the software.

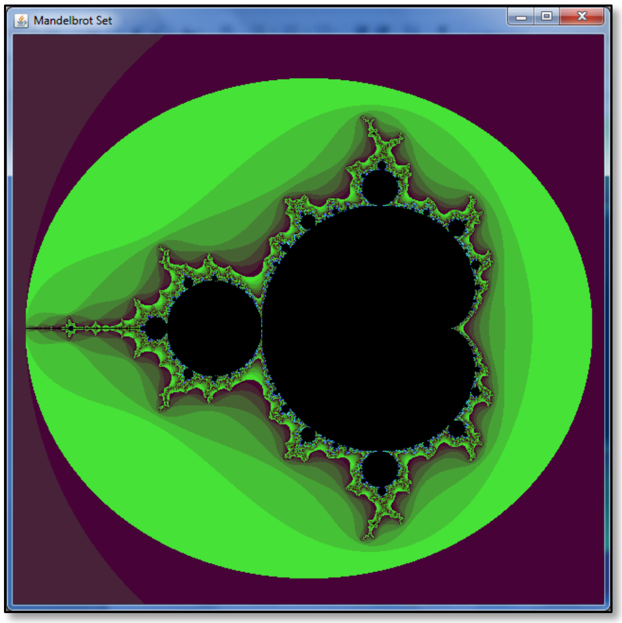
We will look at several examples of working with JAR files using Microsoft Windows. *JOptionPane* dialog windows, *AWT* GUI applications and *Swing* GUI applications all run well from a JAR file in Microsoft Windows, but console I/O that uses the system command prompt is problematic.

In the remainder of this section, we will look at creating a JAR file from a NetBeans project that has no external files or console I/O, a JAR file that uses external files, and then a JAR file that has console I/O. We will do all of this in Microsoft Windows. Ways to do these things for other operating systems are similar, provided you are familiar with the target operating system.

Creating a JAR file from a NetBeans Project

Java IDEs, including *NetBeans* and *Eclipse*, create JAR files. A JAR file is created for a NetBeans project whenever we ***clean and build*** the project. When an IDE performs a **clean and build** for a Java project, it erases all files created by previous builds for the project and rebuilds them, then creates a JAR file for the project using the newly built files.

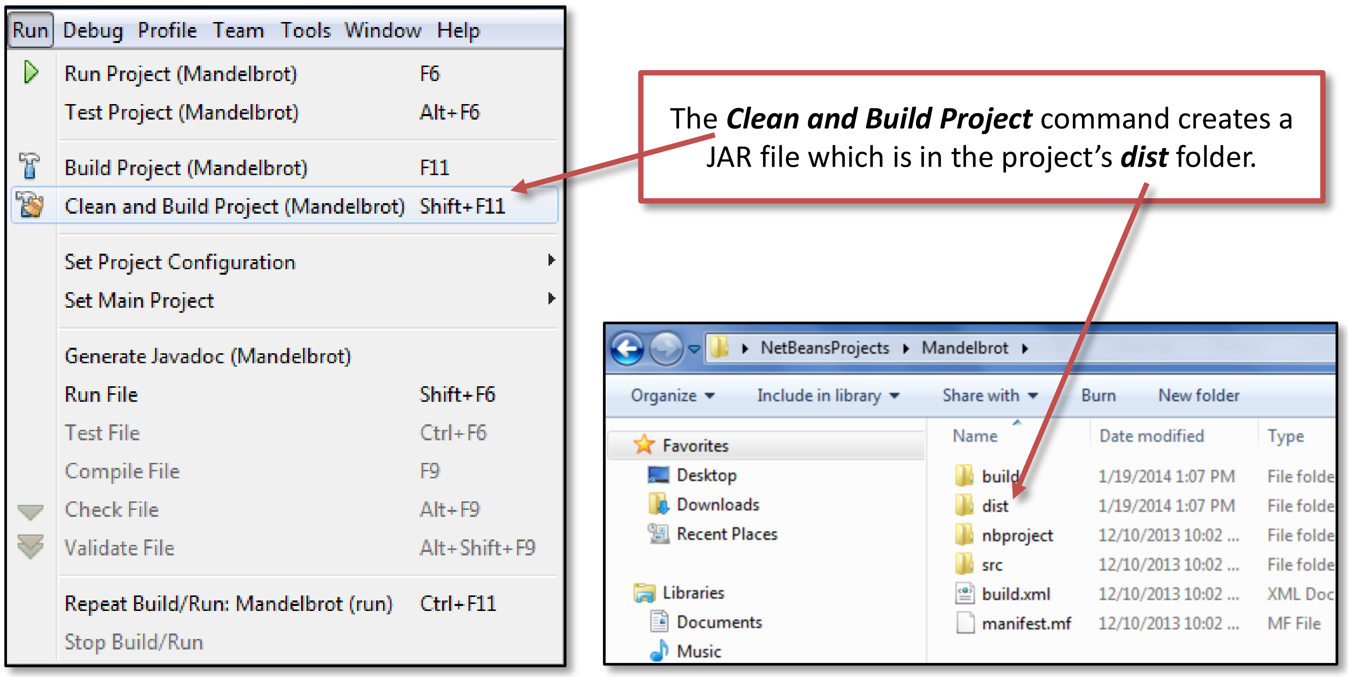
Let's look at an example. The *NetBeans* project ***MandelBrot*** contains Java software to create an image of a Mandelbrot set from fractal geometry. Here is what the output looks like:



This isn't a stored picture; the software draws the Mandelbrot set in a JFrame using the paint() method.

The first step in building a JAR file using NetBeans is to make sure the project runs properly in Netbeans. This project does. (The zipped project file ***mandebrot.zip*** is included with the files for Chapter 14.*)*

After you are sure a project runs in Netbeans, then you can use the ***clean and build*** command on the **Run** menu to rebuild all of the files. This will also create a distribution folder named ***dist*** which contains a JAR file for the software.


Run Debug Profile Team Tools Window Help
Run Project (Mandelbrot) F6
Test Project (Mandelbrot) Alt+F6
Build Project (Mandelbrot) F11
Clean and Build Project (Mandelbrot) Shift+F11
Set Project Configuration
Set Main Project
Generate Javadoc (Mandelbrot)
Run File Shift+F6
Test File Ctrl+F6
Compile File F9
Check File Alt+F9
Validate File Alt+Shift+F9
Repeat Build/Run: Mandelbrot (run) Ctrl+F11
Stop Build/Run
NetBeansProjects ▸ Mandelbrot ▸
Organize Include in library Share with Burn New folder
Favorites
Desktop
Downloads
Recent Places
Libraries
Documents
Music
Name Date modified Type
build 1/19/2014 1:07 PM File folde
dist 1/19/2014 1:07 PM File folde
nbproject 12/10/2013 10:02 ... File folde
src 12/10/2013 10:02 ... File folde
build.xml 12/10/2013 10:02 ... XML Doc
manifest.mf 12/10/2013 10:02 ... MF File

The ***Clean and Build Project*** command creates a JAR file which is in the project's ***dist*** folder.

Once the JAR file exists, we can run it on other systems that are equipped with a JVM. Usually, we only need to click the JAR file icon to run a Java application that has no external files and no console I/O.

Short Exercise - Creating and Running a JAR File

The zipped NetBeans project file ***mandebrot.zip*** is included as with the files for Chapter 14. We will unzip and open the project, create a JAR file, then run the JAR file from its native directory and from the system's desktop.

STEP 1.

Download the file ***mandebrot.zip*** from the files for Chapter 14 to a computer equipped with NetBeans and Java.

STEP 2.

Extract the file, keeping track of where the ***Mandelbrot*** NetBeans project folder is located.

STEP 3.

Open NetBeans then find and open the Netbeans project folder for the ***Mandelbrot*** project.

STEP 4.

Run the project without cleaning and building the project to make sure it works. It might take a moment to compile before it runs. Close Mandelbrot set window after the Mandelbrot set is displayed.

STEP 5.

Open the Netbeans project folder for the ***Mandelbrot*** project and notice that there is no ***dist*** folder and no manifest file present.

STEP 6.

Clean and Build the project using the **Clean and Build Project** command on the **Run** menu.

STEP 7.

navigate to the project folder for the ***Mandelbrot*** project and notice that now there is a ***dist*** folder and a *manifest* file present. Open the ***dist*** folder and notice that it contains a ***Mandelbrot.jar*** file.

STEP 8.

Click the ***Mandelbrot.jar*** file to run the program. Close the window after the Mandelbrot set is displayed.

STEP 9.

Copy the ***Mandelbrot.jar*** file to another location, such as the desktop. Click the ***Mandelbrot.jar*** file to run the program. Close the window after the Mandelbrot set is displayed.

The Java application can be run by clicking on the icon for the ***Mandelbrot.jar*** file on almost any system that has a JVM. There may be a problem on some systems that have features installed to prevent applications from running. If this happens you will need to consult with the systems administrator for that system.

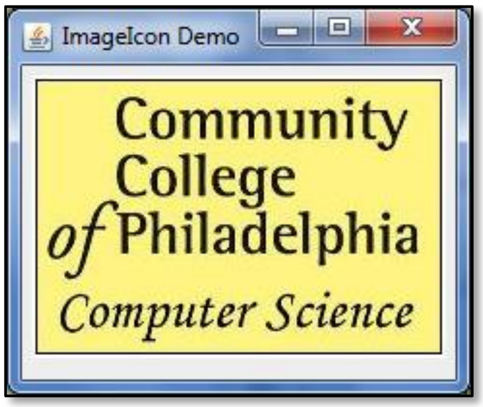
JAR Files that Access External Files

Creating and running a JAR file that accesses external data files can be almost as simple as creating and running a JAR file that does not access external files, provided we do not get involved with the complication of system path names. To do this, we simply make sure the JAR file and the necessary external files are in the same directory.

An easy way to distribute the JAR file and the external files together is to zip them into a single package. The JAR file itself is actually a special form of a zipped file. It is possible to include external files in the JAR file, but this adds an unnecessary complication for most users. It is easier on most systems to simply zip the JAR file and the needed external file or files together, and then distribute them as a zipped file. People wanting to run the software will then only need to unzip the zip file and click on the JAR file icon to run the application. This works on almost every system with a JVM installed.

Short Exercise - Creating and Running a JAR File that Access an External File

The NetBeans project ***ImageIconDemo*** has a Java application that opens and displays the file ***CSCI.jpg***, an image of the CCP Computer Science logo. The application displays the image as an *imageIcon* on a *Jlabel* in a *JFrame* window.



The file ***ImageIconDemo.zip*** is included as with the files for Chapter 14. It contains the ***ImageIconDemo*** Netbeans project folder, with the ***CSCI.jpg*** file in the project folder. You may recall that a NetBeans project can easily find files stored in the project folder.

We will unzip and open the project, create a JAR file for the project, then zip the JAR file and ***CSCI.jpg*** file together for distribution. We will copy the zipped application to a new location and then unzip and run the application.

STEP 1.

Download the file ***ImageIconDemo.zip*** from the files for Chapter 14 to a computer equipped with NetBeans and Java. Extract the file, keeping track of where the ***ImageIconDemo*** NetBeans project folder is located.

STEP 2.

Open NetBeans then find and open the Netbeans project folder for the ***ImageIconDemo*** project. **Run** the project without cleaning and building the project. It might take a moment to compile before running. Close the window after the Computer Science logo is displayed.

STEP 3.

Clean and Build the project using the **Clean and Build Project** command on the **Run** menu.

STEP 4.

Open the Netbeans project folder for the ***ImageIconDemo*** project and notice that there now is a ***dist*** folder. Also notice that the folder contains the ***CSCI.jpg*** file.

STEP 5.

Copy the ***CSCI.jpg*** file into the ***dist*** folder which should already contain ***ImageIconDemo.jar*** file. We want to have both files in the same folder. The ***dist*** folder usually contains a README.TXT file, which is not needed for our purposes.

WARNING: NetBeans will erase the ***dist*** folder and rebuild it whenever you use the ***Clean and Build Project*** command. Place any files to be included in the zip folder in the ***dist*** folder after you clean and build the project, or copy the JAR file and any other needed files to another folder for packaging.

STEP 6.

Zip the ***ImageIconDemo.jar*** and the ***CSCI.jpg*** file together into a new zip file named ***ImageIconDemo*.zip**

STEP 7.

Copy the new ***ImageIconDemo.zip*** file to another location, such as the desktop, or even to another computer that has a JVM.

STEP 8.

Unzip (extract) the ***ImageIconDemo.zip*** file in its new location. Both the ***ImageIconDemo.jar*** and the ***CSCI.jpg*** files should now be in the same folder. Some people prefer to put them in their own new folder while others prefer to just leave them as is, either on the desktop or in the user's *Documents* folder. It is probably neater and more orderly to put them in their own folder.

STEP 9.

Click the ***ImageIconDemo.jar*** file to run the application from its new location. You can close the window after the Community College of Philadelphia Computer Science logo is displayed.

As with the ***Mandelbrot.jar* file in the** previous exercise, the new Java application can be run by clicking on the icon for the ***ImageIconDemo.jar*** file on almost any system that has a JVM. However, in this case, the ***CSCI.jpg*** file must be in the same folder.

JAR files that Contains Console I/O

It can be awkward to run a Java application in a Windows environment if the application uses console I/O. The best way to do this is to run the application from the system's command prompt, but there is a complication – command prompt environments are very sensitive to file locations and Operating system *path* settings.

The command ***java -jar*** followed by the name of the JAR file will run a Java application from most command prompt environments. For example, to run the ***Mandelbrot.jar*** file from the command prompt we would enter the command ***java -jar Mandelbrot.jar*** . On some systems this environment is called the command shell. A command terminal environment, in which the user types commands as if communicating with the processor from an old fashioned communications terminal, will also work on many systems.

The example above, ***java -jar Mandelbrot.jar***, requires the computer to be able to find the Java software and the ***Mandelbrot.jar*** file to run the command. Each system has a ***path*** setting that tells the system where to look for programs and files. Java uses ***classpath*** parameters keep track of where class files are located on the system. In a Windows environment, many of the settings, such as the ***path*** setting, are managed automatically by the system. In a command driven environment it can be very awkward to manipulate these settings. The user in a command-driven environment could change directories manually or use full path names for programs and files when entering commands into the system, but this also can become quite awkward on a modern system with thousands of files and directories. It requires the user to be familiar with system commands and the arrangement of files and folders on the system. It is also very easy to make a mistake when typing full path names such as:

```
C:\Users\cherbert\Desktop\NetBeansProjects\ImageIconDemo\dist\ImageIconDemo.jar
```

We will take a look at a fairly simple way to run Java applications stored as JAR files from the command prompt. The technique involves using a text editor to create a batch file or shell script that will invoke the necessary command or commands to run the application. A **batch file** is simply a text file that contains one or more operating system commands. Microsoft Windows uses batch files. A **shell script** is a text file that contains one or more operating system commands on a Linux system. Other operating systems have similar features. Both batch files and shell scripts can be created using simple text editors.

In the example below we will create a Windows batch file to run a Java application that uses console I/O. When the user clicks on the icon for the batch file, the batch file will run the application from the command prompt so that we may see the console I/O. This saves the user from having to enter commands to change directories, entering long full-context file path names, or editing system *class* and *classpath* settings.

Batch files are not just limited to console I/O. They can also be used to run applications that use windows I/O.

A batch file to run a Jar file from the console simply needs to have the command to run the application:

```
java –jar jarfilename
```

For example, the command to run the *ImageIconDemo.jar* file is:

```
java –jar ImageIconDemo.jar
```

Windows Batch File Command Resources

A batch file is just a text file with one or more commands for the operating system. The file extension ***.bat*** identifies a batch file. A Microsoft Windows command line reference is online at:

http://technet.microsoft.com/en-us/library/bb490890.aspx

Princeton University's Windows Command Line Tutorial for Java programmers is online at:

http://www.cs.princeton.edu/courses/archive/spr05/cos126/cmd-prompt.html

Short Exercise – Creating A Batch File to Run a JAR file with Console I/O

The NetBeans project ***TwoPoints*** uses console I/O to ask the user to enter the X and y coordinates for two points in a Cartesian plane. It then calculates and tells the user the distance between the points and which quadrant each point is in.

The NetBeans project ***TwoPoints*** already has a JAR file, ***TwoPoints.jar,*** in its ***dist*** folder***.*** In this example we will create a batch file to run the JAR file, then zip the *jar* file and the *batch* file together in a package that can be unzipped to run on almost any Microsoft Windows system that has a JVM. We will test the file by unzipping it in a new location and running the program.

STEP 1.

Download the file ***TwoPoints.zip*** from the files for Chapter 14. Extract the file, keeping track of where the ***TwoPoints*** NetBeans project folder is located. We will not use the project in NetBeans, since it has already been processed with the **Clean and Build Project** command, but it is included in case you want to see the source code and edit or run the project outside of this exercise.

STEP 2.

Navigate to the ***dist*** folder inside the ***TwoPoints*** NetBeans project folder. This is where we will place our new batch file.

STEP 3.

Open a simple text editor such as ***Notepad***. Enter the following two lines:

```
java –jar  TwoPoints.jar
pause
```

Make sure you press enter after the second line. The *pause* command will stop the Windows command line interface from closing immediately after the Java program runs. It will display a "*Press any key to continue…*" message.

STEP 4.

Save the Notepad file as a text file with the name ***TwoPoints.bat*** in the ***dist*** folder with the ***TwoPoints.jar*** file or save the file and copy it to this folder. Be sure that it is saved with the extension ***.bat*** and not ***.txt***.

STEP 5.

Zip the ***TwoPoints.bat*** and the ***TwoPoints.jar*** files together into a new zip file named ***TwoPoints.zip.***

STEP 6.

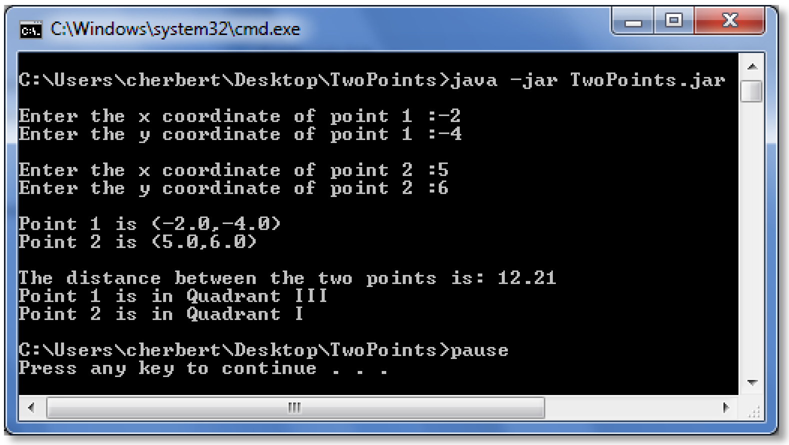
Copy the new ***TwoPoints.zip*** file to another location, such as the desktop, or to another Microsoft Windows computer that has a JVM.

STEP 7.

Unzip (extract) the ***TwoPoints.zip*** file in its new location. Both the ***TwoPoints.bat*** and the ***TwoPoints.jar*** files should now be in the same folder.

STEP 8.

Click icon for the ***TwoPoints.bat*** file to run the application in its new location. Close the command line window after the program finishes. A sample run of the program is shown on the right of this paragraph.



```
C:\Users\cherbert\Desktop\TwoPoints>java -jar TwoPoints.jar

Enter the x coordinate of point 1 :-2
Enter the y coordinate of point 1 :-4

Enter the x coordinate of point 2 :5
Enter the y coordinate of point 2 :6

Point 1 is (-2.0,-4.0)
Point 2 is (5.0,6.0)

The distance between the two points is: 12.21
Point 1 is in Quadrant III
Point 2 is in Quadrant I

C:\Users\cherbert\Desktop\TwoPoints>pause
Press any key to continue . . .
```

14.3 Java Applets

A **Java Applet** is a Java program that will run on a Java enabled Web browser. Java plug-ins for most browsers are freely available from Oracle. Most machines equipped with a JRE can run Java applets in a browser.

There are two ways to run applets – as part of a Web page downloaded from a server, and as part of a local Web page that is stored and run on the user's computer. For the past several years, security concerns have caused most systems to block applets run for local Web pages.

In this chapter we will see a brief overview of the use of Java applets as a way to deploy Java software. We will not discuss the creation and deployment of Java applets in detail. NetBeans has features that allow users to create and test Java applets. The oracle Java Tutorials has a set of lessons about applets and their creation online at: http://docs.oracle.com/javase/tutorial/deployment/applet/index.html

Changes in Java technology have made applets less common than in the past. Java Web Start technology, the increased presence of the Java Run Time Environment (JRE) across systems, and increased use of Java server-side applications have all diminished the demand for applets. In addition, HTML 5 and other developments in Web technology have replaced the need for some low-level applets.

Applet creation and deployment is also complicated by operating system and browser security settings, and the increase in use of some proprietary software. Over the years, Microsoft, Apple, and Google have all developed their own proprietary technologies that either compete with Java or area platform-specific version of Java and Java APIs. Java technology is available free of charge, but the license for it prohibits development of proprietary versions of Java or versions limited to a specific vendors products. First Sun Microsystems and later Oracle (which bought out Sun) have sued those attempting to develop their own proprietary versions of Java Technology.

In the late 1990's, soon after the release of Java, Microsoft attempted to deploy a *Visual J++* version of Java and their own MSJVM (Microsoft Java Virtual Machine) that would only work with Microsoft operating systems and browsers. Sun sued Microsoft and won. In 2001 further development of Visual J++ was halted, Microsoft was ordered to cease distribution of the MSJVM, pay a $20 million penalty.

Microsoft violated the court order by continuing to bundle its own MSJVM as the default Java system with Internet Explorer and attempted to block other implementations of Java from running on Microsoft systems. In addition to the original suit, Sun (and others) filed antitrust lawsuits against Microsoft. The case was settled quickly. Microsoft was ordered to pay Sun more than $1.9 billion dollars and to enable Java technology with Internet Explorer and Windows. By the end of 2002, Java technology was growing as the technology of choice for software development, especially for Web apps, including applets.

More recently, both Apple and Google have attempted to establish their own proprietary technologies. Apple tablets and phones will not run Java. In fact, all software that runs on these devices must be created using Apple's *Objective C* and must be acquired through the Apple app store. Many newer browsers on Apple Macs block the use of Java applets. IPhones and IPads once dominated the smart phone and tablets market, and while their sales are still strong, they are no longer the market leaders, due in part to the limitations on what software they can run.

Google attempted to develop its own proprietary Java APIs and to block use of any other version of Java on Chrome and many Android systems. They were sued by Oracle in 2010. In 2012 the US District Court for Northern California ruled, in a jury trial, that Google could distribute proprietary Java APIs, but that they could not copyright nor patent those APIs. The ramifications of this court action are still unclear, but Java applets will not run on many Android or Chrome OS systems, although they will work on Chrome browsers on other platforms.

The bottom line on all of this is that the use of applets is diminishing. Proprietary rather than open source systems have cut into the use of applets, as have newer technologies, mostly from Oracle itself. Things are currently is a state of change, but the use of Java applications, unlike Java applets, has actually increased, and there are more Java Virtual Machines in use today than there are people on earth. Java application development is a growing field. Java applet development is fading.

The Structure of a Java Applet

A Java applet must be created as GUI software that is a subclass extending either the *Applet* class or the *JApplet* class. This is true even if the applet primarily is being used for things like calculations rather than graphics. Anyone who wishes to create an applet must be familiar with both Java GUI programming and inheritance in Java.

Extension of the *Applet* class allows for the creation of AWT applets. Extension of the *JApplet* class allows for the creation of Swing applets. Both types of applets have a similar structure. An applet is created as a class that can be packaged as a JAR file and invoked from HTML code running in a browser. The applet must extend the JApplet class if any Swing components are to be used such as Swing labels, text boxes, checkboxes, radio buttons, etc. (See chapter 7.)

An HTML window acts as a container for the applet. In most cases, a Java GUI application that runs in a Window using a *JFrame* container can be cut and pasted into applet code and used for a Java applet.

Both the *Applet* and *JApplet* classes require several methods to manage Java applets. the *paint()* method can be used to draw the applet in the browser, just as it could be used to draw in a JFrame application. Several other applet methods are related to the lifecycle of an applet. In its lifecycle, an applet can:

- initializeitself using the *init()* method
- start running using the *start()* method
- stop running using the *stop()* method
- perform some housekeeping by closing files, releasing resources, etc. , before being unloaded by using the *destroy()* method.

Here is code from the Oracle Java Tutorials showing the structure of an applet:

```
import java.applet.Applet;
import java.awt.Graphics;

//No need to extend JApplet, since we don't add any components;
//we just paint.
public class Simple extends Applet {

    StringBuffer buffer;

    public void init() {
        buffer = new StringBuffer();
        addItem("initializing... ");
    }

    public void start() {
        addItem("starting... ");
    }

    public void stop() {
        addItem("stopping... ");
    }

    public void destroy() {
        addItem("preparing for unloading...");
    }

    private void addItem(String newWord) {
        System.out.println(newWord);
        buffer.append(newWord);
        repaint();
    }

    public void paint(Graphics g) {
   //Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0,
          getWidth() - 1,
          getHeight() - 1);

   //Draw the current string inside the rectangle.
        g.drawString(buffer.toString(), 5, 15);
    }
}
```

In general, paint can draw items in an applet, or Swing components can be created and added to an applet just as they are in JFrame applications. The applet can then provide a level of interactivity and programming power well beyond HTML code, or that found with client-side scripting languages like *Javascript*.

The following example shows the code for a simple *Hello World!* Applet with a *JLabel*.

```
import javax.swing.JApplet;
import javax.swing.SwingUtilities;
import javax.swing.JLabel;

public class HelloWorld extends JApplet {
    // the init()method is called when the applet is loaded into the browser
    public void init() {
        // invokeAndWait creates this applet's GUI event thread using its run() method
        try {
            SwingUtilities.invokeAndWait(new Runnable() {
                public void run() {
                    // code from most JFrame applications will work here
                    JLabel lbl = new JLabel("Hello World");
                    add(lbl);
                }  // end run
            }); // end try

        } catch (Exception e) {
            System.err.println("createGUI didn't complete successfully");
        } // end catch

    } // end init()

} // end class HelloWorld
```

In many cases, Java *Web Start* technology, which allows us to run Java applications form the Web, has replaced applets. It is discussed in the next section.

14.4 Java Web Start Technology

Java Web Start technology is software that allows a user to easily launch a Java application on a network, such as the World Wide Web or a local area network. According to Oracle:

Using Java Web Start technology, standalone Java software applications can be deployed with a single click over the network. Java Web Start ensures the most current version of the application will be deployed, as well as the correct version of the Java Runtime Environment (JRE).

– from *Java SE Technologies*, on the Web at: http://www.oracle.com/technetwork/java/javase/javawebstart/index.html

Java Web Start technology has been included in the JRE since version 5 of Java. It uses the **Java Network Launching Protocol (JNLP),** which will download and run a Java application from a server. A **JNLP file** is an XML file ending with extension *.jnlp* that contains information used to run an a Java Web Start application. The first time the user on a particular machine clicks to run the application, it is downloaded from the server, stored locally, then run. After that, anytime the user clicks to run the software the system will check to see if a newer version is available on the server. If a newer version is not available, the system will run the local copy. If a newer version is available, the system will download it and replace the local copy before running the application.

Java Web Start simplifies software deployment and eliminates the need for more complicated installation and upgrade techniques. It uses existing Internet technology, such as the HTTP protocol and Web servers, for deploying Java applications to many client systems.

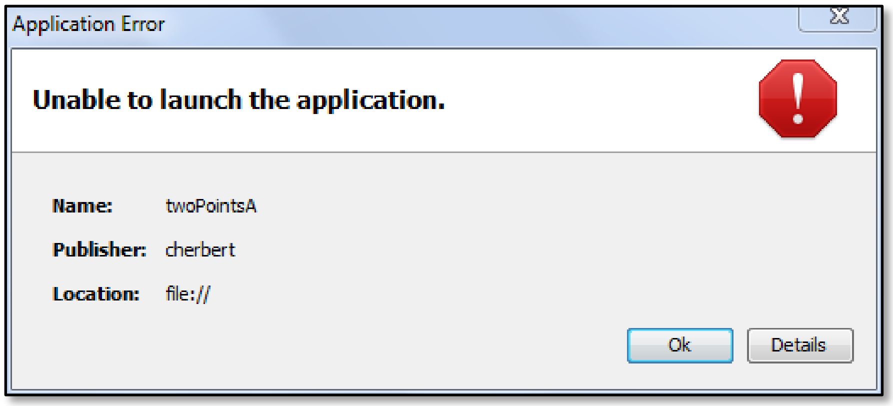
Java Web Start technology should not be confused with server side software, such as Java Server Pages (JSP). **Server side software** stores and runs applications on a server, with the local client machine basically acting as an I/O terminal for the application running on the server. With Java Web Start technology, the application is downloaded from the server, then stored and run on the user's local client machine.

Java Web Start technology can also be used to launch standalone Java applications on a user's host computer without being connected to a network. That is what we will do in the exercise later in this section so that we can avoid, for now, the complications of having to download and run software from a server.

A good description of what Java Web Start technology is and how it works is on the Web at:
http://java.com/en/download/faq/java_webstart.xml

The Oracle Java Tutorials include a section on Java Web Start technology, on the Web at:
http://docs.oracle.com/javase/tutorial/deployment/webstart

As with applets, Java Web Start technology is intended to be used with Java GUI software, such as applications that have a GUI with Swing components. Applications that include GUI software using JOPtionPane dialog windows, AWT, or Swing components will work fine with Java Web Start, but console I/O will generally not work. The application can be launched using Java Web Start, but usually it will not run if it has console input. Instead, an error an error message will appear, similar to the message window below that was generated by trying to run a version of the TwoPoints console I/O application seen earlier in this chapter.


Application Error

Unable to launch the application.

Name: twoPointsA
Publisher: cherbert
Location: file://

Later in this course, we will spend more time looking at GUI programming and the use of APIs. As part of that material we will see more about Java Web Start Technology. In this a chapter, we will look at using NetBeans to prepare a Java GUI application to run locally using Java Web Start Technology.

Java Web Start and JNLP Security Issues

There are some complications with Java Web Start technology and JNLP related to security considerations. Applications used over the Web often require security certificate. A **security certificate** is a digitally signed and encrypted file that says software is safe to use – that it does not seem to contain any viruses or other malware, or any features that could jeopardize the security of your system. There are three levels of security when certificates are used. Network-based software (including Web pages) may have:

- no security certificate;
- a self-signed security certificate;
- a security certificate signed by a third party.

Each security certificate is signed by the person or agency issuing the certificate. A **self-signed security certificate** is signed by the person or agency – called the software vendor – who developed or who is distributing the software. Software with self-signed security certificates are usually treated the same as software without a security certificate. A self-signed security certificate is like a note saying that a student may skip class signed by the student himself rather than by a teacher, principal, or parent. They often are not worth the paper that they are not printed on.

A security certificate may be issued by a known, trusted source or by an unknown source. Systems administrators can establish a list of trusted sources and only allow software from those sources to run on their systems. Many trusted sources charge a fee to issue certificates. For example, the Web hosting company *GoDaddy.com* issues security certificates. See: http://www.godaddy.com/ssl/ssl-certificates.aspx

Agencies that issue certificates often have insurance policies to cover their certifications and can provide such policies to software vendors. This is an a area in which we can expect to see increasing legislation and regulation over the next few years to avoid fraud, data theft, theft of services, etc.

There are different levels of access to system resources that can be prevented or allowed, depending on the nature of the software and its security certificates. Generally, the default settings for JNLP-launched applications have the following restrictions:

- No access to local disk drives is allowed.
- All JAR files must be downloaded from the same host.
- Network connections are enabled only to the host from which the JAR files are downloaded.
- No security manager can be installed by the application (it might override security settings.)
- No local native libraries may be used.
- Access to system properties is limited. Some can't be accessed, some are read-only, some can be changed.
- Access to system calls is limited, but an application is allowed to use the *System.exit* call.

Some of these restrictions can be overcome by the use of Java's *JNLP API* to access the file system and other system resources. An application that needs access to restricted parts of the system will need to be delivered in a set of signed JAR files. All entries in each JAR file must be signed.

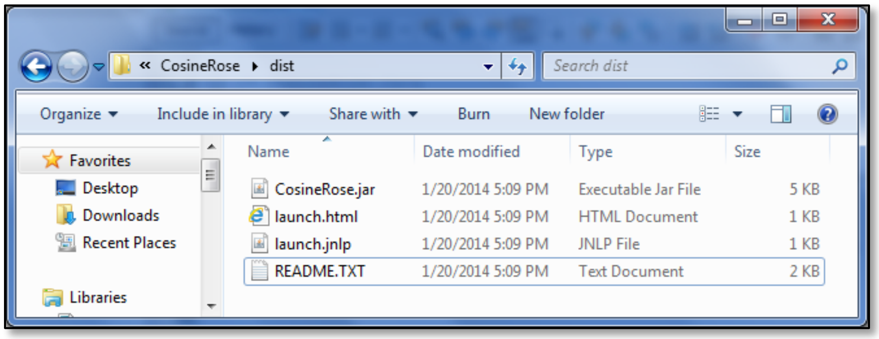
The **Java Control Panel** allows us to manipulate the settings for how Java works on a system. It can often be accessed from the operating system's control panel. The Java control panel has security settings that will determine how Java reacts to security certificates. By default, Java SE 7 update 21 and later versions of Java (and the corresponding JREs) will not let unsigned or self-signed applications run. To do so, the security settings must be changed. We will see this in the exercise later in this chapter.

Using NetBeans to Prepare Applications for Java Web Start Deployment

NetBeans can be used to prepare a working GUI-based Java application for deployment using Java Web Start technology. An easy way to do this is:

1. make sure the application runs correctly in NetBeans ;
2. change the NetBeans project's properties to enable Java Web Start;
3. clean and build the project.

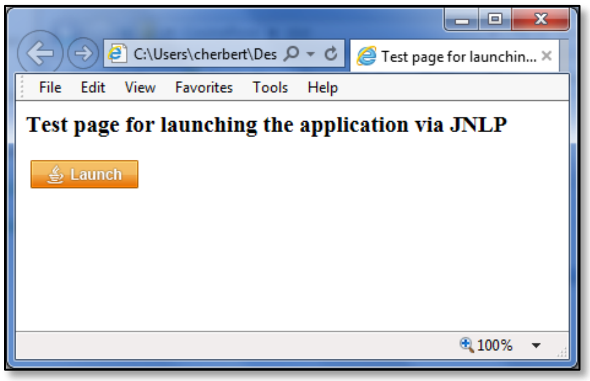
When we clean and build a NetBeans project with Java Web Start enabled, Netbeans will create a working JAR files for the project, a JNLP file for the project, and an HTML page that has a link to the JNLP file, as seen in the image below for a Cosine Rose application that has been cleaned and built in this way.



Name	Date modified	Type	Size
CosineRose.jar	1/20/2014 5:09 PM	Executable Jar File	5 KB
launch.html	1/20/2014 5:09 PM	HTML Document	1 KB
launch.jnlp	1/20/2014 5:09 PM	JNLP File	1 KB
README.TXT	1/20/2014 5:09 PM	Text Document	2 KB

By default The JAR file will have the same name as the project. The HTML page and the JNLP page will each have the name *launch*.

The application can be run by clicking on the JAR file, by clicking on the JNLP file, or by opening the Web page (the HTML file) in a browser and clicking the button linked to the application. Here is what the generated Web page looks like:

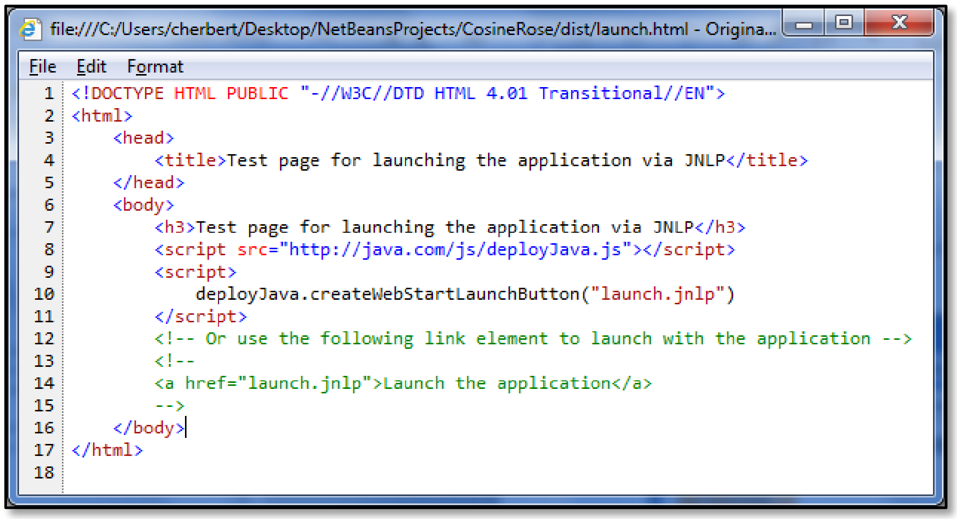


C:\Users\cherbert\Des
Test page for launchin...
File Edit View Favorites Tools Help
Launch
100%

Test page for launching the application via JNLP

The application can be launched by clicking the Launch button. It runs a sa standalone application, just as if you ran the JAR file itself or the JNLP file. This is not the same as running an applet, which runs in the browser.

Here is the HTML code from the generated Web page:


file:///C:/Users/cherbert/Desktop/NetBeansProjects/CosineRose/dist/launch.html - Origina...
File Edit Format

```
1  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
2  <html>
3      <head>
4          <title>Test page for launching the application via JNLP</title>
5      </head>
6      <body>
7          <h3>Test page for launching the application via JNLP</h3>
8          <script src="http://java.com/js/deployJava.js"></script>
9          <script>
10             deployJava.createWebStartLaunchButton("launch.jnlp")
11         </script>
12         <!-- Or use the following link element to launch with the application -->
13         <!--
14         <a href="launch.jnlp">Launch the application</a>
15         -->
16     </body>
17 </html>
18
```

The code accesses a script on the *Java.com* sever, so it will *not* launch the application unless the system is connected to the Internet. (The JNLP file will launch the application when there is no Web connection.)

The script code, comment and link in lines 8 to 15 can be copied and pasted for use in almost any Web page and will work on almost any Java-enabled browser. The HTML file, JNLP file, and JAR file should be in the same directory for local use. More sophisticated uses of Java Web Start technology, such as using it for real downloading and deployment over the Web, allow programmers and system administrators to select where files will be stored and how they will run.

Exercise – Preparing and Running an Application with Java Web Start Technology

In this exercise we will use NetBeans to prepare an application to run using Java WebStart technology. We will not use a server, but will simply create the JNLP and HTML files and run them locally. This exercise should work on any computer that is running Netbeans, but if you have a version of java older than Java 7 update 10 you may want to install a new version of NetBeans with Java before continuing (See Appendix A.)

Be aware that default java security settings on most systems prevent Java Web Start applications without security certificates from running. We will not acquire security certificates for our application, but will use the Java Control Panel to change the Java security settings to allow us to run the application without a certificate. You will need system administrator privileges to do this. If you do not have such privileges should still do the exercise, but you can’t run the resulting software as Web Start application without a certificate. You should be able to run the resulting application as a JAR file.

We will download and open the completed CosineRose NetBeans project, make sure it works, then prepare and run it using Java WebStart technology.

STEP 1.

Download the file **CosineRose.zip** from the files for Chapter 14. Extract the file, keeping track of where the **CosineRose** NetBeans project folder is located, then open the project in NetBeans .

STEP 2.

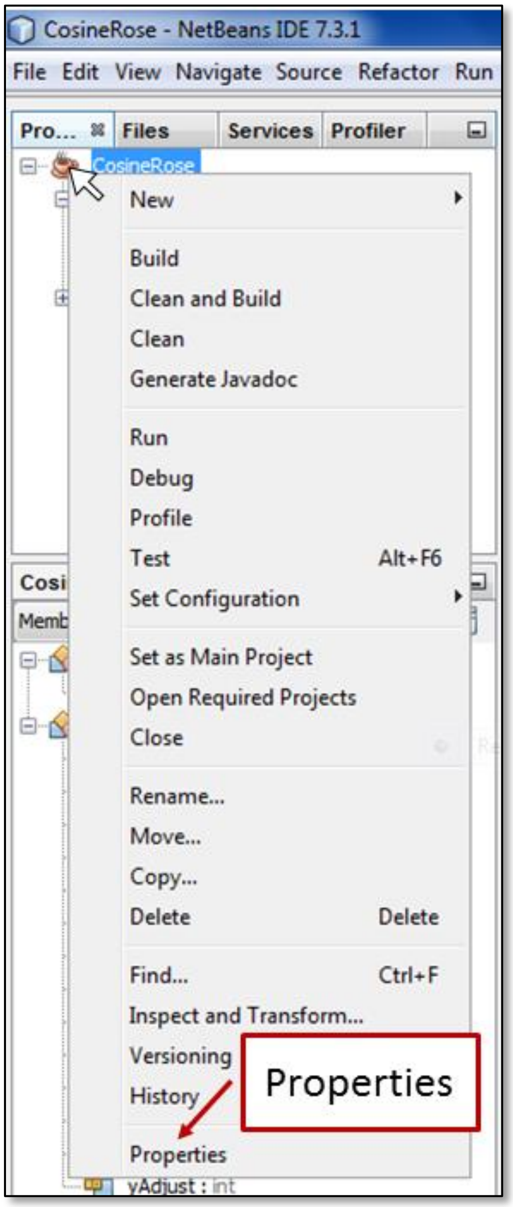
Run the program to see how it works. **Enter a small integer**, such as 2 or 5, when asked to do so. The program uses a *JoptionPane* dialog box to get input form the user, then draws a cosine rose in a *JFrame* Window. The pattern of the rose depends on the value entered.

STEP 3.

Run the program again, but this time **do not input an integer** when asked to do so. Instead, enter a letter of the alphabet or some other String. Notice the error message in the NetBeans output zone. The program crashes because it does not handle a data mismatch error or exception. This in important – in a few minutes we shall what happens with a JNLP-based application that crashes when errors and exceptions are not addressed.

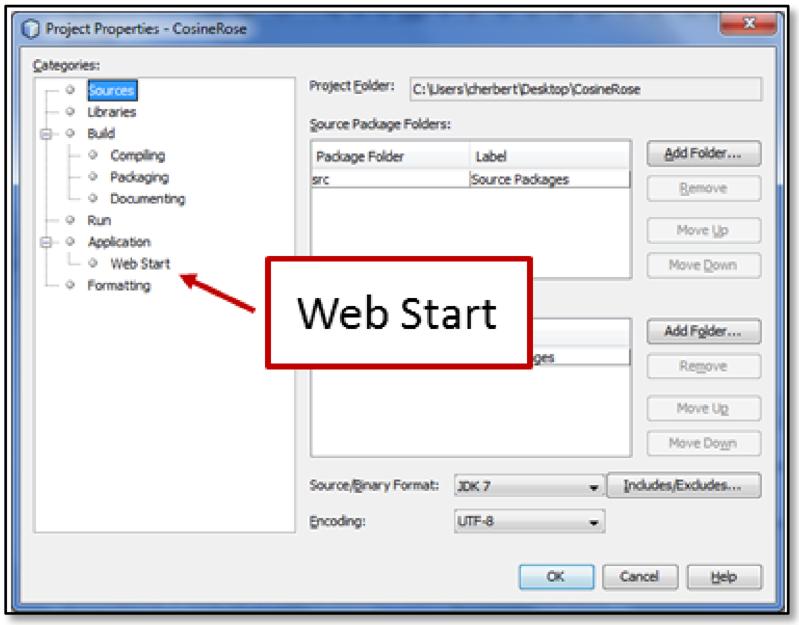
STEP 4.

Right click the **CosineRose** project icon in Netbean's project window, then select **Properties** on the bottom of the menu that appears, as shown in the right.



STEP 5.

The project properties window will appear, as shown below. Select **Web Start** in the categories window.



Project Folder: C:\Users\cherbert\Desktop\CosineRose

Source Package Folders:

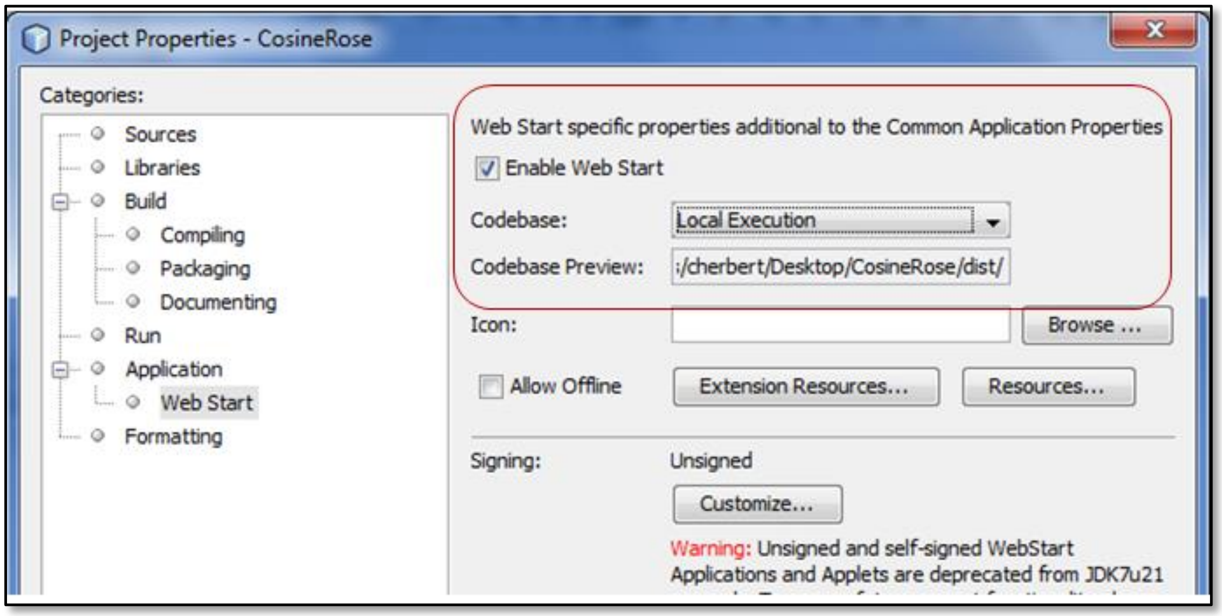
Package Folder	Label
src	Source Packages

Source/Binary Format: JDK 7

Encoding: UTF-8

STEP 6.

The Web start panel of the properties window will appear. Make sure the **Enable Web Start** box is checked and that Local Execution is selected from the **Codebase** dropdown menu. These things are circled in the image below. Do not change the Codebase Preview setting.

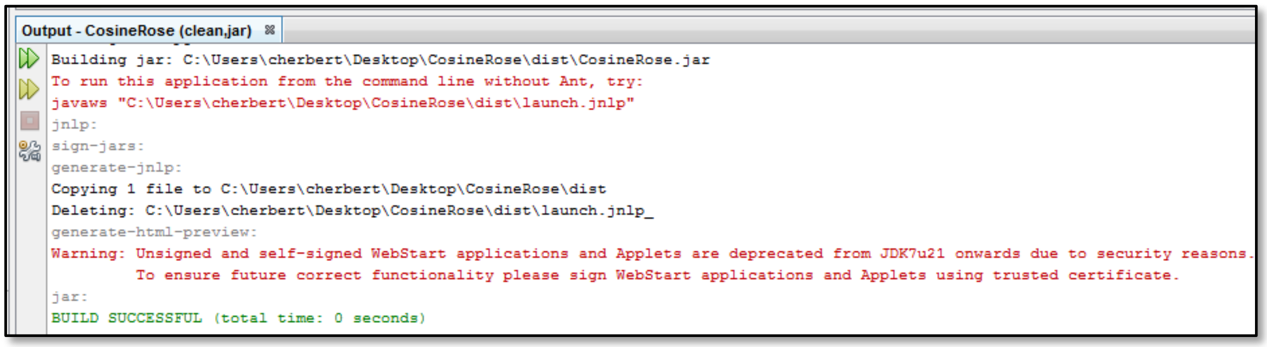


Web Start specific properties additional to the Common Application Properties
[✓] Enable Web Start
Codebase: Local Execution
Codebase Preview: :/cherbert/Desktop/CosineRose/dist/
Icon:
[ ] Allow Offline
Signing: Unsigned
Warning: Unsigned and self-signed WebStart Applications and Applets are deprecated from JDK7u21

STEP 7.

Click the **OK** button to close the window, then clean and build the project using the **Clean and Build Project** command on the **Run** menu.

The project is now ready for use as a Java Web Start application, but before we go any further, read the messages in red in the NetBeans output zone:



```
Building jar: C:\Users\cherbert\Desktop\CosineRose\dist\CosineRose.jar
To run this application from the command line without Ant, try:
javaws "C:\Users\cherbert\Desktop\CosineRose\dist\launch.jnlp"
jnlp:
sign-jars:
generate-jnlp:
Copying 1 file to C:\Users\cherbert\Desktop\CosineRose\dist
Deleting: C:\Users\cherbert\Desktop\CosineRose\dist\launch.jnlp_
generate-html-preview:
Warning: Unsigned and self-signed WebStart applications and Applets are deprecated from JDK7u21 onwards due to security reasons.
         To ensure future correct functionality please sign WebStart applications and Applets using trusted certificate.
jar:
BUILD SUCCESSFUL (total time: 0 seconds)
```

The first message says:

```
To run this application from the command line without Ant, try:
javaws "C:\Users\cherbert\Desktop\CosineRose\dist\launch.jnlp"
```

Notice that JNLP files can be run from the system command prompt by entering **javaws** and the name of the JNLP file to run the application as a Java Web start application. We won't use this for now, but knowing about this may come in handy in the future.

The second message says:

```
Warning: Unsigned and self-signed WebStart applications and Applets are deprecated
from JDK7u21 onwards due to security reasons.
To ensure future correct functionality please sign WebStart applications and
Applets using trusted certificate.
```

This message is warning us that unsigned and self-signed java Web Start applications will not run with newer versions of Java with the default security settings. We'll deal with that in a moment.

For now, we are finished preparing the project in NetBeans

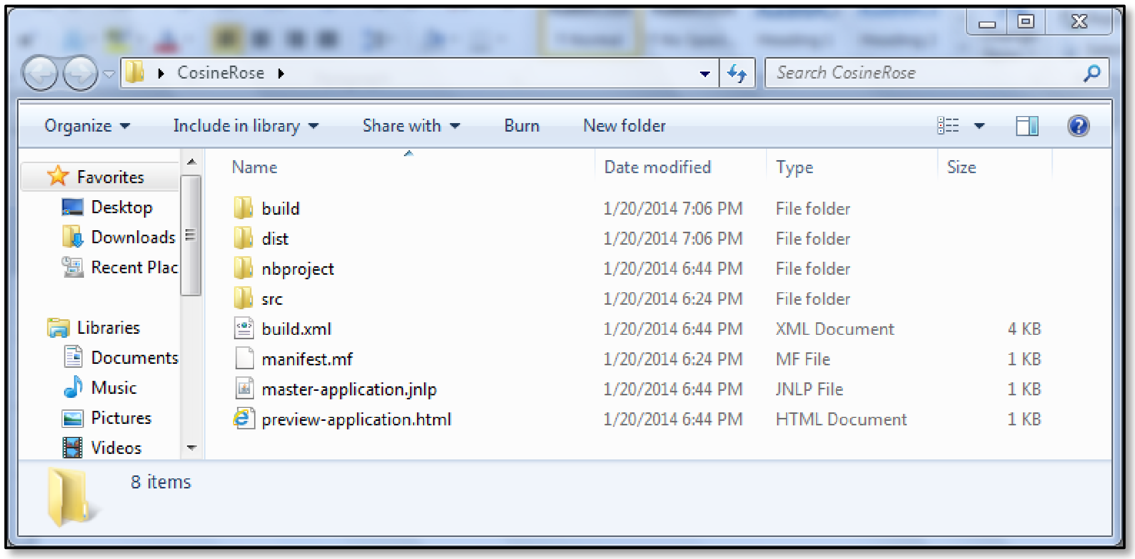
STEP 8.

Close the NetBeans CosineRose project, then **Exit** NetBeans itself.

We will now look at the files that were created by our clan and build and will attempt to run the application.

STEP 9.

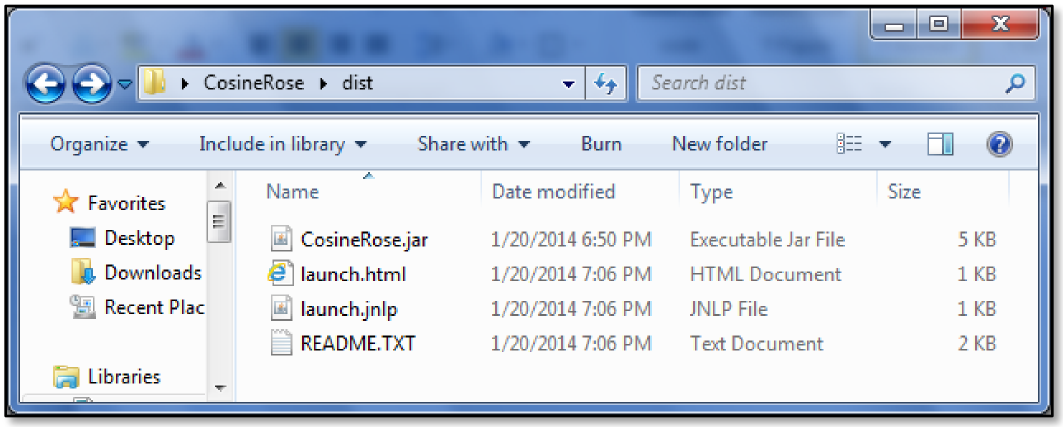
Find and open the project folder for the NetBeans CosineRose project, shown below. Notice that it now has a build folder and several new files, including a *JNLP* file, an *HTML* file, and a manifest. We will not use these files, they are for more sophisticated Web deployment.



Name	Date modified	Type	Size
build	1/20/2014 7:06 PM	File folder	
dist	1/20/2014 7:06 PM	File folder	
nbproject	1/20/2014 6:44 PM	File folder	
src	1/20/2014 6:24 PM	File folder	
build.xml	1/20/2014 6:44 PM	XML Document	4 KB
manifest.mf	1/20/2014 6:24 PM	MF File	1 KB
master-application.jnlp	1/20/2014 6:44 PM	JNLP File	1 KB
preview-application.html	1/20/2014 6:44 PM	HTML Document	1 KB

STEP 10.

Open the **dist** distribution folder, shown below. Notice that it contains a *JAR* file for the project, and *JNLP* and *HTML* launch files. This is similar to the **dist** folder we saw earlier in this chapter.



Name	Date modified	Type	Size
CosineRose.jar	1/20/2014 6:50 PM	Executable Jar File	5 KB
launch.html	1/20/2014 7:06 PM	HTML Document	1 KB
launch.jnlp	1/20/2014 7:06 PM	JNLP File	1 KB
README.TXT	1/20/2014 7:06 PM	Text Document	2 KB

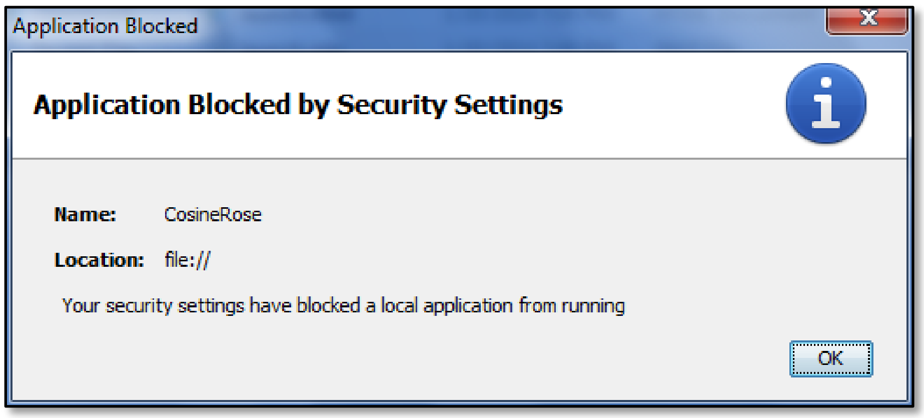
STEP 11.

We will now try to run the application. The JAR file should work, but the JNLP and HTML files probably won't work yet because of your system's Java security settings.

Run the application by double-clicking on the CosineRose.jar file. The program should run correctly. If it doesn't, then turn off your computer, drop the course, and change your major to golf course management. Wait – I'm only kidding. This kind of thing happens and demands patience to debug. Any one of a dozen things could be preventing the program from running. You may need to check what you've done, do a little research, or ask someone who knows more about your system for help. In the end, it will all be worth it. You will make more money than a golf course manager and have less chance of developing hay fever or getting hit by a golf ball.

STEP 12.

We will now try to run the JNLP and HTML files. Try the JNLP file first. **Double-click the launch.jnlp file**. If it runs then everything is okay, but Java on your system is probably either old or insecure. I expect you will get an error message similar to this one:


Application Blocked

Application Blocked by Security Settings

Name: CosineRose

Location: file://

Your security settings have blocked a local application from running

This warning doesn't make it clear, but the security settings that are blocking the application are the Java security settings. We'll deal with those in a minute.

STEP 13.

Close the warning message window (if it appeared) and now **double-click the launch.html file**. It will open a web page like the one below or perhaps give you a warning that the Webpage is restricted from running a script with a button to *Allow Blocked Content* . If you see this, it's okay allow the script on this page to run, it won't do any harm. Different versions of different browsers have different warning messages and buttons to click, but in general, it's okay to run the script and allow blocked content. Eventually the application may run, but most likely you will get to the same security warning shown above. This means the HTML file invoked the JNLP file, but your Java security settings won't allow the application to run because it does not have a valid security certificate.

At this point, we will change the Java security setting then try again to run the application. The application should run with a warning message after we change the settings instead of the warning blocking it from running.

STEP 14.

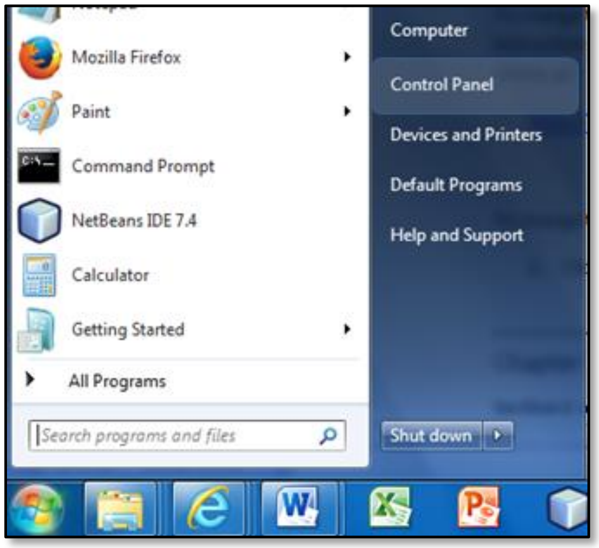
There are three levels of Java security - *Medium*,*High* and *Very High*. *Very High* is the default.

To change the setting, we will need to finds the *Java Control Panel* on your system. The following instructions are for doing so on a Windows 7 computer. Directions for doing this on other systems are online at:

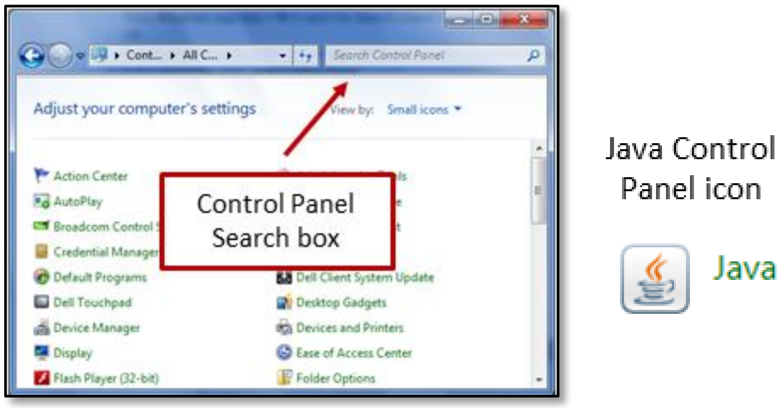
http://www.java.com/en/download/help/jcp_security.xml#control panel

To change the security settings on a Windows 7 computer:

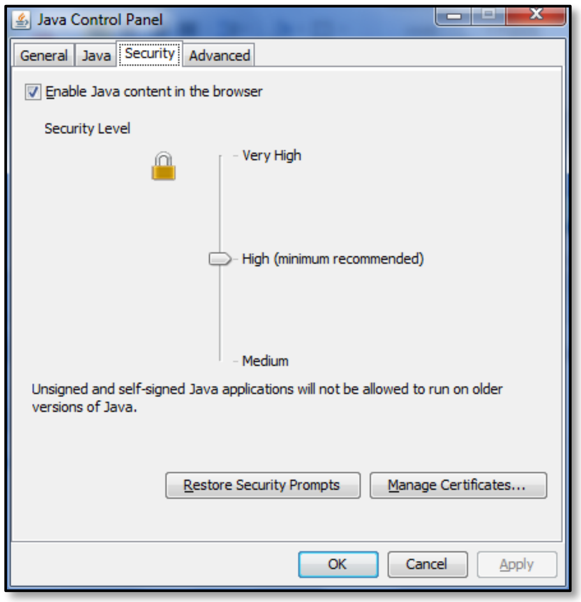
A. Click the Start button, then click the Control Panel option on the menu that appears.



B. The *Windows Control Panel* will appear, as shown below. In the *Control Panel Search* box in the upper right corner, enter Java, then click on the Java Control Panel Icon that appears. The Java contreol panel should appear.



C. Cick the security tab in the Java Control Panel, shown below, then change the security setting from **High** to **Medium.** Click **OK** to close the Java Control Panel. Close your system's control panel if it is still open.



☑ Enable Java content in the browser

Unsigned and self-signed Java applications will not be allowed to run on older versions of Java.

STEP 15.

Now that the Java security has been changed to *Medium*, we can try the JNLP and HTML applications again. First, go back to the project's **dist** folder and double click the JNLP file to run the application. You may get a warning message, but this time it will have options to run the application anyway. If it works, then try the HTML application. If the JNLP application works, then the HTML application should work. The HTML file simply invokes the JNLP file.

That's it, we're done. You may want go back reset the Java security setting, but If your system has a good firewall and good security software, then it shouldn't be a problem to leave the Java security setting on medium. The primary difference is that the system in *Medium* will warn you and allow you to run or block questionable Java applications rather than just blocking them as with a *High* setting.

We will see more about Java Web Start later in the semester, including running Web Start applications form a server.

14.5 Java Applications as Traditional Executable Files

As we have seen, the Java language was intended to be used for cross-platform applications, especially Web-based applications. Yet, there are still times when some people or some enterprises may want to create traditional executable files from Java source code.

A Java application can be harder to install and configure than an executable file such as a Microsoft Windows EXE file. Java applications involve and depend on a variety of files throughout the system. Java programs must access the JVM, which will be stored somewhere on the system. More sophisticated java software is filled with reusable code, referencing many class files. Operating systems *path* settings, java *classpath* settings, and the presence of the JVM are all necessary for a java program to run.

In contrast, once an application has been packaged into an EXE file then the user only needs to run the EXE file to run the application. The EXE file can be moved to other similar platforms and it will run without a lot of fiddling with the system or checking to see if Java is installed, etc.

As one post on a popular programming language bulletin board said, "*a .jar file is sufficient only if Java is installed on the target machine, and java.exe is on the PATH, and the main class is specified in the manifest file. in general, .exes are a bit simpler for non-technical people to execute*." (Posted on Stackoverflow.com, by Richard Le Mesurier, Mar. 5 2013)

Standard Java IDEs do not allow for the creation of executable files, but there are third party products available that can be used to create EXE files, primarily for Microsoft Windows platforms. Most of these products are Java EXE wrapper software. An **EXE wrapper** bundles everything needed to run an application – JAR files and so on – into one package that runs from an EXE file. Some even bundle a JRE into the file. They wrap everything needed to run an application as a standalone unit.

There are a few products that will actually compile Java applications into native source code as an EXE file that will run in the same way as traditional EXE files.

In general products that create EXE files from Java code work at three different increasingly sophisticated levels:

Level 1 –creates an EXE wrapper, but depends on the JVM or JRE already being installed on the system.

JexePack, previously called ***Java2ex****e*, is an example of a level 1 Java application to EXE packager. See http://www.duckware.com/jexepack/index.html for details.

Jar2Exe is a similar product that works on both Windows and Linux systems.
See http://www.jar2exe.com

Jsmooth is free java to EXE wrapper software released under the terms of the GNU General Public License. See: http://jsmooth.sourceforge.net

Level 2 –creates an EXE wrapper that includes a JRE.

Launch4j – is a Java to EXE product that works at a level 1 or level 2. It can create EXE files and similar files for several different platforms. It can bundle java application to look for a host JRE or can bundle the JRE within the EXE file. See http://launch4j.sourceforge.net

Level 3 – creates an EXE file in the traditional manner, compiling Java code into a native machine code program, and then building an executable from it for a specific platform.

Excelsior JET is a Level 3 product described by the vendor as a "*Java to native code compiler*". It is regarded by many as the most professional (and most expensive) java to EXE product.
See: http://www.excelsiorjet.com

GCJ, the GNU Compiler for Java is open source software that, according to their Website can "*compile Java source code to Java bytecode (class files) or directly to native machine code, and Java bytecode to native machine code."* However, it works best with console I/O and does not work very well with most AWT and Swing classes. It is part of the *GCC GNU Compiler Collection*.
See: http://gcc.gnu.org/java

Chapter Review

Section 1 of this chapter discussed the traditional approach to the creation of executable files, how java s approach differs, and several way that Java application can be deployed. Key terms included: executable file**,** Java Archive file**,** JAR**,** Java applet**,** Java Web Start**,** Java enterprise application**,** tier**,** server-side application**,** client-side applications**,** Web Archive files – WAR files.

Section 2 described the creation and use of Java Archive Files (JAR files). Key terms included: manifest**,** clean and build**,** batch file, shell script.

Section 3 discussed tJava applets, their structure, and why they are in decline.

Section 4 describe the use of jav Web Start technology. Key terms included: Java Network Launching Protocol (JNLP), JNLP file, Server side software**,** security certificate**,** self-signed security certificate**,** Java Control Panel.

Section 4 told us what an EXE wrapper is for java software and described three levels of products for converting Java code to EXE files.

Chapter Questions

1. What is the traditional approach for deploying software? What is an executable file?
2. How does the Java approach to software deployment differ from the traditional approach? What design goals contributed to this approach?
3. What are four widely used “official” ways to deploy finished Java software? What other way is used to deploy java applications?
4. What is the difference between a server-side application and a client-side application? How does Java Web Start technology fit in with these two categories?
5. How can we create a JAR file from a NetBeans project? Which Java applications will not run well from a JAR file on Microsoft Windows? What can we do to get such applications to run on the command line from Windows with a single click of an icon?
6. Where should external files that are used by a JAR file application be stored so that we do not need to worry about operating system *path* settings ?
7. What Java technologies have contributed to the decline in demand for Java applets?
8. One of which two methods must be extended to create a java applet? What methods are used in the lifecycle of an applet and what do they do?
9. What does Java Web Start technology do? What Protocol does it use to launch applications? How does it simplify software deployment?
10. What type of java applications run with Java Web Start technology? What usually happens if we try to run a Java application with console input using Java Web Start technology?
11. With default Java security settings, what type of security certificate is needed to run a java application? How can we change this?

12. What is an EXE wrapper for a Java application? How do the three levels of products that create EXE files from Java code differ? Why would someone want to convert java code to an EXE file in Windows?

Chapter Exercises

1. **A JOptionPane JAR File**

The NetBeans project ***jOptionPaneWindowSamples*** is included with the files for Chapter 14 as a zipped folder. It demonstrates the use of different types of *JOptionPane* message boxes. Download and unzip the file, then create and run a JAR file from the application.

2. A JAR application to Sort an External File

The Netbeans project ***tutorials*** is included with the files for Chapter 14 as a zipped file. It has a text file name "*unsorted.txt*" with a list of programming language tutorials and a Java application that will sort the list, display the file using console I/O, and write a copy of the sorted file with the name "*tutorials.txt*".

Your task is to create and run a JAR file for the application. You will need to put the input data file in the same folder as the JAR file, and create a batch file to run the application from the command line so that we can see the console I/O.

You should get the application to run from the JAR file, then zip the necessary files (JAR file, input text file, batch file) together in a single package that can be deployed to other systems. The JAR file will run on almost any system with Java, but the batch file will only run on Microsoft Windows systems. If you are familiar with Linux, you could create a shell script instead of a batch file to run the application. A real distribution might include both the shell script and the batch file, as well as the JAR file and data file.

You should submit a zipped file with the JAR file, the input text file and the batch file (or shell script file).

3. A Webstart Application that Demonstrates Java Graphics commands

The Netbeans project ***drawDemo*** is included with the files for Chapter 14 as a zipped file. It demonstrates commands that can be used with a *paint()* method to draw on *aJFrame* canvas.

Your task is to prepare the application to run as a WebStart application, then get it to launch from the HTML file. It uses the external data file ***logo.jpg***, so this file will need to be in the same folder as the finished HTML and JNLP files.

You should submit a zipped file with the JAR file, the JNLP file, the HTML file, and the *logo.jpg* file.