

17 package, jars and deployment

Release Your Code



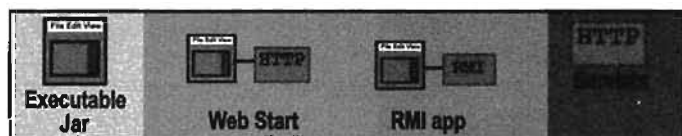
It's time to let go. You wrote your code. You tested your code. You refined your code.

You told everyone you know that if you never saw a line of code again, that'd be fine. But in the end, you've created a work of art. The thing actually runs! But now what? *How* do you give it to end users? *What* exactly do you give to end users? What if you don't even know who your end users are? In these final two chapters, we'll explore how to organize, package, and deploy your Java code. We'll look at local, semi-local, and remote deployment options including executable jars, Java Web Start, RMI, and Servlets. In this chapter, we'll spend most of our time on organizing and packaging your code—things you'll need to know regardless of your ultimate deployment choice. In the final chapter, we'll finish with one of the coolest things you can do in Java. Relax. Releasing your code is not saying goodbye. There's always maintenance...

Deploying your application

What exactly *is* a Java application? In other words, once you're done with development, what is it that you deliver? Chances are, your end-users don't have a system identical to yours. More importantly, they don't have your application. So now it's time to get your program in shape for deployment into The Outside World. In this chapter, we'll look at local deployments, including Executable Jars and the part-local/part-remote technology called Java Web Start. In the next chapter, we'll look at the more remote deployment options, including RMI and Servlets.

Deployment options



100% Local Combination 100% Remote

① Local

The entire application runs on the end-user's computer, as a stand-alone, probably GUI, program, deployed as an executable JAR (we'll look at JAR in a few pages.)

② Combination of local and remote

The application is distributed with a client portion running on the user's local system, connected to a server where other parts of the application are running.

③ Remote

The entire Java application runs on a server system, with the client accessing the system through some non-Java means, probably a web browser.

But before we really get into the whole deployment thing, let's take a step back and look at what happens when you've finished programming your app and you simply want to pull out the class files to give them to an end-user. What's really *in* that working directory?

A Java program is a bunch of classes. That's the output of your development.

The real question is what to do with those classes when you're done.



Brain Barbell

What are the advantages and disadvantages of delivering your Java program as a local, stand-alone application running on the end-user's computer?

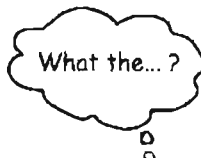
What are the advantages and disadvantages of delivering your Java program as web-based system where the user interacts with a web browser, and the Java code runs as servlets on the server?



Imagine this scenario...

Bob's happily at work on the final pieces of his cool new Java program. After weeks of being in the "I'm-just-one-compile-away" mode, this time he's really done. The program is a fairly sophisticated GUI app, but since the bulk of it is Swing code, he's made only nine classes of his own.

At last, it's time to deliver the program to the client. He figures all he has to do is copy the nine class files, since the client already has the Java API installed. He starts by doing an `ls` on the directory where all his files are...



Whoa! Something strange has happened. Instead of 18 files (nine source code files and nine compiled class files), he sees 31 files, many of which have very strange names like:

`Account$FileListener.class`

`Chart$SaveListener.class`

and on it goes. He had completely forgotten that the compiler has to generate class files for all those inner class GUI event listeners he made, and that's what all the strangely-named classes are.

Now he has to carefully extract all the class files he needs. If he leaves even one of them out, his program won't work. But it's tricky since he doesn't want to accidentally send the client one of his *source* code files, yet everything is in the same directory in one big mess.

Separate source code and class files

A single directory with a pile of source code and class files is a mess. It turns out, Bob should have been organizing his files from the beginning, keeping the source code and compiled code separate. In other words, making sure his compiled class files didn't land in the same directory as his source code.

The key is a combination of directory structure organization and the `-d` compiler option.

There are dozens of ways you can organize your files, and your company might have a specific way they want you to do it. We recommend an organizational scheme that's become almost standard, though.

With this scheme, you create a project directory, and inside that you create a directory called **source** and a directory called **classes**. You start by saving your source code (.java files) into the **source** directory. Then the trick is to compile your code in such a way that the output (the .class files) ends up in the **classes** directory.

And there's a nice compiler flag, `-d`, that lets you do that.



Compiling with the `-d` (directory) flag

```
%cd MyProject/source
%javac -d ../classes MyApp.java
```

tells the compiler to put the compiled code (class files) into the "classes" directory that's one directory up and back down again from the current working directory.

the last thing is still the name of the java file to compile

By using the `-d` flag, you get to decide which *directory* the compiled code lands in, rather than accepting the default of class files landing in the same directory as the source code. To compile all the java files in the source directory, use:

```
%javac -d ../classes *.java
```

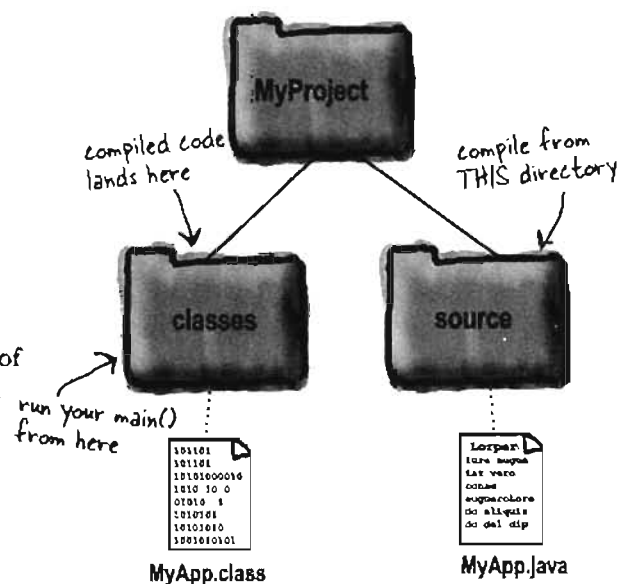
*.java compiles ALL source files in the current directory

Running your code

```
%cd MyProject/classes
```

```
%java Mini
```

run your program from the 'classes' directory.



(troubleshooting note: everything in this chapter assumes that the current working directory (i.e. the ".") is in your classpath. If you have explicitly set a classpath environment variable, be certain that it contains the ".")

Put your Java in a JAR



A **JAR** file is a Java **AR**chive. It's based on the pkzip file format, and it lets you bundle all your classes so that instead of presenting your client with 28 class files, you hand over just a single JAR file. If you're familiar with the tar command on UNIX, you'll recognize the jar tool commands. (Note: when we say JAR in all caps, we're referring to the archive *file*. When we use lowercase, we're referring to the *jar tool* you use to create JAR files.)

The question is, what does the client *do* with the JAR? How do you get it to *run*?

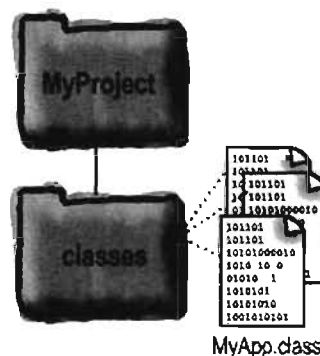
You make the JAR *executable*.

An executable JAR means the end-user doesn't have to pull the class files out before running the program. The user can run the app while the class files are still in the JAR. The trick is to create a *manifest* file, that goes in the JAR and holds information about the files in the JAR. To make a JAR executable, the manifest must tell the JVM *which class has the main() method!*

Making an executable JAR

- **Make sure all of your class files are in the classes directory**

We're going to refine this in a few pages, but for now, keep all your class files sitting in the directory named 'classes'.

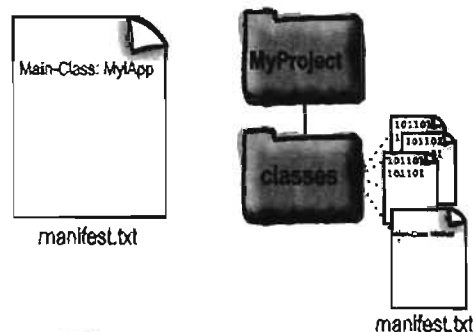


- **Create a manifest.txt file that states which class has the main() method**

Make a text file named manifest.txt that has a one line:

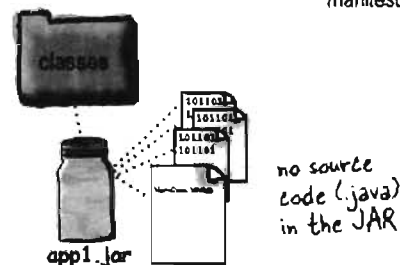
Main-Class: MyApp ← don't put the .class on the end

Press the return key after typing the Main-Class line, or your manifest may not work correctly. Put the manifest file into the "classes" directory.

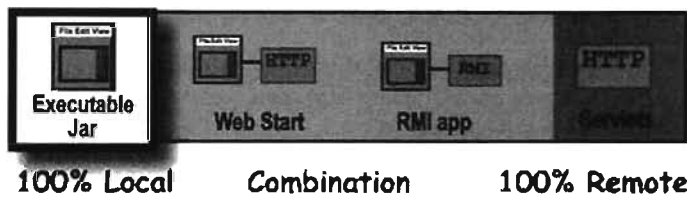


- **Run the jar tool to create a JAR file that contains everything in the classes directory, plus the manifest.**

```
%cd MiniProject/classes
%jar -cvmf manifest.txt app1.jar *.class
OR
%jar -cvmf manifest.txt app1.jar MyApp.class
```



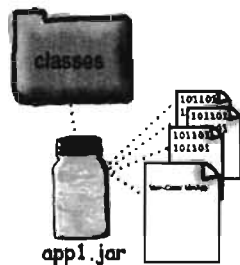
executable JAR



Most 100% local Java apps are deployed as executable JAR files.

Running (executing) the JAR

Java (the JVM) is capable of loading a class from a JAR, and calling the `main()` method of that class. In fact, the entire application can *stay* in the JAR. Once the ball is rolling (i.e., the `main()` method starts running), the JVM doesn't care *where* your classes come from, as long as it can find them. And one of the places the JVM looks is within any JAR files in the classpath. If it can *see* a JAR, the JVM will *look* in that JAR when it needs to find and load a class.



`%cd MyProject/classes`

`%java -jar appl.jar`

The `-jar` flag tells the JVM you're giving it a JAR instead of a class.

The JVM looks inside this JAR for a manifest with an entry for `Main-Class`. If it doesn't find one, you get a runtime exception.

The JVM has to 'see' the JAR, so it must be in your classpath. The easiest way to make the JAR visible is to make your working directory the place where the JAR is.

Depending on how your operating system is configured, you might even be able to simply double-click the JAR file to launch it. This works on most flavors of Windows, and Mac OS X. You can usually make this happen by selecting the JAR and telling the OS to "Open with..." (or whatever the equivalent is on your operating system).

there are no Dumb Questions

Q: Why can't I just JAR up an entire directory?

A: The JVM looks inside the JAR and expects to find what it needs *right there*. It won't go digging into other directories, unless the class is part of a package, and even *then* the JVM looks only in the directories that match the package statement?

Q: What did you just say?

A: You can't put your class files into some arbitrary directory and JAR them up that way. But if your classes belong to packages, you can JAR up the entire package directory structure. In fact, you *must*. We'll explain all this on the next page, so you can relax.

Put your classes in packages!

So you've written some nicely reusable class files, and you've posted them in your internal development library for other programmers to use. While basking in the glow of having just delivered some of the (in your humble opinion) best examples of OO ever conceived, you get a phone call. A frantic one. Two of your classes have the same name as the classes Fred just delivered to the library. And all hell is breaking loose out there, as naming collisions and ambiguities bring development to its knees.

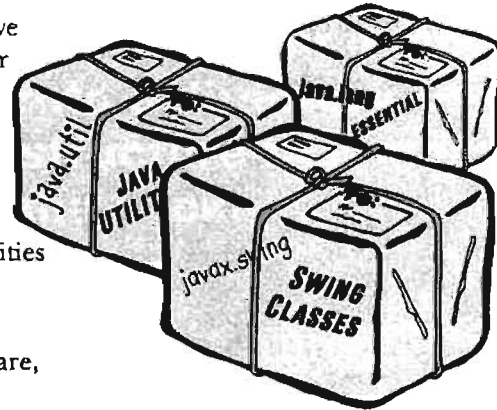
And all because you didn't use packages! Well, you did use packages, in the sense of using classes in the Java API that are, of course, in packages. But you didn't put your own classes into packages, and in the Real World, that's Really Bad.

We're going to modify the organizational structure from the previous pages, just a little, to put classes into a package, and to JAR the entire package. Pay very close attention to the subtle and picky details. Even the tiniest deviation can stop your code from compiling and/or running.

Packages prevent class name conflicts

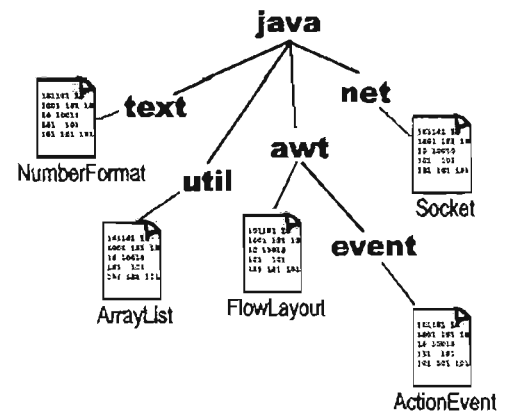
Although packages aren't just for preventing name collisions, that's a key feature. You might write a class named `Customer` and a class named `Account` and a class named `ShoppingCart`. And what do you know, half of all developers working in enterprise e-commerce have probably written classes with those names. In an OO world, that's just dangerous. If part of the point of OO is to write reusable components, developers need to be able to piece together components from a variety of sources, and build something new out of them. Your components have to be able to 'play well with others', including those you didn't write or even know about.

Remember way back in chapter 6 when we discussed how a package name is like the full name of a class, technically known as the *fully-qualified name*. Class `ArrayList` is really `java.util.ArrayList`, `JButton` is really `javax.swing.JButton`, and `Socket` is really `java.net.Socket`. Notice that two of those classes, `ArrayList` and `Socket`, both have `java` as their "first name". In other words, the first part of their fully-qualified names is "java". Think of a hierarchy when you think of package structures, and organize your classes accordingly.



Package structure of the Java API for:

```
java.text.NumberFormat
java.util.ArrayList
java.awt.FlowLayout
java.awt.event.ActionEvent
java.net.Socket
```



What does this picture look like to you? Doesn't it look a whole lot like a directory hierarchy?



Preventing package name conflicts

Putting your class in a package reduces the chances of naming conflicts with other classes, but what's to stop two programmers from coming up with identical *package* names? In other words, what's to stop two programmers, each with a class named `Account`, from putting the class in a package named `shopping.customers`? Both classes, in that case, would *still* have the same name:

`shopping.customers.Account`

Sun strongly suggests a package naming convention that greatly reduces that risk—prepend every class with your reverse domain name. Remember, domain names are guaranteed to be unique. Two different guys can be named Bartholomew Simpson, but two different domains cannot be named `doh.com`.

Packages can prevent name conflicts, but only if you choose a package name that's guaranteed to be unique. The best way to do that is to preface your packages with your reverse domain name.

`com.headfirstbooks.Book`
package name class name

Reverse domain package names

`com.headfirstjava.projects.Chart`

start the package with your reverse domain, separated by a dot (.), then add your own organizational structure after that

the class name is always capitalized

`projects.Chart` might be a common name, but adding `com.headfirstjava` means we have to worry about only our own in-house developers.

To put your class in a package:

1 Choose a package name

We're using `com.headfirstjava` as our example. The class name is `PackageExercise`, so the fully-qualified name of the class is now: `com.headfirstjava.PackageExercise`.

2 Put a package statement in your class

It must be the first statement in the source code file, above any import statements. There can be only one package statement per source code file, so all classes in a source file must be in the same package. That includes inner classes, of course.

```
package com.headfirstjava;

import javax.swing.*;

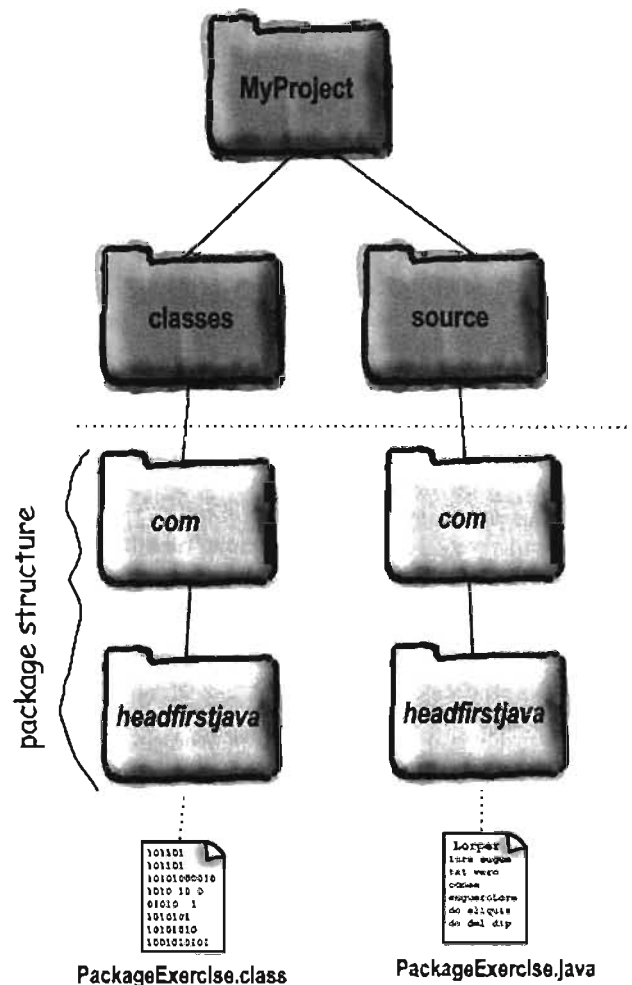
public class PackageExercise {
    // life-altering code here
}
```

3 Set up a matching directory structure

It's not enough to say your class is in a package, by merely putting a package statement in the code. Your class isn't *truly* in a package until you put the class in a matching directory structure. So, if the fully-qualified class name is `com.headfirstjava.PackageExercise`, you must put the `PackageExercise` source code in a directory named `headfirstjava`, which must be in a directory named `com`.

It is possible to compile without doing that, but trust us—it's not worth the other problems you'll have. Keep your source code in a directory structure that matches the package structure, and you'll avoid a ton of painful headaches down the road.

You must put a class into a directory structure that matches the package hierarchy.



Set up a matching directory structure for both the source and classes trees.

Compiling and running with packages

When your class is in a package, it's a little trickier to compile and run. The main issue is that both the compiler and JVM have to be capable of finding your class and all of the other classes it uses. For the classes in the core API, that's never a problem. Java always knows where its own stuff is. But for your classes, the solution of compiling from the same directory where the source files are simply won't work (or at least not *reliably*). We guarantee, though, that if you follow the structure we describe on this page, you'll be successful. There are other ways to do it, but this is the one we've found the most reliable and the easiest to stick to.

Compiling with the `-d` (directory) flag

`%cd MyProject/source` ← stay in the source directory! Do NOT `cd` down into the directory where the .java file is!

`%javac -d ../classes com/headfirstjava/PackageExercise.java`

tells the compiler to put the compiled code (class files) into the classes directory, within the right package structure!! Yes, it knows.

Now you have to specify the PATH to get to the actual source file.

To compile all the java files in the `com.headfirstjava` package, use:

`%javac -d ../classes com/headfirstjava/*.java`

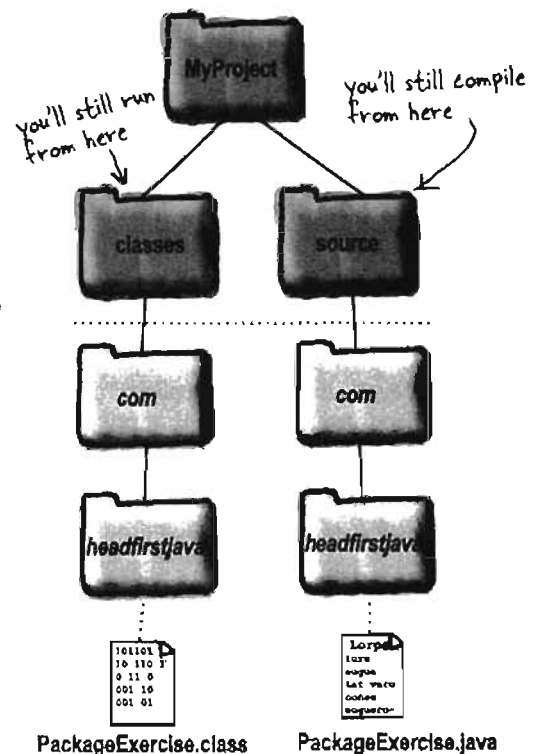
compiles every source (.java) file in this directory

Running your code

`%cd MyProject/classes` run your program from the 'classes' directory.

`%java com.headfirstjava.PackageExercise`

You **MUST** give the fully-qualified class name! The JVM will see that, and immediately look inside its current directory (classes) and expect to find a directory named `com`, where it expects to find a directory named `headfirstjava`, and in there it expects to find the class. If the class is in the "com" directory, or even in "classes", it won't work!

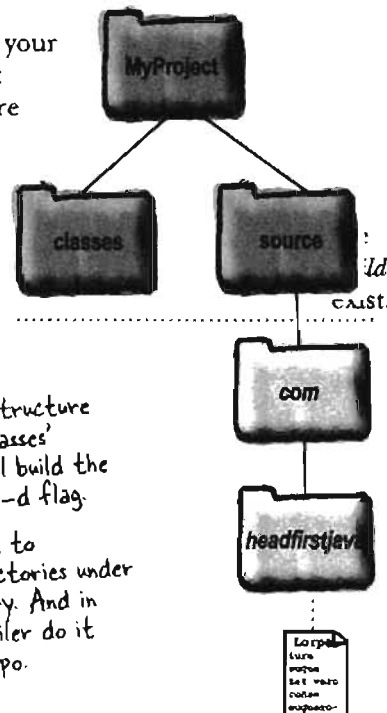


The -d flag is even cooler than we said

Compiling with the `-d` flag is wonderful because not only does it let you send your compiled class files into a directory other than the one where the source file is, but it also knows to put the class into the correct directory structure for the package the class is in.

But it gets even better!

Let's say that you have a nice directory structure all set up for your source code. But you haven't set up a matching directory structure for your classes directory. Not a problem! Compiling with `-d` tells the compiler to not just put your classes into correct directory tree, but to create the directories if they don't



If the package directory structure doesn't exist under the 'classes' directory, the compiler will build the directories if you use the `-d` flag.

So you don't actually have to physically create the directories under the 'classes' root directory. And in fact, if you let the compiler do it there's no chance of a typo.

The `-d` flag tells the compiler, "Put the class into its package directory structure, using the class specified after the `-d` as the root directory. But... if the directories aren't there, create them first and then put the class in the right place!"

there are no Dumb Questions

Q: I tried to `cd` into the directory where my main class was, but now the JVM says it can't find my class! But it's right THERE in the current directory!

A: Once your class is in a package, you can't call it by its 'short' name. You MUST specify, at the command-line, the fully-qualified name of the class whose `main()` method you want to run. But since the fully-qualified name includes the *package* structure, Java insists that the class be in a matching *directory* structure. So if at the command-line you say:

```
% java com.foo.Book
```

the JVM will look in its current directory (and the rest of its classpath), for a directory named "com". **It will not look for a class named Book, until it has found a directory named "com" with a directory inside named "foo".** Only then will the JVM accept that it's found the correct Book class. If it finds a Book class anywhere else, it assumes the class isn't in the right structure, even if it is! The JVM won't for example, look back up the directory tree to say, "Oh, I can see that above us is a directory named com, so this must be the right package..."

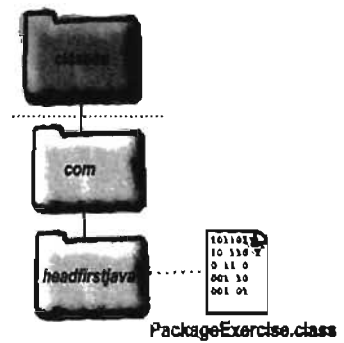
Making an executable JAR with packages



When your class is in a package, the package directory structure must be inside the JAR! You can't just pop your classes in the JAR the way we did pre-packages. And you must be sure that you don't include any other directories above your package. The first directory of your package (usually `com`) must be the first directory within the JAR! If you were to accidentally include the directory *above* the package (e.g. the "classes" directory), the JAR wouldn't work correctly.

Making an executable JAR

- Make sure all of your class files are within the correct package structure, under the classes directory.

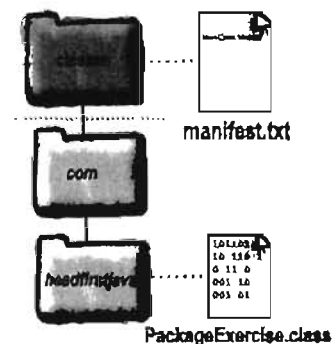


- Create a manifest.txt file that states which class has the main() method, and be sure to use the fully-qualified class name!

Make a text file named `manifest.txt` that has a single line:

```
Main-Class: com.headfirstjava.PackageExercise
```

Put the manifest file into the classes directory



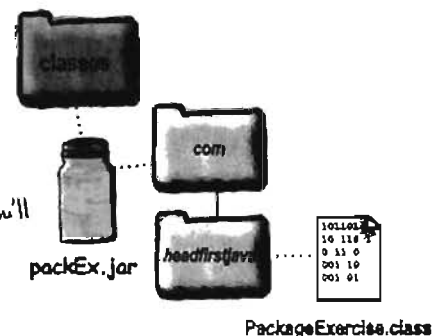
- Run the jar tool to create a JAR file that contains the package directories plus the manifest

The only thing you need to include is the 'com' directory, and the entire package (and all classes) will go into the JAR.

```
%cd MyProject/classes
```

```
%jar -cvmf manifest.txt packEx.jar com
```

All you specify is the com directory! And you'll get everything in it!



So where did the manifest file go?

Why don't we look inside the JAR and find out? From the command-line, the jar tool can do more than just create and run a JAR. You can extract the contents of a JAR (just like 'unzipping' or 'untarring').

Imagine you've put the packEx.jar into a directory named Skyler.

jar commands for listing and extracting

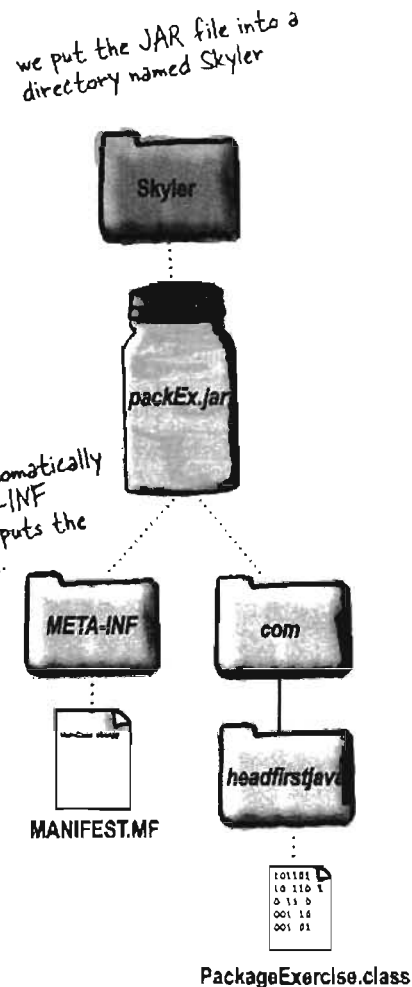
① List the contents of a JAR

```
% jar -tf packEx.jar
```

↑ -tf stands for 'Table File' as in "show me a table of the JAR file"

```
File Edit Window Help Pickle
% cd Skyler
% jar -tf packEx.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/headfirstjava/
com/headfirstjava/
PackageExercise.class
```

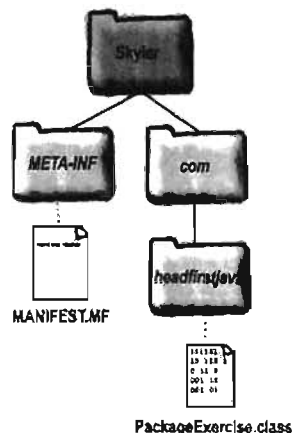
the jar tool automatically builds a META-INF directory, and puts the manifest inside.



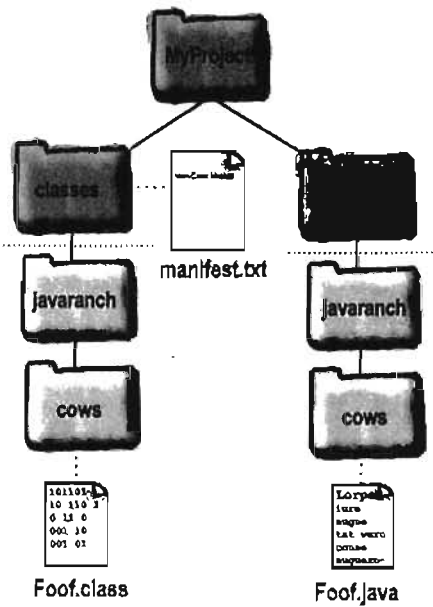
② Extract the contents of a JAR (i.e. unjar)

```
% cd Skyler
% jar -xf packEx.jar
```

↑ -xf stands for 'Extract File' and it works just like unzipping or untarring. If you extract the packEx.jar, you'll see the META-INF directory and the com directory in your current directory



META-INF stands for 'meta information'. The jar tool creates the META-INF directory as well as the MANIFEST.MF file. It also takes the contents of your manifest file, and puts it into the MANIFEST.MF file. So, your manifest file doesn't go into the JAR, but the contents of it are put into the 'real' manifest (MANIFEST.MF).



Given the package/directory structure in this picture, figure out what you should type at the command-line to compile, run, create a JAR, and execute a JAR. Assume we're using the standard where the package directory structure starts just below *source* and *classes*. In other words, the *source* and *classes* directories are not part of the package.

Compile:

```
%cd source
%javac _____
```

Run:

```
%cd _____
%java _____
```

Create a JAR

```
%cd _____
% _____
```

Execute a JAR

```
%cd _____
% _____
```

Bonus question: What's wrong with the package name?

there are no Dumb Questions

Q: What happens if you try to run an executable JAR, and the end-user doesn't have Java installed?

A: Nothing will run, since without a JVM, Java code can't run. The end-user must have Java installed.

Q: How can I get Java installed on the end-user's machine?

Ideally, you can create a custom installer and distribute it along with your application. Several companies offer installer programs ranging from simple to extremely powerful. An installer program could, for example, detect whether or not the end-user has an appropriate version of Java installed, and if not, install and configure Java before installing your application. InstallShield, InstallAnywhere, and DeployDirector all offer Java installer solutions.

Another cool thing about some of the installer programs is that you can even make a deployment CD-ROM that includes installers for all major Java platforms, so... one CD to rule them all. If the user's running on Solaris, for example, the Solaris version of Java is installed. On Windows, the Windows version, etc. If you have the budget, this is by far the easiest way for your end-users to get the right version of Java installed and configured.

BULLET POINTS

- Organize your project so that your source code and class files are not in the same directory.
- A standard organization structure is to create a *project* directory, and then put a *source* directory and a *classes* directory inside the project directory.
- Organizing your classes into packages prevents naming collisions with other classes, if you prepend your reverse domain name on to the front of a class name.
- To put a class in a package, put a package statement at the top of the source code file, before any import statements:
`package com.wickedlysmart;`
- To be in a package, a class must be in a *directory structure that exactly matches the package structure*. For a class, `com.wickedlysmart.Foo`, the `Foo` class must be in a directory named *wickedlysmart*, which is in a directory named *com*.
- To make your compiled class land in the correct package directory structure under the *classes* directory, use the `-d` compiler flag:

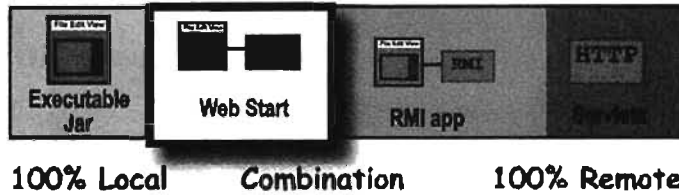
```
% cd source
% javac -d ../classes com/wickedlysmart/Foo.java
```
- To run your code, `cd` to the *classes* directory, and give the fully-qualified name of your class:

```
% cd classes
% java com.wickedlysmart.Foo
```
- You can bundle your classes into JAR (Java ARchive) files. JAR is based on the `pkzip` format.
- You can make an executable JAR file by putting a manifest into the JAR that states which class has the `main()` method. To create a manifest file, make a text file with an entry like the following (for example):
`Main-Class: com.wickedlysmart.Foo`
- Be sure you hit the return key after typing the `Main-Class` line, or your manifest file may not work.
- To create a JAR file, type:
`jar -cvfm manifest.txt MyJar.jar com`
- The entire package directory structure (and *only* the directories matching the package) must be immediately inside the JAR file.
- To run an executable JAR file, type:
`java -jar MyJar.jar`

wouldn't it be dreamy...

Executable JAR files are nice, but wouldn't it be dreamy if there were a way to make a rich, stand-alone client GUI that could be distributed over the Web? So that you wouldn't have to press and distribute all those CD-ROMs. And wouldn't it be just wonderful if the program could automatically update itself, replacing just the pieces that changed? The clients would always be up-to-date, and you'd never have to worry about delivering new





Java Web Start

With Java Web Start (JWS), your application is launched for the first time from a Web browser (get it? *Web Start?*) but it runs as a stand-alone application (well, *almost*), without the constraints of the browser. And once it's downloaded to the end-user's machine (which happens the first time the user accesses the browser link that starts the download), it *stays* there.

Java Web Start is, among other things, a small Java program that lives on the client machine and works much like a browser plug-in (the way, say, Adobe Acrobat Reader opens when your browser gets a .pdf file). This Java program is called the **Java Web Start 'helper app'**, and its key purpose is to manage the downloading, updating, and launching (executing) of *your* JWS apps.

When JWS downloads your application (an executable JAR), it invokes the `main()` method for your app. After that, the end-user can launch your application directory from the JWS helper app *without* having to go back through the Web page link.

But that's not the best part. The amazing thing about JWS is its ability to detect when even a small part of application (say, a single class file) has changed on the server, and—without any end-user intervention—download and integrate the updated code.

There's still an issue, of course, like how does the end-user *get* Java and Java Web Start? They need both—Java to run the app, and Java Web Start (a small Java application itself) to handle retrieving and launching the app. But even *that* has been solved. You can set things up so that if your end-users don't have JWS, they can download it from Sun. And if they *do* have JWS, but their version of Java is out-of-date (because you've specified in your JWS app that you need a specific version of Java), the Java 2 Standard Edition can be downloaded to the end-user machine.

Best of all, it's simple to use. You can serve up a JWS app much like any other type of Web resource such as a plain old HTML page or a JPEG image. You set up a Web (HTML) page with a link to your JWS application, and you're in business.

In the end, your JWS application isn't much more than an executable JAR that end-users can download from the Web.

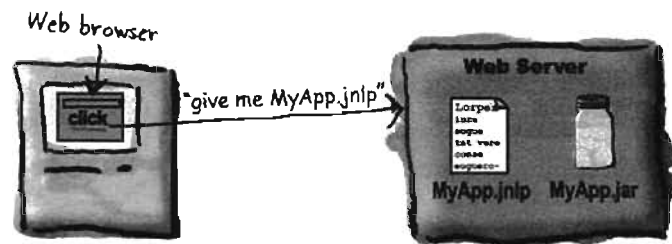
End-users launch a Java Web Start app by clicking on a link in a Web page. But once the app downloads, it runs outside the browser, just like any other stand-alone Java application. In fact, a Java Web Start app is just an executable JAR that's distributed over the Web.

How Java Web Start works

- 1 The client clicks on a Web page link to your JWS application (a .jnlp file).

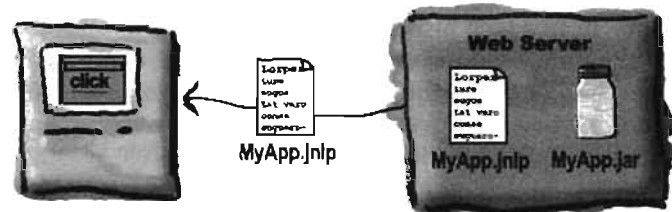
The Web page link

```
<a href="MyApp.jnlp">Click</a>
```



- 2 The Web server (HTTP) gets the request and sends back a .jnlp file (this is NOT the JAR).

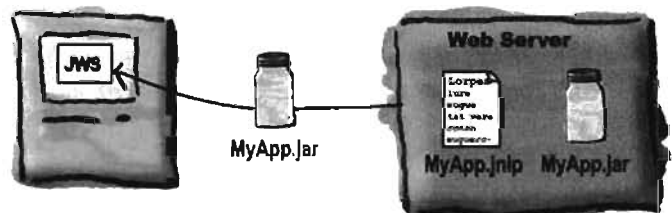
The .jnlp file is an XML document that states the name of the application's executable JAR file.



- 3 Java Web Start (a small 'helper app' on the client) is started up by the browser. The JWS helper app reads the .jnlp file, and asks the server for the MyApp.jar file.



- 4 The Web server 'serves' up the requested .jar file.



- 5 Java Web Start gets the JAR and starts the application by calling the specified main() method (just like an executable JAR).

Next time the user wants to run this app, he can open the Java Web Start application and from there launch your app, without even being online.

HelloWebStart (the app in the JAR)



The .jnlp file

To make a Java Web Start app, you need to .jnlp (Java Network Launch Protocol) file that describes your application. This is the file the JWS app reads and uses to find your JAR and launch the app (by calling the JAR's `main()` method). A .jnlp file is a simple XML document that has several different things you can put in, but as a minimum, it should look like this:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<jnlp spec="0.2 1.0"
```

```
  codebase="http://127.0.0.1/~kathy"
```

```
  href="MyApp.jnlp">
```

The 'codebase' tag is where you specify the 'root' of where your web start stuff is on the server. We're testing this on our localhost, so we're using the local loopback address "127.0.0.1". For web start apps on our internet web server, this would say, "http://www.wickedlysmart.com"

This is the location of the .jnlp file relative to the codebase. This example shows that MyApp.jnlp is available in the root directory of the web server, not nested in some other directory.

```
<information>
```

```
  <title>kathy App</title>
```

```
  <vendor>Wickedly Smart</vendor>
```

```
  <homepage href="index.html"/>
```

```
  <description>Head First WebStart demo</description>
```

```
  <icon href="kathys.gif"/>
```

```
  <offline-allowed/>
```

```
</information>
```

Be sure to include all of these tags, or your app might not work correctly! The 'information' tags are used by the JWS helper app, mostly for displaying when the user wants to relaunch a previously-downloaded application.

This means the user can run your program without being connected to the internet. If the user is offline, it means the automatic-updating feature won't work.

```
<resources>
```

```
  <j2se version="1.3+"/>
```

This says that your app needs version 1.3 of Java, or greater.

```
  <jar href="MyApp.jar"/>
```

```
</resources>
```

The name of your executable JAR! You might have other JAR files as well, that hold other classes or even sounds and images used by your app.

```
<application-desc main-class="HelloWebStart"/>
```

```
</jnlp>
```

This is like the manifest Main-Class entry... it says which class in the JAR has the `main()` method.

Steps for making and deploying a Java Web Start app

- ① Make an executable JAR for your application.



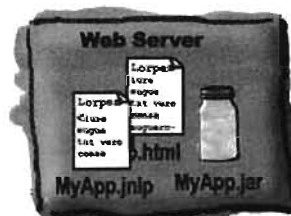
MyApp.jar

- ② Write a .jnlp file.



MyApp.jnlp

- ③ Place your JAR and .jnlp files on your Web server.



- ④ Add a new mime type to your Web server.
application/x-java-jnlp-file

This causes the server to send the .jnlp file with the correct header, so that when the browser receives the .jnlp file it knows what it is and knows to start the JWS helper app.



- ⑤ Create a Web page with a link to your .jnlp file



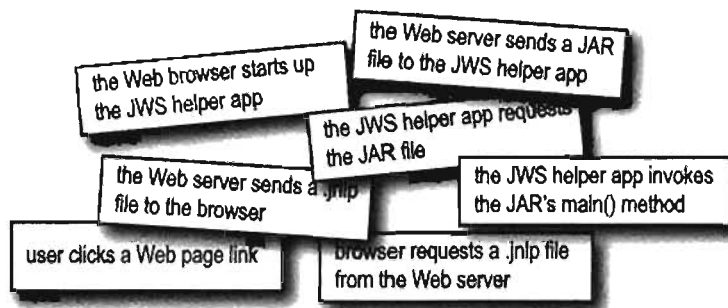
MyJWSApp.html

```
<HTML>
<BODY>
  <a href="MyApp2.jnlp">Launch My Application</a>
</BODY>
</HTML>
```



What's First?

Look at the sequence of events below, and place them in the order in which they occur in a JWS application.



package, jars and deployment

- 1.
- 2.
- 3.
- 4.
- 5.
- 6.
- 7.

there are no Dumb Questions

Q: How is Java Web Start different from an applet?

A: Applets can't live outside of a Web browser. An applet is downloaded from the Web as part of a Web page rather than simply from a Web page. In other words, to the browser, the applet is just like a JPEG or any other resource. The browser uses either a Java plug-in or the browser's own built-in Java (far less common today) to run the applet. Applets don't have the same level of functionality for things such as automatic updating, and they must always be launched from the browser. With JWS applications, once they're downloaded from the Web, the user doesn't even have to be using a browser to relaunch the application locally. Instead, the user can start up the JWS helper app, and use it to launch the already-downloaded application again.

Q: What are the security restrictions of JWS?

A: JWS apps have several limitations including being restricted from reading and writing to the user's hard drive. But... JWS has its own API with a special open and save dialog box so that, with the user's permission, your app can save and read its own files in a special, restricted area of the user's drive.

BULLET POINTS

- Java Web Start technology lets you deploy a stand-alone client application from the Web.
- Java Web Start includes a 'helper app' that must be installed on the client (along with Java).
- A Java Web Start (JWS) app has two pieces: an executable JAR and a .jnlp file.
- A .jnlp file is a simple XML document that describes your JWS application. It includes tags for specifying the name and location of the JAR, and the name of the class with the main() method.
- When a browser gets a .jnlp file from the server (because the user clicked on a link to the .jnlp file), the browser starts up the JWS helper app.
- The JWS helper app reads the .jnlp file and requests the executable JAR from the Web server.
- When the JWS gets the JAR, it invokes the main() method (specified in the .jnlp file).

exercise: True or False



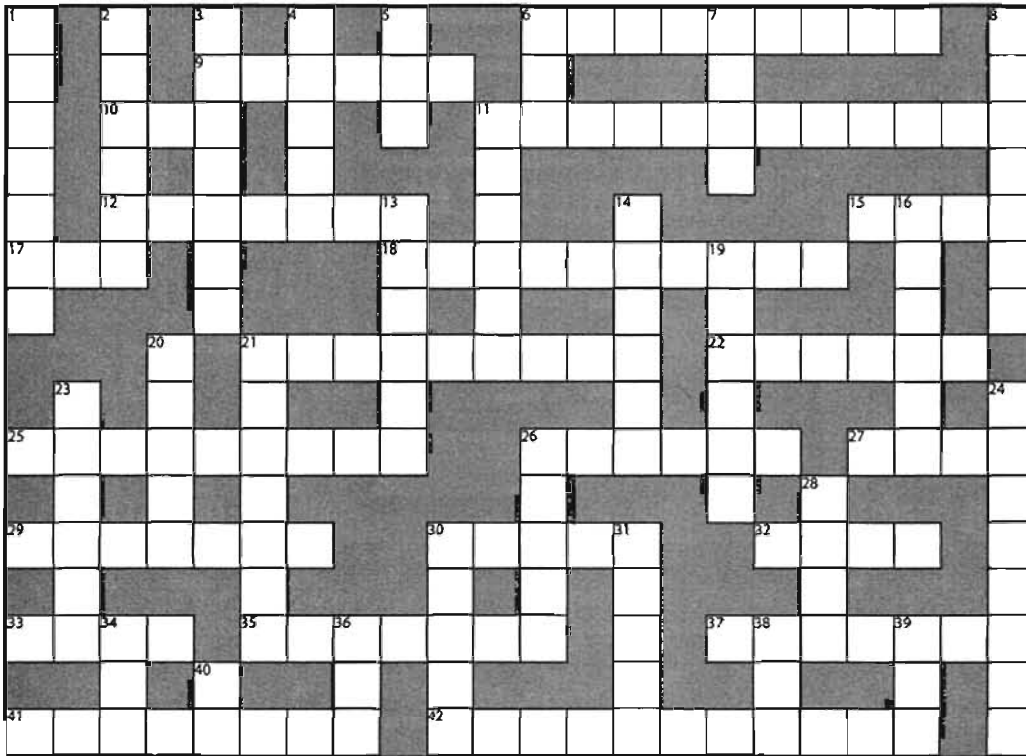
We explored packaging, deployment, and JWS in this chapter. Your job is to decide whether each of the following statements is true or false.

👍 TRUE OR FALSE 👎

1. The Java compiler has a flag, `-d`, that lets you decide where your `.class` files should go.
2. A JAR is a standard directory where your `.class` files should reside.
3. When creating a Java Archive you must create a file called `jar.mf`.
4. The supporting file in a Java Archive declares which class has the `main()` method.
5. JAR files must be unzipped before the JVM can use the classes inside.
6. At the command line, Java Archives are invoked using the `-arch` flag.
7. Package structures are meaningfully represented using hierarchies.
8. Using your company's domain name is not recommended when naming packages.
9. Different classes within a source file can belong to different packages.
10. When compiling classes in a package, the `-p` flag is highly recommended.
11. When compiling classes in a package, the full name must mirror the directory tree.
12. Judicious use of the `-d` flag can help to assure that there are no typos in your class tree.
13. Extracting a JAR with packages will create a directory called `meta-inf`.
14. Extracting a JAR with packages will create a file called `manifest.mf`.
15. The JWS helper app always runs in conjunction with a browser.
16. JWS applications require a `.nlp` (Network Launch Protocol) file to work properly.
17. A JWS's main method is specified in its JAR file.



Summary-Cross 7.0



Anything in the book
is fair game for this
one!

Across

- | | |
|---------------------------|-----------------------|
| 6. Won't travel | 26. Mine is unique |
| 9. Don't split me | 27. GUI's target |
| 10. Release-able | 29. Java team |
| 11. Got the key | 30. Factory |
| 12. I/O gang | 32. For a while |
| 15. Flatten | 33. Atomic * 8 |
| 17. Encapsulated returner | 35. Good as new |
| 18. Ship this one | 37. Pairs event |
| 21. Make it so | 41. Where do I start |
| 22. I/O sieve | 42. A little firewall |
| 25. Disk leaf | |

Down

- | | | |
|-------------------------|-----------------------|-------------------|
| 1. Pushy widgets | 16. Who's allowed | 30. I/O cleanup |
| 2. ____ of my desire | 19. Efficiency expert | 31. Milli-nap |
| 3. 'Abandoned' moniker | 20. Early exit | 34. Trig method |
| 4. A chunk | 21. Common wrapper | 36. Encaps method |
| 5. Math not trig | 23. Yes or no | 38. JNLP format |
| 6. Be brave | 24. Java jackets | 39. VB's final |
| 7. Arrange well | 26. Not behavior | 40. Java branch |
| 8. Swing slang | 28. Socket's suite | |
| 11. I/O canals | | |
| 13. Organized release | | |
| 14. Not for an instance | | |

exercise solutions



Exercise Solutions



1. user clicks a Web page link
2. browser requests a .jnlp file from the Web server
3. the Web server sends a .jnlp file to the browser
4. the Web browser starts up the JWS helper app
5. the JWS helper app requests the JAR file
6. the Web server sends a JAR file to the JWS helper app
7. the JWS helper app invokes the JAR's main() method

- | | |
|--------------|--|
| True | 1. The Java compiler has a flag, <code>-d</code> , that lets you decide where your .class files should go. |
| False | 2. A JAR is a standard directory where your .class files should reside. |
| False | 3. When creating a Java Archive you must create a file called jar.mf. |
| True | 4. The supporting file in a Java Archive declares which class has the main() method. |
| False | 5. JAR files must be unzipped before the JVM can use the classes inside. |
| False | 6. At the command line, Java Archives are invoked using the <code>-arch</code> flag. |
| True | 7. Package structures are meaningfully represented using hierarchies. |
| False | 8. Using your company's domain name is not recommended when naming packages. |
| False | 9. Different classes within a source file can belong to different packages. |
| False | 10. When compiling classes in a package, the <code>-p</code> flag is highly recommended. |
| True | 11. When compiling classes in a package, the full name must mirror the directory tree. |
| True | 12. Judicious use of the <code>-d</code> flag can help to assure that there are no typos in your tree. |
| True | 13. Extracting a JAR with packages will create a directory called meta-inf. |
| True | 14. Extracting a JAR with packages will create a file called manifest.mf. |
| False | 15. The JWS helper app always runs in conjunction with a browser. |
| False | 16. JWS applications require a .nlp (Network Launch Protocol) file to work properly. |
| False | 17. A JWS's main method is specified in its JAR file. |



Summary-Cross 7.0

