Managing Your Programming Environment

Table of Contents

- Managing Your Programming Environment
  - The CLASSPATH Environment Variable
  - Creating and Using Packages
    - Roll Your Own Packages
    - Using Objects and Classes from Other Packages
    - Clarifying Object and Class Names
    - Directory Structure

Managing Your Programming Environment

Table of Contents

Managing Your Programming Environment

For the most part how you organize and manage your programming environment is as personal as decorating your home. However, there are some things that you must do for your classes to work properly with the Java compiler and interpreter, and there are some things that you should do to make your life easier.

The CLASSPATH Environment Variable

You use the environment variable called CLASSPATH to indicate to the interpreter where your classes are located on your system.

Creating and Using Packages

As your code base of Java classes grows, you will need to organize your classes and your directory structure in a way to provide an efficient and convenient programming environment. One of the easiest ways to organize your work is to group related classes and interfaces together into Java packages.

Table of Contents

The CLASSPATH Environment Variable

To run a stand−alone Java application, you specify the name of the Java application that you wish to execute to the Java interpreter. To run an applet, you specify the applet's name with an APP tag within an HTML file. The HotJava browser passes the applet's name to the Java interpreter. In either case, the application or applet that you are running could be anywhere on your system or on the network. Also, the application or applet might use other classes or objects that are in the same or different location.

Because your classes could be anywhere, you must indicate to the Java interpreter where it can find the classes that you are trying to run. You do this with the CLASSPATH environment variable. The CLASSPATH environment variable is comprised of a list of directory names that contain compiled Java classes. The actual construct of CLASSPATH depends on the system you are running.

When the interpreter gets a classname, either from the command line, from the HotJava browser, or from an application or applet, the interpreter searches each directory in the CLASSPATH until it finds the class it's looking for.

You should put the top−level directory that contains your Java classes in your CLASSPATH. By convention, many people have a `classes` directory in their home directory where they put all of the Java code. If you have such a directory, you should put that directory in your CLASSPATH. However, when you are working with applets, it's convenient to put the applet in a `classes` directory underneath the directory where the HTML file is that contains the applet. For this and other reasons, it's often convenient to put dot ('.')−−the current directory−−in the CLASSPATH as well.

The classes included with the Java development environment are automatically available to you because the interpreter automatically appends the correct directory to your CLASSPATH when it starts up.

Order is Important

When the Java interpreter is looking for a class, it searches the directories indicated by your CLASSPATH in order until it finds a class with the correct name. The Java interpreter runs the first class with the correct name that it encounters and does not search the remaining directories. Normally, it's best to give your classes unique names, but if you can't avoid it, make sure that your CLASSPATH searches your classes in the proper order. Keep this in mind when setting up your CLASSPATH and your source code hierarchy.

Managing Your Programming Environment

Table of Contents

Creating and Using Packages

The Java development environment includes several packages of reusable classes. These packages are discussed at length in the previous lesson *Package Tour* . The Java language provides a way for you to create packages of classes and interfaces as well.

Roll Your Own Packages

Packages are groups of related classes and interfaces and provide convenient a mechanism for managing a large set of classes and interfaces and avoiding naming conflicts.

Using the `package` statement you can create a package and indicate that certain classes are part of that package.

Using Objects and Classes from Other Packages

To use objects that are contained in package, whether it's a package provided with the development environment, or one you've created, you need to import the package.

Clarifying Object and Class Names

When two classes in two different packages have the same name, you have to prepend the package name to the class name.

Directory Structure

The interpreter expects the directory structure of classes to be the same as the package hierarchy.

Table of Contents

Roll Your Own Packages

Packages are groups of related classes and interfaces and provide convenient a mechanism for managing a large set of classes and interfaces and avoiding naming conflicts. Several packages are included as part of the Java development environment: the Java language packages, the browser packages, and the window toolkit package just to name a few.

In addition to the Java packages, you can create your own packages using Java's `package` statement. Suppose that you had a group of classes that implement a collection of graphic objects such as circles, rectangles, lines, and points. In addition to these classes you also defined an interface Draggable [PENDING: does this make sense?] It makes sense to collect all of these classes and interfaces together into a package called "Graphics". In this way other programmers can easily determine what the group of classes are for and how they relate to one another and to other classes and packages.

You create a package implicitly when you put a class or classes in it. Here's an outline of the Graphics package described in the previous paragraph.

```
package Graphics;

class Circle {
    . . .
}
```

The first line in the preceding code snippet indicates that all of the classes defined in this compilation unit are members of a package named `Graphics`. The package statement refers to everything in the current compilation unit (in most cases a compilation unit is a file). You can define a class per file and have each class be in the same package.

All classes and interfaces are in a package. Default package.

Using Objects and Classes from Other Packages

To import a specific object in a package, like Applet in the java.applet package, use the `import` statement.

```
import java.applet.Applet;
```

To import all objects in a package, for example, the entire java.applet package, use the `import` statement with the '*' wildcard character.

```
import java.applet.*;
```

Importing an object, or an entire package of objects makes the definitions and implementations available to the *current package*. For example, suppose that you're writing a small Applet:

```
class MyApplet extends Applet {
    . . .
}
```

If you try to compile this applet without importing the java.applet.Applet class, the compiler will issue this fatal error:

```
MyApplet.java:2: Super class Applet of class MyApplet not found.
```

Clarifying Object and Class Names

To disambiguate two classes with the same name you have to prepend the package name to the beginning of the class.

Managing Your Programming Environment

Directory Structure

[PENDING: Under construction.]

Managing Your Programming Environment