

## 5: Using the Javelin Stamp IDE

The Javelin Stamp IDE (Integrated Development Environment) provides a work environment where you can write, run, and debug your Javelin Stamp programs. In addition, you can view the javadoc documentation from within the IDE.

### Starting the IDE

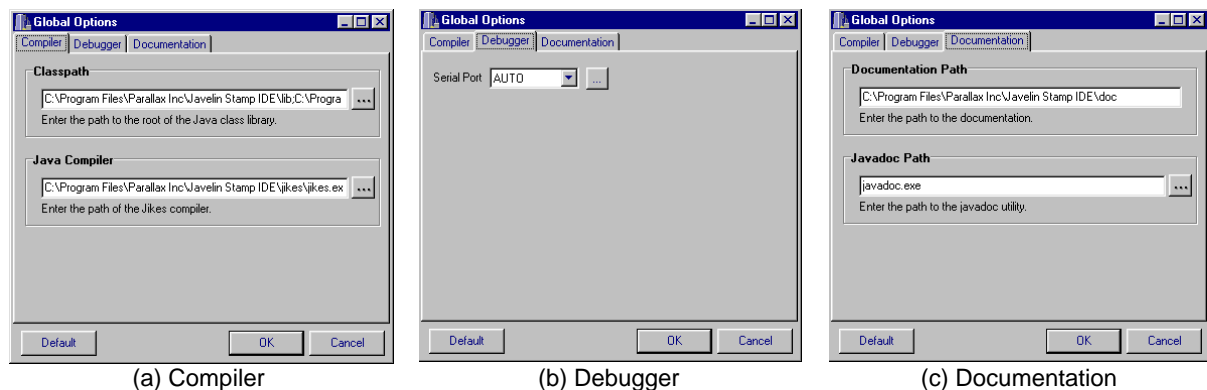
You can run the IDE by selecting the icon from your Start menu. From Windows, press on the Start button on your menu bar. Mouse up to Programs, scroll over and mouse to the Javelin Stamp, scroll over once more and select the IDE and the program will begin. You may wish to maximize the window (double click on the title bar, use the system menu on the left-hand side of the title bar, or use the maximize button to the right-hand side of the title bar).

By default, you'll see two command areas just below the title bar. The first area holds the main menu (which has items for File, Edit, etc.). The second area is a toolbar that has small icons to execute common methods. Below the tool bar, you'll see a tab that reads Untitled.java. This is the name of the file you are editing. If you open multiple files, each will have its own tab and you can switch between them by clicking on the tabs. The area below the tab is where the text will appear. The gray area to the left will contain indicators while debugging, as you'll see shortly.

### Setting Global Options

Before you get started, it is a good idea to review the option settings found within the Global Options... under the Project menu. The dialog (see Figure 5.1) that appears has three tabs. The first tab, Compiler, should contain the Class Path and the path to the compiler. Having the correct Class Path is vital so that the IDE can find the library files required for your programs. Be careful not to change the settings unless you are certain you know what you are doing (you'll learn more about changing the Class Path at the end of this chapter).

Figure 5.1 Global Options for IDE



## 5: Using the Javelin Stamp IDE

---

The Debugger tab has a single option that allows you to set the COM port you've used when connected your Javelin Stamp. The IDE uses this port to communicate with the Javelin Stamp. You can press the Auto button and the IDE will attempt to detect the Javelin Stamp automatically.

The final tab, Documentation, allows you to set the path to the javadoc files and the javadoc program. You'll read more about javadoc later in this chapter.

If you change things inadvertently, you can push the Default button to restore everything to its original state. For now, the only thing you should change is the COM port setting on the Debugger tab.

### Starting a Project

To start a project, you can just begin defining a class in the Untitled.java window. However, it is easier if you use the Insert Template under the File menu to insert a prototypical class into the editor workspace.

Here is the code inserted by the Insert Template command:

```
import stamp.core.*;

/**
 * Put a one line description of your class here.
 * <p>
 * This comment should contain a description of the class. What it
 * is for, what it does, how it use it.
 *
 * You should rename the class and then save it in a file with
 * exactly the same name as the class.
 *
 * @version 1.0 Date
 * @author Your Name Here
 */
public class MyClass {

    // Put variables here.
    static int myVar;

    public static void main() {
        // Your code goes here.
    }

}
```

You'll need to change **MyClass** to an appropriate name. You'll also want to alter the comments and **myVar** variable to suit your program. Java requires that each file have only one public class and that the class have

## 5: Using the Javelin Stamp IDE

exactly the same name as the Java file (including the case of the name). So if your class is **MyFirstClass**, you should save the file as **MyFirstClass.java** Save or Save As under the File menu.

You can also ask the IDE to help you write your code by invoking specific templates. If you press CONTROL+J while editing a file, you'll see a list of templates you can insert. For example, if you select the **for (count)** template, this will appear in your file:

```
for ( int i = 0; i <; i++ ) {  
  
}
```

If you've already typed a partial statement, pressing CONTROL+J will automatically insert the correct template without displaying a list. For example, if you enter **if** and then press CONTROL+J, the IDE will automatically insert the code template for **if**.

Table 5.1 shows the available templates and the keywords that will automatically invoke them.

**Table 5.1: Javelin Templates**

Menu Item	Menu Item	Menu Item
Template	Keyword	Example
Array declaration	Arrayii	int [ ] = {1, 2, 3};
Class declaration	Class	public class {  }
Class declaration (with extend)	Classes	public class extends Object {  };
Complete program	Program	See above example
For statement	For	for ( ; ; ) {  }
For statement (count)	Forc	for ( int i = 0; i <; i++ ) {  }
If	If	if ( ) {  }

## 5: Using the Javelin Stamp IDE

---

Table 5.1: Javelin Templates

Menu Item	Menu Item	Menu Item
If else	lfe	<pre>if ( ) { } else { }</pre>
Try/catch	Tryc	<pre>try { } catch ( ) { }</pre>
Try/catch/finally	Tryf	<pre>try { } catch ( ) { } finally { }</pre>
While	While	<pre>while ( ) { }</pre>
Do while	Whiled	<pre>do { } while ( );</pre>
Switch statement	switch	<pre>switch ( ) { case a: break;  case b: break; }</pre>

## 5: Using the Javelin Stamp IDE

Table 5.1: Javelin Templates

Menu Item	Menu Item	Menu Item
Switch statement (with default)	switchd	<pre>switch ( ) { case a: break;  case b: break;  default:  }</pre>
Method declaration	method	<pre>/**  *  *  * @param  * @return  */ void () {  }</pre>
Method declaration (public)	methodp	<pre>/**  *  *  * @param  * @return  */ public void () {  }</pre>
Method declaration (with throws)	methodt	<pre>/**  *  *  * @param  * @return  */ public void () throws Exception {  }</pre>

## 5: Using the Javelin Stamp IDE

---

Table 5.1: Javelin Templates

Menu Item	Menu Item	Menu Item
Field declaration	field	/** * */ int ;
Debugging output	debug	System.out.println("");

### Building your Program

There are several ways to build your program depending on what you want to do with it. On the Project menu you'll find five important menu items:

- **Compile** – This option simply converts your source code into a class file. This will catch any compile-time errors, but it won't send any code to the Javelin Stamp.
- **Link** – Linking takes all the class files referred to by your program and binds it together for transmission to the Javelin Stamp. However, it doesn't actually send any code to the Javelin Stamp either.
- **Program** – This is the most common option. It compiles, links, and downloads your program to the Javelin Stamp.
- **Debug** – This command is similar to the Program command, but it also adds the necessary code that allows the IDE to debug your program.
- **Resume Debug** – If you are debugging a program and you get interrupted (perhaps you shut your computer down and restart it later), you can start a new debugging session without having to recompile, relink, and download. This does not resume your previous debugging session. It simply allows you to start a new one without reprogramming the Javelin Stamp.

When you use the Program option, the Javelin Stamp will run the program by itself. If you use one of the Debug options, the Javelin Stamp will require commands from the IDE to execute, so you'll want to use Program before you detach the Javelin Stamp. The Compile and Link options are handy for testing your program's syntax before downloading it to the Javelin Stamp.

### Dealing with Errors

Of course, sometimes you'll have compile-time errors (such as syntax errors) that will prevent any of the above commands from working. For example, suppose you left the **static** keyword off of the two variable declarations in the example program:

#### Program Listing 5.1 - My Test Class (Dealing With Errors)

```
// (This program contains intentional errors)
import stamp.core.*;
```

## 5: Using the Javelin Stamp IDE

```
class MyTestClass {
// Put variables here.
int pause=5000;
boolean state=false;

    public static void main() {
        while (true) {
            CPU.writePin(CPU.pin0,state);
            state=!state;
            CPU.delay(pause);
        }
    }
}
```

The two variable declarations should really look like this:

```
// Put variables here.
static int pause=5000;
static boolean state=false;
```

When you try to compile, run, or debug the program a window appears at the bottom of the IDE. This window will contain four error messages (see Figure 5.2). The error message shows the type of error, the file name, the line number, and the actual error message. In complex programs, compiling one file may cause other files to compile, so pay close attention to the file name, as it may not be the same as the current file name.

Regardless of the file name, double clicking one of the error messages will take you to the part of your program where the compiler detected the error. Notice that this is not always the same place as where you created the error.

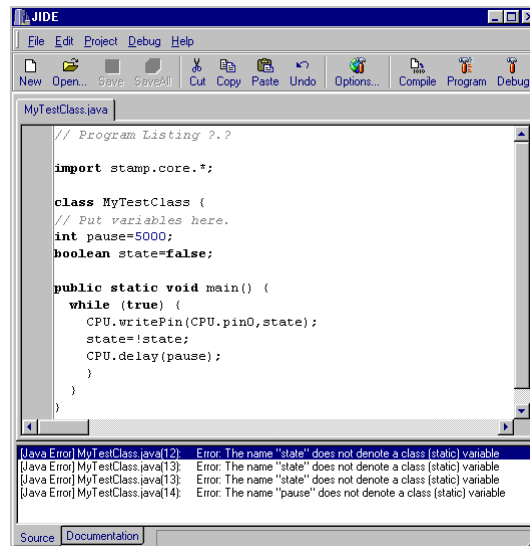
In this case, for example, the first error message is:

The name “state” does not denote a class (static) variable

This error appears on the **CPU.writePin** line. However, the real error is not on this line. The mistake here is that the **state** variable is an instance variable while the **main** method is (by necessity) **static**. A **static** method can't directly access instance variables, so an error occurs. All by itself, there is nothing wrong with creating an instance variable named **state**, so the compiler can't guess that this line is in error. That's because syntactically it isn't in error. The only reason the line is incorrect is because the program uses the variable contrary to its declaration and the compiler detects the error when the program tries to use the variable. The real mistake, of course, is in the declaration and that is where you'll have to fix the program.

## 5: Using the Javelin Stamp IDE

Figure 5.2  
Error  
Messages



The other three errors follow the same logic. Even though there are four errors on three different lines, only two things require repair and they aren't on any of those lines at all. Of course, you need to make the two variable declarations **static**. You could also elect to have **main** create a new **MyTestClass** object and call an instance method (which could then directly refer to non-static fields). However, that's a major change to the program's design, not a repair.

### Using the Debugger to Look Inside the Javelin

In a perfect world, you would write your program, download it to the Javelin, and be finished. In real life unfortunately, it isn't unusual for a program not to behave as you expected. Luckily, the Javelin's built-in debugger makes it very easy to troubleshoot misbehaving programs.

Of course, debugging won't help you locate syntax errors and other problems that prevent your program from compiling. You can find these by reading the messages the compiler and linker generate. However, just because the compiler thinks your program is correct doesn't mean the program does what you think it does. The compiler can accept a program that doesn't do what you want it to do (that is, your program contains an error in its logic). That's where the debugger comes into play.

To start the debugger (Figure 2.11), press the Debug button on the toolbar (or press CONTROL+D or select Debug from the Project menu). The debugger window that appears has several useful buttons and tabs:

- Run – This button starts your program executing under the debugger. The program will stop at a breakpoint, if any are set. You can set a breakpoint clicking in the gray area to the left of a program

## 5: Using the Javelin Stamp IDE

---

- line, using CONTROL+B, or using the Toggle Breakpoint from the Debug menu (in the main Javelin window). A line with a breakpoint appears highlighted in red and has a red dot in the left margin.
- Stop – If the program is running, the Stop button will cause execution to halt as though a breakpoint had occurred.
  - Step Into – When the program is stopped, this will cause one line of program code to execute. If the line makes a method call, the new stop location will be inside the called method.
  - Step Over – When the program is stopped, this will cause one line of program code to execute. If the line makes a method call, the Javelin will attempt to execute the entire method before stopping again. Notice that some program lines make multiple method calls, so the stop position will appear not to move until you press the Step Over button multiple times.
  - Toggle Breakpoint – Push this button to place (or remove) a breakpoint on the current line. When the program executes this line, the debugger will stop and wait for further user commands.
  - Reset – Press this button to restart the program from the beginning.
  - Show Message Window – This button displays the window the Javelin uses to display messages.
  - Call Stack – This tab shows you the method calls that are currently active. So if the **main** method calls method A, and method A calls method B, you'll see **main**, A, and B in the display window when this tab is active. The window also shows local variables and fields.
  - Static Variables – This tab allows you to examine the static variables of each class in your program. Click on a '+' sign to expand the display to see details, then click the '-' sign to hide those details again.
  - Memory Usage – Use this tab to display statistics about how much memory your program is using.

The easiest way to learn to use the debugger is to load a simple example program and start the debugger. Use the *Step Over* and *Step Into* commands while examining the different tabs in the debugger window. Set a breakpoint on a line and use the Run command.

The compiler can detect your syntax errors, but it can't find mistakes in the logic of your program – only you can do that. That's why the IDE was programmed to have sophisticated debugging support to help you examine what your program is actually doing and make it easier to spot mistakes.

Once you clean up any compile errors you can begin to debug the program. Select Debug under the Project menu (or CONTROL+D) to begin the debugging process. Once the IDE downloads the program to the Javelin Stamp, you'll see a green bar indicating the first line of your program that will execute. You'll also see a debugging window (see Figure 5.3).

## 5: Using the Javelin Stamp IDE

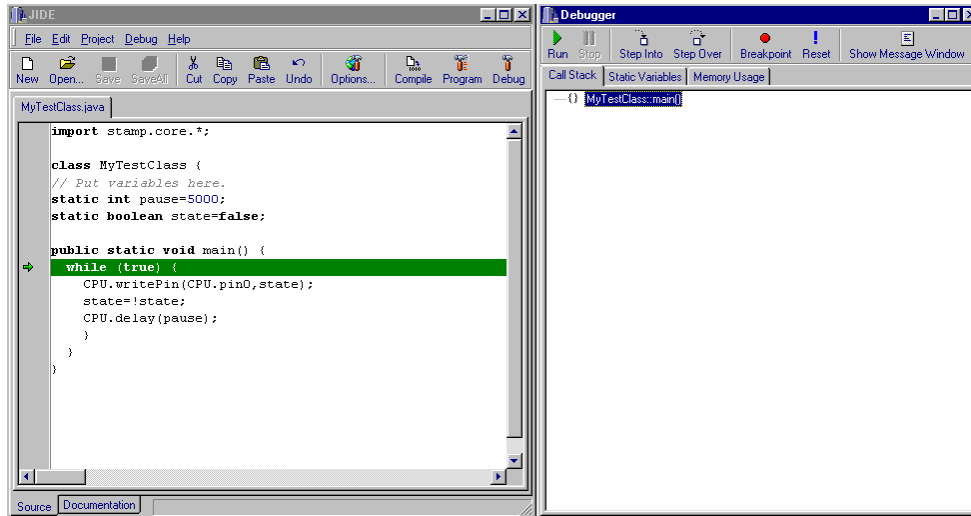


Figure 5.3 Javelin Stamp IDE and Debugger

The debugging window has a toolbar that mimics the method on the main Debug menu (covered shortly). It also has three tabs: Call Stack, Static Variables, and Memory Usage. The Call Stack tab shows the current method executing, along with the return path from methods that are in the middle of calling this method. The Static Variables tab shows you the name and value of all static variables. Finally, the Memory Usage tab allows you to examine how much memory your program is using and how much of that memory is code or data. You can also use the table at the bottom of the debug window to examine memory usage for each class in your program. If the debug window is small, you may have to increase its size vertically (by dragging the window border) to make the table visible.

If you lose the debug window accidentally, you can always get it back by selecting Show Debug Window from the Debug menu. In addition, you can make the message window visible by selecting Show Message Window from the Debug menu. The message window shows any output your program sends using **System.out** or **CPU.message**. You can use **System.out.println** to write debugging messages to the message window to help you debug your program.

There are several ways to execute your program. If you select Run from the Debug menu (or the green arrow in the toolbar, or F9) then the program will execute normally. To stop the program, you can select Stop from the Debug menu (or the double red bars in the toolbar, or F8).

If you want the program to stop at a particular spot, you can do this by setting a breakpoint. Place the cursor on the line in question and select Toggle Breakpoint from the Debug menu (CONTROL+B), or use the stop sign on the toolbar. You can also click on the gray area to the left of the line. In any event, you'll see a red stop sign

## 5: Using the Javelin Stamp IDE

---

icon appear in that left-hand area to indicate the breakpoint. Repeating the step will turn the breakpoint off and make the stop sign icon disappear.

Sometimes you don't know where you want the program to stop. In that case, you can single step through the program. The Step Into menu item (on the Debug menu) causes your program to execute one line of source, and steps into method calls. Step Over is the same, except that any method calls will run to completion. The green execution bar will show you which statement will execute next. You can use F7 for Step Into and F8 for Step Over. On the toolbar, these operations show a small box with an arrow pointing into the box (Step Into) or jumping over the box (Step Over).

While stepping through the program or if you are stopped at a breakpoint, you can always resume execution with the Run command. This will cause the program to continue until it ends or it encounters another breakpoint.

The only other command on the debug menu is Reset (CONTROL+F2). This causes the Javelin Stamp to prepare to run the program again. In other words, a Step Into, Step Over, or Run command will start the program at the beginning after a Reset.

### An Example Debugging Session

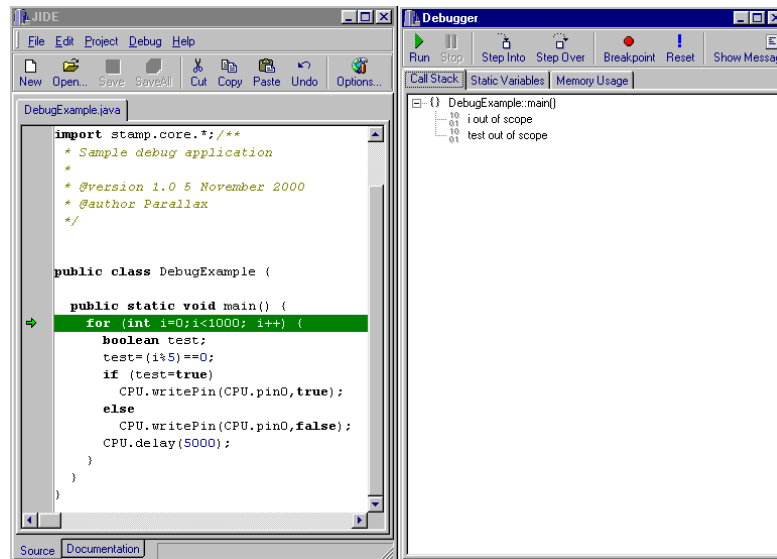
Using the DebugExample (see Program Listing 5.1) type it in exactly as you see it (if you think you see a mistake, leave it as it is). Save the file in **DebugExample.java**. The intent of this program is to blink an LED with a 2 second off time and a half-second on time. The idea is to use a half-second (that's 5000 100us periods) time base and only turn the LED on every fifth count. Of course, you can wire the LED so that the LED will be off every fifth count and on the remainder of the time – the important point is that the LED will be in one state for a single 500 ms period and in the opposite state for 2 seconds.

You can run the program by selecting Program from the Project menu. However, it doesn't work. Why not? You might be able to find the answer by inspecting the program, but often debugging is easier.

To prepare for debugging, select Debug from the Project menu (or press CONTROL+D). Your screen should look like the one in Figure 5.4. The green bar and arrow in the source code tells you that you the next line that will execute. The Call Stack tab in the debug window shows you that you are in the **main** method.

## 5: Using the Javelin Stamp IDE

Figure 5.4  
Stepping  
through  
Code



Use F7 to step through the program a line at a time. Notice that the Call Stack tab also shows the local variables (like **i** and **test**). Press F7 until you make one pass through the loop and notice the state of the local variables at each step.

On the first loop (where **i** is 0) everything seems to work, as you'd expect. On the second pass however, pay particular attention to the **if** statement. Press F7 until the green bar rests on the **if** statement (and **i** is equal to 1). Before executing the **if** statement, the **test** variable is **false**. That's right because 1 is not evenly divisible by 5 so **i%5** is not equal to 0. Now press F7 to step through the **if** statement. Suddenly, **test** is now **true** and the incorrect branch of the **if** executes. Do you see why?

Careful observation of the **if** statement shows that there is only one equal sign! Instead of testing to see if **test** is true, this statement sets **test** to **true** and therefore assures that the **else** clause will never execute. The answer – or at least, one answer – is to change the single equal sign to two equal signs. On the other hand, you could rewrite **main** like this:

```
public static void main() {
    for (int i=0;i<1000; i++) {
        CPU.writePin(CPU.pin0,(i%5)==0);
        CPU.delay(5000);
    }
}
```

## 5: Using the Javelin Stamp IDE

### Editing Text

The IDE text editor window works the same as any other Windows editor. You can use the File and Edit menus as shown in Table 5.2 and Table 5.3.

**Table 5.2: File Menu Commands**

Menu Item	Command	Shortcut
New	Start a new document	CONTROL+N
Insert Template	Insert a sample class definition	CONTROL+J
Open...	Open an existing file	CONTROL+O
Reopen	Opens a recently used file	ALT, F, R
Save	Save the current document	CONTROL+S
Save As...	Save the current document with a new name	ALT, F, A
Close	Close the current document	CONTROL+F4
Print	Print the current document	CONTROL+P
Exit	Ends IDE	ALT, F, E

You can also use common Windows shortcuts to perform common editing operations shown in Table 5.3.

**Table 5.3: Edit Menu Commands**

Menu Item	Command	Shortcut
Undo	Undo the last editing action	CONTROL+Z
Cut	Remove the selection to the clipboard	CONTROL+X
Copy	Copy the selection to the clipboard	CONTROL+C
Paste	Paste the clipboard contents to the document	CONTROL+V
Select All	Select all text	CONTROL+A
Find and Replace...	Find or find and replace text	CONTROL+F
Find Again	Repeat last find operation	F3

### Toolbars and Menubars

You can move the main menu to different locations by grabbing the double vertical bar to the left-hand side of the menu and dragging. You can move the menu anywhere horizontally, and you can move the menu to two different vertical locations.

You can also drag the toolbars around in this fashion. In addition, you can drag the toolbars into the main window area to convert them into floating windows. If you want to restore them to their bar state, you can drag them to the top window border and they will stick. By grabbing the double bar to the left-hand side of the toolbar, you can move the toolbar to different locations.

## 5: Using the Javelin Stamp IDE

---

Another way to issue commands is to right click on the file's tab at the top of the editor screen. Right clicking will display a menu that will allow you to compile, debug, program, manipulate projects (covered shortly), or close the current file. Note that the menu commands always apply to the current document, even if you right click another document's tab.

### Class Path Considerations

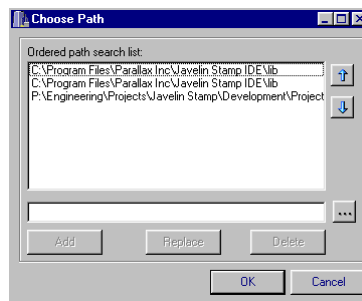
One of the most critical aspects of working with any Java or Java-like development tools is the CLASSPATH. Each time you name a class in your program, the compiler searches for the appropriate class file by searching the directories named in the CLASSPATH (you'll find more about this topic in Chapter 3).

It is crucial that the directories in the CLASSPATH refer to the correct class files, and not class files aimed at another target system (like the PC, for example). In addition, if you create your own libraries of code, you'll want to place the correct directories for that code in the CLASSPATH.

Selecting Global Options under the Project menu will give you a Global Options window. You can select the Compiler tab to view the CLASSPATH variable. You can directly change the string you find there if you like. It is simply a list of paths separated by semicolons. The paths should be absolute (e.g., `c:\myclasses\lib1` instead of `..\lib1`).

However, it is easier to change the CLASSPATH by pressing the ... button next to the path (see Figure 5.5). Here, you can change each part of the path separately. You can use the ... button to browse your files and the up and down arrow buttons to alter the order of each directory in the CLASSPATH. The order is important, because the compiler begins searching with the first directory, and proceeds in order. Once it finds a suitable class file, it stops searching, so if two directories in the CLASSPATH contain class files named the same, the first one mentioned in the CLASSPATH will override any subsequent directories.

**Figure 5.5**  
Class Path  
Settings



### Working with Packages

If you make a class or a group of classes that you want to reuse, you might consider putting them in a package. First, at the start of the java files that contain your classes, you'll put a **package** statement. The convention is

## 5: Using the Javelin Stamp IDE

---

to use your inverted Internet domain name (for example, com.parallaxinc) to begin the package name. After that, you can use as many words as you like separated by periods.

For example, consider this class:

```
package com.parallaxinc.testlib;

public class doubler {
    private int val;
    public doubler(int v) { val=2*v; }
    public int value() { return val; }
}
```

This class (*doubler*) is part of the *com.parallaxinc.testlib* package. You need to save the file (or at least the class file) in a file that is in several subdirectories. In particular, the file name must be *com\parallaxinc\testlib\doubler.java*. This is a relative path name. The compiler will look in the current directory and in all the CLASSPATH directories for this directory structure. So imagine that your CLASSPATH had a single directory in it (*c:\classes*) and that the current directory is *c:\projects*. The compiler will look for *com.parallaxinc.testlib.doubler* in the *doubler.java* file. It will search for that file in:

```
c:\projects\com\parallaxinc\testlib and in c:\classes\com\parallaxinc\testlib
```

To use the **doubler** class, you'd need to refer to its entire name, or use an **import** statement. For example, you might write:

```
com.parallaxinc.testlib.doubler dbl = new com.parallaxinc.testlib.doubler(20);
```

Notice that you have to use the entire name every time you refer to the object. This is not very convenient so you'll usually use an import:

```
import com.parallaxinc.testlib.doubler;

doubler dbl = new doubler(20);
```

Remember, the packaged class must be in the correct directory tree and that directory tree's root directory must be in the CLASSPATH.

### Working with Projects

You can organize your work into projects. From the main file of the project, you can select Make Project from the Project menu. You can also right click the file's tab and select Make Project from the resulting menu (if this option is gray, you have not saved the file yet).

## ***5: Using the Javelin Stamp IDE***

---

Once you've made a project, the tab for that file will have a file folder icon to the left of the file name. One project can be active at a time. The active project will have a green checkmark in the file folder.

Projects are useful when you want your Java file to have its own options. The active project has its own private options that you can access by selecting Project Options from the Project menu. From here you can set the class path for compilation, the debugger settings, and packages you want to include in the javadoc documentation. You can also specify the directory where the IDE will create documentation.