

Eclipse for Visual Studio developers

Skill Level: Intermediate

[Scott Kellicker \(scott.kellicker@acm.org\)](mailto:scott.kellicker@acm.org)

Software Developer

ACM

31 May 2005

This tutorial demonstrates how to use the Eclipse IDE to develop Java applications. It is geared toward the Visual Studio developer, and will present Eclipse concepts and terminology in the context of Visual Studio. The tutorial walks through a simple example, from installation through debugging.

Section 1. Introduction

Who should take this tutorial?

Take this tutorial if you have an interest in using the Eclipse IDE to develop Java applications, but have not yet used Eclipse. Although it is not required to be a Visual Studio developer to benefit from this tutorial, we will introduce Eclipse in the context of some Visual Studio concepts and terminology. Basic knowledge of Java™ technology is also helpful.

What is this tutorial about?

Eclipse is a free open source development environment. There are many projects that fall under the Eclipse umbrella, so it is also known as the Java Development Tools (JDT). This tutorial covers the basics of developing Java applications using the JDT, but as you work through it, it is useful to remember that Eclipse is also:

- An open platform for tool integration using an architecture that allows

plug-ins to be discovered, integrated and executed at run time.

- A platform designed for easy extension by third parties.
- An IDE platform that has generalized interactions like managing resources, launching programs, and debugging capabilities.
- A growing set of projects that build on the core platform, such as the C/C++ Development Toolkit (CDT) and Web Tools Platform (WTP) Project.
- Core technologies, such as the Standard Widget Toolkit (SWT), that can be used separately from the rest of the platform.

To demonstrate the Eclipse IDE, this tutorial will guide you through the development of two stack classes written in the Java language. This will demonstrate how to:

- Install the Java SDK and the Eclipse IDE.
- Set up a Java project using Eclipse.
- Use the Java Editor to write Java code.
- Run and debug a program in Eclipse.
- Use some of Eclipse's productivity tools to streamline development.

Prerequisites

The tutorial is geared toward the Visual Studio and Visual Studio .NET developer, so it assumes you will be running on Windows® (This tutorial will use the term Visual Studio to refer to both Visual Studio and Visual Studio .NET). However, because Java technology and Eclipse are available for many platforms, the tutorial can easily be followed for non-Windows platforms. The tutorial also assumes a basic knowledge of object-oriented concepts, such as inheritance, interfaces, and classes.

Before beginning, make sure you have downloaded and installed the following:

- The [Java 2 Standard Edition Software Developers Kit \(J2SE SDK\)](#), V1.4.2 or higher -- Run the setup and install the files to a location such as c:\j2sdk1.4.2.
- The [Eclipse SDK V3.0.2](#) or later -- The Eclipse installation is distributed as a zip file. Unzip the contents to c:\Programs.

Section 2. Starting Eclipse

The Eclipse Workspace

To start Eclipse, double-click on `c:\Programs\eclipse\eclipse.exe`. The Eclipse logo will appear. Shortly thereafter, you are prompted to select the Workspace. The default workspace location is the `workspace` folder in the Eclipse distribution (`c:\Programs\eclipse\workspace`). Select **OK** to use this.

The Eclipse workspace is similar to Visual Studio's concept of a solution. One workspace may be open by Eclipse at a given time. Each workspace may contain one or more projects. One big difference is that a Visual Studio solution may contain items, such as files, that are common to all projects inside a solution. The Eclipse workspace has no mechanism for common resources among projects.

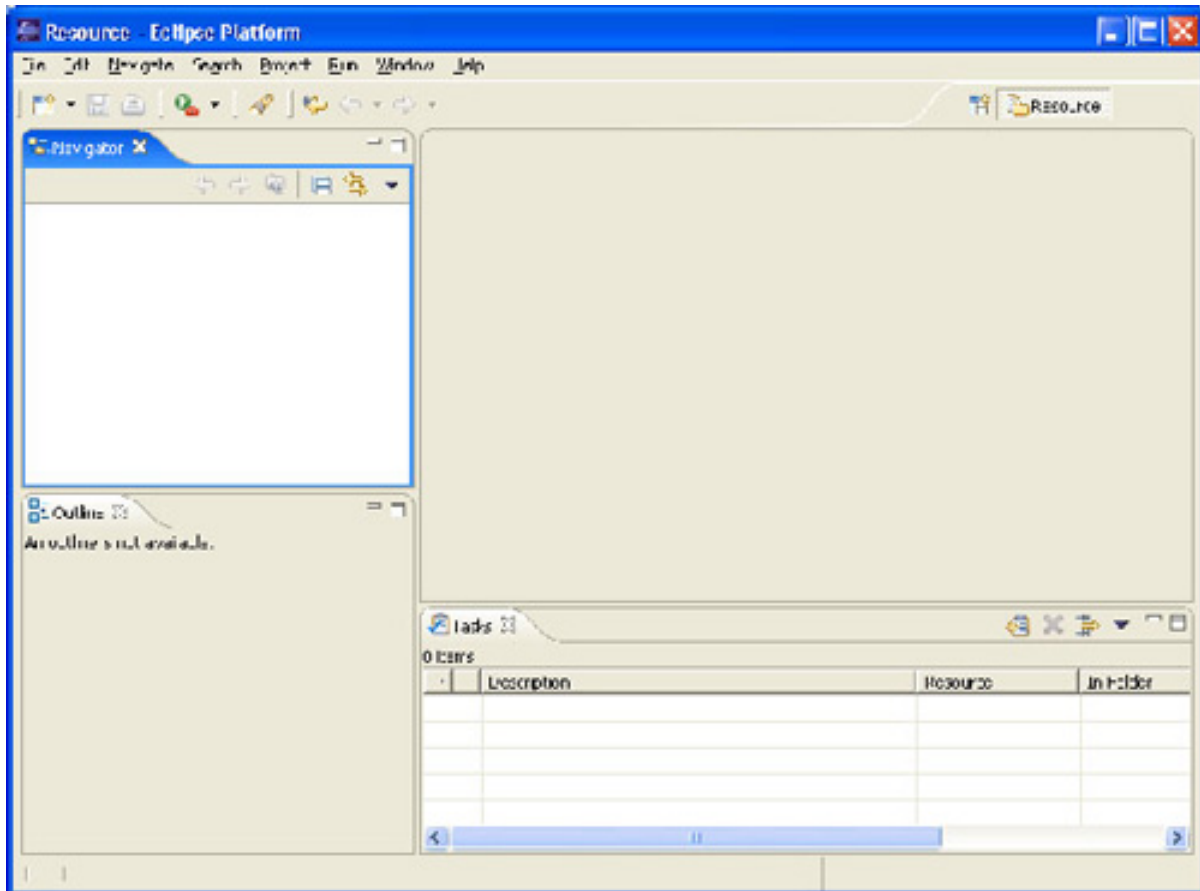
Each project contains resources, which are files and folders. The Eclipse project file and folder hierarchy corresponds 1:1 with the physical file system. Unlike Visual Studio, it is often difficult to move projects to a different location on the file system. This file hierarchy is created by default inside the workspace folder you specified above. However, projects may also live elsewhere on the file system.

Note: One problem that users new to Eclipse have is trying to use an existing Java project in Eclipse. This should be done with an import wizard. Note that you should not copy the files into the workspace folder and expect Eclipse to find them. In fact, you cannot copy them into the workspace folder and use the import wizard. Instead, select **File > Import > Existing Project**. Then Browse to the top-level folder of the project. This leaves the files where they are on the file system and allows Eclipse to operate on them as a project.

The Eclipse landscape

Eclipse will now start. The welcome screen appears and contains four buttons: Overview, Tutorials, Samples, and What's New. For now, skip this page by clicking the X in the Welcome tab. (To explore this content later, select **Help > Welcome**.) Eclipse now looks like Figure 1.

Figure 1. The Resource Perspective



The Eclipse interface is composed of three main elements:

- **Editors** -- Like Visual Studio, an *Editor* is a tool used to modify files in the project. In Figure 1, the Editor area is blank. The Java Editor will be explored in detail later in the A tour of the Editor section.
- **Views** -- *Views* are the various windows that surround the Editor area. The Navigator view appears to the left of the Editor area in the figure. This view is used to manage projects and resources. The Task view is below the Editor area and is used to manage outstanding tasks.
- **Perspective** -- A *Perspective* is a collection of views and editors designed around a specific task. The figure shows a Resource Perspective, as denoted in the upper-right tab and next to the Eclipse icon in the window banner.

This tutorial will explore the Java and Debug Perspectives.

Section 3. Creating a Java project

The stack project

As mentioned, we will step through the development of two stack classes. The stacks will allow objects to be pushed and popped, and allow the current stack depth to be queried. A simple class will be created to exercise the stacks. The example is simple enough to keep the focus on the Eclipse IDE and Visual Studio. However, the example is substantial enough to demonstrate many of the features in Eclipse. If you are new to Java technology, some of the syntax will be unfamiliar, but you should be able to easily follow along.

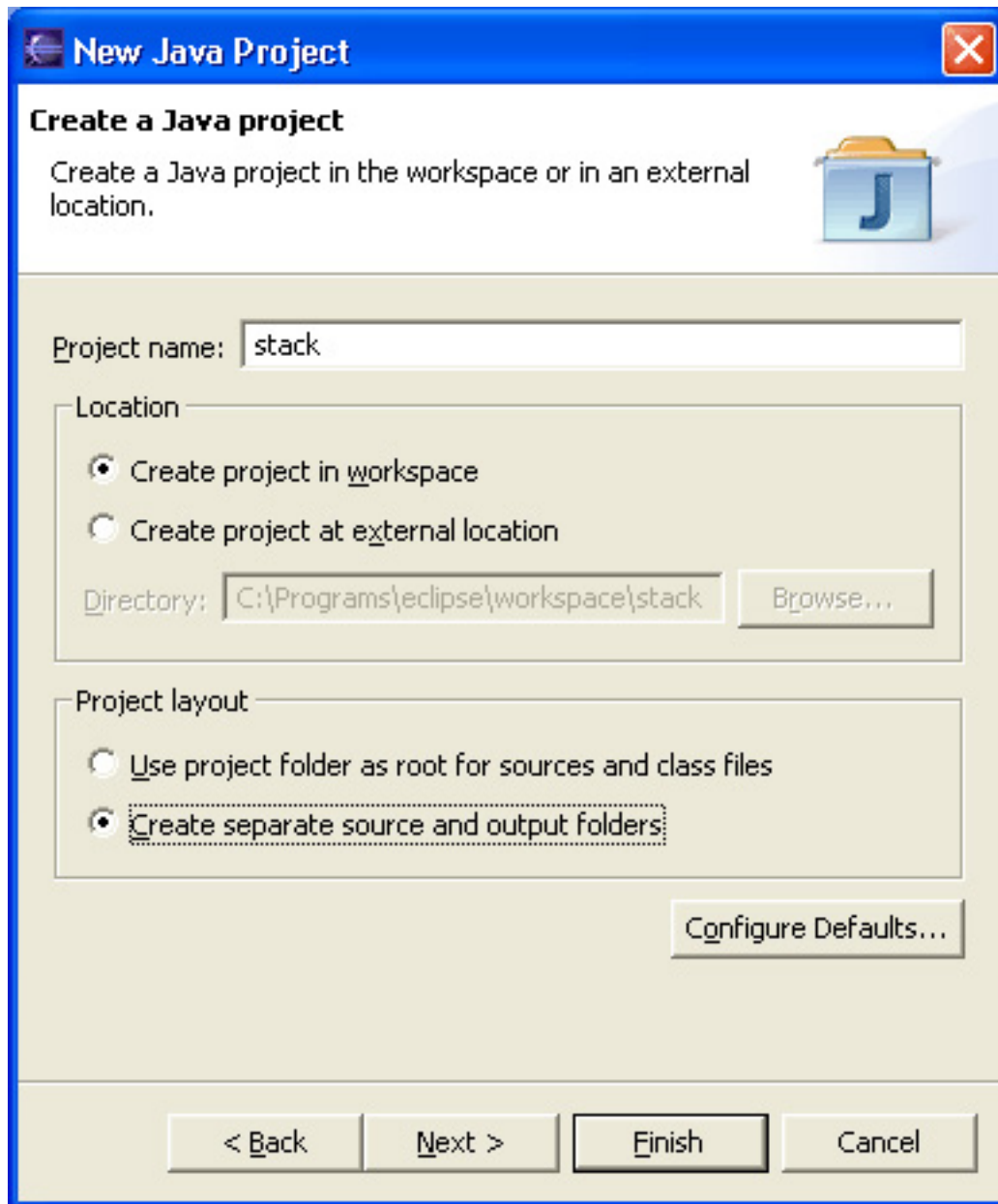
In this section, you will create a new project, tour the Java Perspective, and create stubs for our first two Java classes.

New project wizard

Like Visual Studio, Eclipse can create a new project using a wizard. You will now create the stack project:

1. Select **File > New > Project** or right-click inside the Package Explorer and select **New > Project**
2. Select **Java Project** and **Next** (you will see the dialog in Figure 2)
3. Enter a project name: `stack`
4. Change the project layout to "Create separate source and output folders" (this can ease development by keeping the Java source files separate from the compiled output files)
5. Click **Finish**
6. At this point, you are asked if you would like to switch to a Java Perspective; answer **Yes**

Figure 2. New Java Project wizard



The Java Perspective

The Java Perspective, shown below in Figure 3, has a similar layout to the Resource Perspective. However, the IDE now has a set of views specific to writing Java code. The empty Editor area is in the middle, surrounded by tabbed views.

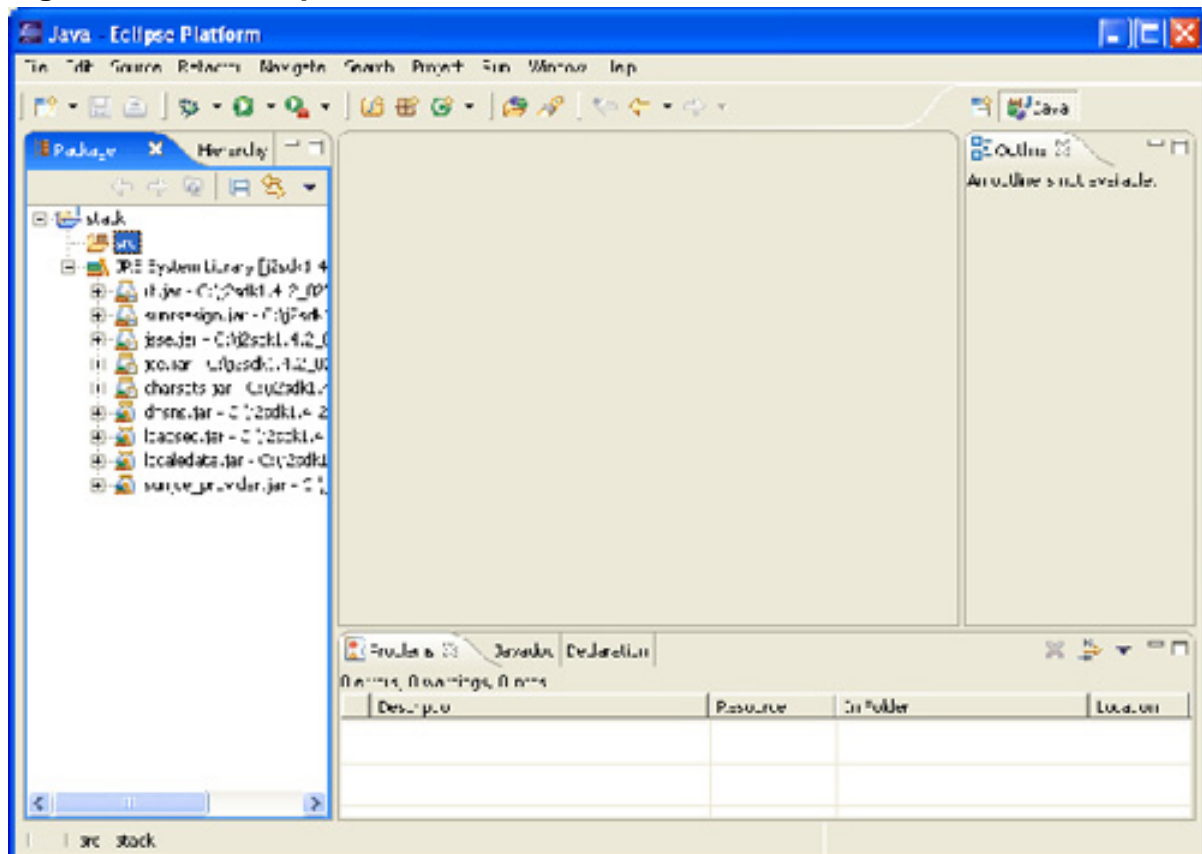
On the left side of the interface, the Resource Perspective's generic Navigator view has been replaced by the similar but Java language-aware Package Explorer. The Package Viewer is a hierarchical tree view of your project resources. Visual Studio developers can equate the Package Explorer to the Solution Explorer.

Click the + signs to expand the stack project. You will see the JRE System Library entry. This contains all the libraries available as part of the Java Runtime Environment (JRE). There is an empty src folder. Soon, you will populate this folder with the stack source code.

Some other views worth mentioning include:

- A Problems view, at the bottom, to alert you of Java compilation warnings and errors
- An Outline view, on the right, to display members and methods of the currently selected class
- A Hierarchy view, behind the Package Explorer, to view class hierarchies

Figure 3. Java Perspective



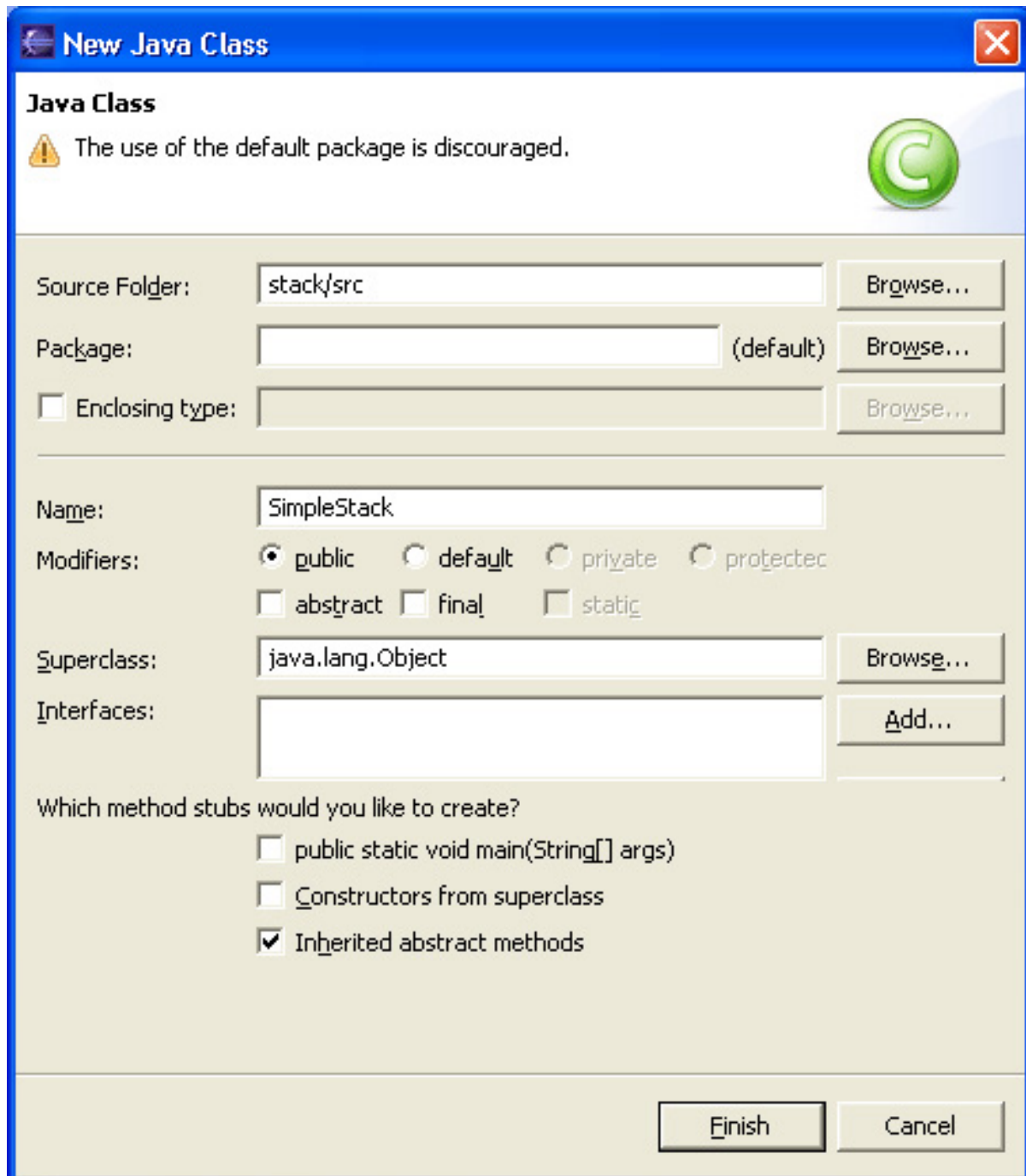
Create a class

Again, like Visual Studio, Eclipse provides tools from the Package Explorer to create generic resources, such as files and folders, as well as more specific resources, such as classes and interfaces. You can create the stack class using the New Java

Class wizard.

1. Right-click on the `src` folder in the Package Explorer
2. Select **New > Class**; the New Java Class wizard will appear (as shown in Figure 4)
3. Enter `SimpleStack` in the Name field
4. A warning states that using the default package is discouraged. Java packages are a mechanism for organizing related groups of code and defining a namespace. Typically, you would provide a package name, such as `com.mycompany.stackexample`. It is considered bad practice to use the default package, but to keep the example simple, ignore the warning and use the default namespace.
5. Like all Java classes, `SimpleStack` inherits from `java.lang.Object`. It is worth noting that this wizard allows the new class to inherit from any Java class and implement zero or more interfaces. If you wish, the wizard will even generate the method stubs of the superclass and interfaces. `SimpleStack` does not require this, so select **Finish**. The Package Explorer and Editor change to reflect the addition of `SimpleStack`.

Figure 4. The New Java Class wizard



A class to exercise `SimpleStack` is required. Create a class called `TestStack` using the same wizard.

1. Select **New > Class** from the `src` folder
2. Enter `TestStack` for the Name

3. This time, check **public static void main(String[] args)** to create a main stub. This is the entry point into `TestStack`.

Now that you've set up the environment and created the classes, you'll jump into writing the actual code.

Section 4. Editing code

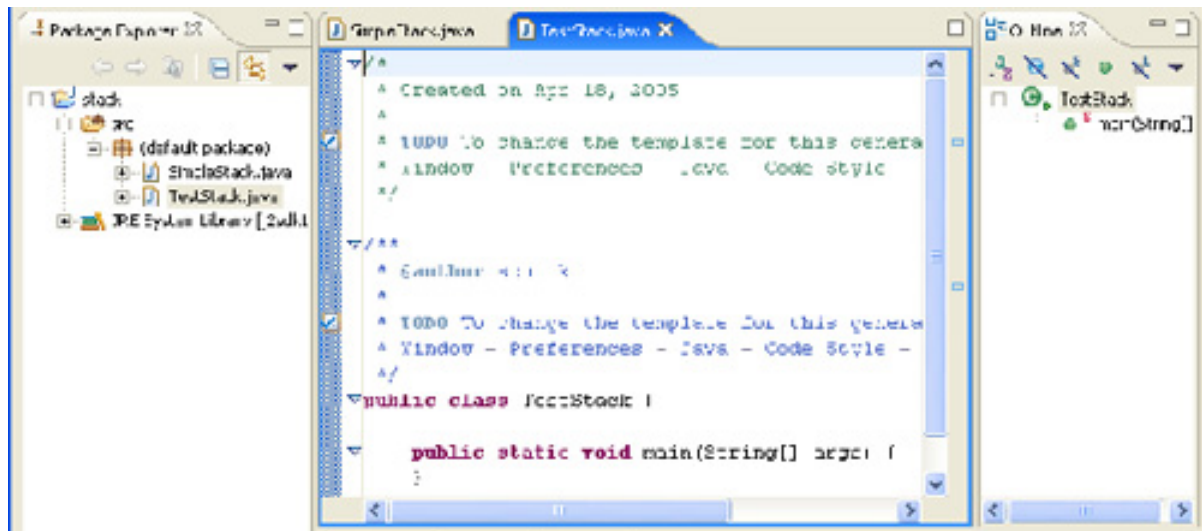
A tour of the Editor

Before writing code for the stack project, which you'll do in this section, a quick tour of the Editor is useful. Visual Studio users will note many similarities, as well as some features unique to Eclipse.

The various views are linked to the Editor. For example, double-clicking on `TestStack.java` in the Package Explorer will open the Editor window for this file. Note that as the edited file changes, the Outline view changes to reflect the current class and its members. This is shown in Figure 5. The top of the Editor window is a tabbed interface to allow quick switching between files. Double-clicking an Editor tab will maximize the Editor to take the entire Eclipse window area. Double-clicking it again will return it to its original size and location. In fact, all views in Eclipse have this same functionality.

Much of the Eclipse user interface provides context-sensitive menus via right-click. For example, right-clicking an Editor tab will present a menu for managing the various open editors. Right-clicking inside the Java Editor presents menu choices geared toward writing Java code, such as code generation and refactoring tools.

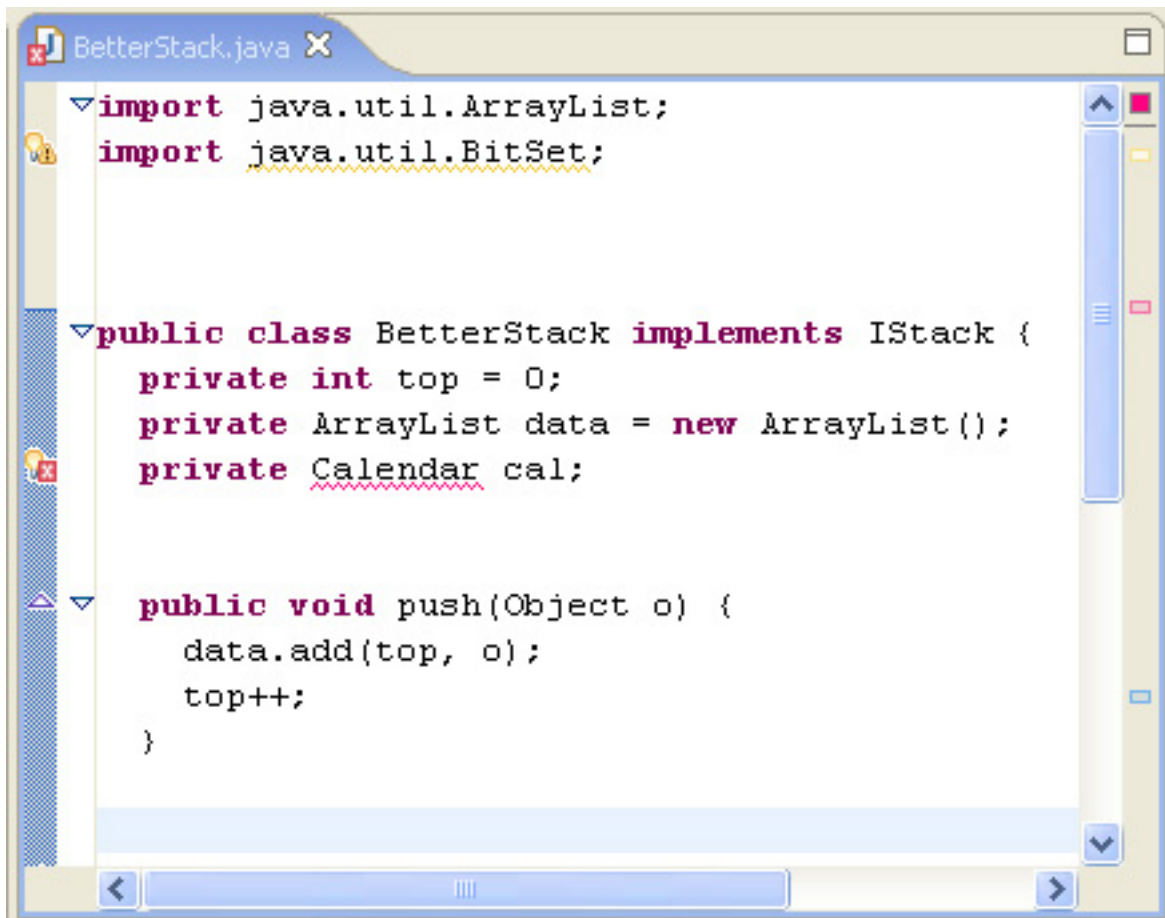
Figure 5. Dynamically linked Editor



On the far left side of the Editor is a vertical channel called the marker bar. The marker bar displays marker icons denoting compiler errors, compiler warnings, tasks, quick assists, breakpoints and overridden methods. Hovering over an icon will give more information about that marker. Figure 6 shows an Editor view of `BetterStack.java`, (which you will develop later in the tutorial), adorned with three markers: a Java compiler warning with a Quick Fix, a Java compiler error with a Quick Fix, and an overridden method. Quick Fixes will be covered in the Quick fix section.

To the right of the marker bar is a ruler to manage code folding. Visual Studio calls this Code Outlining. Hovering over the triangle shows the scope of the code folding. Clicking the triangle folds the code, and clicking again unfolds it. The Java Editor automatically defines the folding regions, but you may also define your own. In the Editor shown in Figure 6, the import statements, the entire `BetterStack` class, and the push method may all be folded.

Figure 6. Marker bar and overview ruler



On the right side of the Editor is the overview ruler, which includes color-coded marks that indicate where warnings, errors, tasks, and search matches are within the current file. The overview ruler is useful for navigating files that do not fit entirely in your Editor. Hover over the overview mark for more information. Click on the mark to go to that line in the file. In Figure 6, there are three markers: a yellow warning, a red error, and a blue task mark. Note that the blue task mark is associated with a task defined in a part of `BetterStack`, which is currently off the screen. Clicking it would take you to that line in `BetterStack`.

Building the project

Building a project in Eclipse is quite different from building a project in Visual Studio. Visual Studio has a Build menu that contains all the build commands: Clean, Compile, Build Solution, Rebuild, etc. You explicitly compile a file or build a solution. One surprising aspect of Eclipse is that there is no Build menu. Eclipse compiles the Java file automatically each time it is saved. Developers coming from another IDE like Visual Studio often find this confusing. However, most people find it second nature once they understand it.

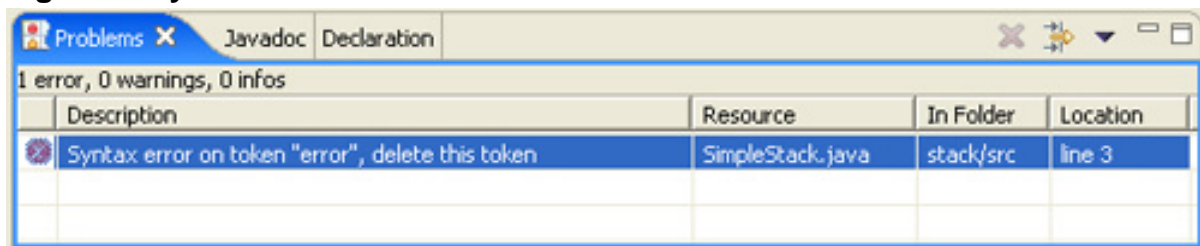
You can see this by adding some code to your skeleton project that fails to compile:

1. Open `SimpleStack.java` in the Editor
2. Select and delete the automatically generated comments. Notice that the Editor tab now has an asterisk before the file name; this denotes that the file is dirty.
3. Change the `SimpleStack` Editor class as shown to generate a syntax error.

```
public class SimpleStack {  
    error  
}
```

4. Even before saving and full compilation, Eclipse performs an incremental compilation. The error is underlined, as well as denoted in the marker bar and overview ruler.
5. Select **Save** (or **File > Save**) to save and compile the file. The syntax error appears in the Problems view, as shown in Figure 7.
6. Double-click the error, and the offending line will be selected in the Editor for an easy change. Remove the line and save again.

Figure 7. Syntax error



Quick fix

Now you will write some `SimpleStack` code to demonstrate the Quick Fix feature. `SimpleStack` requires two variables: an integer representing the current top of the stack and a data structure to hold the objects. Java technology provides a data structure called an `ArrayList` that manages a dynamically resized collection of objects.

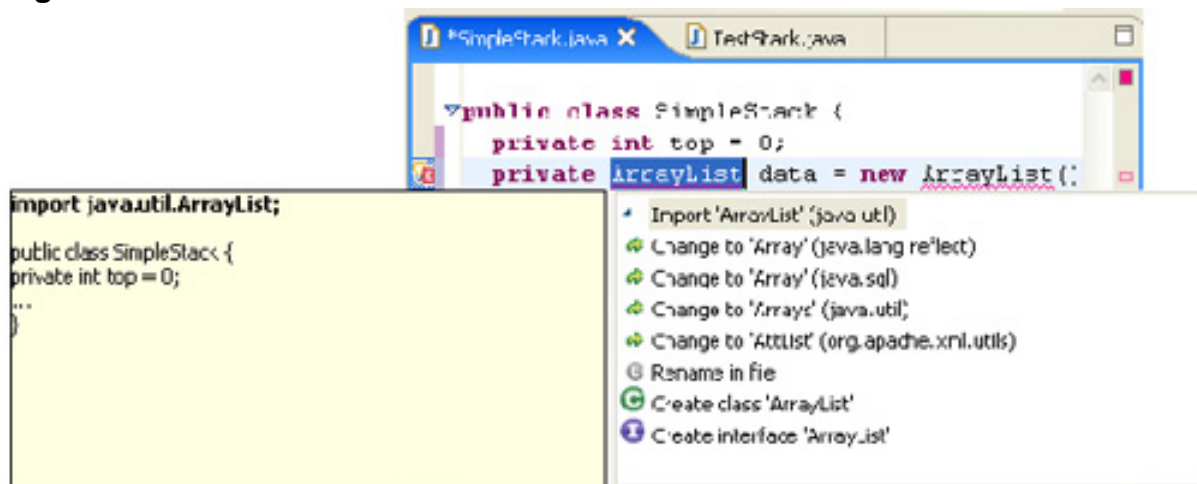
```
public class SimpleStack {  
    private int top = 0;  
    private ArrayList data = new ArrayList();  
}
```

```
}
```

Add the member variables to `SimpleStack`. As you type, notice how the Editor performs error detection and syntax highlighting dynamically. When the second member variable is added, a Quick Fix Error marker appears in the marker bar. Hover over the icon to see more information regarding the error. Click on the light bulb to see what Eclipse suggests to fix the problem. A right panel shows the list of suggestions. As the cursor moves over each suggestion, a left panel previews how the code would change. This is shown in Figure 8.

The problem is that the import statement for `ArrayList` is missing. Select **Import 'ArrayList' (java.util)** to fix it automatically. The Quick Fix icon disappears, and the import statement is added to the code.

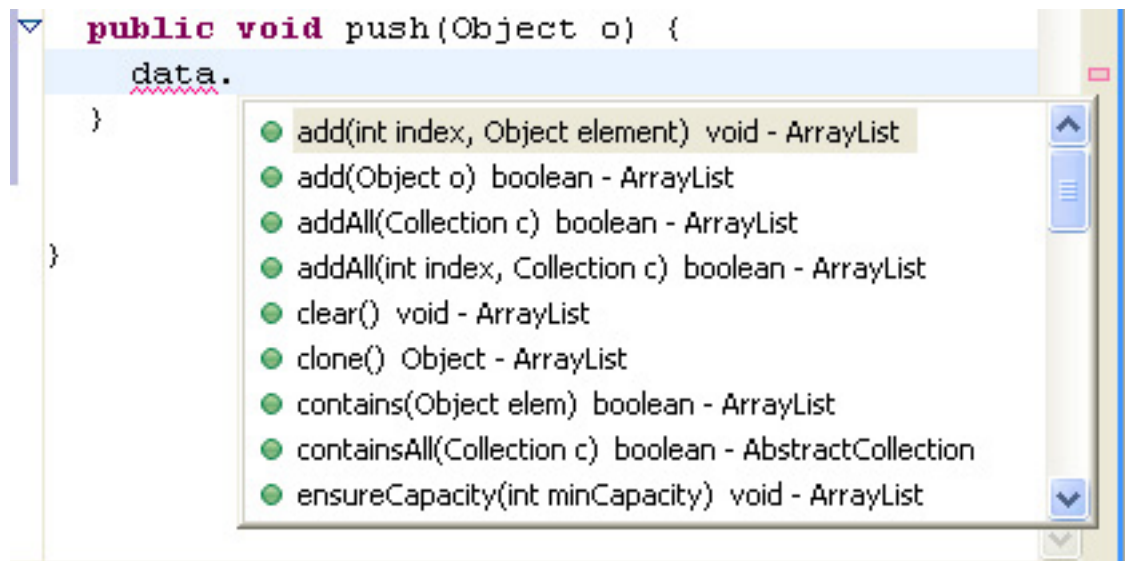
Figure 8. Quick fix



Getting help

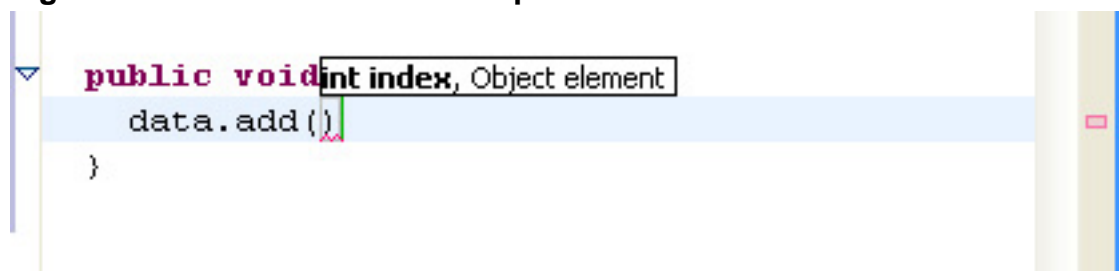
Suppose you wish to know more about the mysterious `ArrayList` class. Eclipse provides several convenient ways to find out more about a class, even those classes that might be in an external library.

- Hover over the `ArrayList` declaration in `SimpleStack`. A window pops up under your cursor with the (Javadoc) documentation of the class.
- Right-click `ArrayList` to see the various context-sensitive actions. Among these is "Open Declaration," which opens the `ArrayList` source code in an editor.
- Another right-click action is "Open Type Hierarchy." This opens the Hierarchy view as shown in Figure 9. In the Hierarchy view, you will see `ArrayList`'s super and sub-classes, as well as the individual class



1. Use the up and down arrows to preview possible selections.
2. You know that you need an add method, so type `a` to narrow the list of candidates. Then use the mouse or arrows keys to choose the first add method, then **Enter**.
3. Content Assist now prompts for each of the add method's parameters, as shown in Figure 11.
4. Type `top` and a comma. Content Assist now prompts for the second parameter.

Figure 11. Content Assist shows possible methods



Finish SimpleStack methods

Now finish the `push` method as shown. Add the `pop` and `getSize` methods using Content Assist (**Ctrl+Space**) to simplify selecting the appropriate member.

```
public void push(Object o) {
    data.add(top, o);
    top++;
}
```

```
}  
public Object pop() {  
    return data.remove(--top);  
}  
  
public int getSize() {  
    return data.size();  
}
```

More Content Assist

Now you'll add code to `TestStack` to exercise `SimpleStack` and continue to explore Content Assist. The code you are going to write will construct a `SimpleStack`, push three `String` objects onto it, check the stack size, pop two objects, and check the stack size again. You will print the results to `stdout` using `println`. The code will look like this:

```
public static void main(String[] args) {  
    SimpleStack stack = new SimpleStack();  
    stack.push("first");  
    stack.push("second");  
    stack.push("third");  
  
    System.out.println("Stack size: "+stack.getSize());  
    System.out.println("Pop "+stack.pop());  
    System.out.println("Pop "+stack.pop());  
    System.out.println("Stack size: "+stack.getSize());  
}
```

1. Select the `TestStack` Editor tab.
2. Inside the empty `TestStack` main method, start typing the line that constructs a `SimpleStack`. Type `sim` and activate Content Assist using **Ctrl+Space**. Content Assist automatically knows about the `SimpleStack` class. Select `SimpleStack` and continue with the first line. Remember to use Content Assist to enter the `SimpleStack` constructor at the end of the line.
3. Experiment with Content Assist as you type in the rest of the main method.
4. Save both classes using **File > Save All**. They should compile without error.

Section 5. Running and debugging

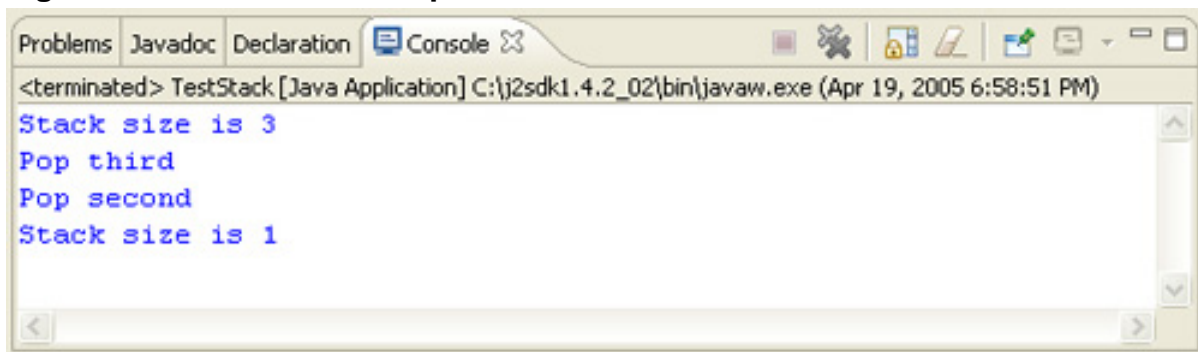
Running TestStack

Running programs in Eclipse is different from in Visual Studio. Eclipse has the concept of a Launch Configuration, which defines the information needed to run a Java program. This includes arguments to the Java Virtual Machine (JVM), input parameters to the program, environment variables, and the classpath.

A program can be run in different modes, as well. For example, you can launch `TestStack` in Run or Debug mode. To create a launch configuration, select **Run > Run** or **Run > Debug**. However, because launch configurations is such a broad topic, this tutorial shows the simplest method for running `TestStack`.

1. To run `TestStack`, select `TestStack.java` in the Package Explorer.
2. Right-click and select **Run > Java Application**.
3. The Console view will appear with the output shown in Figure 12. The Console view captures Java's `stdout` and `stderr`.

Figure 12. Console view output from TestStack



Debugging TestStack

As you have probably noticed, `SimpleStack` is indeed simple. It is going to have problems if a client pops more items than it has pushed. Let's modify `TestStack` to show `SimpleStack`'s flaws and demonstrate some debugging techniques.

1. Modify `TestStack` as shown below to do two additional pops.

```

public static void main(String[] args) {
    SimpleStack stack = new SimpleStack();
    stack.push("first");
    stack.push("second");
    stack.push("third");

    System.out.println("Stack size: "+stack.getSize());
    System.out.println("Pop "+stack.pop());
    System.out.println("Pop "+stack.pop());
    System.out.println("Stack size: "+stack.getSize());
    System.out.println("Pop "+stack.pop());
    System.out.println("Pop "+stack.pop());
}

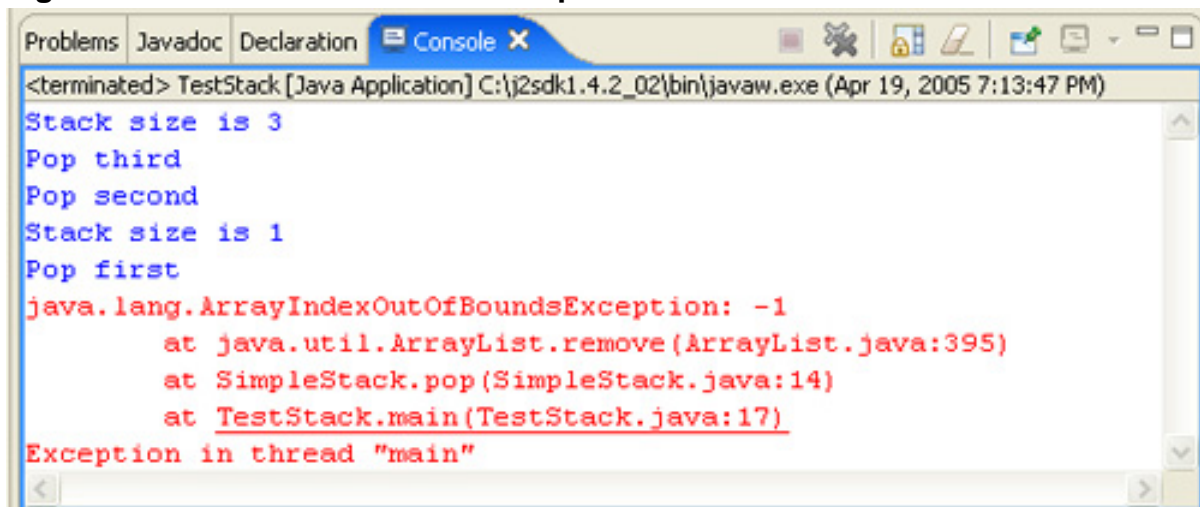
```

2. Now save TestStack and re-run it.

As shown in Figure 13, the Console displays the stack trace where `ArrayList` threw an exception. Note that the information we printed with `System.out.println` appears in blue because it was printed to `stdout`. The error appears in red because it was printed by Java technology to `System.err` (`stderr`).

Like Visual Studio, the lines in the stack trace are hyperlinks. Clicking on one will open the Editor to that line. Clicking on the stack trace and reading the code is probably enough to understand and resolve the problem. However, you can use the Eclipse debugger to solve this problem.

Figure 13. TestStack throws an exception

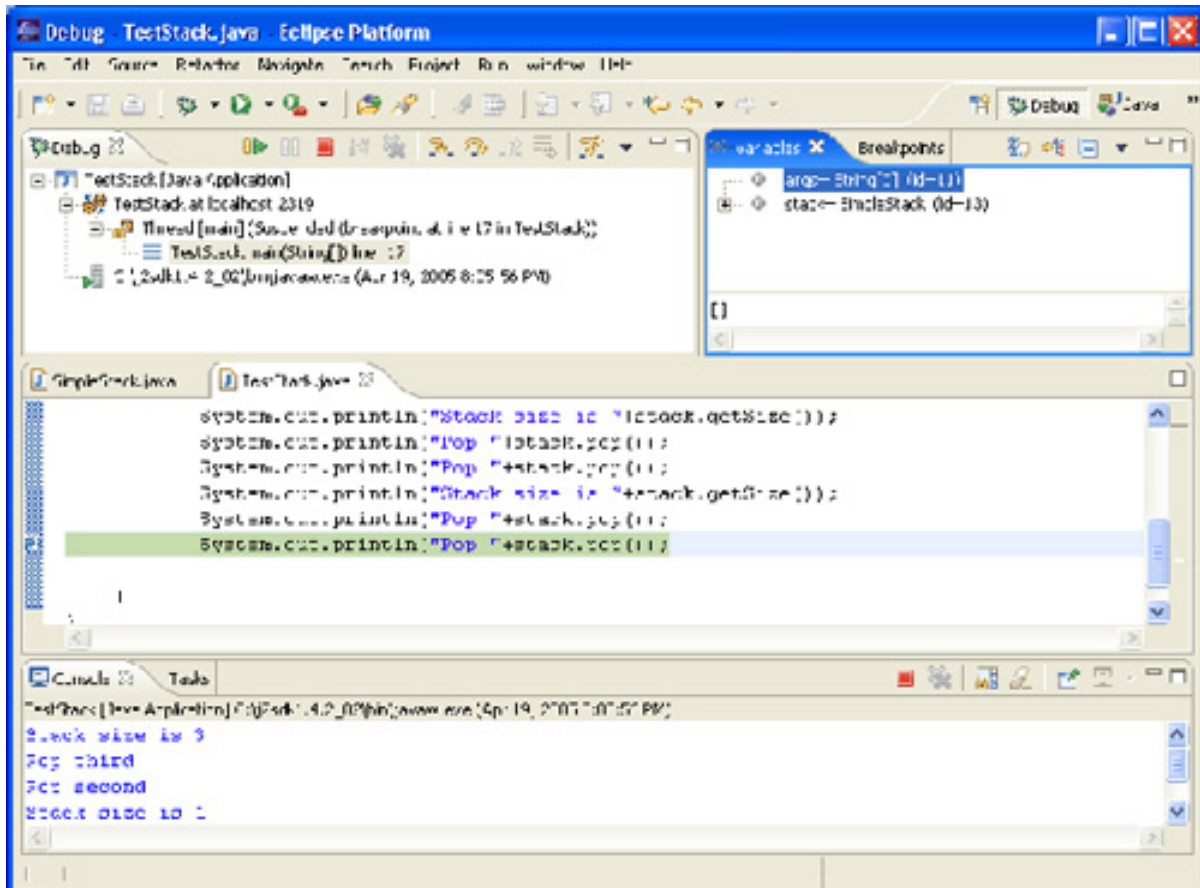


If you have used the Visual Studio debugger, you should find the Eclipse debugger easy to navigate. You can try it to debug `SimpleStack`.

1. Click on the stack trace in the Console view to go to the offending line in `TestStack.java`. As expected, the last pop statement is causing the exception to be thrown.

2. Create a breakpoint at this line by right-clicking in the marker bar and selecting **Toggle Breakpoint**. A small blue circle will appear in the marker bar denoting a breakpoint.
3. Right-click `TestStack.java` in the Package Explorer and select **Debug > Java Application**. (Alternatively, you can click on the bug icon in the toolbar. This will launch the last launch configuration in debug mode.)
4. Eclipse prompts to ensure that it is OK to switch to the Debug view; answer **Yes**.
5. `TestStack` will stop executing at the breakpoint, and the Debug Perspective will be shown, as in Figure 14.

Figure 14. The Debug Perspective



The Debug Perspective contains many elements already explored in the Java Perspective.

- The Editor appears in the middle of the perspective. It is open to the `TestStack` class, and the line at which you have stopped is highlighted.

Also notice that the marker bar has two overlapping markers: a blue arrow denoting the current instruction pointer and the blue circle denoting the breakpoint set previously.

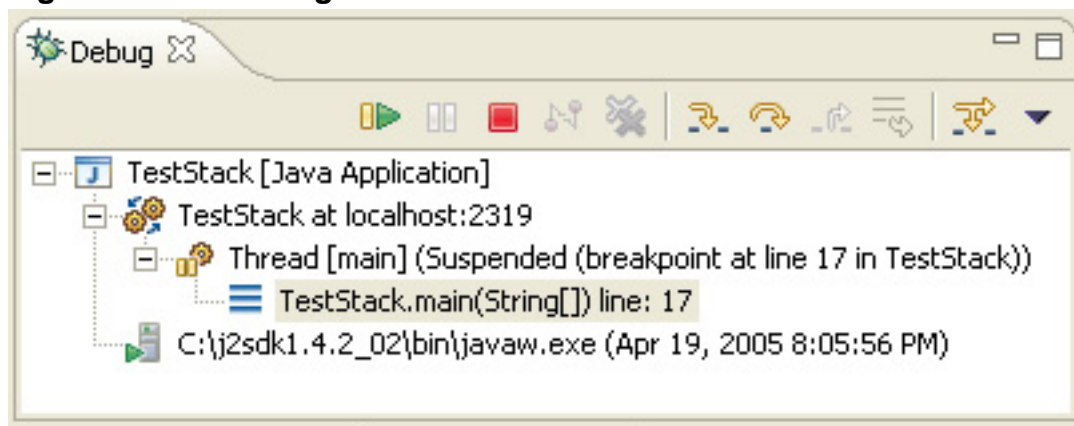
- The Console view appears at the bottom and contains the `TestStack` output you expect to see before the breakpoint is hit.

The Debug view

The Debug view, as shown in Figure 15, appears in the upper left of the Debug Perspective. This view displays the current launch configuration, process, thread, and stack frame information. You see that `TestStack` launch configuration contains a single process called `TestStack` with a process ID of 2319. The `TestStack` process contains a single thread called `main` that is suspended at line 17 in `TestStack`. This thread contains the stack frame information. (At the current breakpoint, the stack is only one level deep.)

The Debug view also has a toolbar for controlling the debugger. The tools are grayed if they are not active in a given context. Hover over the tools to see a tool tip describing them. In Figure 15, the active and available tools are Resume, Terminate, Step Into, Step Over, and Use Step Filters.

Figure 15. The Debug view



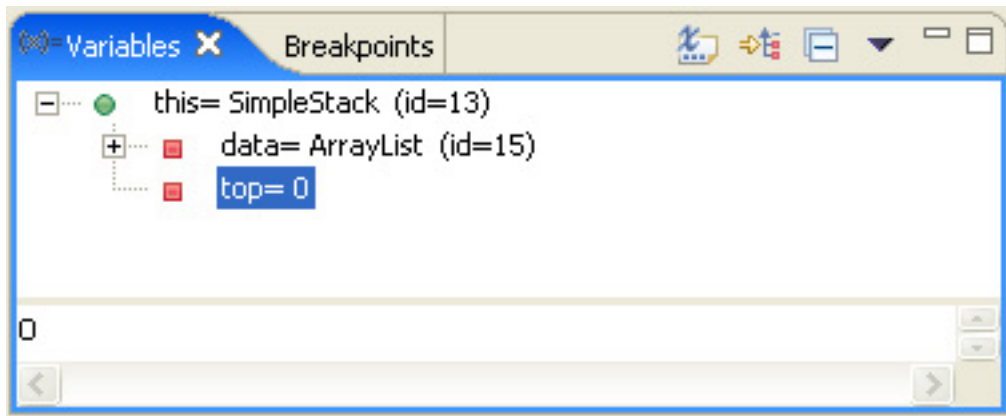
The Variables view

The Variables view, as shown in Figure 16, appears in the upper-right corner of the Debug Perspective. This view shows variable information regarding the Debug view's currently selected stack frame. Complex variables can be expanded by clicking on the **+** to show their contents. Right-clicking on a value allows many operations, including the ability to change the variable's value before resuming the program.

When the stack frame changes, the Variables view and Editor change to reflect the

current state of the debugging session.

Figure 16. The Variables view



Fixing SimpleStack

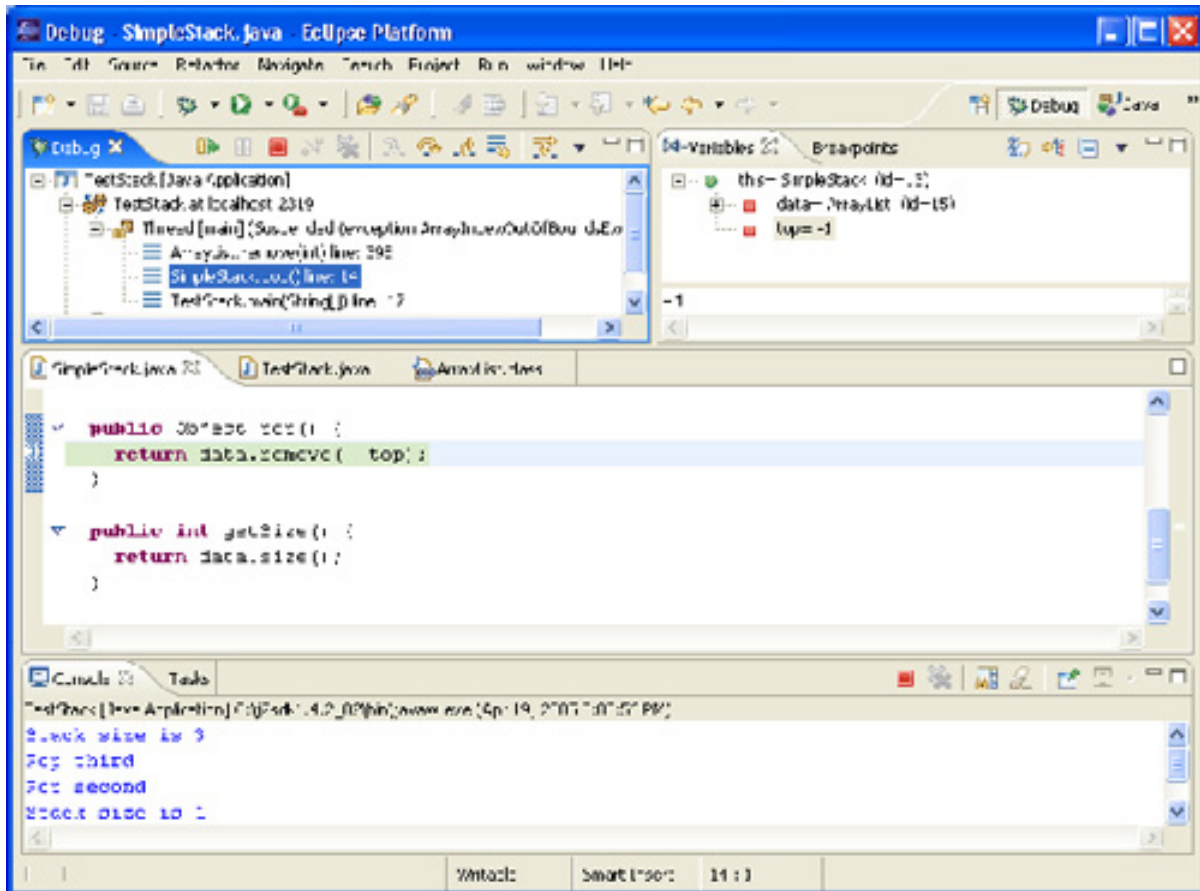
You could step through each line of the code to debug the problem, but because the Java language is throwing an exception, there is a quicker way. Simply select **Resume** in the Debug view, and the Java language will stop execution where the exception is thrown.

The Debug view stack trace shows that `ArrayList.remove` is failing. This is being called by the `SimpleStack.pop` method. Double-click on this stack frame entry in the Debug view. Eclipse should look like Figure 17.

The Variable view shows that `SimpleStack`'s `top` is set to -1. You can also see this by hovering over the variable 'top' in the Editor. As you expected, `SimpleStack` needs some bounds checking.

Click **Terminate** to step debugging. Click **Remove** to remove the debugging session.

Figure 17. The Debugger stops at the Java exception



Advanced debugging

As shown, the Eclipse debugger is intuitive if you've debugged in Visual Studio, but there are some unique Eclipse debugging features it is useful to know about. These features are beyond the scope of this tutorial.

- **Drop To Frame** allows you to backtrack to an earlier state in the program's execution without having to restart the debugging session.
- **Hot Code Replace (HCR)** allows you to change the code you are currently debugging and replace the running code without having to restart the debugging session.
- **Class Load Breakpoint** allows the debugged program to break upon the initial load of a named class.
- The Threads and Monitors view shows thread state, monitors, and thread deadlock situations. In Eclipse V3.1, this view no longer exists because this information is available directly in the Debug view.

Section 6. Source and refactoring tools

Refactoring

Refactoring is a software development practice by which the internal structure of a program is improved, but the external behavior does not change. Even if you are not familiar with the term, you have probably done it. Eclipse has extensive refactoring tools that make this "best practice" convenient and painless. Eclipse also has source tools to generate, organize, and format code. Both the refactoring and source tools are available from the main menu, and also have context menu capabilities.

As demonstrated by your venture into the Eclipse debugger, `SimpleStack` has some fatal run-time behavior if you pop more than you push. `SimpleStack` also violates one of the cardinal rules of object-oriented development: It requires callers to program to a concrete class instead of to an interface. This makes it difficult to replace `SimpleStack` with an improved version in the future without also changing and recompiling the caller.

You will fix the faulty `SimpleStack` and solve the weak design using refactoring and source tools. After fixing the example, you will experiment with other refactoring tools.

Extracting an interface

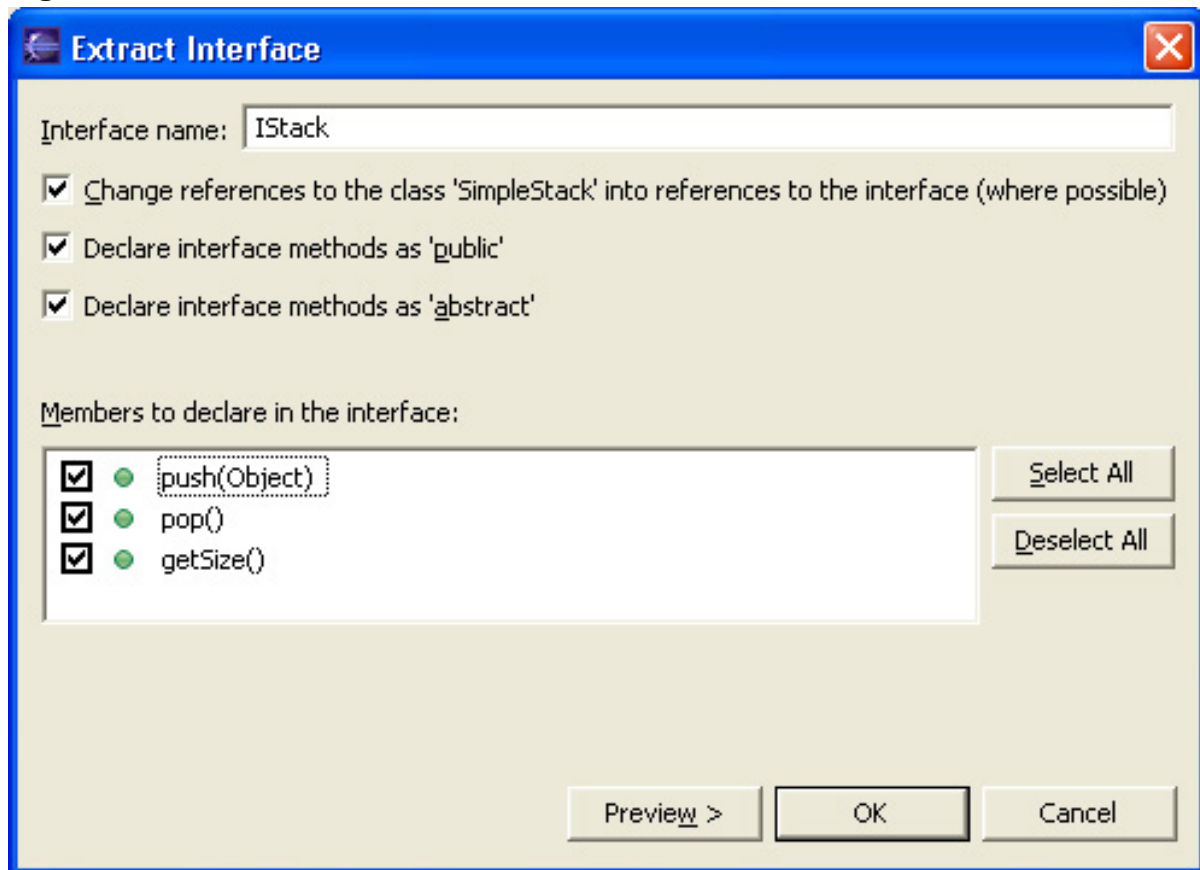
First, you will extract the interface from `SimpleStack` to an interface called `IStack`. Our callers can then call `IStack` methods and be protected from changes in the underlying concrete class. `SimpleStack` will implement the methods in `IStack`. `TestStack` will then program to the abstract `IStack`, instead of the concrete `SimpleStack`. This operation would be rather time-consuming and error-prone if you did it by hand. Eclipse does it with a single tool.

1. Switch to the Java Perspective.
2. Right-click `SimpleStack.java` and select **Refactoring > Extract Interface**. The dialog in Figure 18 appears.
3. Name the interface `IStack`.
4. Select the first checkbox. This will change `TestStack` to use references

to `IStack`, instead of `SimpleStack`.

5. All three members of `SimpleStack` need to be extracted, so check `push`, `pop`, and `getSize`.
6. Click **Finish**.

Figure 18. Extract interface



There are several code changes to note. There is a new interface file called `IStack.java` that looks like this:

```
public interface IStack {  
    public abstract void push(Object o);  
    public abstract Object pop();  
    public abstract int getSize();  
}
```

`SimpleStack` now implements the `IStack` interface. `TestStack` still instantiates the `SimpleStack` class, but references it as an `IStack`.

Save the code and run it. The behavior should be the same. Everything works, except the last `pop` still throws an exception. You'll fix that `pop` next.

A BetterStack

Now you'll fix `SimpleStack` by creating a new subclass of `IStack` with improved behavior. The improved stack will pop a null object and print an error if we pop past the top of the stack.

1. Select `SimpleStack.java` in Package Explorer and select **edit > Copy**. Then Select **Edit > Paste**.
2. When prompted for a new class name, enter `BetterClass`. Select **OK**.
3. This new class appears in Package Explorer.
4. Change the `BetterClass` pop method to print an error if you pop too many times:

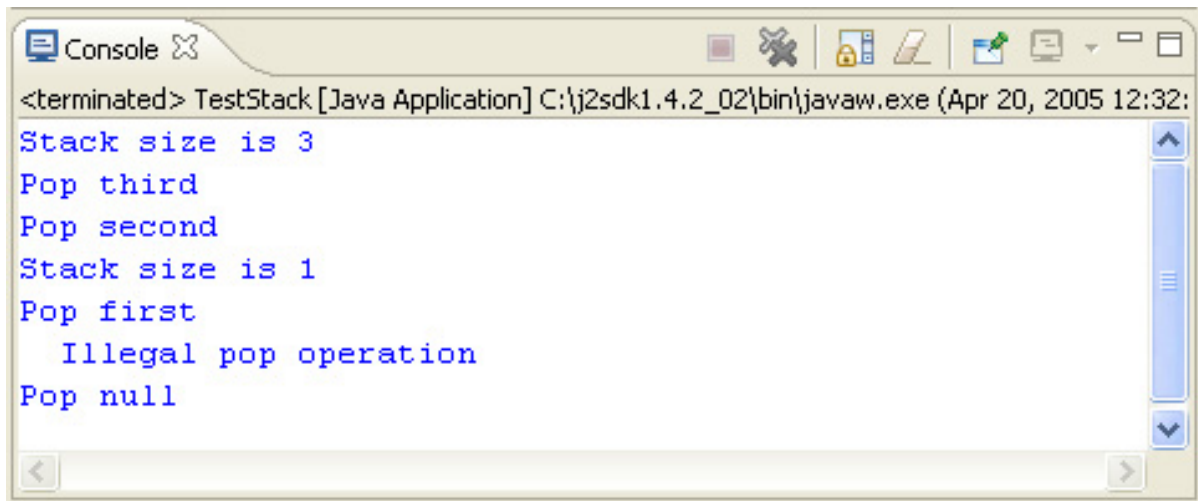
```
public Object pop() {
    --top;
    if (top < 0) {
        System.out.println(" Illegal pop operation");
        return null;
    }
    return data.remove(top);
}
```

5. Modify `TestStack` to call the `BetterStack` constructor, instead of the `SimpleStack` constructor.

```
IStack stack = new BetterStack();
```

6. Save and run `TestStack`. The Console should look like Figure 19.

Figure 19. BetterStack yields the expected output



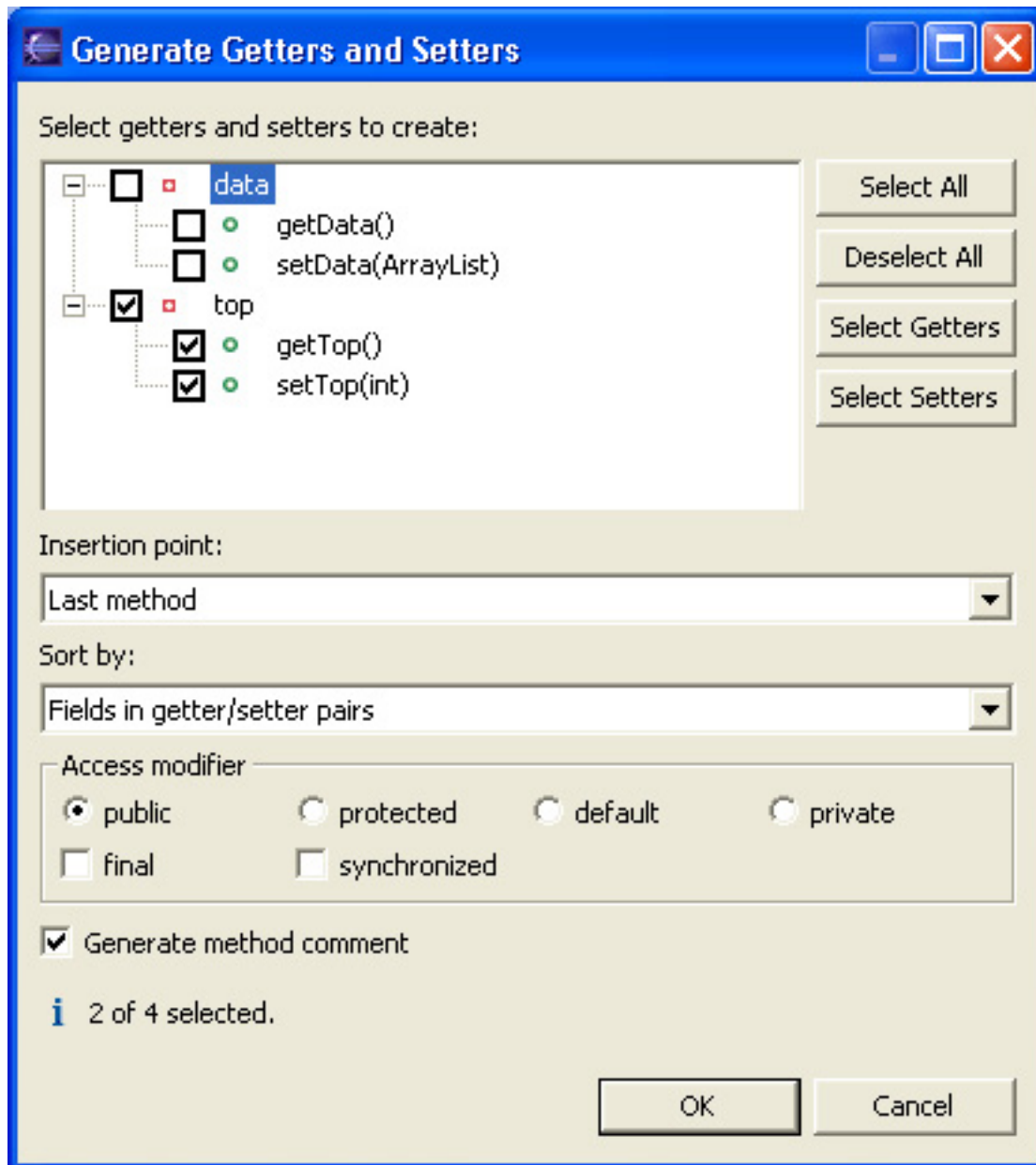
```
<terminated> TestStack [Java Application] C:\j2sdk1.4.2_02\bin\javaw.exe (Apr 20, 2005 12:32:
Stack size is 3
Pop third
Pop second
Stack size is 1
Pop first
  Illegal pop operation
Pop null
```

More refactoring

There are many other useful tools in the Refactor menu, including:

- **Rename** -- This is a frequently used tool for renaming an element, as well as any references to it. For example, select the `getSize` method in `IStack`. Now select **Refactor > Rename**. Change the name to `getStackSize`. Select **Preview** to view all the code that will change. Click **Finish**.
- **Undo** -- This allows the last refactoring to be undone. Select **Refactor > Undo** to see `getStackSize` restored to `getStack`.
- **Extract Method** -- This allows a selected section of code to be extracted into a method. Open `TestStack` and select the three push method calls. Select **Refactor > Extract Method**. The dialog in Figure 20 is shown. Name the method `initStack` and click **Finish**. The `initStack` method is created with the required parameters.

Figure 20. Extracting a method



Source tools

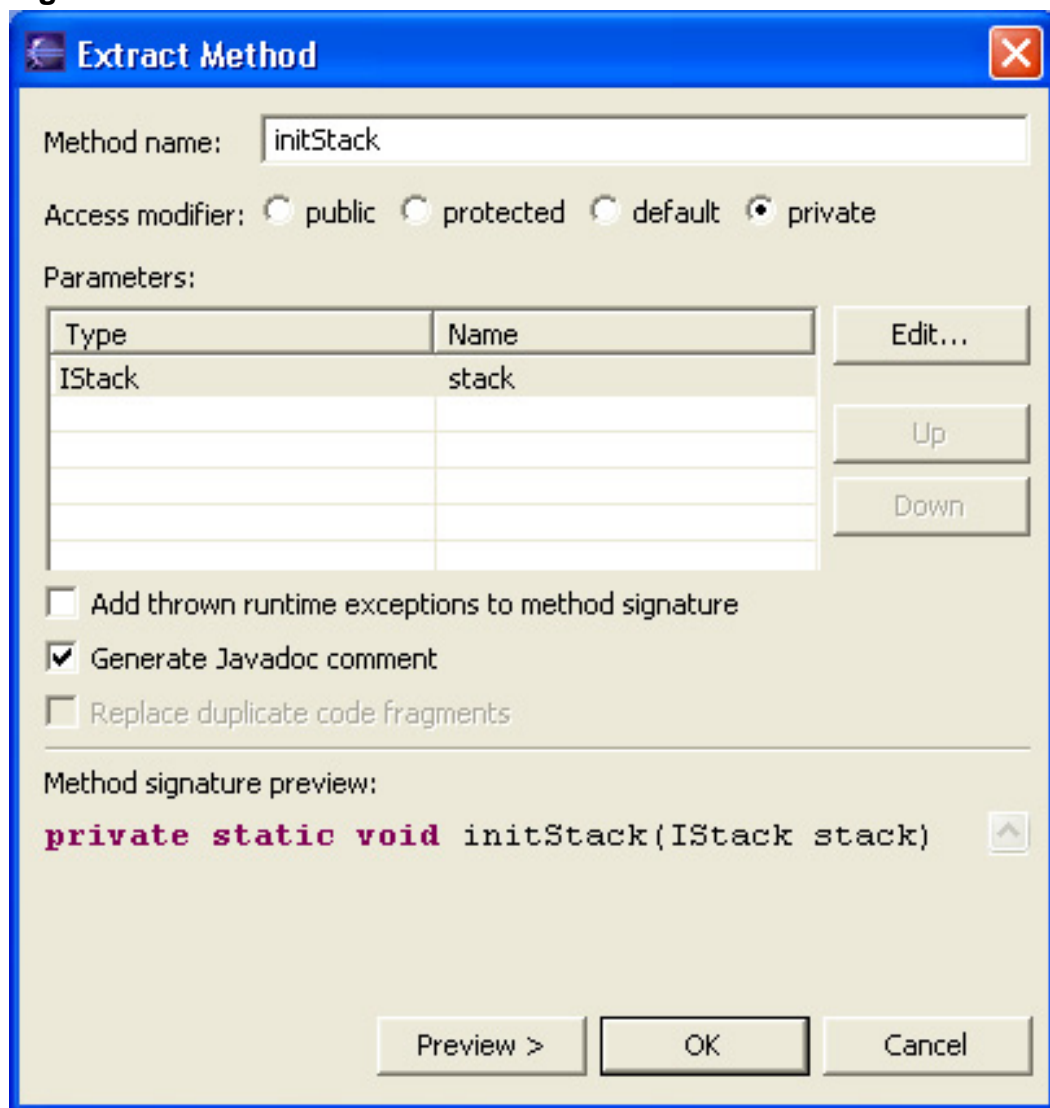
Beside the refactoring tools, Eclipse provides source tools to make the coding experience more efficient. Some worth trying with the sample code include:

- **Toggle Comment** -- Select a block of code and choose **Source > Toggle Comment**. If the code block is not commented, comments are added. Select **Source > Toggle Comment** again to uncomment it.
- **Shift Right and Shift Left** -- Select a block of code and select **Source >**

Shift Right to indent one level. Select **Source > Shift Left** to unindent the code one level.

- **Format** -- Select all the lines in one of the example Java files. Select **Source > Format** to format the code per the defined style. The style may be changed in **Window > Preferences > Java > Code Style > Code Formatter**.
- **Generate Getters/Setters** -- This tool allows `get` and `set` methods to be automatically generated for selected data members. Select `SimpleStack`, then **Source > Generate Getters/Setters**. The dialog in Figure 21 appears. Check methods to be generated. Click **Finish** to generate the methods.

Figure 21. Generate Getters/Setters



- **Add Javadoc Comment.** Javadocs are beyond the scope of this tutorial,

but these specially tagged comments are pulled out of the Java source code with a Javadoc tool to automatically create API documentation. The Javadoc comments are used by Eclipse in hover help and the Javadoc window. Select one or more methods or data members. Select **Source > Add Javadoc Comment** to automatically generate the comment. The Javadoc templates may be changed in **Window > Preferences > Java > Code Style > Code Templates**.

Section 7. Summary

In this tutorial, you've learned to use Eclipse to develop Java applications by developing a stack project. You found that Eclipse has many similarities (and a few key differences) to Visual Studio.

- You installed Eclipse and found your way around Perspectives, Views, and Editors.
- You created a Java project and used the Java Perspective to write the code.
- You learned how to run and debug a Java application.
- You explored the refactoring and source tools.

This tutorial touched on the basics of Java development using Eclipse. Where does one go from here? Here are some ideas:

- There is much more functionality to explore in the JDT.
- There are many plug-ins available to extend the functionality of Eclipse. These range from small open source tools to large commercial offerings.
- You may tailor the JDT by writing a plug-in specific to your own programming needs.

Resources

Learn

- [Eclipse.org](#) has downloads, articles, plug-ins, source code, projects, and newsgroups to explore.
- Learn more about Eclipse refactoring in "[Refactoring for everyone.](#)"
- Learn how to develop an Eclipse plug-in in "[Developing Eclipse plug-ins.](#)"
- A good book that covers developing with the Java IDE and developing Eclipse plug-ins is *The Java Developer's Guide to Eclipse* by Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy (Addison-Wesley, 2004).
- Refactoring is covered in great depth in *Refactoring: Improving the Design of Existing Code* by Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts (Addison-Wesley, 1999).
- Check out the "[Recommended Eclipse reading list.](#)"
- Browse all the [Eclipse content](#) on developerWorks.
- Users new to Eclipse should look at the [Eclipse start here.](#)
- Expand your Eclipse skills by checking out IBM developerWorks' [Eclipse project resources.](#)
- To listen to interesting interviews and discussions for software developers, check out [developerWorks podcasts.](#)
- For an introduction to the Eclipse platform, see "[Getting started with the Eclipse Platform.](#)"
- Stay current with developerWorks' [Technical events and webcasts.](#)
- Check out upcoming conferences, trade shows, webcasts, and other [Events](#) around the world that are of interest to IBM open source developers.
- Visit the developerWorks [Open source zone](#) for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Get products and technologies

- Get [Eclipse plug-ins](#) at the Eclipse Plugin Resource Center and Marketplace.
- Check out the latest [Eclipse technology downloads](#) at IBM [alphaWorks.](#)
- Innovate your next open source development project with [IBM trial software](#), available for download or on DVD.

Discuss

- The [Eclipse Platform newsgroups](#) should be your first stop to discuss questions regarding Eclipse. (Selecting this will launch your default Usenet news reader application and open eclipse.platform.)
- The [Eclipse newsgroups](#) has many resources for people interested in using and extending Eclipse.
- Participate in [developerWorks blogs](#) and get involved in the developerWorks community.

About the author

Scott Kellicker

Scott Kellicker has been a professional software developer for more than 15 years. He has used a variety of languages, operating systems, and development tools, including Visual Studio and Eclipse. He recently developed bridge technologies using Java Native Interface (JNI) technology to allow integration of Java, C++, and proprietary languages. He also has developed applications for the computer-aided design (CAD), geographic information systems (GIS), imaging, Internet, and scientific domains.