1 ECLIPSE 3.1 .................................................................................................................................................. 1
  1.1 INTRODUCTION ................................................................................................................................. 1
  1.2 INTERACTIVE DEVELOPMENT ENVIRONMENTS ........................................................................... 1
  1.3 THE ECLIPSE IDE ............................................................................................................................... 1
  1.4 INSTALLING ECLIPSE ........................................................................................................................ 2

2 ECLIPSE 3.1 AND JAVA .............................................................................................................................. 7
  2.1 INTRODUCTION ....................................................................................................................................... 7
  2.2 CREATING A JAVA PROJECT .............................................................................................................. 7
    2.2.1 Creating a Project ............................................................................................................................ 7
    2.2.2 Creating a Source File .................................................................................................................... 9
    2.2.3 Compiling a Source File ................................................................................................................ 11
    2.2.4 Executing a Project ....................................................................................................................... 12
    2.2.5 Creating a Run Configuration ..................................................................................................... 13
    2.2.6 The Console .................................................................................................................................. 16
  2.3 TAILORING THE ECLIPSE WORKSPACE ......................................................................................... 17
    2.3.1 Resizing Windows .......................................................................................................................... 17
    2.3.2 Adding Line Numbers to Source Files ......................................................................................... 18
    2.3.3 The Outline View .......................................................................................................................... 19
    2.3.4 Hot Keys ....................................................................................................................................... 21
    2.3.5 Code Assist ................................................................................................................................... 22
    2.3.6 Incremental Compilation ............................................................................................................... 24
    2.3.7 Code Style Preferences ................................................................................................................ 24
    2.3.8 Expanding and Collapsing Source Code ........................................................................................ 27
  2.4 WORKING WITH MULTIPLE SOURCE FILES .................................................................................. 28
  2.5 WORKING WITH MULTIPLE PROJECTS ............................................................................................. 31
  2.6 OPENING AND CLOSING PROJECTS ................................................................................................. 33
  2.7 RELOADING (REFRESHING) PROJECT FILES ..................................................................................... 36
  2.8 WORKING WITH DATA FILES ........................................................................................................... 36
  2.9 RUN-TIME (COMMAND-LINE) PARAMETERS .................................................................................... 40
  2.10 VERSION CONTROL (LOCAL HISTORY) ............................................................................................ 42
  2.11 OTHER ECLIPSE FEATURES ............................................................................................................ 45
    2.11.1 Developing the Source Code ....................................................................................................... 45
    2.11.2 Refactoring ................................................................................................................................ 46
  2.12 JAVA DEBUGGER ................................................................................................................................. 46
  2.13 SUMMARY ......................................................................................................................................... 50

3 ECLIPSE 3.1 AND C/C++ .......................................................................................................................... 51
  3.1 INTRODUCTION .................................................................................................................................. 51
  3.2 INSTALLING A C/C++ COMPILER ...................................................................................................... 51
    3.2.1 Cygwin ......................................................................................................................................... 51
    3.2.2 MinGW ....................................................................................................................................... 52
  3.3 INSTALLING THE CDT ECLIPSE PLUG-IN ...................................................................................... 53
  3.4 CREATING A C PROJECT ................................................................................................................... 55
    3.4.1 Terminating an Executing Project ................................................................................................. 60
  3.5 CREATING A C++ PROJECT ................................................................................................................ 63
  3.6 CREATING A STANDARD MAKE C PROJECT .................................................................................. 68
  3.7 CREATING A STANDARD MAKE C++ PROJECT ............................................................................. 75
  3.8 WORKING WITH MULTIPLE PROJECTS ............................................................................................ 77
  3.9 THE C/C++ DEBUGGER ...................................................................................................................... 80
  3.10 SUMMARY ....................................................................................................................................... 84
1 ECLIPSE 3.1

1.1 Introduction

Developing programs using a text editor such as TextPad is simple and straightforward. While such editors allow students to begin programming with a minimum of distractions, as programs become larger and more complicated, a more sophisticated tool for developing programs is quite valuable. In this chapter, we examine one particular tool for developing large programs, the Eclipse system. Eclipse can be used to develop programs in Java and in C/C++ (plus various other languages as well). In this chapter, we examine how to install Eclipse.

1.2 Interactive Development Environments

Interactive development environments (IDE) provide a wealth of features to support the programmer that are not available in a simple editor. There are many IDE’s currently available, some are shareware and some must be purchased and are often expensive. Most IDE’s provide similar facilities, such as the ability to view different classes at the same time, debuggers that step through a program line-by-line in order to assist in locating errors, etc. One of the newest and most popular IDE’s is the Eclipse IDE which is available at no cost from the Eclipse website (http://www.eclipse.org/).

1.3 The Eclipse IDE

Eclipse is an IDE (actually it is more than an IDE but we will concentrate on Eclipse as an IDE) that was initially controlled by IBM but then IBM released the source code for Eclipse, making the project open source. Eclipse development is taking place in several cities, including Winnipeg, Ottawa, and Zurich, although there is also a large community of programmers who provide extensions to Eclipse in order to improve the Eclipse environment.

The facilities in Eclipse can be extended by defining “plug-ins” which provide additional features or support additional languages. The site http://www.eclipseplugincentral.com/ contains a wide variety of Eclipse plug-ins.

Eclipse is written primarily in Java and requires recent version of the Java SDK. While there are various versions of both Java and Eclipse, in these notes we will be using the 1.5.0_04 version of the Java SDK and version 3.1.0 of Eclipse. (The notes were originally developed using the 1.4 Java SDK so that system works equally well.) All of the examples shown in this chapter were run on a Windows XP Professional workstation. Eclipse runs on most operating systems, including Unix, Linux, Windows, and Mac OS X.
1.4 Installing Eclipse

Eclipse can be downloaded from the Eclipse website: www.eclipse.org.

Click on the downloads link and then click on the link for Eclipse 3.1 if you want to download the Windows version of Eclipse. If you want to download Eclipse for a different operating system, click on the Other downloads link.

Save the file to your computer. Unzip the file using WinZip or a similar decompression utility to create the folder/directory Eclipse. You can move this directory to any location but the top of the C: or D: drive is a common location.
Double-click on the eclipse.exe file to start Eclipse.

The Eclipse logo is displayed briefly.

The first time that Eclipse is launched, you are asked to indicate where your Eclipse source files are to be stored. The default is with your Windows XP profile “C:Documents and Settings\Fred\workspace”.

You may change the location of the workspace to any convenient location. If you are the exclusive user of a machine, you may choose to store your files in the Eclipse folder itself.

Then a progress bar that shows the progress of loading the various parts of Eclipse (referred to as plug-ins) is displayed.

When Eclipse has loaded all of its plug-in, the initial screen contains some icons in the middle of the screen. Clicking on these icons will provide additional information on Eclipse.
Once you are ready to begin using Eclipse, click on the Workbench icon in the top right corner of the window.

![Welcome to Eclipse 3.1](image)

The workbench screen is then displayed. Notice that the four icons that appeared in the middle of the initial screen are now displayed in the bottom-left corner of the workbench.

![Eclipse Workbench](image)

If you now quit Eclipse, you will notice that a workspace folder has been added in the Eclipse folder (assuming that you did not create the workspace in another location). All source code that is created is stored in this folder.
Chapter 2

Eclipse 3.1 And Java

2 ECLIPSE 3.1 AND JAVA

2.1 Introduction

In this chapter, we will begin with a simple description of how to develop Java programs in Eclipse.

2.2 Creating a Java Project

In this section we examine how to create a Java source file, compile the source file, and then run the resulting program.

2.2.1 Creating a Project

Source code in Eclipse is stored in a Project. Eclipse makes creating a new Java project relatively easy.

First, click on the New Java Project icon.
Enter a name for the project.

Normally programmers store the .java and .class files in separate directories; this can be accomplished by selecting the “Create separate source and output folders” button. Click the Finish button.
2.2.2 Creating a Source File

Click the New Java Class icon.

Enter the name of the class and select “public static void main (String[] args)” to cause an empty main method to be created automatically.
Type “System.out.println(“ into the main method. When the left bracket is typed, the matching right bracket is added automatically.

```java
public class Java1 {

/**
 * @param args
 */

public static void main(String[] args) {
    // TODO Auto-generated method stub
    System.out.println();
}
```

When you type the first double-quote character, the matching double-quote character is automatically added. This process of completing portions of statements is referred to as “auto-completion” and is a very useful feature in Eclipse. (Auto-completion can be turned off if desired.)

```java
/**
 * @param args
 */

public static void main(String[] args) {
    // TODO Auto-generated method stub
    System.out.println(""");
}
```

Note that there is a green vertical line after the ending double-quote mark.
Typing the Tab character causes Eclipse to move the cursor to the green line.

At this point, Eclipse inserts another green line after the matching bracket; again typing the tab character moves the cursor to this point.

### 2.2.3 Compiling a Source File

After the source file has been created, if you save the file (Ctrl-s), the file is automatically compiled and the results of the compilation are displayed at the bottom of the workspace in the Problems tab.

Eclipse generates a message that there is a missing semicolon and marks the location of the problem in the source code with a .

Adding the missing semicolon and saving the file again results in a clean compile.
2.2.4 Executing a Project

To execute the program, select the source file (by clicking on its window) and then click on the dropdown button beside the Run button. Select Run As † Java Application. (If the Run As Java Application option is not available, see the next section.)

The expected output is displayed in the Console window at the bottom of the screen.
Once a project has been run, it can be run again by typing Ctrl-F11 or by clicking on the Run button ( ).

### 2.2.5 Creating a Run Configuration

In the previous section, we were able to select the appropriate way to run the Java program by using the Run As menu item. As projects become more complicated, the configuration of the execution environment may require a small amount of additional effort.

To begin, select the Run menu item from the dropdown beside Run.
If the project has already been run, then a run configuration will already exist. The run configuration normally has the same name as the project (although this does not have to be true).

If the project has not yet been run, a run configuration must be created. Right-click on the Java Application item and select New.
This creates a run configuration for the current project with most of the entries already filled in.

Clicking on the Run button runs the application as before.

Once a run configuration has been created, it is visible when the dropdown (▼) beside Run is clicked.
2.2.6 The Console

The system console at the bottom of the screen contains any output sent to the console and can also be used to enter information into a program (via System.in).

When a program is executing, the Terminate icon is active (bright red). A program can be terminated at any time by clicking the Terminate icon.

The console may be cleared of its current contents by clicking on the Clear Console icon.

If a program does not terminate, for example, it goes into an infinite loop or it is waiting for something (Godot?), you can resume editing its source files (or the source files of any other project) and re-run the project but system performance will suffer because the original application is still running. If you forget to terminate an application, you can terminate it later by clicking Display Selected Console.

Select the other console that is still running and terminate that application.
Running multiple projects at the same time is usually a mistake but there may be occasions when it is useful to run multiple projects at the same time. You can switch from one running project to another using this technique by changing consoles (there is one console for each running application). Note though that system performance will probably be sluggish if several applications are active at the same time. If you run applications concurrently, you should probably send some information to the console from time to time to indicate the status of the application.

2.3 Tailoring the Eclipse Workspace

In these notes we normally describe the simplest way of performing a task. However, there are often several other equally appropriate ways of performing the same task. Similarly, there are many preferences that can be modified to tailor the workspace to the programmer’s needs. In this section, we examine some of the more commonly-used features. There are many more features that we will not examine that are available to the programmer – if you want to do something, you probably can in Eclipse.

2.3.1 Resizing Windows

Eclipse provides simple mechanisms for resizing and showing/hiding the various windows used in a perspective (formally, they are panes not windows, but windows is the more commonly used term).

To change the size of an existing window, move the cursor to the area between two window panes and drag left or right (or up or down).

Dragging the source window to the left decreases the size of the Package Explorer window.
To increase the size of a window to full screen, select the window and then type Ctrl-M or double-click on the window’s header (the portion highlighted in blue when the window is active). Repeat the process to return the window back to its normal size.

2.3.2 Adding Line Numbers to Source Files

To add line numbers to the source files, just right-click on the left margin of a source file and turn Show Line Numbers on or off.
2.3.3 The Outline View

The Outline view contains a list of all methods and variables in the current class. As you navigate from one section of code to another, the current method/variable is selected in the outline view. For example, click beside the “{” that begins the Java class. Two changes occur on the screen. First, the matching “}” is highlighted by Eclipse. Secondly, the associated element in the outline view is selected (Java1 in this case).
If we now click on the “{” that opens the main method, the same two changes are made to the display of the source code: the matching “}” is highlighted and the corresponding item (main) in the outline view is selected.

Conversely, if an item in the outline view is selected (with a single or double click), the corresponding portion of the source code is indicated. For example, if we click on “main” in the outline window, the main method declaration is highlighted.

To produce more visible and permanent highlighting of the current class/method/variable, select the Mark Occurrences button at the top of the screen (when the source code window is
active). This causes the name of the class/method/variable to be highlighted in yellow and to remain highlighted as long as the user is editing that portion of the code.

If you want to remove the outline view completely, just click its close icon ( ). To restore the view, select Window ➤ Show View ➤ Outline.

The windows/panes in a perspective may also be re-arranged as well as resized. Try dragging a window to a different location to experiment with this facility.

### 2.3.4 Hot Keys

Eclipse provides many keyboard shortcuts for the programmer. Select Help ➤ Key Assist
and the following summary of keyboard shortcuts is displayed.

<table>
<thead>
<tr>
<th>Command</th>
<th>Key Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activate Editor</td>
<td>F12</td>
</tr>
<tr>
<td>Add Block Comment</td>
<td>Ctrl+Shift+J</td>
</tr>
<tr>
<td>Add Import</td>
<td>Ctrl+Shift+M</td>
</tr>
<tr>
<td>Add Javadoc Comment</td>
<td>Alt+Shift+J</td>
</tr>
<tr>
<td>Backward History</td>
<td>Alt+Left</td>
</tr>
<tr>
<td>Breakpoints</td>
<td>Alt+Shift+Q</td>
</tr>
<tr>
<td>Build All</td>
<td>Ctrl+B</td>
</tr>
<tr>
<td>Change Method Signature</td>
<td>Alt+Shift+C</td>
</tr>
<tr>
<td>Cheat Sheets</td>
<td>Alt+Shift+Q</td>
</tr>
<tr>
<td>Close</td>
<td>Ctrl+F4</td>
</tr>
<tr>
<td>Close All</td>
<td>Ctrl+Shift+F</td>
</tr>
<tr>
<td>Collapse</td>
<td>Ctrl+M</td>
</tr>
<tr>
<td>Collapse Subtrac</td>
<td>Ctrl+Shift+Q, C</td>
</tr>
<tr>
<td>Console</td>
<td>Alt+Shift+T, C</td>
</tr>
<tr>
<td>Content Assist</td>
<td>Ctrl+Space</td>
</tr>
<tr>
<td>Context Information</td>
<td>Ctrl+Shift+Space</td>
</tr>
<tr>
<td>Convert Local Variable to Field</td>
<td>Alt+Shift+F</td>
</tr>
<tr>
<td>Cursor</td>
<td>Ctrl+C</td>
</tr>
</tbody>
</table>

Press "Ctrl+Shift+L" to open the preference page.

Then type Ctrl-Shift-L to view the keyboard shortcut preferences page. These preferences may be modified by the programmer (just like Emacs but without the work).

### 2.3.5 Code Assist

We have already seen examples of the Eclipse code completion feature whereby Eclipse attempts to assist the programmer by adding matching brackets, quotes, etc. Eclipse also provides a code assist feature in which Eclipse provides a list of the variables or methods that are valid in a particular context. For example, if we type “System.”, Eclipse pops up a small box that contains valid completions for methods and variables in the System class.
For example, the programmer can double-click on “out” in the list (or select the entry and then type Enter) and this variable is added to the current statement. Note that the type of each variable and the signature of each method are displayed.

As you continue typing, other choices are provided. For example, the methods that complete System.out are displayed, along with their signatures. Again, double-clicking on an entry causes Eclipse to add the entry to the source code.
If the code assist box goes away or if you have already selected a variable/method and want to examine alternatives, click to the right of the period that precedes the item in the source code and then type Ctrl-Space to bring up the code assist box again.

### 2.3.6 Incremental Compilation

Eclipse performs an incremental compilation as code is entered rather than waiting until the code is saved to perform a full compilation of the class. For example, in the program below, typing “fred” instead of “println” causes Eclipse to add an error indicator beside the statement.

If the cursor is moved over the error indicator (棺), Eclipse displays the error message. In this example, Eclipse indicates that fred(String) has not been defined. The problem may have been caused by a typing error or it may be that fred(String) is not defined.

### 2.3.7 Code Style Preferences

Programmers (and organizations) have preferences and/or standards for the way in which code is formatted. Eclipse provides significant flexibility in the style used to generate code.

Select Window ▼ Preferences. Expand the preferences in the left pane and click on Formatter under Java ▼ Code Style.
Click on Show to display the properties. You will find a large number of preferences that can be set so that your code is formatted exactly as you desire.
For example, if you want to change the location of the brace brackets that enclose methods and classes, click on the Braces tab. By default, the left brace is on the same line as the class/method.

You can change this position by selecting Next line so that the brace is placed at the beginning of the next line. (There are other options available.)

When you apply any changes that were based on a predefined profile, you must create a new profile because you are not permitted to modify the predefined profiles.

© David Scuse
University of Manitoba 26
When your profile is complete and you return to the workbench, you will find that the code does not reflect the new profile.

To apply the new profile, select Source \( \Rightarrow \) Format.

### 2.3.8 Expanding and Collapsing Source Code

If you look at the diagram above, you may notice that there are minus-sign icons (\( - \)) beside some of the lines. Clicking on one of these icons causes the corresponding section of source code to be collapsed to make working with the source code easier. In the diagram below, the main method is collapsed. Any portion of code that has been collapsed can be expanded by clicking on the plus sign (\( + \)) that is now beside the collapsed code.
2.4 Working With Multiple Source Files

Eclipse provides excellent facilities for manipulating multiple source files at the same time. We can create a new class by clicking on the New Java Class button ( ). If the main class already exists, select the “Constructors from superclass” option to cause Eclipse to create an empty constructor.

Note that the outline view now defines the components in the new class.
You can switch from one source file to another simply by clicking on the name of the source file at the top of the source window.

You may close a source file window by clicking on its close icon. If you later need to edit a file that is not in the source window, just double-click on the file name in the package explorer window.

We can now add some statements to the new class.

If we switch back to Java1 by clicking on its header, we can add the statement that creates a new SecondClass object. We can now create the object from the main method in Java1.
You may notice that we did not save the changes to SecondClass before switching back to Java1. (This fact is indicated by an asterisk beside the class name in the window header.) If we attempt to run the program, Eclipse warns that there are unsaved changes to the file SecondClass and provides the opportunity to have the changes saved and the file recompiled before beginning execution.
The execution of the program produces the following output:

![Image of Eclipse IDE](image)

### 2.5 Working With Multiple Projects

The examples developed so far involved only one project. This was done simply to avoid having too much information available at one time. However, Eclipse allows the programmer to have any number of active projects in a workspace.

If we create a new project named Java2, the workspace now contains the following information.

![Image of Eclipse IDE with Java2 project](image)

We can create a class file for Java2 in the same manner that a new class file was created earlier. The source file is added to the source window, along with the source files for the project Java1.
At this point, if you do not need the source files for Java1, they could be removed from the source window by clicking on their close icons.

If the “Hello World” statement is added to the main class, the project can be run by selecting the Run drop-down icon and then selecting Run As Java Application.

Once both projects have a run configuration defined, either project may be run by selecting the desired run configuration from the run drop-down icon. Thus, the Java1 project may be run even though we are currently editing the Java2 project.
The run configuration listed at the top of the menu is the one that is executed if the user just clicks on the run button ( ▶️ ) or types Ctrl-F11.

The programmer may create any number of projects and each project may consist of any number of source files.

2.6 Opening and Closing Projects

Each project that is created is shown in the Package explorer window at the left of the screen.

The contents of a project may be displayed by clicking on the + to the left of the project name and may be hidden by clicking on the – to the left of the project name.

A project may be closed by selecting the project, right clicking on the project, and then selecting the Close Project command.
The Close Project command keeps the project in the package explorer but the information about the project is no longer available (the + beside the project name is no longer present). Once a project has been closed, it can no longer be executed.

A project that has been closed may be reopened by right-clicking on the project and selecting the Open Project command.

You may also remove a project from the Package Explorer by right-clicking on the project and then selecting the Delete command. By default, this command does not actually delete the project, it simply removes the project from the Package Explorer but leaves the project in the workspace. If you want to physically remove all files in the project, select the “Also delete contents” radio button.
To restore a file to the Package Explorer, select File $\Rightarrow$ Import $\Rightarrow$ Existing Project into Workspace. Then click Browse on the Select root directory button and click on the project within the workspace.

Then click the Finish button.
2.7 Reloading (Refreshing) Project Files

If you examine the workspace folder, you will find that the source files are stored in the src folder and the class files in the bin folder.

In general, you should not modify the src files outside of Eclipse but if you do, then you must force Eclipse to reload all files in your project. You can do this by right-clicking on the project name and then selecting the Refresh option.

2.8 Working with Data Files

As programs become more sophisticated, most programs use file input and output instead of the system console. File I/O with Eclipse is the same as if a simple editor were used. Any input and output files that do not include a full path to the file are stored at the top level of the Eclipse project folder (i.e. not in the SRC folder).

The following program copies the contents of one file (in.txt) to an output file (out.txt).
If you create the input file and move it into the Java4 folder and then refresh the project, the input file becomes visible in the Package Explorer.

If you double-click on in.txt in the Package Explorer window, the file is loaded into the editor window.

The file can be edited and saved in the same manner as any other file.

Alternatively, the input file can be created within Eclipse using the New File item.
Enter the name of the file and click Finish.

The file now is displayed in the Package Explorer. You can edit and save the file as you would any other file.
After the program is run, if you refresh the project, the output file is also visible in the Package Explorer.

You can double click on the output file and examine its contents.

If the program is run again while the output file is active in the editor window, Eclipse warns you that the contents of the file have changed and asks if you want to load the modified file into the source editor.
Being able to read and modify data files as well as source files from within Eclipse makes the development process much easier.

2.9 Run-Time (Command-Line) Parameters

Parameters may be passed to a Java program at run-time. If we take the previous file copy program, we could pass the names of the input file and the output file as parameters to the main program.
To supply the values, bring up the run configuration for the project, select the Arguments tab and then enter the two file names (separated by a blank).

Now the program can be run with different parameters without having to recompile the program.

In Chapter 3, we will examine how a Java program can be executed outside of Eclipse. Runtime parameters may also be included when a program is executed outside of Eclipse.
2.10 Version Control (Local History)

Eclipse provides a simple form of version control (referred to as the local history) that permits a programmer to recover from lost or incorrectly modified source code. (Eclipse also supports CVS, the team version control system, but that topic is beyond the scope of these notes.)

Suppose that we begin with our simple Hello World program and then make some modifications to it.

If we subsequently decide that the change(s) should not have been made and we would like to revert to the previous version, we can right-click on the left margin of the file and bring up the local history of modifications made to the file. The Compare With command is used to compare the saved versions of a file (but not to replace the current version).

Initially, the current version is displayed and is compared with the previous version of the file.
You can compare the current version with other earlier versions by clicking on the timestamp of any earlier version.

To restore a saved file, use the Replace With ‡ Local History command. (You could also have started with the Replace With command instead of the Compare With command.)
Select the saved version that is to replace the current version.

The current version is now the earlier version.

The version that was replaced is still in the local history and it may be used to replace the current version if you change your mind again.

The parameters for the local history are defined in the Workspace parameters section of Window → Preferences.
2.11 Other Eclipse Features

While it is not the place of these notes to describe every feature that is available within the Eclipse workbench, there are a few significant areas that the programmer should examine.

2.11.1 Developing the Source Code

Eclipse provides various mechanisms that simplify the creation of code. For example, if a try/catch block is required, just select the statement(s) to be enclosed in the try block and select the Surround with try/catch Block menu item from the Source menu.

If you take a look at the Source menu, you will find other helpful commands.
2.11.2 Refactoring

Eclipse provides a wide variety of refactorings that help the programmer improve the quality of his/her code.

<table>
<thead>
<tr>
<th>Refactor</th>
<th>Navigate</th>
<th>Search</th>
<th>Project</th>
<th>Run</th>
<th>Window</th>
<th>Hi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rename...</td>
<td>Alt+Shift+R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move...</td>
<td>Alt+Shift+Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Change Method Signature...</td>
<td>Alt+Shift+C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convert Anonymous Class to Nested...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Move Number Type to New File...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Push Down...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Pull Up...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extract Interface...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Generalize Type...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use Supertype Where Possible...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Infer Generic Type Arguments...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inher...</td>
<td>Alt+Shift+I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extract Method...</td>
<td>Alt+Shift+M</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extract Local Variable...</td>
<td>Alt+Shift+L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extract Constant...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduce Feature...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Introduce Factory...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Convert Local Variable to Field...</td>
<td>Alt+Shift+F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Encapsulate Field...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2.12 Java Debugger

When there are errors in a simple Java program, the errors can normally be identified through the use of a few print statements. As programs become larger, using print statements to track down an error is a time-consuming process. Eclipse includes a Java debugger that makes the debugging task relatively easy. When the following program is run, it generates a run-time error, as shown in the Console window.
To run the program with the debugger, click on the debugger run icon ( ).

This causes Eclipse to switch to the Debug perspective (from the Java perspective) and to run the program until the error occurs.

The statement at line 16 that caused the error is highlighted in the source window and the current value of each program variable is displayed in a debugger window named Variables. Clicking on a variable displays the value of the variable at the bottom of the window.

Note that the program is still active; to terminate it, click on the terminate icon ( ).
Knowing the value of the variables at the time of the error may be sufficient to solve the problem but in many situations, we want to know what happened in the program prior to the error. To enable a breakpoint that stops the execution of the program at a specific line, double-click in the margin on the line in the source code. Note that a blue circle appears beside the line. Also, the breakpoint is identified in the Breakpoints window.

Now when the program is run, the debugger pauses execution prior to executing line 16. The first time through the loop, the values of the variables prior to executing line 16 are shown in the variables window. Note that there is a plus sign (+) beside myArray; clicking on the plus sign expands the array so that the individual array entries are displayed.
To continue the execution, click on the Resume ( ▶️ ) icon. Remember to terminate the application at some point.

If you want to stop at each statement, set a breakpoint at the first statement and then click on the Step Into icon ( ▶️ ) instead of the Resume icon. Step Into causes Eclipse to execute a statement and then pause before executing the next statement.

Eventually you realize that the loop parameters are incorrect and fix the parameters; the program now runs correctly, generating the correct value in the console window.

There are many other facilities that are available with the Java debugger; you can experiment with it by observing what happens when each button is clicked.

You return to the normal Java perspective by clicking on the Java perspective icon which is beside the Debug icon.
2.13 Summary

In this chapter, we have presented a very brief overview of the Eclipse IDE. As you work with Eclipse, you will find many additional useful features. If you take a look at the Eclipse plug-in site (http://www.eclipseplugincentral.com/), you will find a large variety of interesting plug-ins.

In the examples in this chapter, we did not use Java’s Packages. However, Eclipse works with packages quite nicely. Any project can be subdivided into packages and Eclipse will ensure that all source files in all packages are recompiled every time that the source files are modified.

There is also a collection of tutorials dealing with various aspects of Eclipse at http://www.cs.umanitoba.ca/~eclipse/. Many of the tutorials focus on developing GUI-based applications using Java and the Eclipse replacement for Swing, the SWT (the Standard Widget Toolkit). In addition, some tutorials related to refactoring in Eclipse are included. These tutorials were written using Eclipse 2.1 but the content of the tutorials is still relevant even though the appearance of the Eclipse windows has changed.
3 ECLIPSE 3.1 AND C/C++

3.1 Introduction

Although Eclipse is written in Java, Eclipse has been designed so that it can be used to support multiple languages, not just Java. C and C++ programs can be developed by installing the CDT plug-in. Most of the Eclipse features described earlier for the Java environment also apply to the C and C++ environments.

3.2 Installing a C/C++ Compiler

Before you can develop C/C++ projects, you must first install an appropriate C/C++ compiler on your system. There are several open-source compilers available for use on Windows, with most based on the Unix GCC project. We will take a quick look at 2 such systems. Both systems worked correctly with the examples that were tested.

3.2.1 Cygwin

The Cygwin (1.5.18.1) package is available at http://www.cygwin.com. When you install Cygwin, ensure that the GCC compiler tools are included. These tools are not included by default and you will have to check the packages to ensure that the tools are installed. (Expand the package DLevel and you should find the tools.) The Cygwin system requires approximately 945 MB of storage for a install that includes all of the DLevel tools. This can be reduced if you know exactly which tools you require.

Once installed, the directory structure should look something like the following:
To ensure that the tools are available, first ensure that your “path” variable includes the path to the cygwin bin directory:

D:\cygwin\bin;

Test the path by typing the following command into a Windows command window:

gcc –v

The output should be similar to the following:

Similarly, a “make” command must also be available on the system path.

make –v

The output should be similar to the following:

As long as gcc and make are available, you will be able to compile C/C++ programs in Eclipse under Windows.

3.2.2 MinGW

Alternatively, you can install MinGW (Minimalist GNU for Windows), a version of the gcc compiler tools. MinGW is available at http://www.mingw.org. The current version of MinGW (version 4.1.1) has not yet been tested with the new release (3.0.0) of the CDT (see the next section for information on the CDT). However, version 3.1.0-1 of MinGW has been tested with the CDT 3.0.0 and it works correctly.
If you use the MinGW tools, a non-standard name for the “make” command is used to execute the script that compiles C and C++ programs. However, if you make a copy of the make file (named mingw32-make.exe in version 3.1), leave the copy in the same directory, and then rename the copy to “make.exe”, this should eliminate the problem. The MinGW system requires approximately 46 MB of storage. Ensure that your “path” variable includes the path to the bin directory:

D:\mingw\bin;

Then test the path for the gcc compiler by typing gcc -v into a Windows command window:

Similarly, test the path for a “make” command by typing make –v into a Windows command window.

Overall, we have found that Cygwin is less prone to problems than MinGW and the examples shown in this chapter use Cygwin.

3.3 Installing the CDT Eclipse Plug-in

To run C/C++ programs, you must install the Eclipse CDT (C/C++ Development Tools) plug-in. This plug-in can be downloaded from the Eclipse CDT page (http://www.eclipse.org/cdt/).

If you scroll down the CDT home page, you will find the announcement that version 3.0 has been released. Ensure that you locate the CDT 3.0.0 (or higher) download because only this version runs with Eclipse 3.1. (The version used in these notes was org.eclipse.cdt-3.0.0-win32.x86.zip.) Click on the CDT Eclipse 3.1 Based Releases Page.

Scroll down until the CDT 3.0.0 link is visible. Click on the link to the CDT Eclipse 3.1 Based Releases Page.
Scroll down the page until the CDT 3.0.0 link is visible and click on it.

Download the zip file that is appropriate for your operating system. Move the zip file so that it is at the same level as your Eclipse folder (i.e. not inside the Eclipse folder). Your Eclipse folder must be named Eclipse – if it isn’t, temporarily rename your folder to Eclipse and then change the name back to the original name after installing the CDT. Unzip the CDT file and you should be ready to work with C/C++ projects.

To ensure that the CDT was installed correctly, select Help ‡ About Eclipse SDK. Then click the Feature Details button. You should see CDT 3.0.0 at the top of the list (the other Eclipse features are version 3.1.0).
3.4 Creating a C Project

To create a new C project, click the dropdown icon ( ) beside the New icon, select Project… expand the C entry and select Managed Make C Project.

Click Next and enter the name of the project.
Click Finish and associate the project with the C/C++ perspective.

The perspective has now changed from Java to C/C++.

Note the Include libraries that will be used to provide standard C header files etc.

Click the dropdown beside the New C/C++ source file icon.

Select Source File.
Note that unlike using Java, you must enter the full name of the source file, including its extension (which should be “.c”). Click Finish.

Note that the Console displays some error messages because there is no source code yet.
Enter some source code into the main.c source window. Save the file. There should not be any problems listed in the Problems window and the Console should indicate that the Build was completed successfully.

Note that the Outline view contains information similar to that generated for Java programs.

Click on the dropdown beside the Run icon and select Run As ➤ Run Local C/C++ Application.

Sometimes, when you attempt to run a C project, Eclipse does not recognize the project correctly and Run Local C/C++ Application is not visible.
In this case, you must use the Run menu entry to create a run configuration (see the corresponding section for Java in Chapter 1).

If the C/C++ Application information is not filled in automatically, click the Search Project button and Eclipse will locate the corresponding exe file.

The first time that you run a C application, you may be prompted for information about the debugger: double-click on the generic GDB Debugger entry or the Cygwin GDB Debugger and the project is then executed.
The results are displayed on the system Console.

Occasionally, running a project several times in a row without performing a compile between runs does not generate any output. The reason for this is not known but if you perform a compile (even if the compile is not required), the project should then run correctly.

### 3.4.1 Terminating an Executing Project

If a C (or C++) project does not terminate, this will cause a problem if you attempt to remake the exe file that is still running.

For example, the following program contains an infinite loop and does not terminate.
The visual cue that the program is still active is the red Terminate icon at the top of the Console window. If you forget to terminate the program, make some changes to the source, and then attempt to remake the project, the make does not succeed because Eclipse “cannot open output file C1.exe: Permission denied”.

To fix this problem, you must halt the execution of the program that is still running. You can accomplish this by clicking on the dropdown beside the Display Selected Console icon and selecting the other console.
Then terminate the project by clicking on the Terminate icon.

The project can now be rebuilt and executed successfully.
3.5 Creating a C++ Project

Once you understand how C programs are created, you can create and execute C++ programs in an almost identical manner. Click on the New dropdown and select Project.

Select a Managed Make C++ Project.
Enter the name of the project and click Finish.

Associate the project with the C++ perspective.

Click on the New C++ Class dropdown (note that this is not the same icon as that used to create a C program).
Enter the class name (without an extension). Click Finish.

Note that by default, the source file Cpp1.h is the selected file; since you normally don’t need to modify this file, you can close it so that only the Cpp1.cpp source file remains. The code that is automatically generated results in compile errors.

Replace the generated code with the following code.
When saved, the compile and subsequent build work correctly. Click the dropdown beside the Run icon and run the project as a C++ Application.

If Run Local C/C++ Application is not available, create a run configuration as before.
Again, you may be asked to select a debugger – double-click on either the Cygwin version or the generic version of the GDB Debugger.

The program now runs correctly.
3.6 Creating a Standard Make C Project

With a managed make project, Eclipse does the work of generating the necessary make file that compiles the project files and generates the executable. However, you may want to have more control over the project and for this purpose you can create a standard make project (for either C or C++).

Select Standard Make C Project and name the project C2.
Then create a source file and enter the same statements as for project C1.

When the file is saved, various errors are generated. With managed make projects, the make file is generated automatically. With standard make projects, the make file must be created by the programmer. Create a normal file (not a source file) named makefile.

Enter the following statements. This is a very simple make file; the make program supports much more sophisticated options and commands. (Additional information on the make command is available at http://www.gnu.org/software/make/.) Ensure that each command (such as gcc) is preceded by a Tab character.

C2 : main.o
   gcc -o C2 main.o
main.o : main.c
   gcc -ggdb -c main.c
all :
   $(MAKE) C2
clean :
   -del main.o
Save the make file and return to the source file. When the source file is saved, the source file is updated but several error messages are displayed in the Console window. The reason is that some additional information must be supplied for Standard Make C Projects.

First, right-click on the project name and select Create Make Target.
Create the Target “all”.

Right-click on the project again and select Build Make Target.

Select All and click on Build.
There is still one final step. Right-click on the project and select Properties. Select C/C++ Make Project and click on the Binary Parser tab.

Uncheck Elf Parser and check PE Windows Parser.
At this point, save the main.c file and the executable (C2.exe) should be visible under the Binaries entry. The project can now be run.

You can now run the project using Run As ➔ Run Local C/C++ Application (if that works) or create a run configuration.
Once you have manually performed the tasks that Eclipse performs automatically for managed make projects, the program executes correctly.

When a source file is modified, Eclipse does not perform an automatic build for Standard Make Projects. However, this can be changed by right-clicking on the project name in the Projects view and selecting Properties. Select the Make Project item and select “Build on resource save (Auto Build)” as shown below.
3.7 Creating a Standard Make C++ Project

Creating a Standard Make C++ Project is almost identical to creating a Standard Make C Project. The only difference is in the makefile. To begin, create a Standard Make C++ Project.

Create a C++ class file named Cpp2 and enter the following instructions.
Then create a file named makefile. The following statements should be added to the makefile. **Remember to include a tab before each of the commands (g++ etc.).**

```bash
Cpp2 : Cpp2.o
   g++ -o Cpp2 Cpp2.o
Cpp2.o : Cpp2.cpp
   g++ -ggdb -c Cpp2.cpp
all :
   $(MAKE) Cpp2
clean :
   -del Cpp2.o
```

Go through the same process as for the Standard Make C Project: Create Make Target; Build Make Target; select the PE Windows Parser; and finally create a Run configuration for the project. If desired, select the Build on resource save (Auto Build) check box.
And the project now executes correctly.

Once you have gone through this setup, the project will be rebuilt each time that a source file is saved and the project can be executed by typing Ctrl-F11 or by selecting the appropriate Run configuration from the Run icon.

Creating a standard make C or C++ project requires more effort than creating a managed make project. Unless you require more control over the source files, using a managed make project is preferable.

3.8 Working With Multiple Projects

Eclipse allows the programmer to have any number of files in a project and any number of projects active in a workspace. The following diagram illustrates 3 projects (each of which uses a different compiler) active at the same time.
To continue development of a specific project, expand the project and double-click its source file (or several of them) so that they appear in the editor window.

Notice that when the Java project is opened, the perspective does not automatically switch to Java (is this a bug or by design?) and so some of the information that is normally displayed for a Java project in the Package Explorer is not available. You can switch to the Java perspective by clicking on it in the upper-right corner.
Changing from the C++ perspective to the Java perspective does cause some changes to the workbench: some of the icons are different and the window sizes are slightly different. Also, the libraries visible in the Package Explorer change depending on which perspective is active. (See the two diagrams above for an example.)

Even though one project may be active, another project may still be executed by selecting it from the list of executables under the Run icon.

Thus, the C1 project may be executed even though the Java1 project and the Java perspective are active.

The Java Native Interface (JNI) allows the Java programmer to call programs written in C. In Eclipse, being able to switch back and forth between projects written in different languages facilitates the development of systems that use the JNI. (There is a tutorial on the JNI at http://www.cs.umanitoba.ca/~eclipse/)
3.9 The C/C++ Debugger

The CDT uses the GDB debugger that is included with both Cygwin and MinGW. You should ensure that the debugger is on the system path and is at least version 5.2.1.

D:\>gdb -v
GNU gdb 6.3.50_2004-12-28-cvs (cygwin-special)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-cygwin".

The debugger looks much like the Java debugger discussed earlier.

If you execute the following C equivalent to the Java program used to test the Java debugger, the program runs to completion and displays a result.

```c
#include <stdio.h>

void process1(int[], int);

int main(int argc, char *argv[])
{
    #define ARRAYSIZE 5
    int myArray[ARRAYSIZE] = {10, 20, 30, 40, 50};
    process1(myArray, ARRAYSIZE);
    return 0;
}

void process1(int myArray[], int arraySize)
{
    int counter;
    int sum;
    sum = 0;
    for (counter=1; counter<=arraySize; counter++) {
        sum = sum + myArray[counter];
    }
    printf("%d\n", sum);
}
```

Unfortunately, the result generated (139) is not correct. (As we saw earlier with the Java debugger, the correct answer is 150.) Note that C executes the program to completion and does not recognize the subscript out of range error that Java detected.
To switch to the C/C++ debugger, simply click on the debugger icon. Unlike the Java debugger, the C debugger pauses at the first statement in the program (a breakpoint is automatically added to the first statement). You will notice that a separate Windows console appears when the debugger is used.
You can set a breakpoint at any statement in the program by double-clicking in the margin to the left of the statement. Click on the Resume icon to continue the execution of the program until a breakpoint is encountered. At any breakpoint, you can inspect the current values of variables in the Variables window.

Any information sent to the system console during debugging is displayed in the separate Windows console window (not the Eclipse console window).

Unlike Java, when the program finishes executing, the console window disappears and the information in this window is lost. However, you could add a breakpoint to the return statement and this would cause the program to pause before terminating.
At this time, you can view the contents of the console window.

Assuming that the debugging session has indicated the cause of the problem, you can switch back to the C/C++ perspective and run the program normally.
3.10 Summary

In this chapter, we have presented a brief description of how C and C++ programs can be developed using the Eclipse system. The features of Eclipse that were described in Chapter 1 also apply to working with C/C++ programs and so were not repeated in this chapter. For example, data files can be created and examined using Eclipse, run-time parameters may be passed to a C or C++ program by defining them in the run configuration, etc.

More comprehensive information can be found at the CDT web site: http://www.eclipse.org/cdt/. Included at that site is a 400+ page pdf document on using the CDT (The CDT User Guide) although this document has not yet been updated to version 3.

There is also a collection of tutorials dealing with other aspects of Eclipse at http://www.cs.umanitoba.ca/~eclipse/. Some tutorials related to Extreme Programming such as test-driven development and refactoring in Eclipse are included.
4 TEST-DRIVEN DEVELOPMENT

“Mistakes are the portals of discovery.” James Joyce, *The Dubliners*

4.1 Introduction

Test-driven development (TDD) is one of the practices that is used by the agile development community. In this section, we examine how TDD can be used in Eclipse with the JUnit framework.

4.2 Setting Up JUnit

JUnit is a testing framework that was written by Kent Beck and Erich Gamma. It is included with Eclipse and very little effort is required to setup JUnit and then write test cases.

To begin, create a new Java project.

Then you must add junit.jar to the project's class path. Right click on the project name in the Package pane on the left of the screen and select Properties. Click on Java Build Path. Click Add External Jars.
Navigate to org.junit. (The older version, 3.8.1, was selected but you could also select the newer version.) Double-click on the folder.

Double-click on junit.jar inside the folder.

The jar file is now included on the class path for this project.

Click OK to return to the editor.
4.3 Using JUnit to Develop a Linked List Class

The following example illustrates how TDD can be used to drive the development of a LinkedList class. Create a file named LinkedListTest and enter the following statements.

To Run the test class, simply select Run ➜ Run As and JUnit Test should be the only choice.

Alternatively, select Run ➜ Run… to bring up the Run configuration window. Right-click on JUnit and create a new Run configuration named LinkedListTest. Then click the Run button.
The test case `testConstructor()` executed successfully as indicated by the green bar to the left of the editor window. Congratulations, you have successfully run your first test case.

The JUnit pane to the left of the editor window is quite small. If you double-click on the tab at the top of the JUnit pane, the window is expanded to full size. In this case, there isn't much additional information displayed but as tests become more complex, more information is provided in the window.

Now we add some statements that will test the construction of a `LinkedList` object. Note that Eclipse complains about the use of the class `LinkedList` because it does not yet exist. The statement that performs the test is the `assertNotNull(list)` statement. This statement is one of the JUnit test statements that is used to determine whether or not the code being tested is functioning correctly.
Even though LinkedList does not exist, the test can still be run, although this time, we did not get a nice green bar, we got a red bar that indicated that there were errors in the test. However, this is exactly what we expected since we have not yet written the LinkedList class. Once this file is compiled, the test can be run and once again we see a green bar indicating that all tests passed successfully.
We now add an additional test that ensures that the size of the linked list is correct.

Since a size() method has not yet been written, the test fails.

But it is trivial to add a size method to the linked list class. In addition to adding the size method, a pointer to the first Node is defined and then initialized in the constructor.

The Node class was added to the project without adding any tests for it (not a good idea but the point is not to distract from the task at hand.)
Now our test runs successfully again.

Since the basic structure of the linked list has been defined, we can begin to add the processing methods.

First, a test that adds an element to the linked list is defined. This test is defined in its own method and is independent of the testConstructor method (more on this later).
Eclipse indicates that there is a problem with the add method (it doesn’t exist) and the test fails when it is run.

When the following code is added to the linked list class, the problems go away. Note that this code adds each new object to the beginning of the linked list – that is not what an add method is supposed to do but this version is sufficient for now.

```java
public void add(Object object)
{
    // add an object to linked list
    first = new Node(object, first);
    size++;
}
```

In addition to ensuring that the size of the linked list is 1 after the element is added, we should also retrieve the element from the linked list and compare it with the element that was added. The extra statement at the end of testAdd below performs this processing.

```java
public void testAdd()
{
    LinkedList list;
    String string1 = "abc";
    list = new LinkedList();
    list.add(string1);
    assertEquals(1, list.size());
    assertEquals(string1, list.get(0));
}
```

This statement causes the test to fail because the get method has not yet been implemented.

```java
public Object get(int which)
{
    // get the specified object
    return first.getObject();
}
```

Note that the get method always returns the first element in the linked list. We are growing the code slowly to ensure that we understand what is required and so that the system works at all times (instead of writing the entire linked list class at once and then spending hours attempting to remove the bugs from the code). Once we have the basic structure of the LinkedList class correct, we will return and improve the methods that work correctly only in special cases.

Once the get method has been implemented (at least partially), the tests once again are successful.
We now add another test method that tests the remove method in the LinkedList class.

Since remove has not yet been written, the test fails but adding a simple remove method is easy enough to do (although it is the first object that is removed, not the object specified).

```java
public void remove(int which)
{
// remove an object from linked list
    first = first.getNextNode();
    size--;
}
```

And now the new test also passes.

### 4.4 The setUp Method

At this point, the astute programmer should have noticed that each test includes the same initial statements. The reason for this is that each test method is independent of the other test methods – the effects of processing that is performed in one test method are erased before the next test method is executed. This is good because it prevents one test from
having an impact on another test. However, we need to execute the same statements before we can begin the processing for each test. There are two solutions to this problem.

For relatively simple tests, such as the ones above, the tests could be combined into a single test. This strategy works for small tests but is not good as tests become larger.

The second strategy is to refactor the common processing out of each test into a common method. It just so happens that JUnit provides a mechanism for performing common initialization before each test method is executed — the \texttt{setUp} method. Any instructions that are included in the \texttt{setUp} method are automatically executed prior to the execution of each test method.

The following code illustrates the use of the \texttt{setUp} method. Note that the two variables, \texttt{list} and \texttt{string1}, have been made into instance variables (instead of method variables).

```java
import junit.framework.TestCase;

public class LinkedListTest extends TestCase {
    protected LinkedList list;
    protected static final String string1 = "abc";

    protected void setUp() throws Exception {
        super.setUp();
        list = new LinkedList();
    }

    public void testConstructor() {
        assertNotNull(list);
        assertEquals(0, list.size());
    }

    public void testAdd() {
        list.add(string1);
        assertEquals(1, list.size());
        assertEquals(string1, list.get(0));
    }

    public void testRemove() {
        list.add(string1);
        assertEquals(1, list.size());
        list.remove(0);
        assertEquals(0, list.size());
    }

    protected void tearDown() throws Exception {
    }
}
```
There is also a tearDown method that is executed after each test is complete (regardless of whether or not the test was successful). For our tests, the tearDown method is not required but if the tests involved starting a service (such as a web service or a database connection), then the tearDown method could be used to terminate the service.

The tests run correctly once this refactoring is performed.

4.5 Growing the Code

As we proceeded with the development of the LinkedList class, we cut some corners at times (in the methods get, add, and remove) but overall the structure of the class is reasonably sound.

At this point, we should go back and generalize the get, add, and remove methods (one at a time), first improving the existing test and then improving the application code.

4.6 JUnit Assert Methods

We used the assertNotNull and assertEquals methods in the examples in the earlier sections. The following is a list of other useful JUnit methods.

```
assertTrue(boolean condition)         // assert that the condition is true
assertFalse(boolean condition)        // assert that the condition is false
assertEquals(Object expected, Object actual) // assert that the two objects are equal (.equals)
assertEquals(primitive expected, primitive actual) // assert that the two primitive data types are equal (==)
assertSame(Object expected, Object actual) // assert that the two objects are identical (==)
```

© David Scuse
University of Manitoba
assertNotSame(Object expected, Object actual)  assert that the two objects are not identical
assertNull(Object object)  assert that the parameter is null
assertNotNull(Object object)  assert that the parameter is not null
fail()  cause the test to fail

Notes: Each of the methods listed above may also include an initial String parameter; the value of this parameter is displayed if the test fails. For example:

assertTrue("The initial linked list must be empty", list.size()==0);

Also, the assertEquals method is overloaded and may be used with any compatible data types. Primitive data types are compared using the == operator and objects are compared using .equals().

4.7 Eclipse Test Class Wizard

The test class that was developed in the previous sections was created by hand. However, Eclipse provides a wizard that permits the programmer to create a new test class that contains stubs for the various JUnit methods.

To begin, select New Æ JUnit Test Case.

Enter the name of the class in the Name box and select any stubs (such as setUp) that are to be created.
Click Finish and the class is now available.

4.8 Executing a Test Suite

So far we have defined all tests in one test class. As a general rule, there should be at least one test class for each class in a system. It would be a nuisance if each test class had to be run individually. Fortunately, this is not necessary – test classes can be grouped into a test suite that causes all of the tests to be run at the same time.
Before we take a look at test suites, the LinkedListTest is copied so that there are two test classes (even though they are identical). Each test class can be executed individually but by combining them into a test suite, we can run all test classes together.

To begin, select New ➜ Other and select JUnit Test Suite.

Click Next and the following window is displayed. All test classes are listed in the window. By default, all test classes are selected but the programmer may remove any test classes from the test suite.
The class that is generated (AllTests.java) specifies each test class that is to be executed.

The test suite can be run by selecting Run As ‡ JUnit Test or by creating a Run configuration manually.
The results of running the test suite are shown in the JUnit pane.

4.9 The TDD Process

By this time, the TDD process should be becoming clear. Kent Beck describes the process in *Test-Driven Development*, as **Red, Green, Refactor**. A test is written but the corresponding application code does not yet exist so the test fails (the test bar is red). Then the necessary application code is written and the test bar turns to green. This application code is not necessarily perfect, it merely needs to be good enough for now. Once the bar is green, before writing the next test, refactor the application code to improve it but ensure that the test bar remains green.

It takes some practice to become familiar with TDD. The following guidelines may help develop good TDD habits.

First, it is not necessary to write a test for every piece of application code that is written. Remember that the tests are exercising the interface to the application code, ensuring that what the interface contracts to deliver is actually delivered. Thus, the tests are a living specification of how the application system is supposed to function.

The scope of a test and the corresponding application code can vary significantly depending on how comfortable the programmer is with a particular section of code. It is not necessary to take “baby steps” if the programmer understands the code that is being developed. TDD is normally used with the XP practice of simple design where there is not a huge amount of up-front design that takes place before coding begins. If the design is fairly straight-forward, then the programmer can move quickly developing tests and application code. However, if there are potentially difficult portions, then the programmer should slow down and proceed at a more cautious pace.

To ensure that the code is always in a correct state, do not rewrite other parts of the application code when the test bar is red. When the bar is red, the priority is on making the tests run correctly. If this is not possible, back out of the current test to a point where the bar
is green. Then define a test (or modify existing tests) that focuses on the portion of the application code that is to be rewritten. Ensure that the test is green before proceeding to any other changes.

If the TDD process is followed, the application code is grown in small pieces but you should find that the process moves quickly since there is little debugging required. But the most significant advantage of TDD is that the tests remain after the application code has been written. These tests should be run every time that the system is built. This ensures that no subsequent change has broken any of the application classes. So the tests act as a built-in safety net for the system. Even if you are the world’s best programmer (and many of you are), write the tests as you go so that the rest of the programmers who use your code can run the tests after they make modifications to ensure that your code is never broken.

4.10 Summary

This section has provided a brief introduction to test-driven development. While TDD may seem unnatural at the beginning, if you use it as you are developing code, you will find that you create much cleaner, error-free code than before you used TDD.

For more information on JUnit, check out the JUnit web site www.junit.org.

For testing in C or C++, take a look at CUnit (http://cunit.sourceforge.net/) or CppUnit (http://cppunit.sourceforge.net/cppunit-wiki).
5 WORKING OUTSIDE OF ECLIPSE

5.1 Introduction

When a Java program is compiled, a collection of .class files is generated by Java. The program can be run by executing the class files. A Jar (Java archive) file that contains all of the class files can also be produced (either within Eclipse or outside of Eclipse) and then executed. To execute either the class files or the Jar file, the Java run-time environment (JRE) must be installed. When a C or C++ program is compiled, a native Windows executable is created. This executable can be run outside of the Eclipse environment simply by double-clicking on the .exe file. In this chapter, we examine how Java and C/C++ programs can be compiled and executed outside of the Eclipse environment.

5.2 Compiling and Executing a Java Project Outside of Eclipse

Normally Java programs are compiled and run from within a simple editor such as TextPad or a more powerful IDE such as Eclipse. However, occasionally it is necessary to compile and run a Java program from the command prompt (or in a batch file or script). The javac command is used to compile one or more .java files and create the corresponding .class files.

```
javac *.java
```

The javac command above assumes that the Java SDK is on the system path. If this is not true, you can either add the appropriate directory to the path variable or just include it with the javac command, as shown below.

```
"C:\JDK1.5.0_04\bin\javac" *.java
```

To redirect the output of any command to a file instead of to the console, add `>` filename.txt to the end of the command:

```
javac *.java > filename.txt
```

To append the output of any command to the end of an existing file, add `>>` filename.txt to the end of the command:

```
javac *.java >> filename.txt
```

To execute a Java class file, use the command:

```
java ClassName
```
where ClassName is the name of the class file to be executed – ensure that the case of the name of the class file is an exact match to the parameter typed in the command window. Also, do not include the .class extension with the java command. If the Java SDK or JRE directory is not on the system path, the path to the bin directory must be included with the command.

If command-line parameters are used by the program, just add them after the class name. For example,

```
java ClassName parameter1 parameter2 ...
```

Again, to redirect the output of the command to a file instead of to the console, add the following to the end of the command:

```
java ClassName > filename.txt
```

Java does not create native executables (although this can be accomplished using a system described later in this chapter) but it does permit class files to be grouped together into a Java Archive (JAR) file that can be executed in a manner that is similar to a conventional exe file. A Jar file contains a set of class files compressed into one file. To create a Jar file, use the following command to bundle all class files into the Jar file:

```
jar cfm Java1.jar manifest.txt *.class
```

or

```
"C:\JDK1.5.0_04\bin\jar" cfm Java1.jar manifest.txt *.class
```

The manifest file is a text file that identifies the class that contains the main method that is to be executed.

```
Main-Class: Java1
```

The manifest file **must** contain a blank line after the last line of text.

(You can also create a Jar file from within Eclipse.)

Once the Jar file has been created, it can be executed either by double-clicking on it (if this is supported by your operating system) or by executing one of the following commands:

```
java -jar Java1.jar
```

```
"C:\JDK1.5.0_04\bin\java" -jar Java1.jar
```
5.2.1 **Blocking the Execution of a Java Program**

When a Java program is run from a class file or a Jar file, the console window remains visible only while the program is active. If you need to examine the information in the console, you can block the program by attempting to read information from the console. The following modification to the basic Java program causes the program to pause until the user types the enter key.

```java
import java.io.IOException;

public class Java1 {
    public static void main(String[] args) {
        System.out.println("Hello Java World!!");
        try {
            System.in.read();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Alternatively, you could run the class or Jar file from within a Windows command window. Or, you could create a Windows .bat file that executes the class or Jar file. On the next line, add the Windows “pause” statement. This statement in a batch file causes execution to be suspended until the user types the any key. For example, the following commands could be defined in a Windows batch file.

```bash
java -jar Java1.jar
pause
```

The batch file is executed by double-clicking on it.
5.3 Executing a C/C++ Project Outside of Eclipse

The .exe file that is created by the build process is a normal Windows exe file and may be executed outside of Eclipse. For example, in the C2 project, the project directory contains the following files.

One thing to be aware of when using Cygwin is that the executable (.exe file) built by Cygwin requires the dll cygwin1.dll. As long as Cygwin\bin is on the system path, this is not a problem but if the executable is to be run on other machines, then the dll must be made available.

5.3.1 Blocking the Execution of a C/C++ Program

As with Java programs that are executed outside of Eclipse, any results written to the system output file window remain visible only while the program is active. For a C program, the following instructions can be used to cause the program to pause before it terminates so that any output directed to the console can be examined. The statement fflush(stdout) ensures that all output written to stdout has actually been written to the console. (This statement is not always required, it depends on the type of system console that is being used. However, it will not cause any problems if it is included.) The fgetc(stdin) statement then reads one character from the console.
5.4 Compiling a C/C++ Project Outside of Eclipse

In addition to being able to execute a C/C++ project outside of Eclipse, you can also modify the source files and rebuild the project outside of Eclipse. Source files may be edited using any basic text editor.

To recompile a C project that consists of the single source file main.c, bring up a Windows command window and type:

```
gcc -o C2 main.c
```

This command compiles the source file and creates the executable file C2.exe.
5.4 Compiling a C/C++ Project Outside of Eclipse

To compile the C++ source file Cpp2.cpp, bring up a Windows command window and type:

```
g++ -o Cpp2 Cpp2.cpp
```

This command compiles the source file and creates the executable file Cpp2.exe.

If there are multiple files that have to be recompiled, it is easier to group the commands into a script and execute the script. If a standard makefile script has been created, to remake the project, bring up a Windows command window and type:

```
make -k all
```

This should result in the same build messages as were generated within Eclipse.

You can also remake a managed make project from outside of Eclipse. Find the corresponding makefile file and then issue the make command shown above.
5.5 Creating an exe for a Java Project

A special version of the MinGW tools can be used to create native Windows applications from Java source files. These native applications are quite large (usually starting at 4 MB or so) because they include a compiled version of the JRE. However, the ability to create native executables can be very useful.

The version of the MinGW tools used to create Windows native Java executables is available at http://www.thisiscool.com/gcc_mingw.htm. If you are using Eclipse 3.1, ensure that you download the version GCC/GCJ Version 4.0. This system requires approximately 202 MB of storage. While this system is a version of MinGW, it does not contain either make.exe or the gdb.exe debugger. If you are developing C/C++ programs as described in the previous chapter, you should not add this system to your system path since it may interfere with the tools that Eclipse requires.

Test the installation by typing the following command into a Windows command window:

Once the system is installed, you can compile existing Java programs using the appropriate Windows batch (.bat) scripts or makefile scripts. These scripts are run outside of the Eclipse environment.

Our first example compiles a simple hello world Java program into a windows executable.

```
PATH D:\gcc-4.0\bin;D:\gcc-4.0\i686-pc-mingw32\lib;D:\gcc-4.0\i686-pc-mingw32\bin;%PATH%
call gcj-lib orig
@echo on
set MAIN=HelloWorld
gcj --main=%main% -Wall src\%main%.java -o %main%
```

When this script is run, the following output is generated.
5.5 Creating an exe for a Java Project

If the executable is run, the following output is generated.

If the program generates output on the system console, the program can be blocked as described earlier so that the console window is not closed as soon as the program completes executing.

5.5.1 Building SWT GUI’s

If a Java program uses the SWT GUI widgets, then the build script is slightly more complicated. A different library must be included; this library is created by the jcj-lib gui command. In addition, the swt dll that is included with the MinGW system must be included on the system path. This dll (currently named swt-win32-3116.dll) is not necessarily the same as the swt dll that is distributed with the current version of Eclipse. The MinGW swt dll is normally a few releases behind the Eclipse swt dll.

The following script creates an exe for the program SWTHello.java. After the build is complete, the SWTHello.exe file can be executed by double-clicking it.
5.5 Creating an exe for a Java Project

If a project contains multiple java files, then the script is more complicated, as illustrated by the script below. The first set of compile instructions generates the .class files; the second set of compiles generates the .o files; and the final compile instruction links the .o files into the executable file.

```batch
PATH D:\gcc-4.0\bin;D:\gcc-4.0\i686-pc-mingw32\lib;D:\gcc-4.0\i686-pc-mingw32\bin;%PATH%
set SWT_BASE_DIR=D:\gcc-4.0\i686-pc-mingw32
set SWT_GCJ_LIB_DIR=%SWT_BASE_DIR%\lib

call gcj-lib gui
@echo on

gcj --main=%main% -Wall src\%main%.java -o %main%

gcj -C -fclasspath=bin;src; -d bin src\ButtonDemo.java

gcj -C -fclasspath=bin;src; -d bin src\ComboDemo.java

gcj -C -fclasspath=bin;src; -d bin src\CompositeDemo.java

gcj -C -fclasspath=bin;src; -d bin src\GroupDemo.java

gcj -C -fclasspath=bin;src; -d bin src\LabelDemo.java

gcj -C -fclasspath=bin;src; -d bin src\ListDemo.java

gcj -C -fclasspath=bin;src; -d bin src\TextDemo.java

gcj -C -fclasspath=bin;src; -d bin src\BasicWidgetsDemo.java

gcj --classpath=obj;bin; -c src\ButtonDemo.java -o obj\ButtonDemo.o

gcj --classpath=obj;bin; -c src\ComboDemo.java -o obj\ComboDemo.o

gcj --classpath=obj;bin; -c src\CompositeDemo.java -o obj\CompositeDemo.o

gcj --classpath=obj;bin; -c src\GroupDemo.java -o obj\GroupDemo.o

gcj --classpath=obj;bin; -c src\LabelDemo.java -o obj\LabelDemo.o

gcj --classpath=obj;bin; -c src\ListDemo.java -o obj\ListDemo.o

gcj --classpath=obj;bin; -c src\TextDemo.java -o obj\TextDemo.o

gcj --classpath=obj;bin; -c src\BasicWidgetsDemo.java -o obj\BasicWidgetsDemo.o

set MAIN=BasicWidgetsDemo

gcj --main=%MAIN% -s -o %MAIN%.exe obj\*.o
```

5.5.2 Building Multiple Files

If a project contains multiple java files, then the script is more complicated, as illustrated by the script below. The first set of compile instructions generates the .class files; the second set of compiles generates the .o files; and the final compile instruction links the .o files into the executable file.
All of the scripts shown in this section were defined as Windows .bat scripts; however, a corresponding makefile script could also have been used.

5.6 Summary

In this chapter, we have examined the techniques that can be used to compile and execute Java and C/C++ programs outside of the Eclipse environment. Normally it is best to develop projects within Eclipse but these techniques can be used if Eclipse is not available.
6 MISCELLANEOUS FEATURES

6.1 Introduction

The following are descriptions of some miscellaneous features that were not included in the body of these notes.

6.2 Making Class Files Available

Adding a .class file (that does not have an associated .java file) to an Eclipse project does not make the .class file available to Eclipse. For example, in the project below, the file DString.class is available in the project folder but Eclipse does not use the file.

To make the .class file available, create a new folder within the project folder. For example, create a folder named bin2. Place the .class file inside this folder. (Do not place the file inside the bin folder if you keep the source and class files separate – this doesn’t work.

Refresh the project so that the bin2 folder is now visible.
Select Project ➤ Properties and then select Java Build Path.

Click the Add Class Folder button and check the bin2 folder.
You may get the following message: click OK.

![Class Folder Added]

The bin2 folder is now on the build path.

![Properties for Test]

When control returns to the editor window, the missing class file has been found and the error has been fixed.

![Java Build Path](image)
6.3 Eclipse Scrapbook

The Eclipse Scrapbook is a window onto which code may be dropped and then executed.

To create a scrapbook page, select File → New → Other to bring up a list of wizards. Then select Java → Java Run/Debug → Scrapbook page and then click the Next button.

Enter the name of the scrapbook page and then click Finish.
Type or copy and paste some code into the window.

Select the text and then click the Evaluate (△) button

The results of evaluating the code are displayed in the console window.

At this point, the code is still running so the red terminate icon should be clicked to stop the process.
Type an expression and then type the name of the variable that the expression is assigned to; click the Display Result icon.

The result is displayed in the window. Remember to terminate the process.

The scrapbook is not the most intuitive of features but it can be useful when you want to test a small segment of code.

Examine the Eclipse help for more information on the scrapbook feature.