

***Introduction
to
Java
Programming***

EEET C6040

Melbourne Co-Op Books

INTRODUCTION



1 99 9770000135309

6040/6040B/6042/6050

This is a companion text for TAFE students studying the following programs:

- Advanced Diploma of Electronics Engineering C6040**
 - Advanced Diploma of Electronics Engineering C6040B (articulation)**
 - Advanced Diploma of Computer Systems Engineering C6042**
 - Advanced Diploma of Computer Systems Engineering C6042B (articulation)**
 - Advanced Diploma of Electrical Engineering C6050**
 - Advanced Diploma of Electrical Engineering C6050A (articulation)**
-
-

These notes are to be used in conjunction with a series of lectures delivered in the classroom. They are available on the network as:

X:\tedbown\java\student text.pdf

The recommended text is:

Deitel & Deitel, Java:How to Program 5th Ed. ISBN 0-13-120236-7

However, any book that introduces Java and covers Java2 would be acceptable.

The software used to develop the Java programs in the classroom is Borland's JBuilder3. A copy of the software is made available to students for home use. This is proprietary software and is made available under the appropriate restricted use copyright provisions. That is, the software can be used for private use. It cannot be used to create commercial applications.

Some Useful Web Sites:

[Sun Microsystems Java Home Page](http://math.hws.edu/javanotes/index.htm)

[math.hws.edu/javanotes/index.htm](http://www.adeveloper.com/resource.htm)

www.adeveloper.com/resource.htm

www.javacoffeebreak.com

www.ggrweb.com/geojava

devcentral.iftech.com/articles/Java/default.php

www.taxon.demon.nl/JW/jwtut.htm

[Sun's Java Tutorials](http://www.bruceeckel.com)

www.bruceeckel.com

www.freewarejava.com

www.javaside.com

www.brainjar.com

www.xanasoft.com

javaboutique.internet.com

First Edition, September, 2003

This document was written by Ted Bown.

E-mail: tedbown@rmit.edu.au

Faculty of Engineering, TCTCE
Royal Melbourne Institute of Technology,
Building 56, City Campus,
Queensbury Street,
Carlton, Victoria.

P.O. Box 2476V
Melbourne
Victoria. 3001

Telephone: +61 3 9925 4727

Facsimile: +61 3 9663 7974

This work is copyright. Apart from any fair dealing for the purposes of study, research, criticism or review, as permitted under the Copyright Act, no part may be reproduced by any process without written permission of the author. Enquiries should be made to the Training Centre for Telecommunications, Computing and Electrotechnology, Faculty of Engineering, RMIT University.

Table of Contents

<i>Before You Start!</i>	1
<i>Lesson 1: Introduction to Java and JBuilder</i>	2
Opened List CAUTION:	5
Documentation Techniques	7
DOS Console	8
<i>Lesson 2: Introduction to Applets</i>	10
<i>Lesson 3: Introducing The Mouse</i>	17
<i>Lesson 4: Introducing Widgets</i>	22
<i>Lesson 5: Language Flow Control</i>	28
Primitive Data Types	31
Program Constructs	32
The 'if' Statement	32
The 'while' Loop	34
The 'for' Loop	36
<i>Lesson 6: Classes, Methods, Variables and Scope</i>	37
Classes	37
Instantiation	37
Interfaces	37
Abstract	37
Methods	38
Variables	40
<i>Lesson 7: Arrays</i>	47
<i>Lesson 8: Modularity</i>	49
<i>Lesson 9: Testing and Debugging</i>	54
Syntax Errors	54
Run-Time Errors	54
Logical Errors	54
Error Detection Techniques	55
<i>Java Programming Exercises</i>	57

Before You Start!

You will soon learn that the Borland JBuilder program creates a lot of files with the same name but different file extensions. If you cannot see these file extensions you will not know which file to open (see **Figure 2** below). The showing/hiding of the file extensions is a parameter set up in the Windows operating system. You need to set this before launching JBuilder and leave Explorer open to keep these settings. The file extensions will now appear in the JBuilder dialog boxes.

Start Windows Explorer and select **VIEW | OPTIONS** from the top menu bar. This will bring up a screen like **Figure 1**. Click on the checkbox next to “**Hide file extensions for known file types**” and remove the tick to allow the extensions to become visible (**Figure 3**). Now, when JBuilder starts, it will show the extensions (**Figure 4**) and make your programming experience less traumatic.

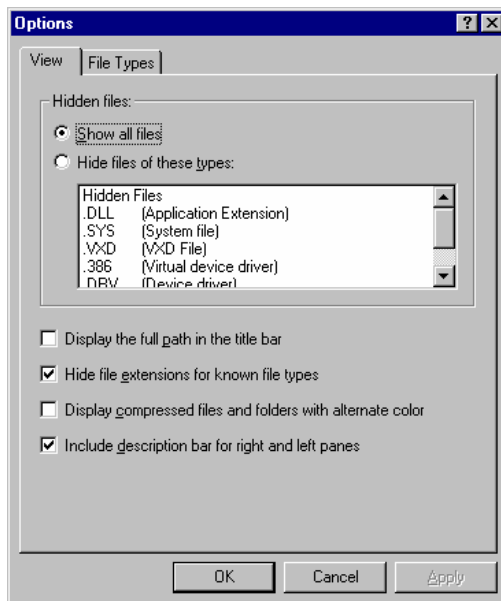


Figure 1. File Extensions Hidden

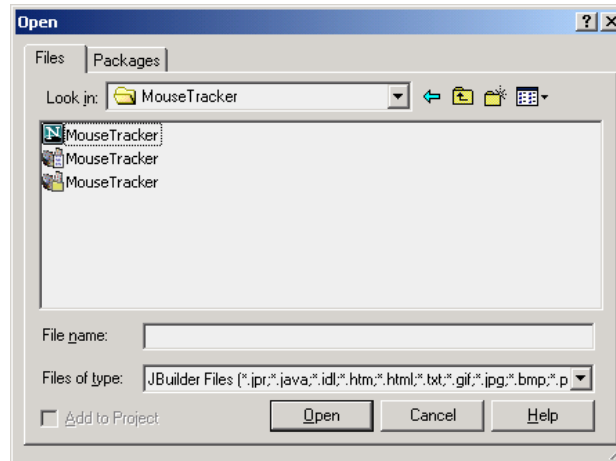


Figure 2. File Extensions Hidden

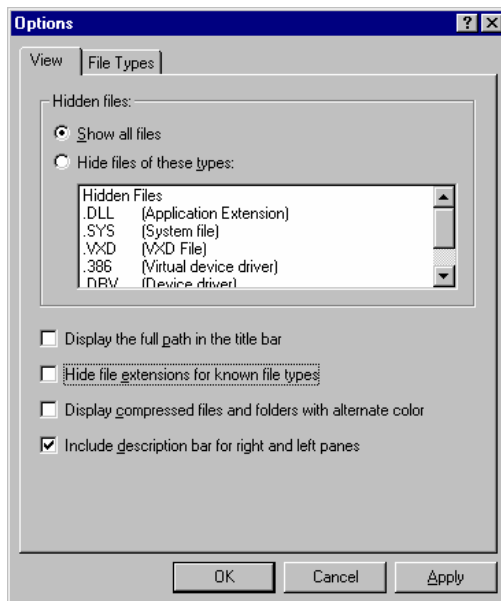


Figure 3. File Extensions Visible

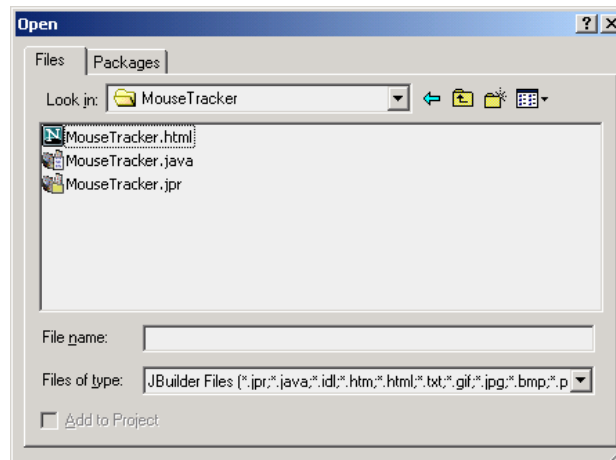


Figure 4. File Extensions Visible

Lesson 1: Introduction to Java and JBuilder

This first lesson is used to introduce you to Java programming and JBuilder3, the IDE (Integrated Development Environment) used in the classroom. The first program you will write is the ubiquitous *Hello World* program. It has become something of a standing joke that the first thing any programmer does on a new computer system or with a new programming language is to write the words “Hello world” on the screen. This seemingly innocent act has a very serious purpose. It shows that a) the system works, and b) that you know how to write and produce a program on the system. The following three lines of java source code make up the entire program. NB: ‘Source Code’ is the name given to the ASCII version (English, readable version) of the program code.

My first Java program:

1. Launch Borland’s JBuilder3 program on your PC.

2. At home you will set the default output directories. This is where the editor will automatically save your work:

PROJECT | DEFAULT PROPERTIES | PATHS

Set the source and output directories to wherever you intend to store your work. If you attempt to set the default properties at school you will get an error message as you are not allowed to write to the C:\ drive where these settings are stored !!

3. Create a new project using the built-in wizard:

FILE | NEW PROJECT

Give it a file name of *Hello.jpr* JBuilder stores a set of related files grouped together as a project.

*****Do not put spaces in file or directory names!!***

Java doesn’t allow it and JBuilder will lose the files.**

Title – The Hello Project

Author – type in your name

Company – enter your student number

Description – enter “My first Java Application”

FINISH

The Content Pane (shown on **Figure 1.1** below) will show these details, which are stored as an **html** file. Note the automatically generated **.html** file in the Navigation Pane (the panel to the left), with the same name as the project. This is where JBuilder has stored those details you put into the previous Wizard screen.

4. In the Navigation Pane (see **Figure 1.1**), right-click on your project’s name. From the drop-down menu select:

PROPERTIES

and the Paths tab. Make sure that the Source and Output are set to H:\ as this is where you want your work stored. This option can also be selected from the top through the

PROJECT | PROPERTIES | PATHS

option. You will learn that JBuilder will always give you several ways to achieve any operation.

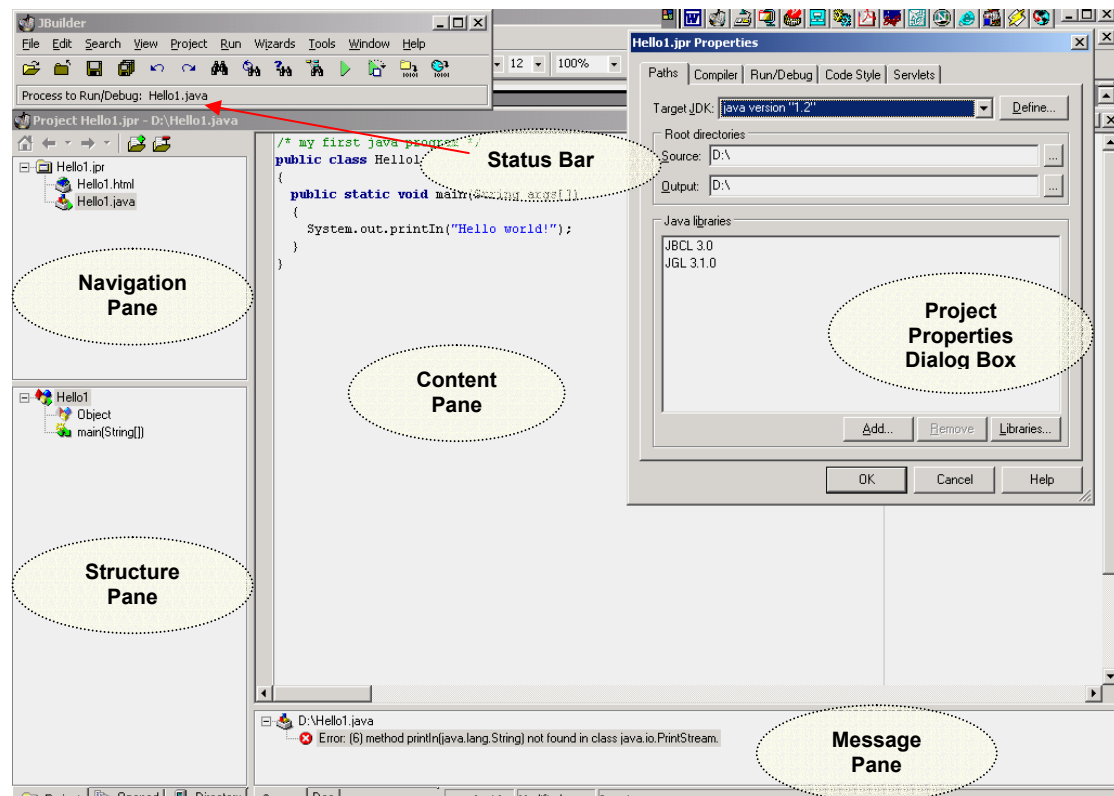


Figure 1.1 Layout of JBuilder Editor Screens

5. In the Navigation Pane just above your project's name is the 'Add To Project' icon. It has a green + in it. Select:

ADD TO PROJECT

In the *Open* dialog box type in the file name Hello1.java and select **OPEN**. This new file name will appear in the Navigation Pane.

6. In the Content Pane add the following text to your new *.java* file, paying close attention to the layout, type of brackets used, spelling and punctuation. Java is case sensitive. *hello* is different to *Hello* is different to *HELLO*. When you get to **System.out.** and type the full stop, JBuilder's Code Help system automatically starts. Here the editor will show you the legal options available. You may select the one you want, or ignore the offered help and continue typing:

```
public class Hello
{
    public static void main(String[ ] args)
    {
        System.out.println("Hello World!");
    }
}
```

7. When you have finished entering the program, select:

FILE | SAVE ALL

to preserve your work. You will see confirmation of the save in the window's status bar at the top of the screen below the toolbar icons. Ensure that the file name you created in paragraph 5 and the class name in the code are **exactly** the same, otherwise Java will complain and your program will not run.

8. Compile your program, that is, create a binary version of your program. The binary version will have the same filename and a **.class** file extension. Move the mouse pointer over the filename in the Navigation Pane, right click and select **MAKE**. If you have not made any typing errors you will see a *Compiler:..No errors* message in the status bar. Any errors will be listed in the Message Pane window below the Content Pane. A new file called Hello1.class will be created by the compiler. This is called a *bytecode* file as it is in binary format. You can use Windows Explorer to look in your H:\ drive to confirm the creation.

9. Your project is now ready to run. However, the programs you create here are not self-contained **.exe** executable applications. In the paragraph above I explained that the output of the *make* command was a **.class** file. The operating system doesn't know what to do with this type of file. It has to be sent to a special Java launching platform called the Java Runtime Environment (JRE). Right click the file name again, select **RUN**. This will launch the JRE. Your program will write information to a black DOS shell screen (called the Console Window).

10. On completion, the editor closes this window and regains control. This means that you will only see a fleeting glance of your program's output. To prevent this from happening, you have two choices. The first and easiest is to select:

RUN | PARAMETERS... | RUN/DEBUG or
PROJECT PROPERTIES... | RUN/DEBUG

Select *Console I/O...Console Window*. Remove the tick from *Close console window on exit*. This will keep the console window open so that you can see the program's output.

The second alternative is to write the program's output to the Execution Log, a log, or record, of the program's output. Once again, go to the run/debug window, then select *Console I/O...Execution Log*. Run your program again. It will look as if nothing happened. Select:

VIEW | EXECUTION LOG

to open the Execution Log window. This time the output will be written into the log window where you have a permanent copy of your program's output.

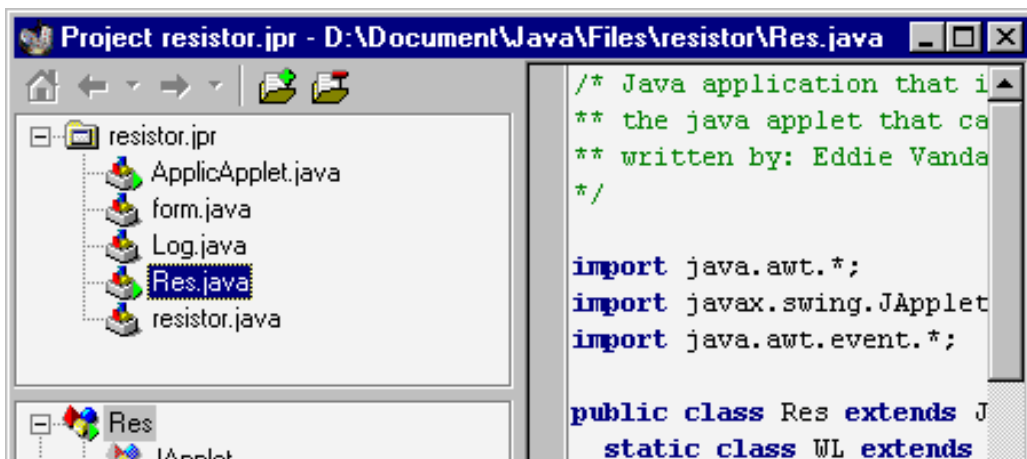


Figure 1.2 Navigation Pane Normal View

Opened List CAUTION:

The Navigation Pane should always show the full set of files belonging to your project. When you use a single click of the mouse to select a file, the contents should be displayed in the Contents Pane to the right as shown in **Figure 1.2**.

If you double-click the file name to select it, the editor will copy your file from the project and open it as a separate file in what it calls the 'opened list' (see **Figure 1.3**). In this mode any changes you make to your source code will no longer be seen in your project. To return to the normal mode of editing, select the Project tab at the bottom of this window.

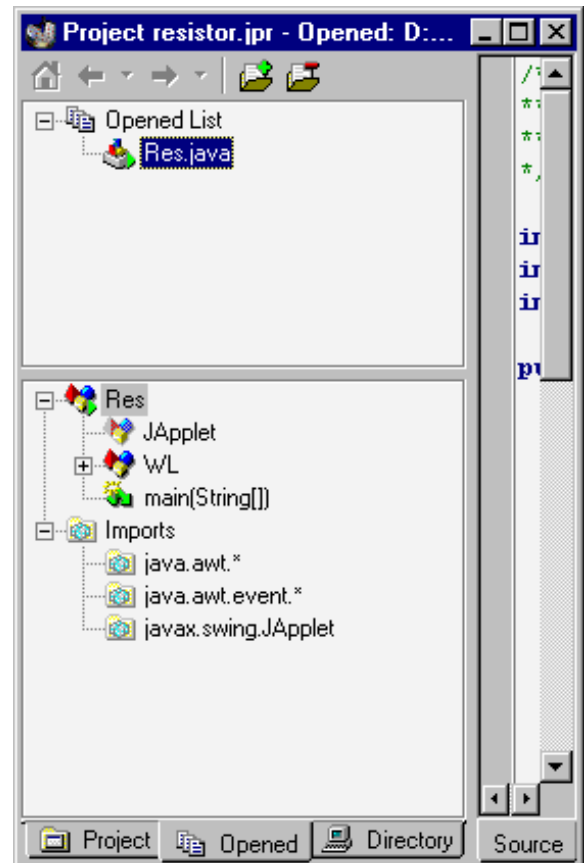


Figure 1.3 Navigation Pane in Opened List Mode

My second Java program:

Create a new project following the previous example. This time call it *Hello2.jpr*. Enter the following program and see what changes there are to the output:

```
public class Hello2
{
    public static void main(String[ ] args )
    {
        System.out.print("\nWelcome\nto");
        System.out.println("\nJava\nprogramming!");
    }
}
```

Here you are using two new concepts.

The first is the use of two versions of the ‘print’ method. The first will display information in the console window (see page 8 for an explanation of the console), but it will leave the cursor at the end of the data. Just in case you want to add more data before moving the cursor to the next line. The second version, ‘println’, which you can think of as “print my data then go to a new line”, will display your data then move the cursor to the beginning of the next line. This is often referred to as a Carriage Return Line Feed, CR/LF.

The second new concept is the use of ‘\n’. This is called an *escape* character. Instead of being displayed on the screen, it is interpreted by *print* and *println* as a special command, to perform a CR/LF. If you refer to the text you will find a list of these escape characters. The effect of both of these new concepts is to put each of the four words on a separate line.

My third Java program:

Create a new project following the previous two examples. You should be getting the idea by now. This time call your project *Hello3.jpr*. Again enter the following program and see what changes there are this time. Note what happens in the Structure Pane (the panel bottom left) as you type:

```
/* Hello3.java
   My first program with windows
*/

import javax.swing.JOptionPane;    // import class JOptionPane
                                   // and be careful with the spelling

public class Hello3
{
    public static void main(String[ ] args)
    {
        JOptionPane.showMessageDialog(null, “Welcome\nto\nJava\nprogramming!”);
        System.exit( 0 );    // politely tells Windows the program has terminated
    }
}
```

The *import* command tells the system that you want to use one of the pre-written class files that is stored in the **javax.swing** library package. This particular file will put Windows style dialog boxes on the screen for you. In one line of code you have become a Windows programmer. The **System.exit(0);** command tells the operating system that your program has finished, and it can have the screen back again that was occupied by your dialog box.

Documentation Techniques

Note the layout of the source code in this program. There is liberal use of space, and indenting to show the different sections of the program. These points may seem trivial here but when you try to read a program that contains thousands of lines of source code you will appreciate the need for a disciplined approach to the layout of code. Often the purpose of a section or even a line of code is not readily apparent. It requires an explanation. This is achieved by the use of comments in the code. The comments are ignored by the compiler as they are there purely for the benefit of a human reading the source code.

To make sure that the compiler does ignore comments, they are enclosed in special brackets, ‘*/**’ and ‘**/*’. This may enclose comments on one line or it may go over many lines. This is a handy way to temporarily remove blocks of code that don’t work when you are developing a project.

The second style of comment is called the *inline* comment. By using a double slash, ‘*//*’, you can insert a one-line comment. The comment automatically terminates at the end of the line.

My fourth Java program:

This time, we will create a program with some user interaction. Ask the user to enter two numbers and we will show the sum. Call this project *Adding.jpr*

```
/* Adding.java
** My first program with user interaction
*/
import javax.swing.JOptionPane;           // import class JOptionPane

public class Adding
{
    public static void main(String[ ] args)
    {
        String    firstNumber,           // Create some places in memory
                 secondNumber;         // where the data can be stored
        int       number1,
                 number2,
                 sum;

        firstNumber = JOptionPane.showInputDialog( "Enter first integer" );
        // read in the first number as a string

        secondNumber = JOptionPane.showInputDialog( "Enter second integer" );
        // read in the second number as a string
```

```
number1 = Integer.parseInt(firstNumber);
number2 = Integer.parseInt(secondNumber);
    // convert the numbers to integers from text strings

sum = number1 + number2;
    // Add the two numbers together

JOptionPane.showMessageDialog (null, "The sum is " + sum,
                                "Results", JOptionPane.PLAIN_MESSAGE);

System.exit( 0 );           // terminates the program
}
}
/***** End of Listing *****/
```

Compile and run the application. Don't forget to save it first. Your program should successfully show the sum of two numbers. Note what happens when you enter letters instead of numbers. We will trap illegal inputs in a future lesson.

You will notice that the input is taken into the program as **String** values. A string is simply a series of characters typed in through the keyboard. Java has been designed so that this is the only way that data can be added to any program. Obviously, we can't do maths on a string of characters, so these must be converted to numbers (if that is appropriate) for calculations to be performed. The *Integer.parseInt(String)* expression does this for us. If the string typed in is in the form of a number (without a decimal point) this string will be converted and stored as an **integer**. We can now perform mathematical operations on the data. If you wanted to manipulate floating point numbers these would be referred to as **float** or **double**.

DOS Console

Java was originally designed to run from the console, or black DOS, window: that is, the command line environment with no GUI components. Programs were to be edited in a basic text editor, like Notepad, then compiled and run with command line input commands. You are getting it easy with JBuilder. Single click compile and run, inbuilt code help (that's when JBuilder helps you with screen lists of available options) colour-coded editing are all designed to speed your programming development. The previous program runs in the windows environment. The following program is the same, except that it has been modified to run in the console window instead (modifications in orange).

```
/* Adding.java
** My first program with user interaction from the console window
*/
import java.io.*;

public class Adding
{
    public static void main(String[ ] args) throws IOException
    {
        BufferedReader stdin = new BufferedReader
            (new InputStreamReader(System.in));
```

```
String  firstNumber,          // Create some places in memory
        secondNumber;        // where the data can be stored
int     number1,
        number2,
        sum;

System.out.print("Enter first integer: ");
firstNumber = stdin.readLine();
    // read in the first number as a string

System.out.print("\nEnter second integer: ");
secondNumber = stdin.readLine( );
    // read in the second number as a string

number1 = Integer.parseInt(firstNumber);
number2 = Integer.parseInt(secondNumber);
    // convert the numbers to integers from text strings

sum = number1 + number2;
    // Add the two numbers together

System.out.println("\n\nThe sum is " + sum);
}
}
/***** End of Listing *****/
```

The `BufferedReader` is a temporary storage area for the input string of characters that have come from the `InputStreamReader` or keyboard.

For a review of this lesson, read Chapter 2 of your text and attempt the self-review exercises. Your homework is to install JBuilder at home and repeat the exercises from this lesson on your home PC. Running *Hello1* at home proves that the software is installed correctly. Remember that you will have to do work at home if you are going to receive any benefit from the classroom sessions.

Lesson 2: Introduction to Applets

The first lesson showed you how to create applications. An application is a stand-alone program that is launched with the JRE. This lesson covers small Java programs that are designed to run inside web pages, called **applets**. Applets are not stand-alone programs, they must be started from within a web page coded in *html*, the language web browsers use to display information on your screen. Because we are using a browser to display our applet, we will have to write an *.html* file. Therefore, launching applets requires an extra step.

Package	Description
java.applet	Classes for developing applets
java.awt	Abstract Windowing Toolkit (AWT) classes for the GUI , such as windows, dialog boxes, buttons, text fields, and more.
javax.swing	Widgets (windows gadgets) used in the creation of windows programs - an updated version of the previous package.
java.net	Classes for networking, URLs, client/server, sockets.
java.io	Classes for various types of input and output.
java.lang	Classes for various data types, running processes, strings, threads, and more.
java.util	Utility classes for dates, vectors, and more.
java.awt.image	Classes for manipulating and managing images.

Table 2.1 Some Java Packages

Packages are pre-written libraries that you can use in the programs you write. You gain access to the libraries by having an “import” statement at the top of your code. These libraries are broken down into sub-libraries that are accessed using full stops to indicate the sub-library name. For example, the last program last lesson included the statement:

```
import javax.swing.JOptionPane;
```

This means that my program has access to the set of methods written to create **option panes**, or windows dialog boxes. The code for these option panes is stored in the **swing** section of the **javax** package.

My first Java applet:

Applets are slightly more complicated to set up than applications. They require a driver program written in html to initiate them. Use the following instructions to set up the applet and its driver.

1. Launch JBuilder3.
2. As you did last lesson, create a new project. Name this one HelloApplet.jpr
****Remember, do not put spaces in directory or file names!!**
****This is a Java restriction.**

3. You may find it convenient to keep each week's work separate to make it easier to organize your H:\ drive. I suggest you save this week's work in a subdirectory called WK-3. You can tell the editor to put your work there by right-clicking on the project's name, select Properties and the Path tab. Set the source and output directories to H:\WK-3.

4. Select **ADD TO PROJECT**. Create a file called HelloApplet.java In the Contents Pane, enter the following code:

```
// HelloApplet.java
import java.applet.*; // the applet classes
import java.awt.*;    // Abstract Windowing Toolkit classes (GUI components)

public class HelloApplet extends Applet
{
    public void paint(Graphics g)
    {
        g.drawString("Hello World!", 20, 50);
    }
}
```

Compiling this code will create a new file called HelloApplet.class.

5. Now create the driver. Select **ADD TO PROJECT**. Create a new file called Driver.html At the bottom of the Contents Pane select *Source*. This will allow you to enter the html source code. Click in the Contents Pane, then enter the following code:

```
<html>
<head>
  <title>Applet Driver File</title>
</head>

<applet
  code="HelloApplet.class"
  width=150
  height=50>
</applet>
</html>
```

Note: Do not forget to close the *applet* tag with > after the height attribute. Save your work.

Hint: You will be able to use this driver again by simply changing the class name inside the code.

6. To view your applet, you will have to launch Netscape or Internet Explorer and use **FILE | OPEN PAGE** to start the driver file, or more easily, in Windows Explorer, double click on the driver file to auto-launch it. Either of these two methods show how your applet appears in a web page. Not all versions of the two popular browsers will start applets. There were versions that were not applet-aware between V3 and V6. Versions from V6.0 onwards for both programs should be OK. To be sure your applet works, use the system in the following paragraph.

7. There is an easier way to see your applet. Sun Microsystems created a program called *appletviewer.exe* that was written specially to view applets. Sun's viewer shows just the applet, not the entire web page. To launch the appletviewer, right-click on the driver file, and select **RUN** or select the **RUN** icon from the top toolbar.

My second Java applet:

Create a new project the same way as you did for the previous example. This time call it *WelcomeApplet.jpr* Enter the following program, saving it as *WelcomeApplet.java*

```
// WelcomeApplet.java
// Displaying multiple strings
import javax.swing.JApplet; // import class JApplet
import java.awt.Graphics;    // import class Graphics

public class WelcomeApplet extends JApplet
{
    public void paint(Graphics g)
    {
        g.drawString("Welcome to", 25, 25);
        g.drawString("Java programming!", 25, 40);
    }
}
```

Modify your driver file to call this new class and save it as *Driver2.html* or something similar. Change the co-ordinates to a width of 300 and a height of 45 and follow the previous example to view the output.

My third Java applet:

This time, modify the previous applet by putting lines above and below the text. Create a new project called *WelcomeApplet2.jpr* Enter the following program, saving it as *WelcomeApplet2.java*

```
// WelcomeApplet2
// Displaying text and lines

import javax.swing.JApplet; // import class JApplet
import java.awt.Graphics;    // import class Graphics

public class WelcomeApplet2 extends JApplet
{
    public void paint(Graphics g)
    {
        g.drawLine(15, 10, 210, 10);
        g.drawLine(15, 30, 210, 30);
        g.drawString("Welcome to Java programming!", 25, 25);
    }
}
/***** End of Listing *****/
```

Again modify your driver file to call this new class and change the co-ordinates to a width of 300 and a height of 40.

My forth Java applet:

This time, we will modify the application we created last week that added two numbers together. We will make two changes (as well as the major modification of turning it into an applet). First change – the numbers will now be floating point (ie. containing a decimal point and a fraction component). Second change – multiply the numbers together and display the product. Start a new project called *MultiplyApplet.jpr* and enter the following code as *MultiplyApplet.java*

```
// MultiplyApplet.java
// multiply two floating point numbers

import javax.swing.*;      // import package javax.swing
import java.awt.Graphics;  // import class Graphics
public class MultiplyApplet extends JApplet
{
    double Product;        // memory location to store the result

    public void init ()
    {
        String  firstNumber,    // first string entered by user
                secondNumber;   // second string entered by user
        double  number1,        // first number to multiply
                number2;        // second number to multiply

        firstNumber = JOptionPane.showInputDialog(
                                "Enter first floating-point value");
        // read in the first number from the user as a string

        secondNumber = JOptionPane.showInputDialog(
                                "Enter second floating-point value" );
        // read in the second number

        number1 = Double.parseDouble(firstNumber);
        number2 = Double.parseDouble(secondNumber);
        // convert the numbers from type String to type double

        Product = number1 * number2;
        // multiply the two numbers together
    }

    public void paint(Graphics g)
    {
        g.drawRect(15, 10, 270, 20);
        g.drawString("The product is " + Product, 25, 25);
    }
}
/***** End of Listing *****/
```

Again modify your driver file to call this new class and change the co-ordinates to a width of 300 and a height of 50.

You will find that Sun supplied many example applets with Java.. There is a list on page 81 of your text. We have installed them at X:\JAVA\DEMO. Call them into the editor along with their supplied driver *.html* files. After you have played with each one, read through the source code and try to work out what they are doing. The easiest way to learn a new language is to mimick the ideas in other programs and borrow the techniques of accomplished programmers. Don't re-invent the wheel. If someone else has solved a problem, copy the idea. Modify the code to suit your purpose. By modifying other people's code and making it work in your program, you are showing that you understand what is happening.

Time to do some more work. Start a new project called *DrawShapes.jpr* Enter the following code as *DrawShapes.java*

```
// DrawShapes .java
// Written by Eddie Vanda, 1998

import javax.swing.JApplet;
import java.awt.*;
import java.awt.Dimension;

public class DrawShapes extends JApplet {

    public void paint (Graphics g) {
        drawResistor (g, 100, 50);
        drawResistor (g, 200, 50);
        drawResistor (g, 300, 50);
    }

    void drawResistor (Graphics g, int x, int y) {
        g.drawLine (x, y, x+10, y);
        g.drawRect (x+10,y-5, 25, 10);
        g.drawLine (x+35,y, x+45, y);
    }
}
```

Reuse your Driver file. Change the name of the class it calls to *DrawShapes.class* and change the dimensions to 400 x 400.

=====

This applet demonstrates the methods that are automatically launched when an applet is started. These methods are inherited from the Applet class and are normally left 'as is'. Occasionally, you will have to modify the *init()* method to modify the way your applet starts (see program 1, next lesson, for example). However, you will always modify the *paint()* method as this is where you fill your applet window with your output details.

Method	Operation (Eckel P.591)
init()	Called when the applet is first created to perform first-time initialisation of the applet.
start()	Called every time the applet moves into sight on the web browser to allow the applet to start up its normal operations (especially those that are shut off by stop()). Also called after init() .
paint()	Called as part of an update() to perform special painting on the canvas of an applet.
stop()	Called every time the applet moves out of sight on the web browser to allow the applet to shut off expensive operations. Also called right before destroy() .
destroy()	Called when the applet is being unloaded from the page to perform final release of resources when the applet is no longer used.

Table 2.2 The Automatic Applet Methods

```
//Applet3.java      Eckel P.595
// shows init(), start() and stop() activities

import java.awt.*;
import java.applet.*;

public class Applet3 extends Applet
{
    String s;
    int inits = 0, starts = 0, stops = 0;
    public void init() { inits++; };
    public void start() { starts++; };
    public void stop() { stops++; };

    public void paint(Graphics g)
    {
        s = "inits: " + inits + ", starts: " + starts + ", stops: " + stops;
        g.drawString(s, 10, 10);
    }
}
/***** End of Listing *****/

=====

/* The PIZZA Program
** This applet allows you to order a pizza
**
** Remember, it is an applet and needs a driver file
**/

import java.applet.*;
import java.awt.*;
import java.awt.event.*;
```

```
public class pizza extends Applet implements ItemListener
{
    private CheckboxGroup pizzaType;
    private Checkbox cheese, pepperoni, seafood, meat, garlicBread;
    private Label theLabel;

    public void init ( )
    {
        Panel p = new Panel ( );
        p.setLayout (new GridLayout (6, 1));
        theLabel = new Label ("");
        p.add (theLabel);
        pizzaType = new CheckboxGroup ( );
        cheese = new Checkbox ("Cheese", pizzaType, true);
        cheese.addItemListener (this);
        p.add (cheese);
        pepperoni = new Checkbox ("Pepperoni", pizzaType, false);
        pepperoni.addItemListener (this);
        p.add (pepperoni);
        seafood = new Checkbox ("Seafood", pizzaType, false);
        seafood.addItemListener (this);
        p.add (seafood);
        meat = new Checkbox ("Meat Eaters", pizzaType, false);
        meat.addItemListener (this);
        p.add (meat);
        garlicBread = new Checkbox ("Garlic bread");
        p.add (garlicBread);
        setLayout (new BorderLayout ( ));
        add (p, "Center");
    }

    public void itemStateChanged (ItemEvent theEvent)
    {
        if (cheese.getState ( ))
            theLabel.setText ("Yummy cheese!");
        else
            theLabel.setText ("No cheese, thanks!");
    }
}
/***** End of Listing *****/
```

Lesson 3: Introducing The Mouse

The first two lessons only used the keyboard to react with your programs. It's time to introduce the mouse. There are many operations that the mouse can perform; clicking buttons, pressing and holding buttons while you move the mouse, moving without pressing - all these distinct operations have to be detected. We also have to know where the mouse is on the screen in relation to the buttons, toolbars, etc. that you put in your windows. **Table 3.1** lists the methods that are available to our programs for mouse monitoring. The following programs illustrate how Java monitors mouse events.

Method	Interface
public void mouseClicked (MouseEvent e)	ML
public void mousePressed (MouseEvent e)	ML
public void mouseReleased (MouseEvent e)	ML
public void mouseEntered (MouseEvent e)	ML
public void mouseExited (MouseEvent e)	ML
public void mouseDragged (MouseEvent e)	MML
public void mouseMoved (MouseEvent e)	MML

**Table 3.1. Methods of the
MouseListener and MouseMotionListener Interfaces**

Using the Mouse:

This first applet only monitors the mouseDrag event. The driver for this applet also introduces another concept; passing information from the web page to your applet.

```

/* HelloMouse.java
** Monitor mouse events
*/
import javax.swing.JApplet;
import java.awt.*;

public class HelloMouse extends JApplet
{
    int messageX = 125, messageY = 95;
    String myMessage;

    public void init () {
        myMessage = getParameter("message");
    }

    public void paint (Graphics g) {
        g.drawString(myMessage, messageX, messageY);
    }

    public boolean mouseDrag(Event evt, int x, int y) {
        messageX = x;
        messageY = y;
        repaint ();
    }
}

```

```
        return true;
    }
}
/***** End of Listing *****/
```

Create a slightly different driver and save it as MouseDriver.html or something similar:

```
<html>
<head>
  <title>Applet Driver File</title>
</head>

<applet
  code="HelloMouse.class"
  width=300
  height=200>
  <param
    name= "message"
    value= "Hello Mouse!!">
  </applet>
</html>
```

The <param> tag is used to pass information across to the applet from a web page.

This next exercise is an extension of last lesson's exercise DrawShapes. We will use the mouse to determine where we draw the resistor.

```
//MovingResistor.java
// Written by Eddie Vanda, 1998
import javax.swing.JApplet;
import java.awt.*;
import java.awt.event.*;

public class MovingResistor extends JApplet implements MouseListener {
    int cx = 5;
    int cy = 20;

    public void init () {
        this.addMouseListener (this);    // listen to my own mouse events
    }

    public void mouseClicked (MouseEvent e) {
        cx = e.getX();
        cy = e.getY();
        repaint ();
        System.out.println ("Mouse clicked at x = " + cx + " y = " + cy);
    }

    public void paint (Graphics g) {
        drawResistor (g, cx, cy);
    }
}
```

```
void drawResistor (Graphics g, int x, int y) {
    g.drawLine (x, y, x+10, y) ;
    g.drawRect (x+10, y-5, 25, 10) ;
    g.drawLine (x+35, y, x+45, y) ;
}

public void mouseEntered (MouseEvent e) { }
public void mouseExited (MouseEvent e) { }
public void mousePressed (MouseEvent e) { }
public void mouseReleased (MouseEvent e) { }
}
/***** End of Listing *****/
```

Compile this applet and reuse your original driver file from last lesson. Set the co-ordinates to 400 x 300 and run the applet. Click on the applet window and note the effect. Note what is happening in the black DOS window at the same time. You may have to reset the co-ordinates of the driver file to 200 x 200 or something similar to view both windows at the same time. What is the console window displaying?

Note, this next program is an application, not an applet:

```
// Fig. 12.17: MouseTracker.java
// Demonstrating mouse events.
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class MouseTracker extends JFrame implements MouseListener,
MouseMotionListener
{
    private JLabel statusBar;

    public MouseTracker()
    {
        super( "Demonstrating Mouse Events" );

        statusBar = new JLabel( );
        getContentPane( ).add( statusBar, BorderLayout.SOUTH );

        // application listens to its own mouse events
        addMouseListener( this );
        addMouseMotionListener( this );

        setSize( 275, 300 );
        show();
    }
}
```

```
// MouseListener event handlers
public void mouseClicked( MouseEvent e )
{
    statusBar.setText( "Clicked at [" + e.getX() + ", " + e.getY() + "]" );
}
public void mousePressed( MouseEvent e )
{
    statusBar.setText( "Pressed at [" + e.getX() + ", " + e.getY() + "]" );
}

public void mouseReleased( MouseEvent e )
{
    statusBar.setText( "Released at [" + e.getX() + ", " + e.getY() + "]" );
}

public void mouseEntered( MouseEvent e )
{
    statusBar.setText( "Mouse in window" );
}

public void mouseExited( MouseEvent e )
{
    statusBar.setText( "Mouse outside window" );
}

// MouseMotionListener event handlers
public void mouseDragged( MouseEvent e )
{
    statusBar.setText( "Dragged at [" + e.getX() + ", " + e.getY() + "]" );
}

public void mouseMoved( MouseEvent e )
{
    statusBar.setText( "Moved at [" + e.getX() + ", " + e.getY() + "]" );
}

public static void main( String args[ ] )
{
    MouseTracker app = new MouseTracker();
    app.addWindowListener(
        new WindowAdapter() {
            public void windowClosing( WindowEvent e )
            {
                System.exit( 0 );
            }
        }
    );
}
}
/***** End of Listing *****/
```

```
/*Painter.java
** A basic painting program that uses MouseMotionAdapter
*/
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class Painter extends JFrame
{
    private int xValue = -10, yValue = -10;
    public Painter()
    {
        super("A simple paint program");

        getContentPane().add(new Label("Drag the mouse to draw."),
                               BorderLayout.SOUTH);
        addMouseListener
        (
            new MouseMotionAdapter()
            {
                public void mouseDragged(MouseEvent e)
                {
                    xValue = e.getX();
                    yValue = e.getY();
                    repaint();
                }
            }
        );
        setSize(300, 300);
        show();
    }
    public void paint(Graphics g)
    {
        g.fillOval(xValue, yValue, 4, 4);
    }
    public static void main(String args[ ])
    {
        Painter fred = new Painter();
        fred.addWindowListener
        (
            new WindowAdapter()
            {
                public void windowClosing(Event e)
                {
                    System.exit(0);
                }
            }
        );
    }
}
/***** End of Listing *****/
```

Lesson 4: Introducing Widgets

Widgets, or window gadgets, are the GUI components that go together to become the interface between your program and the user. The *Widgets.java* program below creates the screen shot shown in **Figure 4.1**. This demonstrates some of the many components that are available to you.

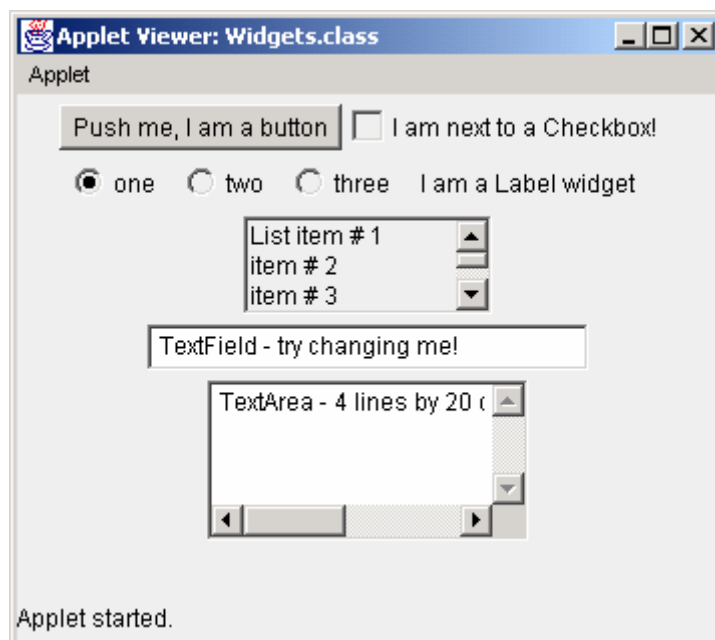


Figure 4.1 Widgets

Widgets:

Note: this program does not use the new *swing* library set. It uses the original AWT.

```
// Widgets.java
import java.applet.*;
import java.awt.*;

public class Widgets extends Applet
{
    Button b = new Button ("Push me, I am a button");
    Checkbox cb = new Checkbox ("I am next to a Checkbox!");
    CheckboxGroup cbg = new CheckboxGroup (); // no label text
    Label lab = new Label ("I am a Label widget");
    List list = new List (3,false);
    TextArea ta = new TextArea (
        "TextArea - 4 lines by 20 columns _ try changing me!", 4, 20);
    TextField tf = new TextField ("TextField - try changing me!");

    public void init ()
    {
        add (b);
        add (cb); // Checkbox
    }
}
```

```

add (new Checkbox ("one", cbg, true));    // three Checkbox
add (new Checkbox ("two", cbg, false));  // widgets form a
add (new Checkbox ("three", cbg, false)); // CheckboxGroup

add (lab);                               // label

list.addItem ("List item # 1");          // set up some
list.addItem ("item # 2");               // items in a
list.addItem ("item # 3");               // list box
list.addItem ("item # 4");
list.addItem ("item # 5");
list.addItem ("item # 6");
list.addItem ("item # 7");
list.addItem ("item # 8");
add (list);                              // add the list box

add (tf);                                // TextField
add (ta);                                // TextArea
}
}
/***** End of Listing *****/

```

When you are designing the layout of your GUI window, you will have to decide where to place the components. You can count the pixels and place them using absolute coordinates. However, you will find any changes will become very tedious very quickly. The easiest way is to let one of the purpose-built layout managers manage the placement of your widgets within the window frame. The three most commonly used managers are listed in **Table 4.1**.

FlowLayout	Places components sequentially (left to right) in the order that they were added. Default manager for <i>java.awt.Applet</i> , <i>java.awt.Panel</i> and <i>javax.swing.JPanel</i>
BorderLayout	Arranges the components into five areas: NORTH, SOUTH, EAST, WEST and CENTER. Default manager for <i>JFrame</i> and <i>JApplet</i>
GridLayout	Arranges the components into rows and columns

Table 4.1 Layout Managers

The following code will do the same as the previous program, but I have modified it to use the new *swing* library, and you will modify it further:

```

// WidgetsSwing.java

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class WidgetsSwing extends JFrame
{
    private JButton b;

```

```

private JCheckBox cb;
private JRadioButton b1, b2, b3;
private JLabel lab;
private JList list;
private String listStuff[ ]={"List item # 1", "item # 2",
    "item # 3", "item # 4", "item # 5", "item # 6",
    "item # 7", "item # 8"};
private JTextField tf;
private JTextArea ta;

public WidgetsSwing( )
{
    super("Widgets in a Box");

    Container c = getContentPane();
    c.setLayout(new FlowLayout( ));

    b = new JButton ("Push me, I'm a button");
    c.add(b);

    cb = new JCheckBox ("I am next to a Checkbox!");
    c.add(cb);

    b1 = new JRadioButton("One");
    c.add(b1);
    b2 = new JRadioButton("Two");
    c.add(b2);
    b3 = new JRadioButton("Three");
    c.add(b3);

    lab = new JLabel ("I am a Label widget");
    c.add(lab);

    list = new JList (listStuff);
    list.setVisibleRowCount(3);
    list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
    c.add(new JScrollPane(list));

    tf = new JTextField ("TextField - try changing me!", 30);
    c.add(tf);
    ta = new JTextArea ("TextArea - 4 lines by 20 columns _ try changing me!", 4, 20);
    c.add(ta);

    addWindowListener          // this windowlistener could have
    (                            // been put in the main() method
    new WindowAdapter( )      // see ShowColours.java in Week5.rtf
    {
        public void windowClosing(WindowEvent e)
        {
            System.exit(0);
        }
    }
}

```

TASK #1:
Add tooltips to some of your components.

TASK #2:
Put some code behind the button so that we can use it to exit the program.

TASK #3:
Save text from the TextField and display it in the TextArea

TASK #4:
Add a menu with one item that will allow us to exit the program this way.

Task #5:
The radio buttons are not in a group. Add code that will allow only one button to be selected.
Put code behind the buttons to change the background colour of the textbox.

```
    }
    }
);
setSize(350, 250);
setLocation(100, 100);
setResizable(false);
show();
}

public static void main (String args[ ])
{
    new WidgetsSwing( );
}
}
/***** End of Listing *****/
```

The next two programs are taken from the textbook and show some basic windows skills.

```
/* Fig 11.5: ShowColours.java
** Dietel P519
** Demonstrating Colours
*/
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ShowColours extends JFrame
{
    public ShowColours( )
    {
        super( "Trial Colour demo");

        setSize(400, 130);
        show();
    }

    public void paint(Graphics g)
    {
        // set new colour using integers
        g.setColor(new Color(255, 0, 0));
        g.fillRect(25, 25, 100, 20);
        g.drawString("Current RGB: " + g.getColor(), 130, 40);

        // set new colour using floats
        g.setColor(new Color(0.0f, 1.0f, 0.0f));
        g.fillRect(25, 50, 100, 20);
        g.drawString("Current RGB: " + g.getColor(), 130, 65);
    }
}
```

```
// set new colour using static colour objects
g.setColor(Color.blue);
g.fillRect(25, 75, 100, 20);
g.drawString("Current RGB: " + g.getColor(), 130, 90);

// display individual RGB values
Color c = Color.magenta;
g.setColor(c);
g.fillRect(25, 100, 100, 20);
g.drawString("RGB values: " + c.getRed() + ", " + c.getGreen() + ", " +
            c.getBlue(), 130, 115);
}

public static void main(String args[ ])
{
    ShowColours app = new ShowColours();

    app.addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        }
    );
}
}
/***** End of Listing *****/
```

```
/* Fig 11.6: ShowColours2.java
** Dietel P521
** Demonstrating JColourChooser
**/
```

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class ShowColours2 extends JFrame
{
    private JButton changeColour;
    private Color colour = Color.lightGray;
    private Container c;

    public ShowColours2( )
    {
        super( "Trial Colour demo choosing colours");
        c = getContentPane( );
        c.setLayout(new FlowLayout( ));
    }
}
```

```
changeColour = new JButton("Change colour");
changeColour.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent e)    {
            colour = JColorChooser.showDialog(ShowColours2.this,
                "Pick a colour", colour);

            if(colour == null)
                colour = Color.lightGray;
            c.setBackground(colour);
            c.repaint();
        }
    }
);
c.add(changeColour);
setSize(400, 130);
show();
}

public static void main(String args[ ])
{
    ShowColours2 app = new ShowColours2();

    app.addWindowListener(
        new WindowAdapter()
        {
            public void windowClosing(WindowEvent e)
            {
                System.exit(0);
            }
        }
    );
}
}
/***** End of Listing *****/
```

Lesson 5: Language Flow Control

Answer to Widget Swing Coding Tasks from last lesson:
(modifications in orange)

```
// WidgetsSwing.java
// with 5 modifications requested last lesson
// I have also added code behind the JList

import javax.swing.*;
import javax.swing.event.*; // needed for the JList listener
import java.awt.*;
import java.awt.event.*;

public class WidgetsSwing extends JFrame
{
    private JButton b;
    private JCheckBox cb;
    private JRadioButton b1, b2, b3;
    private ButtonGroup group;
    private JLabel lab;
    private JList list;
    private String listStuff[] = {"List item # 1", "Peaches", "Apples", "Oranges",
                                "Grapefruit", "Bananas", "Passionfruit", "Olives"};
    private JTextField tf1, tf2;
    private JTextArea ta;

    public WidgetsSwing()
    {
        super("Widgets in a Box");

        Container c = getContentPane();
        c.setLayout(new FlowLayout());

        b = new JButton ("Push me, I'm a button");
        b.setToolTipText("Press to exit program"); //P561
        c.add(b);

        ButtonHandler bh = new ButtonHandler(); //P573
        b.addActionListener(bh);

        putInMenu();

        cb = new JCheckBox ("I am next to a Checkbox!");
        c.add(cb);

        b1 = new JRadioButton("Red", true);
        c.add(b1);
        b2 = new JRadioButton("Green", false);
        c.add(b2);
    }
}
```

```
b3 = new JRadioButton("Blue", false);
c.add(b3);

RadioButtonHandler rbh = new RadioButtonHandler();
b1.addItemListener(rbh);
b2.addItemListener(rbh);
b3.addItemListener(rbh);

group = new ButtonGroup( );
group.add(b1);
group.add(b2);
group.add(b3);

lab = new JLabel ("I am a Label Widget")
c.add(lab);

list = new JList (listStuff);
list.setVisibleRowCount(3);
list.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
c.add(new JScrollPane(list));
list.addListSelectionListener(
    new ListSelectionListener()
    {
        public void valueChanged(ListSelectionEvent e)
        {
            tf1.setText(listStuff[list.getSelectedIndex( )]);
        }
    }
);

tf1 = new JTextField ("TextField - try changing me!", 30);
c.add(tf1);
tf2 = new JTextField ("You can't change this one!", 30);
tf2.setForeground(Color.red);
tf2.setBackground(Color.white);
tf2.setEditable(false);
c.add(tf2);

TextFieldHandler tfh = new TextFieldHandler();
tf1.addActionListener(tfh);

ta = new JTextArea ("TextArea - 4 lines by 20 columns _ try changing me!",
                    4, 20);
c.add(ta);

addWindowListener      // this windowlistener could have
(                       // been put in the main() method
    new WindowAdapter(  // see ShowColours.java in Week5.rtf
    {
        public void windowClosing(WindowEvent e)
```

```
        {
            System.exit(0);
        }
    }
);

setSize(350, 350);
setLocation(200, 300);
show();
}
/*****/
public static void main (String args[ ])
{
    new WidgetsSwing();
}
/*****/
private class ButtonHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        System.exit(0);
    }
}
/*****/
private class TextFieldHandler implements ActionListener
{
    public void actionPerformed(ActionEvent e)
    {
        String text = ""; // create an empty string
        text += e.getActionCommand(); // add the input to the string
        tf2.setText(text); // write the string to the other textfield
    }
}
/*****/
private class RadioButtonHandler implements ItemListener
{
    public void itemStateChanged(ItemEvent e)
    {
        if(e.getSource() == b1)
            tf1.setBackground(Color.pink);
        else if(e.getSource() == b2)
            tf1.setBackground(Color.green);
        else
            tf1.setBackground(Color.cyan);
        tf1.repaint();
    }
}
/*****/
private void putInMenu() // P650 3rd ED.
{ // P747 4th ED.
```

```

// P696 5th ED.
JMenuBar bar = new JMenuBar( );           // create space for the menus
setJMenuBar(bar);                         // put it in the frame

JMenu fileMenu = new JMenu("File");
fileMenu.setMnemonic('F');

JMenuItem exitItem = new JMenuItem("Exit");
exitItem.setMnemonic('x');
exitItem.addActionListener
(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
        {
            System.exit(0);
        }
    }
);
fileMenu.add(exitItem);
bar.add(fileMenu);
}
}
/***** End of Listing *****/

```

Primitive Data Types

Table 5.1 lists the lowest level or primitive data types that are available to Java programmers. Unlike other programming languages, these data types are fixed for all hardware platforms. For example, asking for memory to store data of type **int** will always receive four bytes of memory. There are no unsigned data types, the most significant bit is always used to indicate positive or negative value.

TYPE	SIZE	MIN/MAX
boolean	1 bit	true or false
char	16 bits	\u0000 \uFFFF
byte	8 bits	-128 127
short	16 bits	-32,768 32,767
int	32 bits	-2,147,483,648 2,147,483,647
long	64 bits	-9,223,372,036,854,775,808 9,223,372,036,854,775,807
float	32 bits	1.40239846e-45 3.40282347e+38
double	64 bits	4.94065645841246544e-324 1.79769313486231570e+308

Table 5.1 Java Primitive Data Types

Program Constructs

All computer programs are made up of combinations of three basic constructions.

- sequence (concatenation)
- selection (alternation) *if, if-else, switch-case*
- repetition (iteration) *for, while, do-while*

Figure 5.1 shows a segment of a computer program that has a combination of all three constructs. Laying the program out in this type of diagram allows the programmer to see the ‘big picture’. Converting this flow chart into source code is a simple step.

Block B1 represents an *if* statement. If the test B1 is true, then *sequence* S1, S2 are performed and the program continues on to the next phase. Otherwise, the code in S3 is activated. S3 will be repeated as long as B2 is true. S3, B2 form a *do-while* loop.

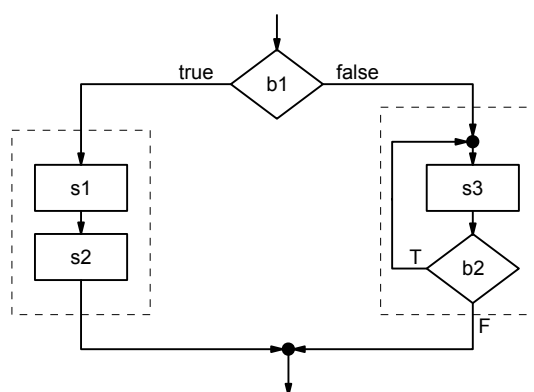


Figure 5.1 Flow Chart of a Code Segment

Boolean operators:		Comparison symbols:	
&&	AND	>	greater than
	OR	>=	greater than or equal to
!	NOT	<	less than
		<=	less than or equal to
		==	is equal to
		!=	is not equal to
		!	a prefix to invert the next value

Table 5.2 Operators Used To Determine Control Flow Decisions

The ‘if’ Statement

The ‘if’ test is used to send your program down either of two alternative paths. The ‘if’ statement is made up of three components: the key word if, an expression to evaluate and a statement to be activated. The ‘if’ statement is then completed with a semi-colon (;).

The program evaluates the expression. If the result of the evaluation is true, the statement following the ‘if’ test will be activated. If the result of the evaluation is found to be false the program ignores this statement and program control is passed to the next statement. If the next statement is the keyword ‘else’ then this will be activated before control is passed on. Multiple statements are placed in a block by using the curly braces ({}).

```

if (expression)
    statement ;
next statement ;

```

```

if(Mark < 50)
    System.out.println("NN");
// continue processing

```

```

if (expression)
    statement ;
else
    statement ;
next statement ;

```

```

if(Mark < 50)
    System.out.println("NN");
else
    System.out.println("PA");
// continue processing

```

```

if (expression)
{
    statement ;
    statement ;
}
else
{
    statement ;
    statement ;
    statement ;
}

```

```

if(Mark < 50)
{
    System.out.println("NN");
    sendFailNotice();
}
else
{
    System.out.println("PA");
    send SuccessNotice();
    enrolNextPhase();
}
// continue processing

```

With multiple *if* statements, it is easy to lose the relationship between matching *if...else* pairs. Laying the source code out in a logical way will help to reduce the confusion. However, there is no substitute for careful planning and attention to detail.

```

if(Mark < 50)
    System.out.println("NN");
else if(Mark < 70)
    System.out.println("PA");
else if(Mark < 80)
    System.out.println("CR");
else if(Mark < 90)
    System.out.println("DI");
else
    System.out.println("HD");

```

The *switch...case* construct is a special form of the *if* statement. It allows a choice from multiple options. The *if* statement only allows two options. The *switch...case* is ideal for constructing menus, where ‘one of many’ is required.

```

switch (expression) {
    case const_expr : statement;
                    break;
    case const_expr : statement;
                    statement;
                    break;

```

```
.....
default : statement;
}
.....
switch (Menu_Choice)
{
    case 'a' : addEmployee( );
                Emp_Count++;
                break;
    case 'u' : updateEmployee( );
                break;
    case 'd' : deleteEmployee( );
                Emp_Count--;
                break;
    case 'q' : System.exit (0);
    default : runMenu( );
}
```

Task 1:

Get a worker's name, the number of hours he has worked and the basic rate at which he is to be paid. Calculate the total wage, allowing for payment of the first 38 hours at the basic rate and any further hours at 1.5 times the basic rate.

Display the name and gross pay.

Hint: Use the *adding* program from lesson 1 as the basis for your program. You may wish to write your program from 'scratch', but at this stage of your programming career it is easier to modify an existing, working program.

Task 2:

Modify the above program. Calculate the income tax payable with the first \$200 being tax-free; the next \$300 taxed at 25¢ in the dollar; and the remainder at 45¢ in the dollar. Calculate the nett wage. Display the name, gross wage, tax and nett wage.

The 'while' Loop

This form of construction allows a section of your program to repeat for an as-yet unknown number of times. For the *while* loop to work correctly, three conditions need to be considered:

1. A precondition needs to be existing that can be tested in the *while* expression. If this condition is false, the *while* loop will never run.
2. The expression must have a true/false result to determine when the loop finishes, and must originally equate to true so the loop starts.
3. Conditions within the loop must change so that the expression becomes false and the loop eventually finishes. If the expression never becomes false, the loop never finishes and you have created an **endless** or **infinite loop**.

while (*expression*)

statement ;
 next statement ;

```

while (expression)
{
    statement ;
    statement ;
    statement ;
}
next statement ;

```

```

.....
ReadNextRecord();           // priming read – i.e. the loop is primed
while (moreRecords == true)
{
    System.out.println(Worker_Name);
    readNextRecord();
}
....
....

```

```

// WhileCounter.java
// Counter-controlled repetition
import java.awt.Graphics;
import javax.swing.JApplet;

public class WhileCounter extends JApplet {
    public void paint( Graphics g ) {
        int counter = 1;           // initialization
        while ( counter <= 10 ) {  // repetition condition
            g.drawLine( 10, 10, 250, counter * 10 );
            ++counter;             // increment
        }
    }
}

```

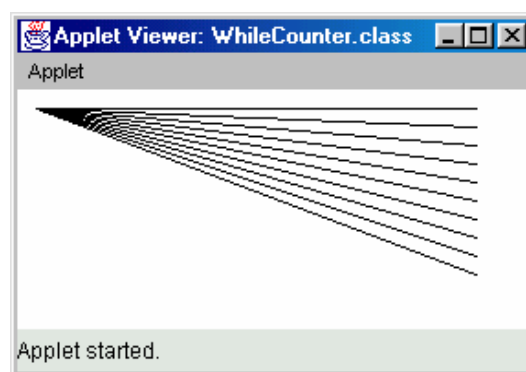


Figure 5.2 Output of WhileCounter Program

Task 3:

Write an application that accepts an integer between 65 and 90 only, then use a **cast** to print out the unicode (ASCII) value of that number. Use a *while* loop to reject inputs until a legal number is entered.

The 'for' Loop

The *for* loop is a variation of the *while* loop. This time the number of iterations, or repetitions, is known. All three conditions that controlled the operation of the *while* loop are placed in the brackets next to the reserved word *for* and are separated by semi-colons. These equate to:

1. *expression1* - **START** - set up the starting conditions (initialise the loop counter)
2. *expression2* - **STOP** - set where the loop stops. Don't forget that the loop will continue while this condition is true and will stop when it becomes false.
3. *expression3* - **STEP** - determine by how much the loop counter is incremented for each iteration.

```
for (expression1 ; expression2 ; expression3)
    statement ;
next statement ;
```

```
for (expression1 ; expression2 ; expression3)
{
    statement ;
    statement ;
    statement ;
}
next statement ;
```

```
....
for (i = 0 ; i < 10 ; i++)    // P.173
    System.out.println( i );
....
....
```

The segment of code above shows a *for* loop. We will now analyse it, looking for the three conditions that control the program flow:

1. *expression1* sets the loop counter **i** to zero.
2. *expression2* is now evaluated to see if the loop is to continue. If the result of this expression is **false**, the loop immediately terminates and control of the program's execution is passed to the next statement following the loop. If the statement is **true** control is passed into the loop. As the value of **i** is less than 10, the *System.out.println* statement is actioned and '0' is printed on the screen.
3. After all the code in the body of the loop has been executed, program control goes to *expression3*. The loop counter is incremented, in our case, by 1.
4. *expression2* is re-evaluated. The expression is still **true**, so our loop is repeated and the *System.out.println* statement prints '1' on the screen.
5. Steps 3. and 4. will now repeat until **i** reaches 10 and *expression2* now becomes **false**.

Task 4:

Write a program to print out Fahrenheit and equivalent Celcius temperatures in the range 0° to 300° in 20° steps. Use the formula $C^{\circ} = (5/9)(F^{\circ} - 32)$.

Lesson 6: Classes, Methods, Variables and Scope

Classes

- pieces of a java program – containing methods & variables
- the *public* class must be the same name as the file name with a *.java* extension
- therefore, *.java* files can only contain one *public* class (p92)
- variables are local or global(called fields or instance variables)
- using a local variable before it is initialised is a syntax error (p92)
- setting the visibility of methods and variables sets their accessibility (see the list below)
- every class must *extend* another class. If the class is not specified, **object** is extended (p329)

Instantiation

Instantiation is the process of creating an instance of a class, or, allocating memory for a class and giving it a name in memory. This class has methods and variables (the first point above) that can be accessed by referring to the name of the class and using the dot (.) operator. Instances can be created in one or two steps.

```
Font myFont;
```

creates a reference to an object of type Font called ‘ myFont ’.

```
myFont = new Font("Arial", Font.BOLD, 18);
```

calls the constructor, writes the object to memory and sets some initial attributes.

These two steps could have been done together at the beginning of the class with the following line:

```
Font f = new Font("Arial", Font.BOLD, 18);
```

We can now access this instance of class Font in memory via its name, ‘myFont’, and use one of the methods that were included in the class template:

```
size = myFont.getSize();
```

This will return an integer value that represents the size of the font.

Interfaces

Every class will extend and enhance a superclass, its parent class. If you have not consciously extended any class, **object** will be extended by default. It is possible to also implement one or more *interfaces* to add more functionality to your class. These interfaces will modify how your program looks and behaves. (P238, P467)

Some of the interfaces we have seen so far are:

- ActionListener
- MouseListener
- MouseMotionListener

Abstract

Abstract classes contain empty method constructors only. They are intended as blueprints for creating child classes. They are intended as repositories for methods and data and can never be instantiated into an object. The idea of abstract classes fits logically with object oriented programming. If you think about a **Car** class, for example, they all

have wheels, engines and accelerate, brake, turn left. It is not logical to create a **Car** object, but you will see Falcon or Corolla objects of type **Car**. Ch.10 P.438
Schaum's Outline: Programming With Java P163

Visibility:

- field access modifiers.
- **Private** - no other class can directly manipulate the private data or method.
- **Protected** - only classes derived from the class containing the protected data or methods can operate on them directly.
- **Public** - any other class can manipulate the designated data or methods.
- **Static** - class-wide information -one copy of the variable is shared by all objects of the class.

Class Relationships:

- **Use** - when a class uses the public methods of another class
- **Inheritance** - when a class is derived from another class
- **Composition** - when a class is composed of other classes

Designing OOP Programs:

1. identify the objects (classes, variables) – look for the nouns
2. identify the responsibilities (methods) – look for the verbs

Methods

- called functions(C, C++), procedures(Pascal), subroutines(basic)
- perform a single task
- layout *from Name(to)*
 {
 }
 }

Methods can be used to modularise your code. If a method is longer than one page, it is too long. If it performs more than one task, it should be subdivided. The method is laid out using the following rules:

1. the name of the method should be meaningful. You can tell from its title what it does.
2. the curly braces denote the boundaries of your method.
3. a new segment of RAM is allocated on the stack in which to run your method.
4. the brackets following the name are used to pass information to your method when it starts. Local memory is allocated according to the parameters within these brackets and the memory is initialised with the incoming data. If no data is to be passed across, the brackets are left empty.
5. the field in front of the name declares the datatype returned when your method is finished. If it does not return a value the word **void** is used to denote no data returned. It is a restriction of the design of the language that only one piece of data can be returned.
6. the reserved word **return** is used to indicate which value is to be returned from your method. Its data type must match that declared in 5. above. This should be the last statement in the method.

7. Java's garbage collection system is used to reclaim this section of RAM. If you needed to retrieve anything from within your method, it's too late. It has been consigned to the Bit Bucket.

Follow this list to create a new method from existing code that is too big and needs to be modularised:

1. identify the block of code that is self-contained and can be easily removed.
2. cut and paste it to the end of the class (before the last closing curly brace)
3. put opening and closing curly braces around the pasted code.
4. give it a meaningful *method_Name ()*
5. if any variables need to be passed to your method, declare data types and data names within the brackets.
6. if data is to be returned declare the data type before the method name.
7. put a **return** statement at the end of the code (optional if no data is to be returned)
8. declare any required local memory.
9. call your new method by putting its name in the main program at place from where you cut the original code.
10. place any data that is to be passed within the brackets of the function name.
11. remember to save any returned data

The following method definition shows a method constructed from the above steps. The four lines of the **if** statement were cut from the previous method.

```
private double calculateWage(int Hours, double Rate)
{
    double Wage;

    if (Hours <= 38)
        Wage = Hours * Rate;
    else
        Wage = (38 * Rate) + (Hours - 38) * Rate * 1.5;

    return Wage;
}
```

This method is going to calculate a worker's wage. It receives two variables, an integer and a floating point number and returns a floating point number. Local memory is declared to store the calculated value. Note that the local data type is the same as that declared as the data type to be returned (the one in red). The following line will be added to the original code at the point where you removed the cut code (point 9 above). This will cause your new method to be run.

```
.....
Salary = calculateWage(hoursWorked, payRate);
.....
```

Note that the variable names in the main program are different to the names we used in the new method. Your method is running in a different, independent area of RAM. As such, what we call these variables has no effect on the main program.

The way that information is passed into a method is determined by the *type* of data being passed. If the data is one of the primitive data types, as listed on page 29, a copy of the information is passed to the method. This is called *pass-by-value*. A copy of the value is passed. If the data is a complex data type, an object, then the memory address of the information is passed so that your method can access the original data, not a copy. This is called *pass-by-reference*. We can refer to the original. The following table demonstrates both methods. In the first, a copy of the integer is given to *sample()*. In the second, the address of **date** *d* is passed to *sample()*.

pass by value	pass by reference
<pre>void test() { int a = 5; sample(a); }</pre>	<pre>void test() { date d = new date(); d.year = 1948; sample(d); }</pre>
<pre>void sample(int b) { b = 6; }</pre>	<pre>void sample(date e) { e.year = 2099; }</pre>

Table 6.1 Passing Parameters

See page 293, **References and Reference Parameters**, for an explanation of the difference between *pass-by-reference* and *pass-by-value*.

Variables

Variables can be declared in the body of the class (instance variables) or within each method (local variables).

Instance variables are initialised at the time of creation and are available throughout the class definition.

Local variables are only available within the scope of the method and are not initialised. The variable **Wage** in the previous method definition is an example. Attempting to compile this code will lead to a syntax error. The variable needs to be initialised.

```
double Wage = 0.0;
```

Constant variables are variables that cannot be changed. They are locked with the key word **final**. To maintain programming documentation standards the variable name should be spelt using uppercase and using underscores between words. (p239)

```
final int MAX_STUDENTS = 100;
```

As with all the programming examples I have shown you, this program is designed to highlight a concept. This concept is the availability, or visibility of variables. If you enter the following code into a file called Scope.java and run it, the output will not seem to have made much of a point. This time, to gain the maximum benefit, run the program on paper, that is, perform a bench check. This involves drawing diagrams and allocating memory in a similar fashion to the way I explain code operation on the board. This is a technique you will have to learn when developing and debugging your own programs.

```
// Scope.java
// Demonstrates the scope of variables

public class Scope
{
    public static void main(String args[ ])
    {
        int x = 4;

        System.out.println("x = " + x);
        doSomething( );
        System.out.println("x = " + x);
    }

    public void doSomething( )
    {
        int x = 5;
        System.out.println("x = " + x);
    }
}
```

Make sure that you understand what this program is showing you. Action each line in a bench check to analyse the operation.

The following program is broken up into two source code files to demonstrate the principle. This time create two **.java** files: one for *Animal* and one for *Main*. You will have to compile *Animal* before *Main* will run.

```
class Animal
{
    String noise = null;

    Animal (String sound)           // this constructor
    {                               // copies input parameter
        this.noise = sound;        // "sound"
                                   // into class variable "noise"
    }

    void makeNoise ( )
    {
        System.out.println (noise);
    }
}

=====

class Main
{
    static public void main (String args[ ])
    {
        Main m = new Main ( );
    }
}
```

```
    m.menagerie ();
}

void menagerie ()
{
    Animal dog = new Animal ("woof");
    Animal cat = new Animal ("miauw");

    System.out.println ("Program o/p: caused by: ");
    System.out.println ("    dog.makeNoise ();");
    dog.makeNoise ();           // should print "woof"

    System.out.println ("    cat.makeNoise ();");
    cat.makeNoise ();          // should print "miauw"

    System.out.println ("    cat = dog;");
    cat = dog;
    System.out.println ("    cat.makeNoise ();");
    cat.makeNoise ();          // what should this print????
}
}
/***** End of Listing *****/
```

The following Date and Test classes further investigate the use of local and global variables. Once again save them as two separate **.java** files.

```
/*****
** Test.java
** This program creates some objects of the "date" class,
** which is defined in "date.java". It demonstrates
** basic class usage.
*****/
```

```
import Date;

public class Test
{
    public static void main(String arg[ ])
    {
        Date a, b;
        a = new Date(2010, 1, 5);
        b = new Date();
        b.Year = 2010;
        b.Month = 1;
        b.Day = 10;
        a.addDays(5);
        if(a.isEqual(b))
            System.out.println("The dates are equal");
        else
```

```

        System.out.println("The dates are not equal");
    }
}
/***** End of Listing *****/

```

```

/*****
** Date.java
** This is a sample date class.
** It is taken from "Talk Java to Me"
** Program "test.java" shows how objects
** of this class can be created and used.
*****/

public class Date{
    int Year;
    int Month;
    int Day;

    Date()                // constructor
    {
        Year = 2000;
        Month = 1;
        Day = 1;
    }

    Date(int year, int month, int day)    // constructor
    {
        this.Year = year;
        this.Month = month;
        this.Day = day;
    }

    void addDays(int days)                // add the given number of days to the date
    {
        int daysInMonths[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

        Day += days;
        daysInMonths[1] = 28 + (leapYear() ? 1 : 0);

        while(Day > daysInMonths[Month-1])
        {
            Day -= daysInMonths[Month-1];
            if(++Month > 12)
            {
                Month = 1;
                ++Year;
                daysInMonths[1] = 28 + (leapYear() ? 1 : 0);
            }
        }
    }
}

```

```
}

boolean leapYear( )
{
    if( (Year % 100) == 0){
        if( (Year % 400) == 0)
            return true;
        else
            return false;
    }
    else{
        if( (Year % 4) == 0)
            return true;
        else
            return false;
    }
}

// determine if the two dates are equal
boolean isEqual(Date d)
{
    if(Year != d.Year)
        return false;
    if(Month != d.Month)
        return false;
    if(Day != d.Day)
        return false;
    return true;
}
}
}
/***** End of Listing *****/
```

The following program taken from your text also illustrates these concepts:

```
// Fig. 8.1: Time1.java
// Time1 class definition

import java.text.DecimalFormat;           // used for number formatting

// This class maintains the time in 24-hour format
public class Time1 extends Object {
    private int hour;           // 0 - 23
    private int minute;        // 0 - 59
    private int second;         // 0 - 59

    // Time1 constructor initializes each instance variable
    // to zero. Ensures that each Time1 object starts in a
    // consistent state.
    public Time1()
```

```

    {
        setTime( 0, 0, 0 );
    }

// Set a new time value using universal time. Perform
// validity checks on the data. Set invalid values to zero.
public void setTime( int h, int m, int s )
{
    hour   = ( ( h >= 0 && h < 24 ) ? h : 0 );
    minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
    second = ( ( s >= 0 && s < 60 ) ? s : 0 );
}

// Convert to String in universal-time format
public String toUniversalString( )
{
    DecimalFormat twoDigits = new DecimalFormat( "00" );

    return twoDigits.format( hour ) + ":" +
           twoDigits.format( minute ) + ":" +
           twoDigits.format( second );
}

// Convert to String in standard-time format
public String toString( )
{
    DecimalFormat twoDigits = new DecimalFormat( "00" );

    return ( (hour == 12 || hour == 0) ? 12 : hour % 12 ) +
           ":" + twoDigits.format( minute ) +
           ":" + twoDigits.format( second ) +
           ( hour < 12 ? " AM" : " PM" );
}
}
}
/***** End of Listing *****/

```

```

// Fig. 8.1: TimeTest.java
// Class TimeTest to exercise class Time1
import javax.swing.JOptionPane;

public class TimeTest {
    public static void main( String args[] )
    {
        Time1 t = new Time1();           // calls Time1 constructor
        String output;

        output = "The initial universal time is: " + t.toUniversalString() +
                "\nThe initial standard time is: " + t.toString() +
                "\nImplicit toString() call: " + t;
    }
}

```

```
t.setTime( 13, 27, 6 );
output += "\n\nUniversal time after setTime is: " + t.toUniversalString() +
        "\nStandard time after setTime is: " + t.toString();

t.setTime( 99, 99, 99 );           // all invalid values
output += "\n\nAfter attempting invalid settings: " + "\nUniversal time: " +
t.toUniversalString() + "\nStandard time: " + t.toString();

JOptionPane.showMessageDialog( null, output, "Testing Class Time1",
    JOptionPane.INFORMATION_MESSAGE );

System.exit( 0 );
}
}
/***** End of Listing *****/
```

Lesson 7: Arrays

An array is an identifier that refers to a collection of data items that have the same name. The items must be of the same type, that is, all integers, or all doubles, or all characters, etc. The array can be a collection of objects. Square brackets after the name of the array declare the number of elements to be stored. As with C/C++ and most programming languages, the array numbering starts at zero. So a declaration:

```
int Scores[ ] = new int[8];
```

declares memory to store 8 integers, the first being `Scores[0]` and the last being `Scores[7]`. The number in the brackets is referred to as the *subscript* or *index* and is used to access the data items in memory. The memory locations are automatically initialised to values of zero. The **new** operator is necessary as array elements are treated as objects and object memory is allocated with the **new** operator.

However, it is possible to initialise them with values of your choosing, like:

```
int Scores[ ] = {71, 144, 16, 55, 72, -18, 68, 42};
```

This will set up memory arranged like **Figure 7.1**. Note, it was not necessary to specify the number of memory cells, or use the **new** operator, the compiler counted the number of integers and automatically allocated sufficient memory.

Figure 7.1
The Scores Array

71	}	Scores[0]
144	}	Scores[1]
16	}	Scores[2]
55	}	Scores[3]
72	}	Scores[4]
-18	}	Scores[5]
68	}	Scores[6]
42	}	Scores[7]

- Arrays are used to store many variables of the same type.
- Array elements are accessed using the index number placed in square brackets
- Array elements are automatically initialised to zero (for primitive data types) or *null* (for non-primitives - objects)
- Arrays can be set up and initialised dynamically:
- Array elements are accessed by putting the appropriate subscript number inside the brackets following the array name.
- Array memory is allocated with the **new** operator as arrays are considered to be objects

```

float num [] = new float [12];    // declare and allocate the array
or
float num [];                      // declare a reference to the array
num = new float [12];             // allocates the array

String [] dow = {"Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
                 "Friday", "Saturday"};

```

The array length is determined by the number of elements in the list.

Every Java array knows its own length:

eg. `dow.length`
will determine the length of this array

```

class Array
{
    public static void main(String args[] )
    {
        int scores[] = {-45, 27, 1096, -3, 42, 8, 391, 66};
        System.out.println(scores[2] + " " + scores[4]);
        System.out.println(scores[2] + scores[4]);
    }
}

```

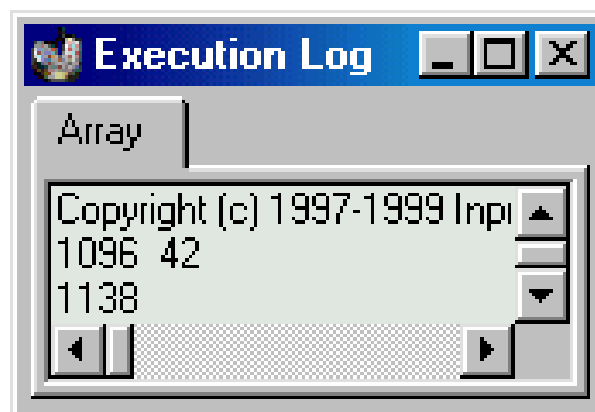


Figure 7.2 Output of Array.class

Be careful what you ask for in a program. The above program shows that you will get what you coded, not what you meant. Both print statements were meant to display the two array elements, however, the second statement added the numbers, then displayed the result.

Task 1:

Modify the program in **Task 4** of Lesson 4 so that it saves the C° and F° data in an array then prints it in a dialog box.

Lesson 8: Modularity

Modularising a Project

Introduction

Much of the initial work in designing a program is in selecting how many classes to use and how to organize them. Putting too much in one class can make that class unwieldy to debug and use. Fragmenting between too many classes can make the whole program hard to understand. The happy medium lies in aligning the class structure with the natural flow of the program.

Lab - Draw Resistors and Capacitors

In keeping with the philosophy of aligning the class structure with the natural flow of the program, Draw is called by the browser to organize things and to interface with the user's screen and mouse, MenuBox tracks the generation and the use of the menu, and Symbol tracks the generation and the drawing of each of the on-screen symbols. To help in this process, Constants provides a single source for the program's main attributes, and Global allows all other classes to access its variables from anywhere.

Step 1 - Preparing the source files:

Each of these files is a java source file and is worked on independently. However, they should all be together in one subdirectory as together they make a package.

Constants.java

The Constants.java file holds symbolic constants that can be quoted in other modules. It is an interface so it can be implemented by all the modules that need to access these symbols's values. The keyword static indicates that this is to be the only constant instance, and final that it may not be modified any further, after all we want it to be a constant! A very common practise is to use all capitals so these constants won't get confused with variables.

```
interface Constants {
    static final int RESISTOR = 1;
    static final int CAPACITOR = 2;
    static final String [] itemA = {" ", "R", "C"};    // for MenuBox
}
```

Global.java

The Global.java file holds the variables that are:
to be accessed by many parts of the program.
to be unique, that is, not able to be instantiated further.

The use of global variables should be reduced as much as possible. They introduce possible side effects which may be hard to track down.

The keyword *static* indicates that this is to be the only instance of these variables. `selectedItem` starts with 0 as neither resistor nor capacitor is selected to be drawn, and may be set to 1 or 2 for resistor or capacitor respectively. The `mba []` array tracks the menu boxes that the user can click on, and the `aSymbol []` array holds the positions and types of the components drawn so far.

```
class Global {
    static int selectedItem = 0;
    static MenuBox [] mba = null;
    static Symbol aSymbol[] = null;
}
```

MenuBar.java

An instance of this class is made for each menu choice. These instances are then put into an array for easier management.

```
01:import java.awt.*;
02:import java.applet.*;
03:
04:class MenuBox implements Constants{
05:  int x, y;
06:  int width = 20;
07:  int height = 20;
08:
09:  int menuType = 0;
10:  Font f = new Font("TimesRoman",Font.BOLD, 20);
11:
12:  MenuBox (int item) {
13:    MenuBox tmba [] = null; // temporary menu box array
14:
15:    if (Global.mba == null)
16:      tmba = new MenuBox [1];
17:    else {
18:      tmba = new MenuBox [Global.mba.length + 1];
19:      for (int j=0; j < Global.mba.length; j++)
20:        tmba[j] = Global.mba [j];
21:    }
22:    tmba [tmba.length - 1] = this; //assign new symbol to last instance
23:
24:    Global.mba = tmba;
25:
26:    menuType = item;
27:    y = 3;
28:    x = item * 30;
29:  }
30:
31:
32:  void drawMenuBox (Graphics g){
33:    g.setFont(f);
34:
```

```

35:  if (Global.selectedItem == menuType)
36:      g.setColor(Color.green);
37:  else
38:      g.setColor(Color.black);
39:
40:  g.drawString (itemA [menuType], x + 3, y + 14);
41:  g.drawRect (x, y, width, height);
42:  }
43:
44:  boolean isItemSelected (Point p) {
45:      boolean selected = false;
46:
47:      if ((p.x >= x) && (p.x <= (p.x + width)) &&
48:          (p.y >= y) && (p.y <= (p.y + height)))
49:          selected = true;
50:
51:      return selected;
52:  }
53:
54:}

```

Symbol.java

An instance of this class is made for each object generated by the user with a mouse click.

```

01:import java.awt.*;
02:
03:class Symbol extends java.applet.Applet implements Constants
04:{
05:  private int symbolType; // 1 for r, 2 for c - see Constants i/f.
06:  private int x = 0;
07:  private int y = 0;
08:
09:
10:  Symbol (int type, Point p) { // constructor
11:      Symbol asymb [] = null;
12:
13:      // Create a temporary Symbol array one bigger than the global array.
14:      if (Global.aSymbol == null)
15:          asymb = new Symbol [1]; // this is the first entry.
16:      else {
17:          asymb = new Symbol [Global.aSymbol.length + 1];
18:          for (int j=0; j < Global.aSymbol.length; j++)
19:              asymb[j] = Global.aSymbol [j]; // copy existing Symbols
20:      }
21:      asymb [asymb.length - 1] = this; //assign new symbol to last instance
22:
23:      Global.aSymbol = asymb;
24:
25:      symbolType = type;

```

```
26:  this.x = p.x;
27:  this.y = p.y;
28:  }
29:
30:  void draw (Graphics g)
31:  {
32:    g.setColor(Color.black);
33:    if (symbolType == RESISTOR) {
34:      g.drawLine (x,  y,  x+10, y) ;
35:      g.drawRect (x+10, y-5, 30, 10) ;
36:      g.drawLine (x+40, y,  x+50, y) ;
37:    }
38:    else
39:      if (symbolType == CAPACITOR) {
40:        g.drawLine (x,  y,  x+18, y) ;
41:        g.drawRect (x+19, y-10, 4, 20) ;
42:        g.drawRect (x+29, y-10, 4, 20) ;
43:        g.drawLine (x+33, y,  x+50, y) ;
44:      }
45:    }
46: }
```

Draw.java

The Draw.java file is the source for the main class called by the browser. It controls the overall operation of the program and handles the mouse events. It is only instantiated once by the browser.

```
01:import java.awt.*;
02:import java.awt.event.*;
03:import java.applet.*;
04:
05:public class Draw extends Applet implements MouseListener, Constants
06:{
07:
08:
09:
10: public void init() {
11:   setBackground(SystemColor.white);
12:   addMouseListener(this);
13:   new MenuBox (RESISTOR);
14:   new MenuBox (CAPACITOR);
15: }
16:
17:
18: //needed to satisfy listener interfaces
19: public void mouseClicked(MouseEvent e) {}
20: public void mouseEntered(MouseEvent e) {}
21: public void mouseExited(MouseEvent e) {}
22: public void mouseReleased(MouseEvent e) {}
23:
```

```
24:
25: public void paint(Graphics g){
26:     int i;
27:
28:     if (Global.mba != null)
29:         for (i = 0; i < Global.mba.length; i++)
30:             if (Global.mba[i] != null)
31:                 Global.mba[i].drawMenuBox(g);
32:
33:     if (Global.aSymbol != null)
34:         for (i = 0; i < Global.aSymbol.length; i++)
35:             if (Global.aSymbol[i] != null)
36:                 Global.aSymbol[i].draw(g);
37: }
38:
39:
40: public void mousePressed(MouseEvent e){
41:     Point p = e.getPoint();
42:     if (p.y < 30)
43:         if (p.x < 50)
44:             Global.selectedItem = 1;
45:         else
46:             Global.selectedItem = 2;
47:     else
48:         if (Global.selectedItem != 0)
49:             Symbol s = new Symbol(Global.selectedItem, p);
50:
51:     repaint ();
52: }
53:}
```

draw.html

This is the file read by the browser to instantiate the Draw applet.

```
<html>
  <head>
    <title>The Draw Applet</title>
  </head>
  <body bgcolor="#000000" text="#FF00FF">
    <center>
      <APPLET
        CODE="Draw.class"
        height=300
        width=300>
      </APPLET>
    </center>
  </body>
</html>
```

Lesson 9: Testing and Debugging

testing - trying to *find* errors (bugs) in a program

debugging - a process performed by the *programmer* to remove the programs errors

Program testing may not show the presence of bugs, but it cannot guarantee their absence.

Carefully designed test data reduces the likelihood of hidden bugs.

Program bugs fall into three categories:

- syntax errors (compile-time errors)
- run-time errors (program stops mid-run)
- logic errors (design errors)

Syntax Errors

Violation of the grammatical rules of a programming language:

- found during compilation
- punctuation
- mis-capitalisation of variable names

Run-Time Errors

Errors that cause the program to prematurely terminate (crash!):

- infinite loops
- over-run the end of an array
- divide by zero

Logical Errors

Errors coming from an incorrectly designed program:

- input variables out of range
 - value range check
 - data type check
- loop
 - incorrect start/stop values
 - incorrect number of iterations
- truncation and rounding errors
- internal representation of values
- -ie. Test like ***while (x != 4.5) ...*** may fail because *x* never exactly equals 4.5.
 - use ***while (x <= 4.5) ...*** instead
- failure to correctly handle null data, zero or out of range data.
- preconditions
 - check that essential preconditions of loops or functions are satisfied and within legal range
- invariants
 - if a variable is to remain unchanged through a loop, this should be verified

Error Detection Techniques

- desk check - feed simple data into the source code
- stub programming - “This method is not written yet”
- walkthrough - a guided tour of the partially completed program
- snap-shots - statements in a program that print out intermediate values of variables.

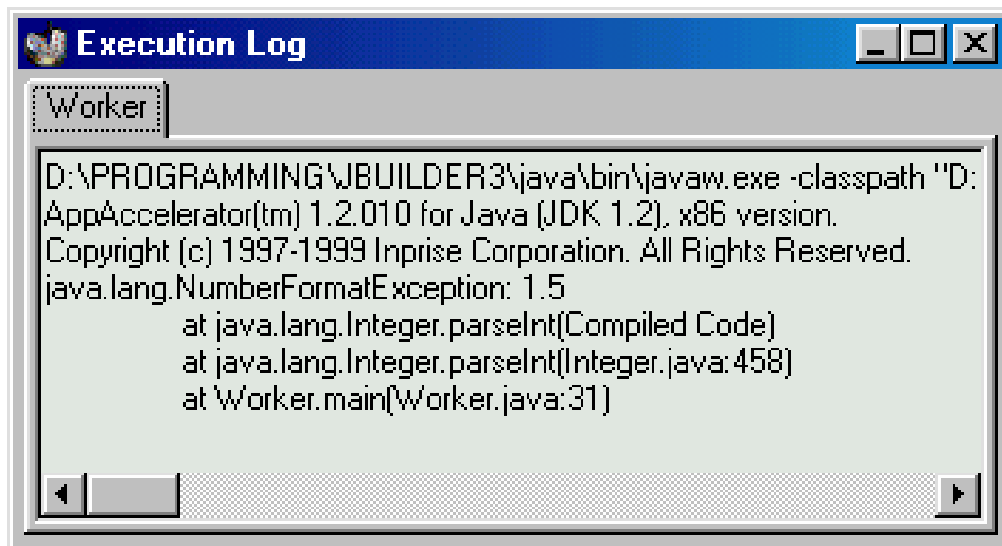


Figure 9.1 A Runtime Error

```
try{
    .....
}
catch( error_type ){
    .....
}
catch( error_type ){
    .....
}
finally{
    .....
}
```

```
try {
    number1 = Integer.parseInt( input1.getText() );
    number2 = Integer.parseInt( input2.getText() );
    result = quotient( number1, number2 );
}
catch ( NumberFormatException nfe ) {
    JOptionPane.showMessageDialog( null,
        "You must enter two integers", "Invalid Number Format",
        JOptionPane.ERROR_MESSAGE );
}
catch ( DivideByZeroException dbze ) {
    JOptionPane.showMessageDialog( null,
        "Attempted to Divide by Zero", "Divide by Zero Error",
        JOptionPane.ERROR_MESSAGE );
}
```

P:705

```
name = JOptionPane.showInputDialog("Worker's name:");

do{
    flag = false;
    try{
        hours = JOptionPane.showInputDialog("hours worked:");
        hoursWorked = Integer.parseInt(hours);
    }
    catch(NumberFormatException nfe){
        JOptionPane.showMessageDialog(null,
            "Number must be an integer",
            "Non-Valid Number Format",
            JOptionPane.ERROR_MESSAGE);

        flag = true;
    }
}while(flag);

rate = JOptionPane.showInputDialog("pay rate:");
```

The boolean variable 'flag' is used here to indicate if an error has occurred. It is initialised to *false* so that the while loop will finish if there is no error. It is set *true* inside the **catch** section as the only time this code will be actioned is when there has been an error.

Task:

Write an application that accepts an integer between 65 and 90 only, then use a **cast** to print out the unicode (ASCII) value of that number. Use a *while* loop to reject inputs until a legal number is entered.

Catch all number format exception

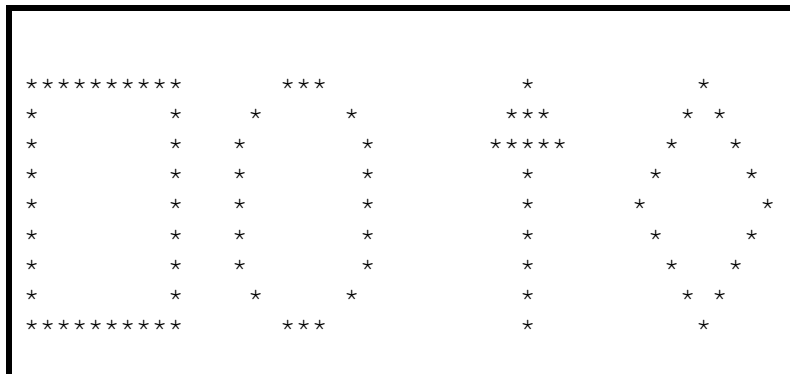
Java Programming Exercises

(x:\tedbown\java\sem-1\exercises.rtf)

- Write an application that calculates the sum of three numbers, 1.23, 55 and 93.82 then displays the result.
- Write an application that rings the bell (unicode value 0007) then prints the following text EXACTLY as shown in a DOS console window:
 Startled by the sudden sound, Sally shouted,
 "By the Great Pumpkin, what was that!"
- Repeat exercise 2 but have the output appear in a message box.
- The mass of a single molecule of water is 3.0×10^{-23} grams. One litre of water weighs 1000 grams. Write an applet that requests an amount of water in litres and displays the number of water molecules in that volume of water.
- Write an application that **reads in** a number then prints it in decimal floating point notation, then in exponential notation. Have the output appear in a JFrame and use the following format:

The input is 21.290000 or 2.129000e+01.

- Write a program that reads in the radius of a circle and prints the circle's diameter, circumference and area. Use **Math.PI**. Do each of the calculations inside the **println()** statements.
- Write an applet that prints a box, an oval, an arrow and a diamond, as follows:



8. Print your initials in block letters down the page. Construct each block out of the letter it represents, as follows:

```
EEEEEEEEEEEEEEEE
E       E       E
E       E       E
E       E       E
E       E       E

I                   I
IIIIIIIIIIIIIIII
I                   I

BBBBBBBBBBBBBBBB
B       B       B
B       B       B
B       B       B
  B   B   B   B
    BB   BB
```

9. Write an application that calculates the squares and cubes of the numbers from 1 to 10 and uses tabs to print the following table of values:

```
number      square      cube
0           1           0
1           1           1
2           4           8
3           9          27
4          16          64
5          25         125
6          36         216
7          49         343
8          64         512
9          81         729
10         100        1000
```

10. Write a program that reads an integer and determines and prints whether it is odd or even. (Hint: use the modulus operator. An even number is a multiple of two. Any multiple of two leaves a remainder of zero when divided by two).

11. Write a program that reads in two integers and determines and prints if the first is a multiple of the second. (Hint: use the modulus operator).

12. Write an application that adds together the integers from 1 to 100. Do it first using a **for** loop, then a **while** loop in the same program. Display the results of both calculations.

13. Modify exercise 12 to exclude from the sum any integers that are even multiples of 3 and to add twice any integers that are even multiples of 10.

14. Write a program that asks you to input an integer in the range 33 to 105 (the program **MUST ONLY** accept input within this range) then use **println()** to print out the unicode character that corresponds to this integer value.