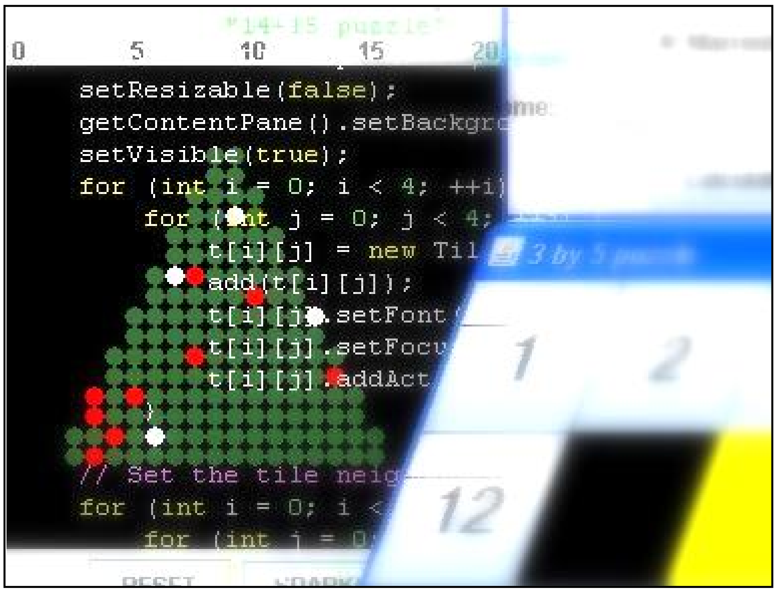
INTRODUCTION TO PROGRAMMING IN JAVA (PART 2)

Softeware Tools and Techniques (COMP1618)



```
setResizable(false);
getContentPane().setBackgr
setVisible(true);
for (int i = 0; i < 4; ++i)
    for (int j = 0; j < 4;
        t[i][j] = new Til
        add(t[i][j]);
        t[i][j].setFont(
        t[i][j].setFocu
        t[i][j].addAct
    }
// Set the tile neig
for (int i = 0; i <
    for (int j = 0
```

University of Greenwich

School of Computing and Mathematical Sciences

2012

Acknowledgements (2nd Edition)

For many years, Dr Don Cowell was the course coordinator for Computer Programming 1 (COMP1148).

This fine workbook (and the sets of exercises that go with it) was almost entirely written, developed and maintained by him.

Thank you, Don!

Chris Walshaw

Acknowledgements (3rd Edition)

From 2009 to 2011 Dr Chris Walshaw was the course coordinator for Computer Programming 1 (COMP1148). He has updated this workbook and has made a major contribution to the course by introducing NetBeans.

Asif Malik

Contents

11. Exceptions and Files in Java

Exceptions

If a Java program compiles OK, it can still go wrong as it is running, as we have seen.

Java distinguishes between run-time *Errors* and run-time *Exceptions*.

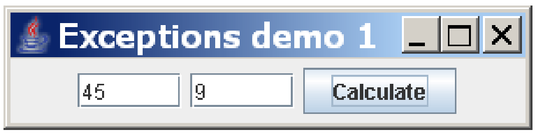
Run-time *errors* are quite rare and it is harder for the program to recover from them. An example would be a program – perhaps running on a small device such as a mobile phone – that runs out of RAM memory: the only way to recover would be to free up some memory, but this may not be possible

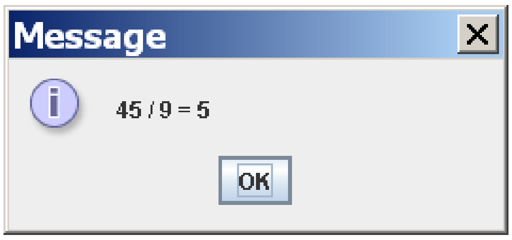
Run-time *exceptions* happen quite a lot, and the program *can* often recover from them.

For example, you may have noticed a lengthy exception message when you ask the user to type a number into a text field and the number isn't formatted properly. Consider the following application which tries to divide one whole number by another:

Exceptions1.java

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

public class Exceptions1 extends JFrame implements ActionListener {
    JTextField num1Txt = new JTextField(5);
    JTextField num2Txt = new JTextField(5);
    JButton calculate = new JButton("Calculate");

    public static void main(String[] args) {
        new Exceptions1();
    }

    public Exceptions1() {
        setLayout(new FlowLayout());
        setSize(300, 70);
        setTitle("Exceptions demo 1");
        add(num1Txt); add(num2Txt); add(calculate);
        calculate.addActionListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        int num1 = Integer.parseInt(num1Txt.getText());
        int num2 = Integer.parseInt(num2Txt.getText());
        int answer = num1 / num2;
        String message = num1 + " / " + num2 + " = " + answer;
        JOptionPane.showMessageDialog(null, message);
    }
}
```

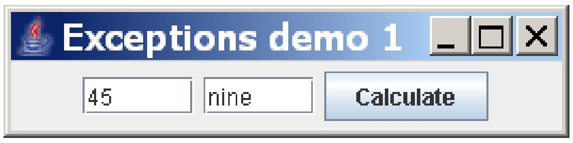




45 / 9 = 5

The screenshots above show what happens when the program runs "normally". But this program can go wrong in two different ways:

- The user might type badly formatted input:

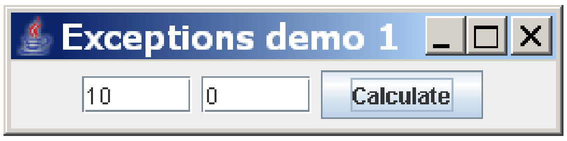


This produces an exception report displayed on the console which looks like this:

```
Exception in thread "AWT-EventQueue-0" java.lang.NumberFormatException:
For input string: "nine"
    ...
    at Exceptions1.actionPerformed(Exceptions1.java:26)
    ...
```

Much of the detail listed (there are several "at" lines omitted here) is a "stack trace" which shows how and where the exception happened. Here we see that it happened at line 26 when an attempt was made to parse the `String "nine"` as an `int`.

- The user might type two whole numbers with the second being 0. This gives rise to a "division by zero" exception:



```
Exception in thread "AWT-EventQueue-0" java.lang.ArithmeticException:
/ by zero
    at Exceptions1.actionPerformed(Exceptions1.java:27)
    ...
```

try ... catch

Rather than letting the Java run-time system handle the exceptions, we can handle them ourselves using the **`try ... catch`** mechanism:

Exceptions2.java

```
...     ...
24      public void actionPerformed(ActionEvent e) {
25          try {
26              int num1 = Integer.parseInt(num1Txt.getText());
27              int num2 = Integer.parseInt(num2Txt.getText());
28              int answer = num1 / num2;
29              String message = num1 + " / " + num2 + " = " + answer;
30              JOptionPane.showMessageDialog(null, message);
31          }
32          catch (Exception ex) {
33              System.out.println(ex);
34              // ex.printStackTrace();
35          }
36      }
```

(the remaining code is identical to `Exceptions1.java`)

At lines 25 – 31 we have a “try” block. This encloses code where we think that exceptions may occur. Each “try” block must be followed by one or more “catch” blocks to handle the exception. When an exception happens, control is passed to the “catch” block at lines 32 – 35. This prints the exception `ex` to the console.

Here we would get either:

```
java.lang.NumberFormatException: For input string "nine"
```

or

```
java.lang.ArithmeticException: / by zero
```

If you comment out line 33 and remove the `//` from line 34 you get the same output as for **Exceptions1.java**.

Self-test Exercise

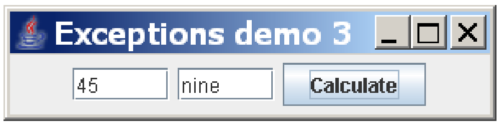
1. What happens if we have `try { ... }` without `catch { ... }`?

Exceptions in Java form a hierarchy with class `Exception` at the root. Instead of catching `Exception` (which would trap *any* exception) it is good practice to have a “catch” block for each separate exception type:

Exceptions3.java

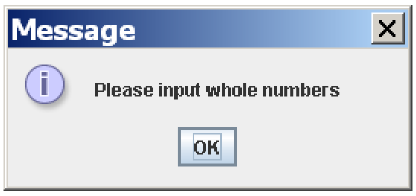
```
...     ...
24      public void actionPerformed(ActionEvent e) {
25          try {
26              int num1 = Integer.parseInt(num1Txt.getText());
27              int num2 = Integer.parseInt(num2Txt.getText());
28              int answer = num1 / num2;
29              String message = num1 + " / " + num2 + " = " + answer;
30              JOptionPane.showMessageDialog(null, message);
31          }
32          catch (NumberFormatException nfe) {
33              JOptionPane.showMessageDialog(null,
34                  "Please input whole numbers");
35          }
36          catch (ArithmeticException ae) {
37              JOptionPane.showMessageDialog(null,
38                  "Cannot divide by zero");
39          }
40      }
```

Here the “catch” block at lines 32 – 35 has been triggered:


Exceptions demo 3

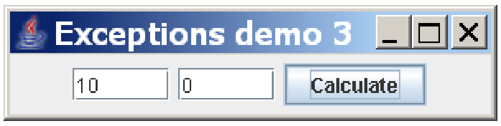
45

nine


Message

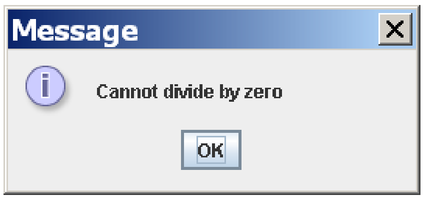
Please input whole numbers

and here the “catch” block at lines 36 – 39 has been triggered:


Exceptions demo 3

10

0


Message

Cannot divide by zero

There are over a hundred different exception types (classes) in the Java API and programmers may easily create new ones. For example in an application to do with airline seat reservations, you might like to have an exception type `NotEnoughSeatsException`.

The most common exception types are listed below:

`NullPointerException`: This happens if you try to use a non-instantiated object (i.e. one with value `null`). This usually indicates a bug in your program.

`NumberFormatException`: as in the above example.

`ArithmeticException`: as in the above example.

`ArrayIndexOutOfBoundsException`: An attempt has been made to access a non-existent array element. This usually indicates a bug in your program, but the next example shows a “legitimate” use.

`IOException` (and subtypes thereof): Something has gone wrong when you try to perform an input or output operation on a file.

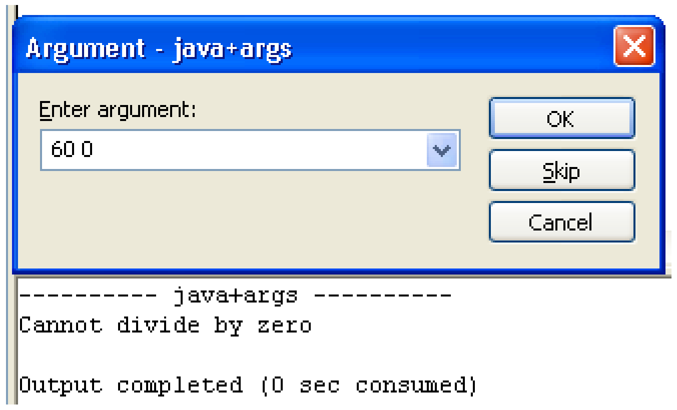
`SQLException` (and subtypes thereof): Something has gone wrong when you try to access a database.

NB. When working with files (databases), you *must* catch `IOException` (`SQLException`), as we shall see later on in this section and in section 14.

Exceptions4.java

Here is a command-line version of the previous example:

```
class Exceptions4
{
   public static void main(String[] args) {
         try {
            int x = Integer.parseInt(args[0]);
            int y = Integer.parseInt(args[1]);
            System.out.println(x + " / " + y + " is " + (x / y));
        }
        catch (ArrayIndexOutOfBoundsException aie) {
            System.out.println("Two arguments needed");
        }
        catch (NumberFormatException nfe) {
            System.out.println("The arguments must be whole numbers");
        }
        catch (ArithmeticException ae) {
            System.out.println("Cannot divide by zero");
        }
   }
}
```



```
---------- java+args ----------
Cannot divide by zero

Output completed (0 sec consumed)
```

This application catches three different exception types. It shows a “legitimate” use of `ArrayIndexOutOfBoundsException`. If the user doesn’t supply any arguments, or supplies only one, then one or both of `args[0]`, `args[1]` will be non-existent and this exception will occur.

Does the order of the “catch” blocks matter?

Normally, no. The blocks can come in any order. BUT if one exception type is a *subtype (subclass)* of another, it *must* come first.

If you are worried that you have forgotten about some exception types, then you could include this “catch” block at the end:

```
catch (Exception ex) {
    JOptionPane.showMessageDialog(null, "Exception: " + ex);
    ex.printStackTrace();
}
```

File handling in Java

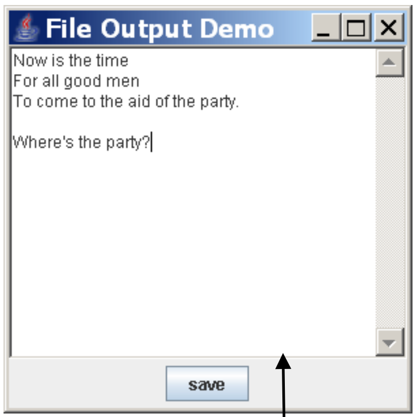
There are many classes for I/O in package **java.io**. There are also classes to handle communication across networks in package **java.net**.

The first example shows how to *write* a text file:

FileDemo1

NB the Swing component `JTextArea` has more functionality than the AWT `TextArea` but is more difficult to use.

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

class FileDemo1 extends JFrame implements ActionListener {
    private TextArea inputTextArea = new TextArea();
    private JButton saveButton = new JButton("save");
    private FileWriter outFile;

    public static void main(String[] args) {
        new FileDemo1();
    }

    public FileDemo1() {
        setSize(300, 300);
        setTitle("File Output Demo");
        add("Center", inputTextArea);
        JPanel bottom = new JPanel();
        bottom.add(saveButton);
        add("South", bottom);
        saveButton.addActionListener(this);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent evt) {
        if (evt.getSource() == saveButton) {
            try {
                outFile = new FileWriter("testout.txt", true);
                outFile.write(inputTextArea.getText());
                outFile.close();
            }
            catch (IOException e) {
```



Now is the time
For all good men
To come to the aid of the party.

Where's the party?

A horizontal scrollbar appears here automatically if it is needed.

```
35                  System.err.println("File Error: " + e);
36                  System.exit(1);
37              }
38          }
39      }
40 }
```

Explanation of the code

Line 01: This imports the java *input-output package* and is necessary for all file handling (except for console I/O `System.out` and `System.in`)

Line 09: This declares a `FileWriter` object. This is a simple class for creating text files. It works on a character-by-character basis, and it has a useful method `write(String s)` that writes a complete string (including newline characters, etc) used at line 31.

Lines 29 – 37: When using files you *must* catch `IOException` or one or other of its subtypes, the most common being `FileNotFoundException` (fairly obvious!) and `EOFException` (attempting to read past the end of an input file)

Line 30 opens `outFile` for output. The argument `true` means output in *append* mode – if the file already exists, any new output will go at the end. If this argument is `false`, or missing, any existing file will be erased, so take care!

Line 31 takes the text typed into the text area and writes it to `outFile`.

Line 32 closes the file.

Lines 34 – 37 As mentioned, you *must* catch `IOException`.

Self-test exercises

2. What do you think would happen if you didn't close the file – i.e. if line 32 was missing?

3. Suppose you replace line 30 with

```
30                  outFile = new FileWriter("B:testout.txt", true);
```

What might happen? (There is no `B:` drive on our network.) What if you replaced `B:` with `T:` (`T:` being the CMS network packages drive)?

4. (In the lab) Run this program and check that clicking the save button *appends* text to file `testout.txt`. Change line 30 to

```
30                  outFile = new FileWriter("testout.txt");
```

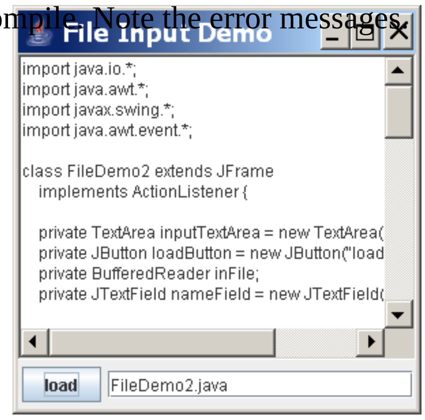
Re-compile and re-run the program and note the difference – file `testout.txt` only contains the latest text saved.

5. (In the lab). Delete lines 29, 33 – 37 and recompile. Note the error messages.

The second example shows how to *read* a text file.

FileDemo2

```
01 import java.io.*;
02 import java.awt.*;
03 import javax.swing.*;
04 import java.awt.event.*;
05
```



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

class FileDemo2 extends JFrame
   implements ActionListener {

   private TextArea inputTextArea = new TextArea(
   private JButton loadButton = new JButton("load
   private BufferedReader inFile;
   private JTextField nameField = new JTextField(
```

FileDemo2.java

```
06 class FileDemo2 extends JFrame
07     implements ActionListener {
08
09     private TextArea inputTextArea = new TextArea();
10     private JButton loadButton = new JButton("load");
11     private BufferedReader inFile;
12     private JTextField nameField = new JTextField(20);
13
14     public static void main (String [] args) {
15         new FileDemo2();
16     }
17
18     public FileDemo2() {
19         setSize(300, 300);
20         setTitle("File Input Demo");
21         add("Center", inputTextArea);
22         JPanel bottom = new JPanel();
23         bottom.add(loadButton);
24         bottom.add(nameField);
25         add("South", bottom);
26         loadButton.addActionListener(this);
27         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
28         setVisible(true);
29     }
30
31     public void actionPerformed(ActionEvent evt) {
32         if (evt.getSource() == loadButton) {
33             String fileName;
34             fileName = nameField.getText();
35             try {
36                 inFile = new BufferedReader(
37                              new FileReader(fileName));
38                 inputTextArea.setText(""); // clear the input area
39                 String line;
40
41                 while( ( line = inFile.readLine() ) != null) {
42                     inputTextArea.append(line + "\n");
43                 }
44                inFile.close();
45             }
46             catch (IOException e) {
47                 JOptionPane.showMessageDialog(this, "File not found");
48                 nameField.setText("");
49             }
50         }
51     }
52 }
```

Explanation of the code

Line 11: Unfortunately the `FileReader` class, which corresponds naturally to the `FileWriter` class, and which also works on a character-by-character basis, does not have a `read` method to return the entire file contents as a `String`. Instead we use a `BufferedReader` to read the file a line at a time. *Buffering* means that text is read ahead in chunks suitable for typical input devices (usually discs) and supplied as needed to the program. This is efficient.

Lines 35 – 49: Again we need

```
try { ... } catch (IOException e) { ... }
```

for the code which works with the file.

Lines 36, 37: A new `FileReader` is created, then this is used to create a new `BufferedReader`.

Lines 41 – 43: It is usual to work with buffered readers on a line-by-line basis, using a `while` loop like this. When the `readLine()` method can't find any more data it returns `null`, thus ending the loop. At line 42 the `append(String s)` method of the `TextArea` class simply appends the string to the text displayed. It is important not to forget to add a newline character here as the string returned by `readLine()` does *not* have one.

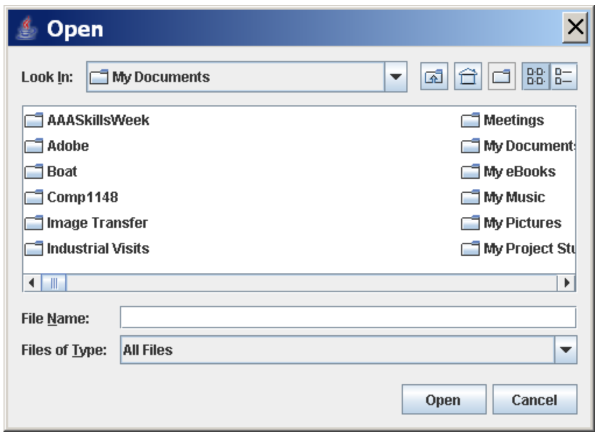
File dialogs

Software running in Microsoft Windows and similar systems normally provides the user with *file dialogs* which browse the available folders. The third example here is a version of the second example which uses the `JFileChooser` class to open a file dialog. Only the relevant lines of code are shown here.

FileDemo3

```
...
27     public void actionPerformed(ActionEvent evt) {
28         if (evt.getSource() == loadButton) {
29             JFileChooser chooser = new JFileChooser();
30             chooser.showOpenDialog(this);
31             File file = chooser.getSelectedFile();
32             try {
33                 inFile = new BufferedReader(new FileReader(file));
...
```

When line 29 is executed the following dialog box pops up:

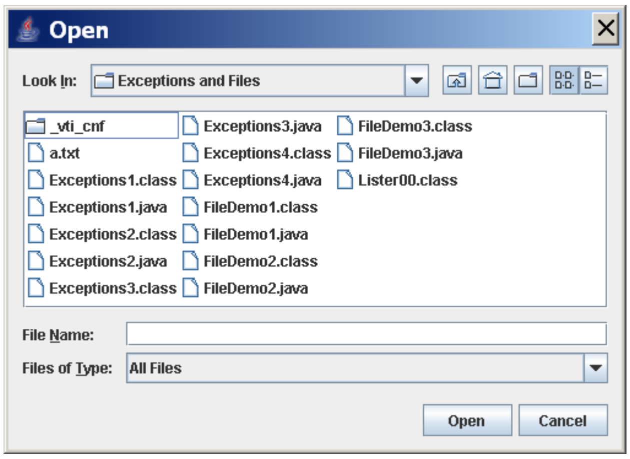

Open

Look In: My Documents

File Name:

Files of Type: All Files

In Windows the `showOpenDialog` method defaults to the **My Documents** folder. Browsing to `J:\Java\Exceptions and Files` (in the usual Windows manner) gives:

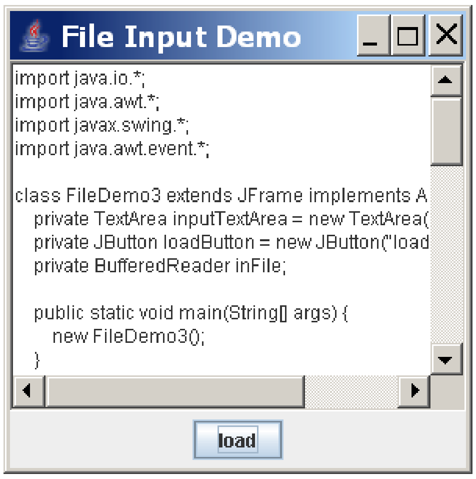


Look In: Exceptions and Files

Files of Type: All Files

Clicking on `FileDemo3.java` and Open, or just double-clicking on `FileDemo3.java` returns a `File` object at line 31 and this is used to create the buffered reader at line 33. (If the user clicks Cancel, `null` is returned at line 31 and a `NullPointerException` happens.)

Note: The `JFileChooser` class is powerful enough to mimic any Windows file dialog. You can use *file filters* to restrict access to specific file types such as `*.txt` and `*.java` and start browsing from other folders. See the Java documentation for details of how to do this – it's a bit fiddly.



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

class FileDemo3 extends JFrame implements A
   private TextArea inputTextArea = new TextArea(
   private JButton loadButton = new JButton("load
   private BufferedReader inFile;

   public static void main(String[] args) {
      new FileDemo3();
   }
```

The next example combines text input and output using a `BufferedReader` and a `PrintWriter`. It is a command line utility program to add line numbers to a file:

Lister00

```
01 import java.io.*;
02 import java.text.DecimalFormat;
03
04 class Lister00 {
05     public static void main(String[] args) {
06         DecimalFormat d2 = new DecimalFormat("00 ");
07         try {
08             BufferedReader inFile =
09                 new BufferedReader(new FileReader(args[0]));
10             PrintWriter outFile =
11                 new PrintWriter(new FileWriter(args[1]));
12             String line;
13             int lineNumber = 0;
14             while ((line = inFile.readLine()) != null) {
15                 lineNumber++;
16                 outFile.println(d2.format(lineNumber) + line);
17             }
18             inFile.close();
19             outFile.close();
20         }
21         catch (ArrayIndexOutOfBoundsException ae) {
22             System.out.println("I need an input and output file!");
23         }
24         catch (IOException ioe) {
25             System.out.println("File error");
26         }
27     }
28 }
```

Lines 08 – 11: This program uses a `BufferedReader` for input and a `PrintWriter` for output. `PrintWriter`s are usually created from `FileWriter`s.

Line 13: A counter `lineNumber` is declared and initialised to 0.

Lines 14 – 17: This `while` loop reads the input file a line at a time. For each line read `lineNumber` is incremented by 1 and a line is printed to the output file, consisting of `lineNumber` formatted to 2 digits plus a space, followed by the line read.

The code listing shown above shows file `Lister00.txt` created by the command

```
J:\Java\Exceptions and Files>java Lister00 Lister00.java Lister00.txt
```

(Many of the code listings in these notes were produced by this simple application).

Self-test Exercise

6. What modification(s) is (are) required to list files with 100 to 999 lines, needing three digit line numbers instead of two?

Network programming in Java (optional)

Network programming in Java is relatively straightforward once you have mastered basic text I/O.

Here is a remarkable little program that connects to any valid Web URL and displays its HTML – in effect a “tiny browser”:

TinyBrowser

```
01 import java.io.*;
02 import java.net.*;
03 import javax.swing.*;
04
05 class TinyBrowser {
06     public static void main(String[] args) {
07         String urlString = "";
08         try {
09             String line;
10             urlString =
11                 JOptionPane.showInputDialog
12                 ("Type a URL address (e.g http://java.sun.com/) :");
13             URL urlAddress = new URL(urlString);
14             URLConnection link = urlAddress.openConnection();
15              BufferedReader inStream =
16                  new BufferedReader
17                  (new InputStreamReader(link.getInputStream()));
18              while ((line = inStream.readLine()) != null) {
19                  System.out.println(line);
20              }
21          }
22          catch (MalformedURLException me) {
23              System.err.println(me);
24              System.exit(2);
25          }
26          catch (IOException ioe) {
27              System.err.println("Error in accessing URL: " + ioe);
28              System.exit(1);
29          }
30      }
31  }
```

Lines 01, 02: We need packages `java.io` and `java.net` for network programming.

Line 13: We attempt to create a URL object from user input. This may give rise to a `MalformedURLException`. Suppose the user typed `wally` for the URL address:

```
java.net.MalformedURLException: no protocol: wally
```

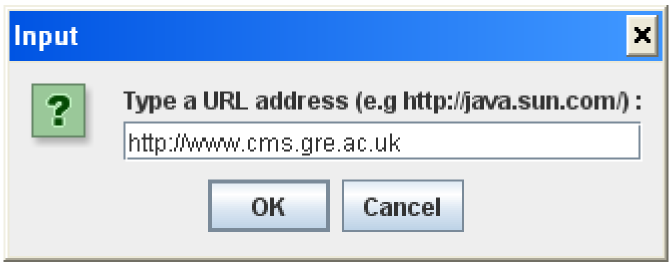
Line 14: This creates a `URLConnection` from the (well-formed) `URL` object.

Lines 15 - 17: This shows the power of the Java network library. We create an `InputStreamReader` from the `URLConnection` and then use this to create a `BufferedReader`. If the URL is not recognised an IO exception occurs. Suppose the user typed `http://wally.com` for the URL address:

```
Error in accessing URL: java.net.UnknownHostException: wally.com
```

(When this workbook was written originally there was no such web site, although there is one now.)

Lines 18 – 20: `while` loop to read the contents of the buffered reader obtained from the URL and display it to the console. Here are the first few lines of output when we type the URL for the CMS School home page:



Type a URL address (e.g http://java.sun.com/) :

http://www.cms.gre.ac.uk

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html lang="en">
<head>
<!-- #BeginEditable "doctitle" -->
<title>the University of Greenwich> Computing & Mathematical Sciences</title>
```

Client / Server programming

We finish with an example of *client / server* programming. One program – the *server* – runs on one machine and waits for *client* programs to connect to it. The server and clients usually run on different machines on a network, but in this simple example they will both run on one machine.

TCPServer

```
import java.io.*;
import java.net.*;

class TCPServer {
    public static void main(String[] args) throws Exception {
        String s, capS;
        ServerSocket ss = new ServerSocket(4000);
        while (true) {
            Socket cs = ss.accept();
            BufferedReader in =
                new BufferedReader
                (new InputStreamReader(cs.getInputStream()));
            DataOutputStream out =
                new DataOutputStream(cs.getOutputStream());
            s = in.readLine();
            capS = s.toUpperCase() + '\n';
            out.writeBytes(capS);
        }
    }
}
```

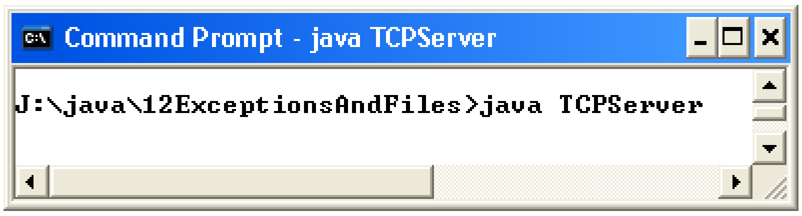
TCPClient

```
import java.io.*;
import java.net.*;

class TCPClient {
    public static void main(String[] args) throws Exception {
        String s, modS;
        BufferedReader inC =
            new BufferedReader(
            new InputStreamReader(System.in));
        Socket cs = new Socket("localhost", 4000);
        DataOutputStream out =
            new DataOutputStream(cs.getOutputStream());
        BufferedReader inS =
            new BufferedReader
            (new InputStreamReader(cs.getInputStream()));
        System.out.println("Type a line to send to the server");
        s = inC.readLine();
        out.writeBytes(s + '\n');
        modS = inS.readLine();
        System.out.println("FROM SERVER: " + modS);
        cs.close();
    }
}
```

To demonstrate this clearly it's best to run the programs from a command shell rather than EditPlus or NetBeans.

To run the program, open a Java Command Promt. Then change folder (directory) to the one where the **class** files are stored, e.g. type "`J:`" and then "`cd java\12ExceptionsAndFiles`". Finally, type run a class file, e.g. MyClass.class, with "`java MyClass`".

To run this demo, open *two* command shells. Run `TCPServer` in one shell first:

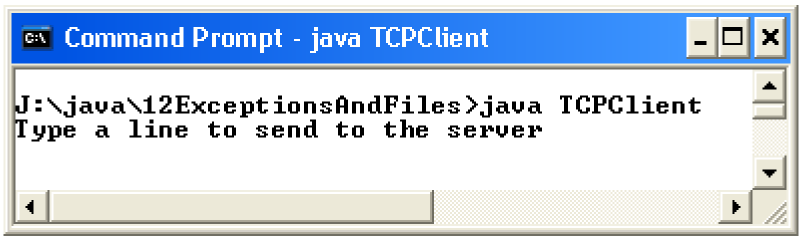


```
J:\java\12ExceptionsAndFiles>java TCPServer
```

The server first creates a new *socket* (line 07). Think of a socket as a two-way communication channel, rather like a telephone line. It is created on the computer's *port* 4000 ("localhost" refers to the machine on which the program is running).

The server enters a never-ending loop waiting for a client to attach to it (line 09).

In the second shell, run `TCPClient`:

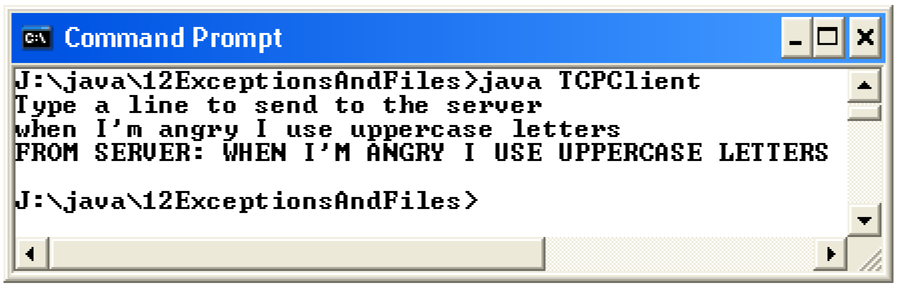


```
J:\java\12ExceptionsAndFiles>java TCPClient
Type a line to send to the server
```

The client sets up a buffered reader to read a line from the console (lines 07 – 09), creates a socket (line 10) to connect to the server on the same machine and port and gets the data output stream and buffered reader from this socket. (data output streams are used for writing streams of bytes) (lines 11 – 15). It then reads a line from the keyboard and sends it to the server (lines 16 – 18).

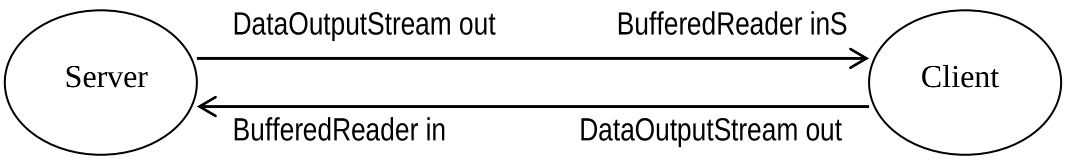
At this point the server has accepted the connection at line 09, creates its own buffered reader and data output stream, reads the line sent by the client, turns it into upper case and sends it back to the client.

The client then reads this response and displays it on its console (lines 19 – 21).



```
J:\java\12ExceptionsAndFiles>java TCPClient
Type a line to send to the server
when I'm angry I use uppercase letters
FROM SERVER: WHEN I'M ANGRY I USE UPPERCASE LETTERS

J:\java\12ExceptionsAndFiles>
```

This may be a little confusing at first sight. The socket connection is a two-way communication channel. The `DataOutputStream` for the server becomes the `BufferedReader` for the client, and vice-versa.



Line 05 in both programs shows a lazy way of dealing with exceptions. When we add the clause `throws Exception` to the `main` method the Java compiler no longer requires us to use exception handling. In general this should be avoided!

There is a problem with this example. The server can only handle one client at a time. If you run the server, then run two clients in separate command shells, the first client must finish before the second one. To overcome this the server needs to employ *multithreading*, which is beyond the scope of this introduction to Java.

Self-test exercises

7. What do you expect would happen if you started the client running before the server?

8. If you do the following:

   - Run the server in one shell
   - Run the client in a second shell but DON'T type a line to send yet
   - Run the client again in a third shell, this time typing a line to send

   you will find that the third shell 'hangs' with no response from the server. If you then go back to the second shell and type a line to send you will find that the server seems to respond to both clients. Explain this behaviour.

Summary

- Exceptions in Java are things that can go wrong with your program as it runs, but from which it is possible to recover.
- Exceptions are handled by means of `try` and `catch` blocks. If an exception happens inside a `try` block, which is followed by a `catch` block for that type of exception, then control passes to that block instead of the program stopping with an exception report
- Some of the most common exception types are:

  ```
  NullPointerException
  NumberFormatException
  ArithmeticException
  ArrayIndexOutOfBoundsException
  IOException
  SQLException
  ```

- File handling is carried out using the `java.io` package and usually requires the programmer to handle `IOException`.
- Network programming (optional at Level 1) in Java is carried out using the `java.net` and `java.io` packages and is no more complex than file handling.

Solutions to self-test exercises

1. As you might expect, the Java compiler complains:

   ```
   'try' without 'catch' or 'finally'
   ```

(a `finally` block is always executed, whether or not exceptions occur. They are not used very often.)

2. Some of the output may be lost. This is because I/O to disc is usually buffered, physically taking place only when a full disc sector has been generated. You should always close files (and databases!) before ending a program.

3. In both cases you get a type of `IOException`. In the first case it is a `FileNotFoundException`, in the second case a `FileNotFoundException` qualified by `"Access is denied"` because you have read-only access to `T:`

4. (in lab)

5. (in lab). The compiler gives an error message for each of lines 30 – 32, namely `"unreported exception java.io.IOException"`.

6. This is trivial! Change `"00 " to "000 "` in line 06. A more challenging exercise is to output a single digit for files with < 10 lines, two digits for files with 10 to 99 lines, three digits for files with 100 to 999 lines and so on. We shall do this later.

7. The client expects the server to be running first. If not a `ConnectException` (Connection refused) occurs and the client crashes.

8. As written, the server can only handle one client at a time. So the second client must wait until the server has finished with the first one.

   Most servers (e.g. web servers) are written to handle a number of simultaneous clients. This requires *multi-tasking*, done in Java by means of *threads*. The server creates a separate thread to handle each client request as it comes in.

   To stop the server in this simple example, type Ctrl+C in its shell. Web servers only stop by accident!

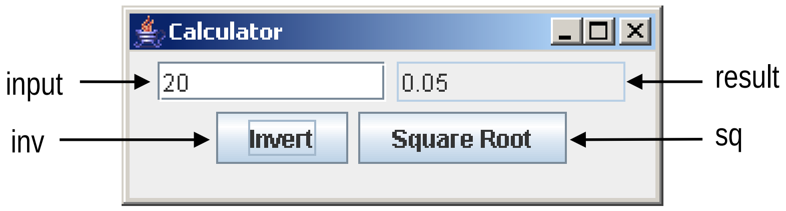
12. Testing programs

The actual coding of a program covers only a small part of the whole life of any software product. One of the other important phases of the software development cycle is **software testing**. The aim of testing is two-fold:

- To investigate whether the system meets its given specifications.
- To expose defects before the system is delivered.

Strangely, testing is about demonstrating the *presence*, not the *absence* of program faults! One of the most difficult tasks for a software developer is to prove the correctness of a software system. As in any other engineering discipline, rigorous testing can expose problems and so increase the reliability of a system. Just think how much testing goes into the release of a new car model. A software system can be quite complex as well as critical for the safety of people.

Let us consider a Java application (`Calculator.java`) that performs some simple calculations:



This program tries to take a number as input and either *invert* it or find its *square root*. The code behind the command buttons is:

```
public void actionPerformed(ActionEvent e) {
   int inp = Integer.parseInt(input.getText());
   if (e.getSource() == inv) {
      if (inp == 0) complain(); // can't invert 0
      else invertCalc(inp);
   }
   if (e.getSource() == sq) {
      if (inp < 0) complain(); // can't find sqrt of -ve number
      else sqrtCalc(inp);
   }
}
```

where the auxiliary methods are:

```
void invertCalc(int i) {
   result.setText("" + 1.0/i);
   // "" + 1.0/i will convert 1.0/i to a string
}

void sqrtCalc(int i) {
   result.setText("" + Math.sqrt(i));
}

void complain() {
   result.setText("Error!");
}
```

Now let us devise a series of tests to pinpoint possible mistakes in the code. (You may be able to spot some already!)

There are two main formal approaches to testing software.

- ***Functional*** or ***Black-box*** testing where the tests are derived from the program specification. The software is treated as a "black box" with inputs producing outputs. The tests are devised to explore various inputs and corresponding outputs, *without any knowledge of the workings of the program itself*. (Figure 1)
- ***Structural*** or ***White-box*** testing where the tests are derived *from the knowledge of the program code*. This knowledge allows the testing of the various paths that the program execution can take. (Figure 2)

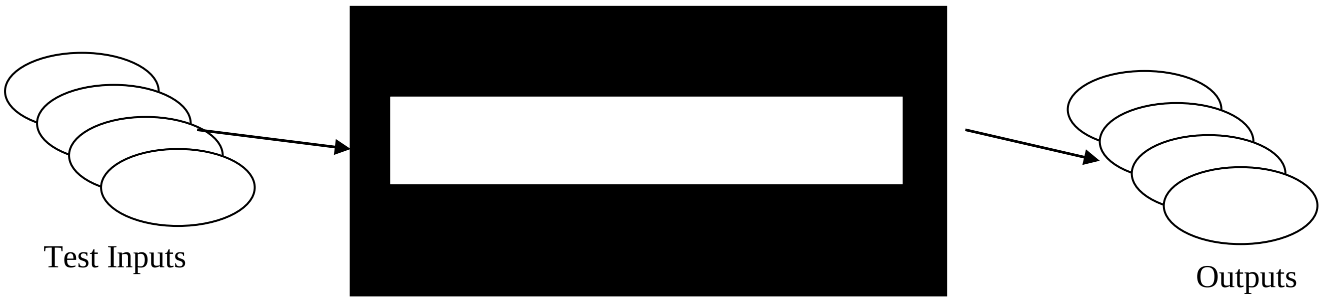


Figure 1 Black-box testing

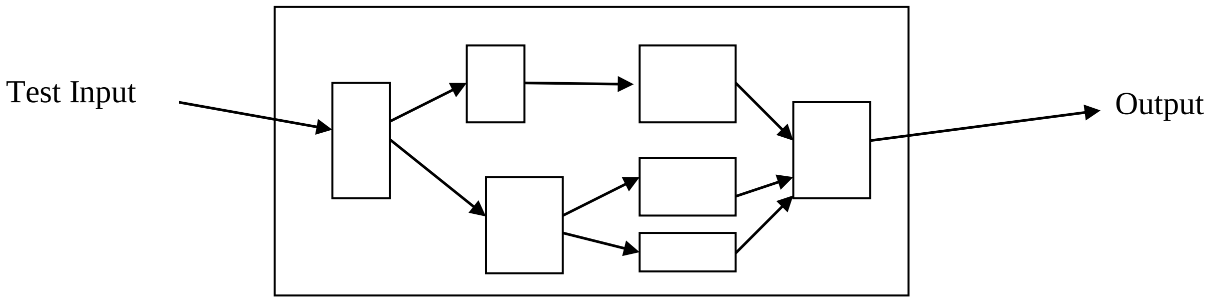


Figure 2 White-box testing

Black box testing

Let us now perform some basic Black-box testing on the simple calculator program. Our task is to prepare a set of inputs to test the program thoroughly. At this stage, we know nothing about the internal structure of the program. Our aim is to identify sets of tests that are more likely to produce incorrect outputs. It is usually the case that the developers would have tested most typical inputs, so we have to identify the most unusual families of inputs. As we cannot test all the possible inputs exhaustively, we have to identify classes of inputs that we hope will all make the program behave in a similar way. Assuming that the input is always a valid number (no strings or misspellings), we can identify the following classes:

- Positive Integers
- Negative Integers
- Positive real numbers
- Negative real numbers
- 0

Within each class, we pick some random numbers for our tests, with particular preference to extreme cases (such as very large or very small numbers).

Such a set could be:

- 1, 2, 59, 10000, 9999999999
- -1, -3, -34, -120000, -1000000
- 0.00001, 0.3333, 1.5, 10.7, 5.0E+6
- -0.00001, -2.5, -10.003, -1.e+5
- 0, 0.0

Note that we use very small and very large numbers within every class. This black-box testing process is very similar to test driving a car in extreme conditions to test its limits.

Black-box testing reveals more than one defect, as it stands.:

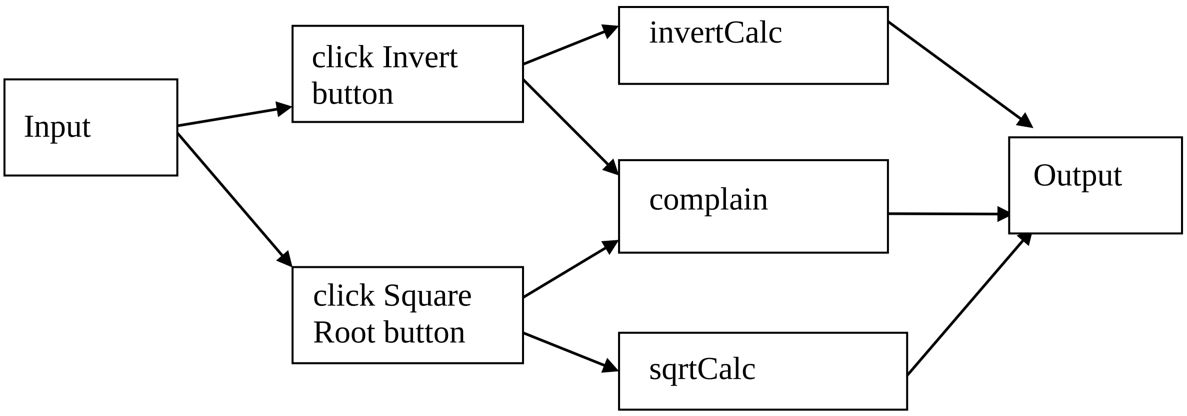
- The program cannot deal with large input numbers. (e.g. 9999999999)
- The program does not deal properly with real numbers. (e.g. 1.5)

In both cases an exception happens.

The error is in the use of ***int*** *variables and formal parameters – we should be using* ***double*** *instead.*

White-box testing

To white-box test the same program, we must consider the different paths that execution can follow (The program's structure). This is:



To devise a set of white-box tests, we try to make sure that each branch point (i.e. each `if` / `else if` decision) in the program is tested both ways. For this simple program, we can exhaust all the possible (four) paths within the program:

• Invert Non zero numbers	e.g.	input 0.5
• Invert zero (should complain)		input 0
• Calculate the square root of non-negative numbers		input 4.0
• Calculate the square root of negative numbers (should complain)		input -4.0

White box and black box testing are complementary to each other. They can lead to a rigorous approach to testing software for defects. ***From the programmer's viewpoint white box testing is obviously best as she/he knows the code.*** However it is well known that programmers often fail to spot errors in their programs (because they don't want to be proved wrong). Someone carrying out black-box testing really wants to show that the program doesn't work!

Many (but not all) of the programs we have looked at so far can be exhaustively white-box tested. In general though, it is impossible to exhaustively test a program.

For instance, how can we exhaustively test the Lottery program introduced in Chapter 10? This makes use of random numbers and there are hundreds of millions of different outputs.)

The *very least* you should do is:

- Make sure that all code is tested at least once!
- Make sure that each branch (***if*** or loop) is tested both ways.
- Bear in mind that a test such as ***if (a && b) …*** has three branches –
  - ***a = false***,
  - ***a = true*** but ***b = false***,
  - ***a = b = true***.
- Similarly ***if (a || b) …*** has three branches –
  - ***a = true,***
  - ***a = false*** but ***b = true,***
  - ***a = b = false***
- A ***switch …*** has as many branches as it has ***case***s, plus one if it doesn't have a ***default***.
- A ***try { … } catch …*** exception handler has as many branches as it has ***catch***es, plus one for normal execution with no exceptions raised.
- Loops can be tricky, but usually you need at least two tests – one 'normal' case where the code in the loop is executed at least once, and one where the code isn't executed at all.

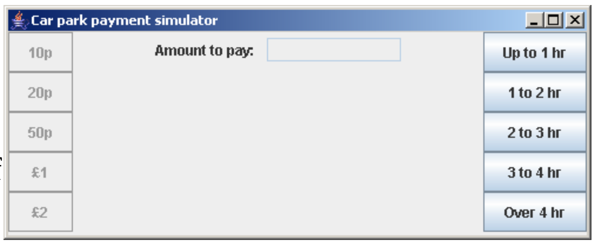
Self-test exercise

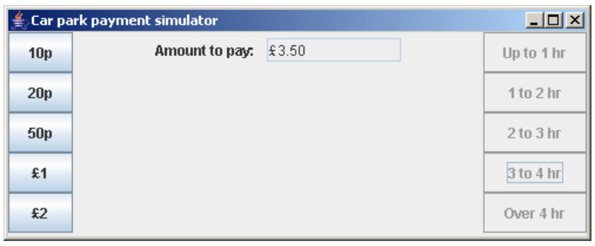
1. Give an example of a loop which might not execute at all. (Consider how you read a text file).

Testing a car park simulator

Consider a simple car park ticket machine simulator. The first screenshot shows the initial state. The user clicks one of the buttons on the right to select the number of hours required. The charges are:

Hours	Amount payable
0 - 1	£1.00
1 - 2	£2.00
2 - 3	£3.50
3 - 4	£5.00
4+	£6.00

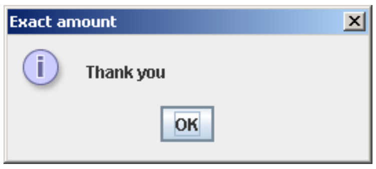




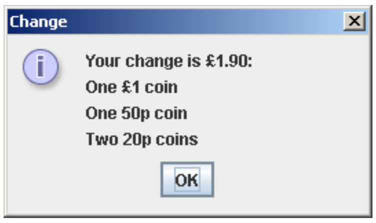
The second screenshot shows the state after the user has selected a number of hours (2 to 3 hr here). The amount to pay is displayed in a text box.The user then clicks the buttons on the left until the exact amount or more is entered. The program should respond "Thank you" if the exact amount was entered, or with a message stating the change using the minimum number of coins if more money was entered.

In the download you will find two proposed solutions, `CarPark1.java` and `CarPark2.java`. They both attempt to meet the above specification.

Black box testing



Thank you



Your change is £1.90:
One £1 coin
One 50p coin
Two 20p coins

All we have to go on is the specification given above.

The GUI has ten buttons, each of which must be clicked at least once during testing.

The change given may be anything from £0.00 (no change, program should respond as shown in the first screenshot opposite) to £1.90 (e.g. if the user selects up to 1 hour, then enters 50p, 2x20p, £2 the program should respond as shown in the second screenshot opposite), in steps of 10p. As we have no idea how the change is calculated, we need to test each of these eventualities – making 20 tests in all. With care, we can make sure that all the buttons are clicked during these 20 test runs.

A minimal "test plan" is given below. (Many others would fit the bill.) In practice we might want to include extra tests.

Test case	Expected output
01. Up to 1 hr, £1	Exact amount
02. 1 to 2 hr, 10p, £2	Change £0.10: 1 x 10p
03. 1 to 2 hr, 20p, £2	Change £0.20: 1 x 20p
04. 1 to 2 hr, 10p, 20p, £2	Change £0.30: 1 x 20p, 1 x 10p
05. 1 to 2 hr, 20p, 20p, £2	Change £0.40: 2 x 20p
06. 2 to 3 hr, £2, £2	Change £0.50: 1 x 50p
07. 2 to 3 hr, 10p, £2, £2	Change £0.60: 1 x 50p, 1 x 10p
08. 2 to 3 hr, 20p, £2, £2	Change £0.70: 1 x 50p, 1 x 20p
09. 2 to 3 hr, 10p, 20p, £2, £2	Change £0.80: 1 x 50p, 1 x 20p, 1 x 10p
10. 2 to 3 hr, 20p, 20p, £2, £2	Change £0.90: 1 x 50p, 2 x 20p
11. 2 to 3 hr, 50p, £2, £2	Change £1.00 : 1 x £1
12. 3 to 4 hr, 10p, £2, £2, £2	Change £1.10: 1 x £1, 1 x 10p
13. Over 4 hr, 20p, £1, £2, £2, £2	Change £1.20: 1 x £1, 1 x 20p
14. Over 4 hr, 10p, 20p, £1, £2, £2, £2	Change £1.30: 1 x £1, 1 x 20p, 1 x 10p
15. Over 4 hr, 20p, 20p, £1, £2, £2, £2	Change £1.40: 1 x £1, 2 x 20p
16. Over 4 hr, 50p, £1, £2, £2, £2	Change £1.50: 1 x £1, 1 x 50p
17. Up to 1 hr, 10p, 50p, £2	Change £1.60: 1 x £1, 1 x 50p, 1 x 10p
18. Up to 1 hr, 20p, 50p, £2	Change £1.70: 1 x £1, 1 x 50p, 1 x 20p
19. Up to 1 hr, 10p, 20p, 50p, £2	Change £1.80: 1 x £1, 1 x 50p, 1 x 20p, 1 x 10p
20. Up to 1 hr, 20p, 20p, 50p, £2	Change £1.90: 1 x £1, 1 x 50p, 2 x 20p

White box test runs

For this we need to see the code! Here the relevant code from `CarPark1.java` is:

```
    ...
    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == less1) { amount = 100; leftButtons(); }
        if (e.getSource() == oneTwo) { amount = 200; leftButtons(); }
        if (e.getSource() == twoThree) { amount = 350; leftButtons(); }
        if (e.getSource() == threeFour) { amount = 500; leftButtons(); }
        if (e.getSource() == overFour) { amount = 600; leftButtons(); }
        if (e.getSource() == tenP) amount -= 10;
        if (e.getSource() == twentyP) amount -= 20;
        if (e.getSource() == fiftyP) amount -= 50;
        if (e.getSource() == onePound) amount -= 100;
        if (e.getSource() == twoPounds) amount -= 200;

        if (amount > 0) message.setText(pounds.format(amount / 100.0));
        else {
            message.setText("");
            if (amount < 0) {
                int change = -amount;
                JOptionPane.showMessageDialog(this, "Your change is "
                    + pounds.format(change / 100.0)
                    + coins(change),
                    "Change", JOptionPane.INFORMATION_MESSAGE);
            } else {
                JOptionPane.showMessageDialog(this, "Thank you",
                    "Exact amount", JOptionPane.INFORMATION_MESSAGE);
            }
            rightButtons();
        }
    }
    ...
    String coins(int change) {
        String answer = ":";
        if (change >= 100) {
            answer += "\nOne £1 coin";
            change -= 100;
        }
        if (change >= 50) {
            answer += "\nOne 50p coin";
            change -= 50;
        }
        if (change >= 40) {
            answer += "\nTwo 20p coins";
            change -= 40;
        }
        if (change >= 20) {
            answer += "\nOne 20p coin";
            change -= 20;
        }
        if (change >= 10) {
            answer += "\nOne 10p coin";
            change -= 10;
        }
        return answer;
    }
}
```

This is taken from the version `CarPark1.java` which stores `amount` as an `int` representing the number of pence required.

The remaining code consists of the `CarPark1` constructor that sets up the GUI and the methods `leftButtons` and `rightButtons` to set the enabled / disabled status of the buttons (see the first two screenshots). As this code has no branches it is only necessary to make sure that it is executed at least once. The constructor executes as soon as the application is run, and the other two methods are called from `actionPerformed`.

The ten `if` statements at lines 63 – 72 are exercised both ways if we make sure that all the buttons have been clicked during testing.

The `if` statements at lines 74 and 77 are exercised both ways if we have at least one test with an exact amount entered and one where change is required.

The `coins` method, which works out the coins needed for change, will be called whenever change is required.

We only need two runs to exercise the `if` statements at lines 121, 125, 129, 133, 137 both ways – one where £1.90 is required (the `ifs` at lines 121, 125, 129 succeeding and the other two failing) and one where £0.30 is required (the reverse being the case).

In fact only six of the above 20 test cases are required:

Test case	**Expected output**
01. Up to 1 hr, £1	**Exact amount**
04. 1 to 2 hr, 10p, 20p, £2	**Change £0.30: 1 x 20p, 1 x 10p**
06. 2 to 3 hr, £2, £2	**Change £0.50: 1 x 50p**
12. 3 to 4 hr, 10p, £2, £2, £2	**Change £1.10: 1 x £1, 1 x 10p**
13. Over 4 hr, 20p, £1, £2, £2, £2	**Change £1.20: 1 x £1, 1 x 20p**
20. Up to 1 hr, 20p, 20p, 50p, £2	**Change £1.90: 1 x £1, 1 x 50p, 2 x 20p**

In practice we might want to include a few more cases, in particular one or two more where the exact amount should have been entered.

Self-test exercise

2. Give a test case which should result in the exact amount being entered, and which involves the customer clicking *all* of the coins buttons at least once and the 10p button three times.

It is interesting to note that the version `CarPark1.java` passes all 20 black box tests whereas `CarPark2.java` gives incorrect results for six of them, two of which are in our white box test plan.

The bug in the second version is to do with using a `double` to represent the amount required in pounds, which runs the risk of slight round-off errors cropping up. For instance case 12 results in the change required being calculated as £1.0999999999999996 instead of the correct £1.10 and the customer is short-changed by 10p.

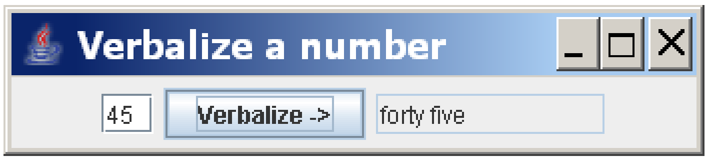
Also, the case (not chosen) where the customer asks for up to 1 hr and puts in ten 10p coins results in being asked to pay £0.00! (In fact the program thinks that an amount roughly 1.4 x 10^{-16} is required.) After adding another 10p coin the program generously returns this as change.

These bugs could have been fixed by using different comparison tests, but it is better to use exact arithmetic, i.e. use an `int` to represent the amount in pence instead. This is what `CarPark2.java` does.

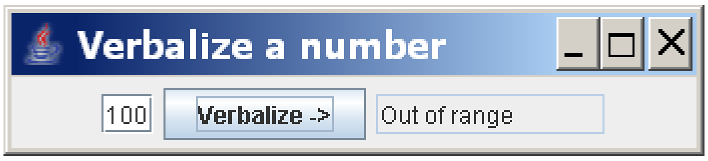
Verbalising numbers – white box testing

Many financial organisations automate the writing of cheques using appropriate software. This involves printing the amount both numerically and ‘linguistically’. For instance, suppose your car insurance company owes you £2399. The software will print this both as **£2,399.00** and as **two thousand three hundred and ninety nine pounds only**.

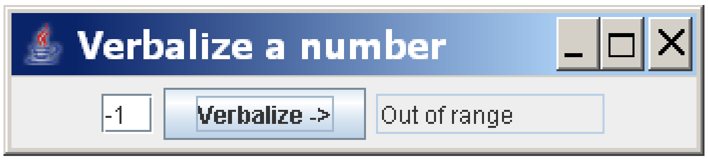
Here is a simpler program which tries to represent any number between 0 and 99 verbally:



45 forty five



100 Out of range



-1 Out of range

This is in the download as V99.java

and here is the code, omitting the GUI ‘housekeeping’.

```
...    ...
05 public class V99 extends JFrame implements ActionListener {
06     JTextField number = new JTextField(2); // input field
07     JTextField vnumber = new JTextField(10); // output field
08     JButton verbalize = new JButton("Verbalize ->");
09
10     static final String[] v19 =
11         {"zero", "one", "two", "three", "four",
12          "five", "six", "seven", "eight", "nine",
13          "ten", "eleven", "twelve", "thirteen", "fourteen",
14          "fifteen", "sixteen", "seventeen", "eighteen", "nineteen"};
15     static final String[] vty =
16         {"", "", "twenty", "thirty", "forty", "fifty",
17          "sixty", "seventy", "eighty", "ninety"};
18     // so v19[5] is "five" and vty[4] is "forty", etc
19
...    ...
35     public void actionPerformed(ActionEvent e) {
36         int n; // number to be verbalized
37         try {
38             n = Integer.parseInt(number.getText());
39         }
40         catch (Exception ex) {
41             number.setText("0");
42             vnumber.setText("zero");
43             return;
44         }
45          if (n < 0 || n > 99) vnumber.setText("Out of range");
46          else vnumber.setText(verbalize(n));
47      }
48
49      String verbalize(int n) {
50          if (n < 20) return v19[n];
51          else {
52              int tens = n / 10, units = n % 10;
53              return vty[tens] + " " + v19[units];
54          }
55      }
56 }
```

Lines 37 – 44: This try … catch block traps badly formatted numbers (it assumes the input is 0).

Lines 45, 46: This tests for numbers out of range – less than 0 or greater than 99, which produces output “Out of range”. For the remaining (100) inputs the output should be the verbalization of the input, as shown.

To white-box test this code we need:

- A badly formatted number, e.g. **fish** Expected outcome: **zero**
- A number < 0 e.g. -5 Expected outcome: **Out of range**
- A number > 99 e.g. 100 Expected outcome: **Out of range**

Other tests need to look at the `verbalise` method. This has just one test – ***if (n < 20) …*** To test this we need

- A number ≥ 0 but < 20 e.g. 10 Expected outcome: **ten**
- A number ≥ 20 but ≤ 99 e.g. 44 Expected outcome: **forty four**

`verbalise` makes use of the constant `String` arrays `v19` and `vty`. Clearly we need to test these out. It seems that an exhaustive test set (including the above five) would be:

Input	Expected result		Input	Expected result
fish	zero		18	eighteen
-5	Out of range		19	nineteen
-1	Out of range		22	twenty two
0	zero		33	thirty three
1	one		44	forty four
10	ten		55	fifty five
11	eleven		66	sixty six
12	twelve		77	seventy seven
13	thirteen		88	eighty eight
14	fourteen		98	ninety eight
15	fifteen		99	ninety nine
16	sixteen		100	Out of range
17	seventeen		105	Out of range

Self-test exercise

3. Spend a few moments checking that these would in fact test each of the `String`s in arrays `v19` and `vty`).

But sadly, testing is not that easy. This program has a bug, even though the above 26 inputs all give the expected output. Can you spot it?

It pays to look at *borderline* cases. Borderline cases here are -1, 0, 1, 98, 99, 100 because they are clustered around the test values in the `if` statement at lies 45, 46. But there are other borderline cases – numbers exactly divisible by 10. What about, say, 40? We expect output **forty** but instead we get **forty zero** – try it!

Clearly 0 (zero) is a special case. We can rewrite the code to fix this bug by isolating input 0. This leads us to the changes:

```
47     String verbalize(int n) {
48         // treat 0 as a special case
49         if (n == 0) return "zero";
50         else return v(n);
51     }
```

This isolates zero and introduces a new auxiliary method `v` to handle the other numbers.

this correct version is program V99OK.java from the download

```
53     String v(int n) {
54         if (n < 20) return v19[n];
55         else {
56             int tens = n / 10, units = n % 10;
57             if (units == 0) return vty[tens];
58             else return vty[tens] + " " + v19[units];
59         }
60     }
```

This tests (lines 57, 58) for a remainder of 0 on division by 10, treating 20, 30, … differently.

Test plans

As a programmer, you should prepare a set of white box test cases and document the outcome. For the original verbalisation program, this might appear as follows:

The following input set tests each branch in the program both ways. Borderline cases are -1, 0, 1, 98, 99, 100, 40. The latter test indicates a bug in the program.

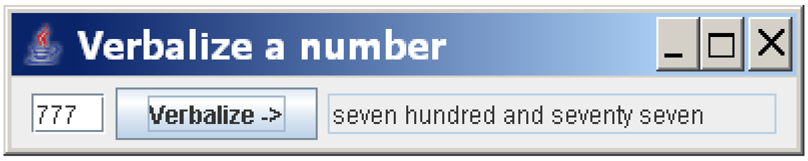
Test	**Input**	**Expected Output**	**Actual Output**
1	fish	zero	zero
2	-5	Number out of range	Number out of range
3	-1	Number out of range	Number out of range
4	0	zero	zero
5	1	one	one
6	10	ten	ten
7	11	eleven	eleven
8	12	twelve	twelve
9	13	thirteen	thirteen

10	14	fourteen	fourteen
11	15	fifteen	fifteen
12	16	sixteen	sixteen
13	17	seventeen	seventeen
14	18	eighteen	eighteen
15	19	nineteen	nineteen
16	22	twenty two	twenty two
17	33	thirty three	thirty three
18	**40**	**forty**	**forty zero**
19	44	forty four	forty four
20	55	fifty five	fifty five
21	66	sixty six	sixty six
22	77	seventy seven	seventy seven
23	88	eighty eight	eighty eight
24	98	ninety eight	ninety eight
25	99	ninety nine	ninety nine
26	100	Number out of range	Number out of range
27	105	Number out of range	Number out of range

The eighteenth test (input 40) documents a bug in the program. Even if you don't know how to correct it, it is good that it is documented. (Of course it would be better if you could fix the bug and re-test – but sometimes that may be difficult!)

Most commercial software systems are released with a list of such documented bugs. The theory is that users might still want to use (and pay for) a buggy system provided the bugs are well documented.

For this program, it is feasible (but a little tedious) to test exhaustively, with inputs -5, -1, 0 to 99, 100, 105. But this is out of the question if we extend the range to 999, as is the case with program `V999.java` in the download.



777

seven hundred and seventy seven

The out of range test becomes

```
        if (n < 0 || n > 999) vnumber.setText("Out of range");
```

and the method `v` is extended to

```
    String v(int n) {
        if (n < 20) return v19[n];
        else if (n < 100) {
            int tens = n / 10, units = n % 10;
            if (units == 0) return vty[tens];
            else return vty[tens] + " " + v19[units];
        }
        else {
            int hundreds = n / 100, remainder = n % 100;
            // e.g 345: hundreds = 3, remainder = 45
            if (remainder == 0) return v19[hundreds] + " hundred";
            else return v19[hundreds] + " hundred and " + v(remainder);
        }
    }
```

Note the *recursive* call of `v`. `remainder` is between 1 and 99 and `v` already knows how to handle this, so we can call it here to do the right job.

Testing auxiliary classes

To test a class we need to test each of its constructors and methods. We could do this as part of testing the whole program which uses the class, but what if the class is designed to be re-used (as is the case with all the Java library classes)? One solution is to write a *harness* class whose only job is to test the class in question.

Let us revisit the income tax calculator application discussed in Chapter 8. This application had an auxiliary class `TaxCalc.java` to perform the tax calculation. This class could be used in many applications (e.g. a payroll program) besides the example we used.

Here is a slightly different version of that class, which has a simple constructor and which carries out the calculation in methods `getTaxable()` and `getTax()` (the previous version did the calculation in the constructor):

```
class TaxCalc {
    // instance variables
    private double gross;
    private int nKids;
    private boolean married;

    // constructor
    TaxCalc(double gross, int nKids, boolean married) {
        this.gross = gross;
        this.nKids = nKids;
        this.married = married;
    }

    // get method to return taxable
    public double getTaxable() {
        // calculate personal allowance using private methods
        double taxable = gross - personalAllowance() - childAllowance();
        return taxable;
    }

    // get method to return tax
    public double getTax() {
        double taxable, remainder, tax;
        taxable = getTaxable();
        if (taxable <= 20000) // first band
            tax = 0.2 * taxable;
        else if (taxable <= 40000) { // second band
            remainder = taxable - 20000;
            tax = 4000 + 0.4 * remainder;
        }
        else {
            remainder = taxable - 40000;
            tax = 4000 + 8000 + 0.6 * remainder;
        }
        return tax;
    }
37
38     // private methods to help calculate the taxable income
39     private double personalAllowance() {
40         double pa;
41         if (married) pa = 3000;
42         else         pa = 2000;
43         return pa;
44     }
45
46     private double childAllowance() {
47         double ca = nKids * 500;
48         return ca;
49     }
50 }
```

To white-box test this we need to execute all the code and both branches of the `if` statements at lines 25, 27 and 41. We could use the original `TaxCalcApp.java`, but here is a simpler command-line harness program which has two advantages:

- It carries out a lot of tests with no user input
- It will be easy to add extra tests

```
import static java.lang.System.out;
class TestTaxCalc {
    public static void main(String[] args) {
        TaxCalc t;

        // first test - single, no kids, gross 20000
        t = new TaxCalc(20000, 0, false);
        out.println("Test 1 - tax should be 3600: " + t.getTax());

        // second test - married, no kids, gross 20000
        t = new TaxCalc(20000, 0, true);
        out.println("Test 2 - tax should be 3400: " + t.getTax());

        // third test - married, no kids, gross 23000 (borderline)
        t = new TaxCalc(23000, 0, true);
        out.println("Test 3 - tax should be 4000: " + t.getTax());

        // fourth test - married, two kids, gross 24001 (borderline)
        t = new TaxCalc(24001, 2, true);
        out.println("Test 4 - tax should be 4000.40: " + t.getTax());

        // fifth test - married, four kids, gross 35000
        t = new TaxCalc(35000, 4, true);
        out.println("Test 5 - tax should be 8000: " + t.getTax());

        // sixth test - single, no kids, gross 42000 (borderline)
        t = new TaxCalc(42000, 0, false);
        out.println("Test 6 - tax should be 12000: " + t.getTax());

        // seventh test - married, one kid, gross 43501 (borderline)
        t = new TaxCalc(43501, 1, true);
        out.println("Test 7 - tax should be 12000.60: " + t.getTax());

        // eighth test - single, two kids, gross 100000
        t = new TaxCalc(100000, 2, false);
        out.println("Test 8 - tax should be 46200: " + t.getTax());
    }
}
```

`TestTaxCalc` creates eight `TaxCalc` objects in turn, calling the `getTax()` method on all of them. This ensures that the code in `TaxCalc` is thoroughly white-box tested.

The output from `TestTaxCalc` is:

```
Test 1 - tax should be 3600: 3600.0
Test 2 - tax should be 3400: 3400.0
Test 3 - tax should be 4000: 4000.0
Test 4 - tax should be 4000.40: 4000.4
Test 5 - tax should be 8000: 8000.0
Test 6 - tax should be 12000: 12000.0
Test 7 - tax should be 12000.60: 12000.6
Test 8 - tax should be 46200: 46200.0
```

which seems to show that the `TaxCalc` class works correctly – but unfortunately there is a bug!

Self-test exercises

4. Check that `TestTaxCalc` does white box test the code in `TaxCalc` – i.e. show that (i) all the code is executed at least once and (ii) that the tests at lines 25, 27 and 41 are executed both ways during these tests.
5. What is the bug? Add another test which will show this bug.

Testing with NetBeans (optional)

There is a major problem with software testing. Programmers don't like doing it!

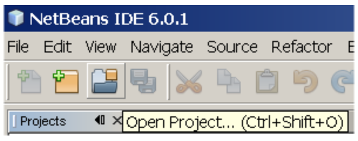
Late in the 20th century a group of programmers put forward the idea of *test driven development*. Essentially this means that you develop testing software alongside the actual software required. In the case of Java programming this is achieved by a freely available utility called JUnit.[†]

Understanding and using JUnit directly is beyond the scope of a level 1 course. However, JUnit is built in to NetBeans and it isn't too difficult to make use of it this way.

Testing the Tax Calculator using NetBeans

The Tax Calculator application has been re-written as a NetBeans project – complete with the above mentioned bug. Follow these steps to add testing to this project:

1. Download TaxCalculator.zip to a suitable location such as J:\NetBeansProjects and unzip it.

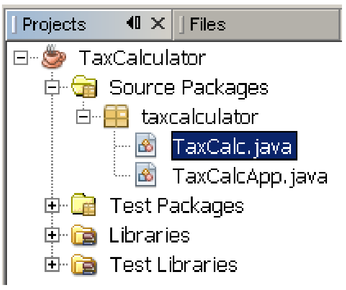

NetBeans IDE 6.0.1
File Edit View Navigate Source Refactor
Projects
Open Project... (Ctrl+Shift+O)

2. Open this project in NetBeans. To do this:
   a. Run NetBeans
   b. Click on the Open Project button (Ctrl+Shift+O)
   c. Navigate to the TaxCalculator folder in the Open Project dialogue and click the Open Project button.

[†] See http://en.wikipedia.org/wiki/JUnit

3. Run the project. (F6, or click the green arrow button). It should run just like the version we developed in Chapter 8 except for one bug.

4. Open up the project folders in the Projects pane so that it looks like the illustration shown.

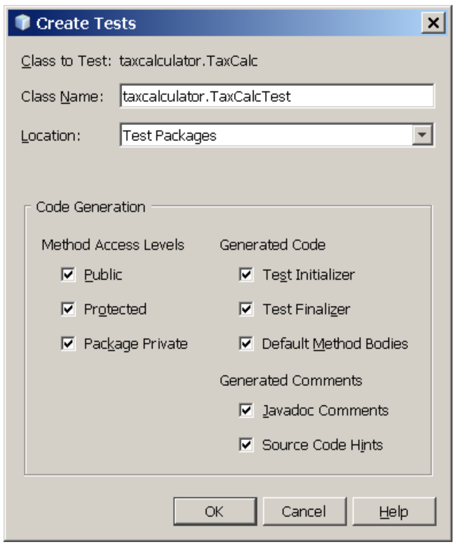


```
TaxCalculator
    Source Packages
        taxcalculator
            TaxCalc.java
            TaxCalcApp.java
    Test Packages
    Libraries
    Test Libraries
```

5. Right-click on TaxCalc.java and select Tools | Create JUnit Tests, or simply make sure TaxCalc.java is selected and type Ctrl+Shift+U.

6. A dialogue box pops up asking you to select the version of JUnit you want to use. Select the default version JUnit 4.x.

7. The Create Tests dialogue box pops up as shown. This will create a *test harness* class TaxCalcTest and store it in the Test Packages folder of your project. We will edit this class to create our test cases. (Leave all the check boxes as shown, just click the OK button.)



Class to Test: taxcalculator.TaxCalc

Class Name: taxcalculator.TaxCalcTest

Location: Test Packages

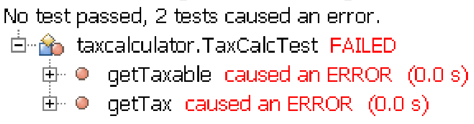
Code Generation

Method Access Levels
- [✓] Public
- [✓] Protected
- [✓] Package Private

Generated Code
- [✓] Test Initializer
- [✓] Test Finalizer
- [✓] Default Method Bodies

Generated Comments
- [✓] Javadoc Comments
- [✓] Source Code Hints

8. You can run the tests as they stand by selection Run | Test "TaxCalculator" (Alt+F6). This produces output:



```
No test passed, 2 tests caused an error.
taxcalculator.TaxCalcTest FAILED
    getTaxable caused an ERROR (0.0 s)
    getTax caused an ERROR (0.0 s)
```

Looking at the code generated for TestCalcTest, we see two ‘dummy’ test cases, one for each public method of TaxCalc:

```
/**
 * Test of getTaxable method, of class TaxCalc.
 */
@Test
public void getTaxable() {
    System.out.println("getTaxable");
    TaxCalc instance = null;
    double expResult = 0.0;
    double result = instance.getTaxable();
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the
default call to fail.
    fail("The test case is a prototype.");
}

/**
 * Test of getTax method, of class TaxCalc.
 */
@Test
public void getTax() {
    System.out.println("getTax");
    TaxCalc instance = null;
    double expResult = 0.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
    // TODO review the generated test code and remove the↲
default call to fail.
    fail("The test case is a prototype.");
}
```

You can safely ignore the rest of the generated code in TestCalcTest.java.

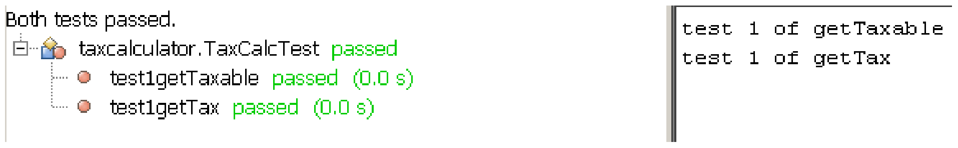
In order to create genuine test cases, we get rid of (or edit) these dummy cases and create our own. But first, note:

- Each test case is a public void method with no arguments, prefaced by the *tag* @Test
- The code for each test case usually finishes with a call to the JUnit method assertEquals. (There are other similar methods such as assertTrue but we won't use them here). The idea is that if the arguments to assertEquals are equal to one another, the test succeeds, otherwise it fails.
- The JUnit fail method always results in a failure.

9. Edit these two dummy tests to provide tests for a single person with no children with gross salary 20,000 doshky. The taxable income should be 18,000 doshky and the tax should be 3,600 doshky:

```
@Test
public void test1getTaxable() {
    System.out.println("test 1 of getTaxable");
    TaxCalc instance = new TaxCalc(20000, 0, false);
    double expResult = 18000.0;
    double result = instance.getTaxable();
    assertEquals(expResult, result);
}

@Test
public void test1getTax() {
    System.out.println("test 1 of getTax");
    TaxCalc instance = new TaxCalc(20000, 0, false);
    double expResult = 3600.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
}
```

10. Re-run the tests by pressing Alt+F6. (No need to recompile first):



```
Both tests passed.
taxcalculator.TaxCalcTest passed
    testlgetTaxable passed (0.0 s)
    testlgetTax passed (0.0 s)
```

```
test 1 of getTaxable
test 1 of getTax
```

(Here the test results in the left hand pane have been unfolded)

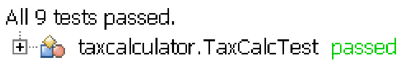
11. Add some extra test cases. Here, for completeness, are all the cases from the previous run which just used EditPlus to test getTax[†]. Note that the method names can be anything, but it is conventional to start with test:

```
@Test
public void test1getTaxable() {
    System.out.println("test 1 of getTaxable");
    TaxCalc instance = new TaxCalc(20000, 0, false);
    double expResult = 18000.0;
    double result = instance.getTaxable();
    assertEquals(expResult, result);
}

@Test
public void test1getTax() {
    System.out.println("test 1 of getTax");
    TaxCalc instance = new TaxCalc(20000, 0, false);
    double expResult = 3600.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
}

@Test
public void test2getTax() {
    System.out.println("test 2 of getTax");
    TaxCalc instance = new TaxCalc(20000, 0, true);
    double expResult = 3400.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
}

@Test
public void test3getTax() {
    System.out.println("test 3 of getTax");
    TaxCalc instance = new TaxCalc(23000, 0, true);
    double expResult = 4000.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
}

@Test
public void test4getTax() {
    System.out.println("test 4 of getTax");
    TaxCalc instance = new TaxCalc(24001, 2, true);
    double expResult = 4000.40;
    double result = instance.getTax();
    assertEquals(expResult, result);
}

@Test
public void test5getTax() {
    System.out.println("test 5 of getTax");
    TaxCalc instance = new TaxCalc(35000, 4, true);
    double expResult = 8000.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
}
```

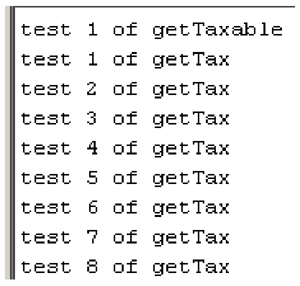
[†] JUnit anticipates that we will have several test cases for each public method. Here we assume that if getTax works as expected, then getTaxable must also have worked; we include just two test cases for getTaxable

```
@Test
public void test6getTax() {
    System.out.println("test 6 of getTax");
    TaxCalc instance = new TaxCalc(42000, 0, false);
    double expResult = 12000.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
}

@Test
public void test7getTax() {
    System.out.println("test 7 of getTax");
    TaxCalc instance = new TaxCalc(43501, 1, true);
    double expResult = 12000.60;
    double result = instance.getTax();
    assertEquals(expResult, result);
}

@Test
public void test8getTax() {
    System.out.println("test 8 of getTax");
    TaxCalc instance = new TaxCalc(100000, 2, false);
    double expResult = 46200.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
}
```

12. Type in as many of these cases in as you can. (Use cut & paste to speed it up). Every time you press Alt+F6, assuming it compiles OK, all the tests will be run:



```
All 9 tests passed.
taxcalculator.TaxCalcTest passed
```

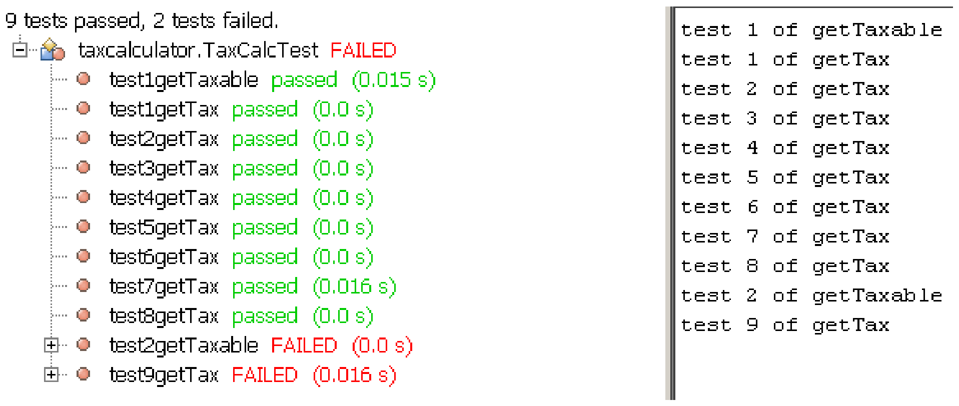


```
test 1 of getTaxable
test 1 of getTax
test 2 of getTax
test 3 of getTax
test 4 of getTax
test 5 of getTax
test 6 of getTax
test 7 of getTax
test 8 of getTax
```

13. Now add tests for Reverend Quiverfull (married, 14 children, gross salary 500 doshky)

```
@Test
public void test2getTaxable() {
    System.out.println("test 2 of getTaxable");
    TaxCalc instance = new TaxCalc(500, 14, true);
    double expResult = 0.0;
    double result = instance.getTaxable();
    assertEquals(expResult, result);
}

@Test
public void test9getTax() {
    System.out.println("test 9 of getTax");
    TaxCalc instance = new TaxCalc(500, 14, true);
    double expResult = 0.0;
    double result = instance.getTax();
    assertEquals(expResult, result);
}
```

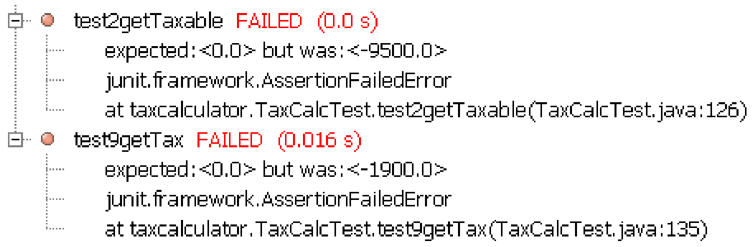
This is the result:



```
9 tests passed, 2 tests failed.
taxcalculator.TaxCalcTest FAILED
    test1getTaxable passed (0.015 s)
    test1getTax passed (0.0 s)
    test2getTax passed (0.0 s)
    test3getTax passed (0.0 s)
    test4getTax passed (0.0 s)
    test5getTax passed (0.0 s)
    test6getTax passed (0.0 s)
    test7getTax passed (0.016 s)
    test8getTax passed (0.0 s)
    test2getTaxable FAILED (0.0 s)
    test9getTax FAILED (0.016 s)
```

```
test 1 of getTaxable
test 1 of getTax
test 2 of getTax
test 3 of getTax
test 4 of getTax
test 5 of getTax
test 6 of getTax
test 7 of getTax
test 8 of getTax
test 2 of getTaxable
test 9 of getTax
```

Unfolding the two failure nodes we see:

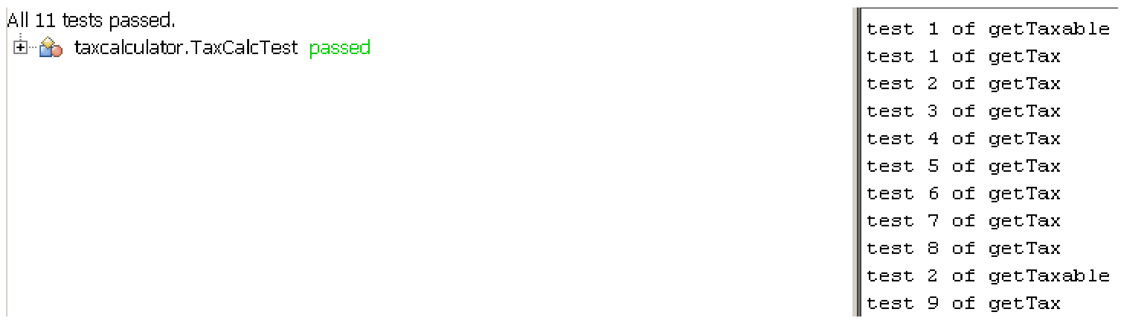


```
test2getTaxable FAILED (0.0 s)
    expected:<0.0> but was:<-9500.0>
    junit.framework.AssertionFailedError
    at taxcalculator.TaxCalcTest.test2getTaxable(TaxCalcTest.java:126)
test9getTax FAILED (0.016 s)
    expected:<0.0> but was:<-1900.0>
    junit.framework.AssertionFailedError
    at taxcalculator.TaxCalcTest.test9getTax(TaxCalcTest.java:135)
```

14. The problem is with the getTaxable method. Here we can see that the taxable income was -9500 doshky which is nonsense. We need to add a statement to set the taxable income to 0 if it is negative. Add

```
        if (taxable < 0.0) taxable = 0.0;
```

just before the return statement in getTaxable and re-run the tests with Alt+F6:



```
All 11 tests passed.
taxcalculator.TaxCalcTest passed
```

```
test 1 of getTaxable
test 1 of getTax
test 2 of getTax
test 3 of getTax
test 4 of getTax
test 5 of getTax
test 6 of getTax
test 7 of getTax
test 8 of getTax
test 2 of getTaxable
test 9 of getTax
```

Why test with JUnit in NetBeans?

You may be forgiven for thinking that this looks like a lot of work. Stub testing as carried out above seems much more compact.

The point is that once you add tests to a NetBeans project they are always there and can be executed simply by pressing Alt+F6. You only get a lot of output if one or more tests fail – no news is good news! If you make any changes to the project, you will naturally add extra test cases and then Alt+F6 runs all of them. Sometimes, changes cause test cases to fail, even if they previously worked! The JUnit approach is much safer, especially with large projects containing many auxiliary classes.

If you use JUnit in a stand-alone manner (*definitely not level 1 material!)* you can set things up in such a way that you can test a large, multi-programmer project automatically at regular intervals (saturday night is a popular time).

You do NOT need to use JUnit, nor even know anything about it, to test your programs and answer exam questions at Level 1. This material is included here as it will prove useful for later study.

Summary

- The aim of *testing* a programs is to check that it meets its specifications, and / or to expose defects.
- With *black box testing*, test cases are derived from the program specification alone.
- Programmers will use *white box testing* for preference. Tests are derived from knowledge of the code. At the very least, all the code should be executed at least once, and each branch point (decision or loop) exercised both ways.
- Programmers should produce test plans to document their testing. For each test, the input, expected output and test result should be given.
- Classes can be tested in isolation using “test harness” programs
- IDEs such as NetBeans automate much of the effort required to create and maintain test harnesses

Solutions to self-test exercises

1. In earlier programs we used a `BufferedReader` `inFile` and `String line` with loops like this:

   ```
   while ((line = inFile.readLine()) != null) {
       ...
   }
   ```

   The loop would execute if the file contained any test, but not if the file was empty. There should be a test for the latter eventuality.

2. Several possibilities. For instance **3 to 4 hr, 10p, 10p, 10p, 20p, 50p, £1, £1, £2**. `CarPark2.java` still asks the customer to pay **£0.00** after this!

3. **zero** is produced by inputs **fish** and **0**; **one – nine** by inputs **1, 22, 33, 44, 55, 66, 77, 88, 99** and **twenty – ninety** by inputs **22 – 99**.

4. All of the tests execute all the code in the `TaxCalc` constructor and the `getTaxable()` method and the private `childAllowance()` method.

   `if (taxable <= 20000) ...` at line 25 succeeds for the first three tests and fails for the rest.

   `if (taxable <= 40000) ...` at line 27 succeeds for the fourth to sixth tests and fails for the seventh and eighth tests.

   `if (married) ...` at line 41 succeeds for the second, third, fourth, fifth and seventh tests and fails for the rest.

   Borderline cases are covered by the third, fourth, sixth and seventh tests .

5. Adding an unlikely (but possible) ninth test such as:

```
// ninth test - married, fourteen kids, gross 500
t = new TaxCalc(500, 14, true);
out.println("Test 9 - tax should be 0: " + t.getTax());
```

reveals the bug (the tax calculated is -1900!)

This bug could be fixed, for example by adding an extra statement

```
if (taxable < 0) taxable = 0;
```

after line 17. One’s taxable income cannot be negative.

13. Java Database Connectivity (JDBC)

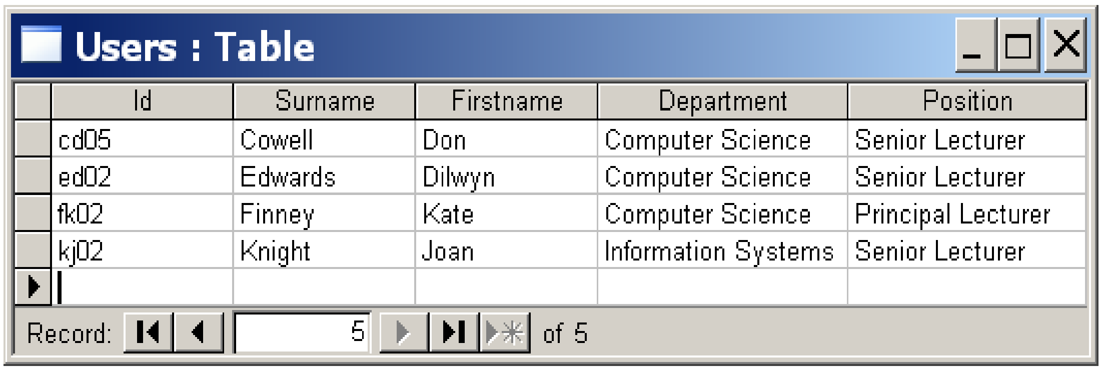
JDBC is a Java API for establishing a connection to a database, executing SQL statements and processing the results.

Each database management system (DBMS) uses its own vendor-specific *call level interface* (CLI). The CLI libraries are responsible for the actual communication with the database server. Due to vendor-specific CLI libraries, a separate driver is needed for each different DBMS.

There are currently four types of JDBC drivers:

1. JDBC-ODBC bridge plus ODBC driver. (Open Database Connectivity or ODBC)
    - Provides access via ODBC.
    - JDBC methods are translated into ODBC function calls.
    - Advantage in that it is straightforward and it works with a large number of ODBC drivers
    - Disadvantages are that it may not be available on all operating systems and that the ODBC drivers need to be loaded locally on the client.
2. Native-API partly-Java driver
    - Java code accesses data through native methods i.e. JDBC calls are converted directly into calls for most major DBMSs.
    - Advantage is – more efficient.
    - Disadvantage is that the CLI libraries must be located on the client.
3. JDBC-Net pure Java driver
    - Translates JDBC calls into DBMS-independent net protocol which then translates the call to a DBMS protocol
    - i.e. the DBMS-independent net protocol is acting as a piece of middleware.
    - The benefit here is that database native libraries do not need to be downloaded to the client therefore this approach can be easily used by Applets.
4. Native-protocol pure Java driver
    - Converts JDBC calls into network protocol used by DBMS directly
    - i.e. the client communicates directly with a database server using the server's native protocol.
    - Advantage is that it is fast and efficient because no protocol translation is required so applets can communicate with a DBMS on a server directly,
    - Disadvantage - it is DBMS specific.

For illustration purposes we use a simple Microsoft Access database **UserDB.accdb**. It only has one table, called **Users**. It has five fields but for these purposes we will only make use of the first three, **Id**, **Surname** and **Firstname**.



Users : Table

Id	Surname	Firstname	Department	Position
cd05	Cowell	Don	Computer Science	Senior Lecturer
ed02	Edwards	Dilwyn	Computer Science	Senior Lecturer
fk02	Finney	Kate	Computer Science	Principal Lecturer
kj02	Knight	Joan	Information Systems	Senior Lecturer

Record: 5 of 5

We will make a connection to this database using the first technique described above – the JDBC-ODBC bridge. The first demo program simply writes a new entry (Fred

Bloggs) to the database and then all the entries in the Users table are read back and displayed.

Two ways of establishing an ODBC connection

- The ODBC connection may be made by means of the Windows Control Panel to create a DSN (Data Set Name) to identify the database, and then referring to this ‘nickname’ in the Java code. The advantage of this way is that in your Java code you only need to know the DSN, not the absolute location of the database.
- More easily, the connection may be made directly as in the example below. Of course you need to know where the database is! For simple demonstrations, it is best to keep the database in the same folder as the Java program(s) accessing it.

DBDemo1.java

```
01  import java.io.File;
02  import static java.lang.System.*;
03  import java.sql.*;
04  import static javax.swing.JOptionPane.*;
05
06  public class DBDemo1 {
07
08  public static void main(String[] args) {
09     try {
10           Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
11           String sourceURL = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb,
12           *.accdb)};DBQ=" + new File("UserDB.accdb").getAbsolutePath() + ";";
13            Connection userDB = DriverManager.getConnection(sourceURL, "admin", "");
14              Statement myStatement = userDB.createStatement();
15              if (showConfirmDialog(null, "add Fred Bloggs to database?") ==
16              YES_OPTION) {
17              String writeString = "INSERT INTO Users(Firstname, Surname, Id)
18              VALUES('Fred', 'Bloggs', 'bf01')";
19              myStatement.executeUpdate(writeString);
20              }
21              ResultSet results = myStatement.executeQuery("SELECT Firstname,
22              Surname, Id FROM Users ORDER BY Id");
23              while (results.next()) {
24              out.print(results.getString(1) + " ");
25              out.print(results.getString(2) + " ");
26              out.println(results.getString(3));
27              }
28              results.close();
29              if (showConfirmDialog(null, "delete Fred Bloggs from database?") ==
30              YES_OPTION) {
31                  String deleteString = "DELETE FROM Users WHERE Surname='Bloggs'
32              AND Firstname='Fred'";
33              myStatement.executeUpdate(deleteString);
34              } else {
35                  showMessageDialog(null, "OK - not deleted\n\ndo not add Fred
36                  Bloggs again\n"
37                          + "or DBDemo1 will throw an SQL exception");
38              }
39              userDB.close();
40          } // The following exceptions MUST be caught
41          catch (ClassNotFoundException cnfe) {
42              out.println(cnfe);
43          } catch (SQLException sqle) {
44              out.println(sqle);
45          }
46      }
47  }
```

Line 01: We must import the Java SQL package.

Line 10: This statement initialises the Jdbc-Odbc driver. Other drivers may be used, for details see http://servlet.java.sun.com/products/jdbc/drivers. If the string used is wrong, a `ClassNotFoundException` happens, and this *must* be caught (lines 41 – 42).

Lines 11, 12: This is the *connection string* used to access the database. For this to work, **UserDB. accdb** should be in same folder as this application. If not, you must include the full pathname of the database after **DBQ=**

Lines 13: This attempts to make a *connection* to the database. The default username is **admin**, with a blank password so this will make a connection provided the database can be found. If it can’t, or password protection has been implemented and the username and / or password are wrong, a `SQLException` happens, and this *must* be caught also (lines 43 – 44).

Line 14: A `Statement` object, created by the connection’s `createStatement()` method, allows you to execute SQL statements on the database.

Lines 17 – 19: we use the `executeUpdate` method to execute a SQL INSERT statement. This inserts a new user *Fred Bloggs* with **Id** set to *bf01*. If there is anything wrong with this statement, a `SQLException` happens. `executeUpdate` is also used for SQL UPDATE and SQL DELETE statements.

Lines 21, 22: `executeQuery` is used for SQL SELECT statements. It returns a `ResultSet`. Think of a `ResultSet` as the list of results retrieved by executing the statement (`ResultSet`s are called Dynasets in MS Access terminology). This `ResultSet` is a list of the first names, surnames and user Ids for all users in the table, in order of user Id.

Lines 23 – 28: `ResultSet`s have a number of useful methods. Here we just use the `next()` method to traverse from start to finish of the `ResultSet`, and `close()` to close it. For more details, look up `ResultSet` in the Java documentation. Note that `getString(1)`, `getString(2)` and `getString(3)` return the `String` values of the first, second and third fields in the `ResultSet` *and that counting starts from 1, not 0*. There are lots of other methods such as `getInt` and `getFloat` for other types of field, but `getString` will always retrieve a field’s value as a `String`.

This program should produce output:

```
Fred Bloggs bf01
Keeran Jamil kj05
Asif Malik ma60
Kevin Mcmanus mk02
Andy Wicks wa02
Chris Walshaw wc05
```

Note that this is in order of the **Id** field.

Self-test exercise

1. If this program is run a second time, it produces a rather unhelpful exception message:

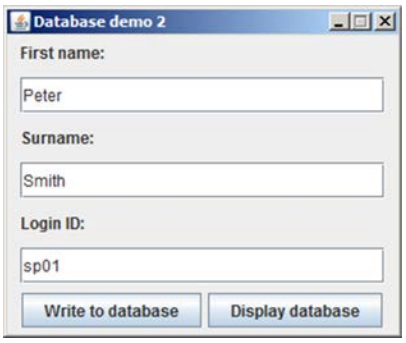
   ```
   java.sql.SqlException: General error
   ```

Can you think of a reason for this? (Hint: The **Id** field was defined as the *primary key* of the **Users** table.)

The next program extends this one by allowing the user to enter Usernames and Ids to the database:

DBDemo2.java

```
import java.awt.*;
import java.awt.event.*;
import java.io.File;
import static java.lang.System.*;
import java.sql.*;
import javax.swing.*;
import static javax.swing.JOptionPane.*;

public class DBDemo2 extends JFrame implements ActionListener {

    JTextField firstName, surname, loginId;
    JButton writeBtn, displayBtn;
    Connection userDB;
    Statement myStatement;

    public static void main(String[] args) {
        new DBDemo2();
    }

    public DBDemo2() {


...    ...     ...

        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String sourceURL = "jdbc:odbc:Driver={Microsof
        *.accdb)};DBQ="
                    + new File("UserDB.accdb").getAbsolute
            userDB = DriverManager.getConnection(sourceURL
            myStatement = userDB.createStatement();
        } // The following exceptions must be caught
        catch (ClassNotFoundException cnfe) {
            out.println(cnfe);
        } catch (SQLException sqle) {
            out.println(sqle);
        }
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == writeBtn) {
            String f = firstName.getText();
            String s = surname.getText();
            String id = loginId.getText();
            // if any field is blank, signal an error
            if (f.equals("") || s.equals("") || id.equals("")) {
                showMessageDialog(this, "One or more fields blank");
                return;
            }
            String writeString = "INSERT INTO Users(Firstname, Surname, Id) VALUES('"
                    + f + "', '" + s + "', '" + id + "')";
            try {
                myStatement.executeUpdate(writeString);
                firstName.setText("");
                surname.setText("");
            } catch (SQLException sqle) {
                showMessageDialog(this, "Duplicate key " + id);
            }
            loginId.setText("");
        }
        if (e.getSource() == displayBtn) {
            try {
                String queryString = "SELECT Firstname, Surname, Id FROM Users ORDER BY Id";
                ResultSet results = myStatement.executeQuery(queryString);
                while (results.next()) {
                    out.print(results.getString(1) + " ");
                    out.print(results.getString(2) + " ");
                    out.println(results.getString(3));
                }
                results.close();
            } catch (SQLException sqle) {
                out.println(sqle);
            }
        }
    }
}
```



Database demo 2

First name:

Peter

Surname:

Smith

Login ID:

sp01

Most of the GUI code has been omitted from this listing. The user can enter a new Username and Id to the database by filling in the fields and clicking the **Write to database** button. The relevant code (lines 65 – 84) first checks that all fields have been filled in, then (at lines 74 – 75) constructs a SQL INSERT statement as a `String`. For the input shown, this statement would be

INSERT INTO Users(Firstname, Surname, Id) VALUES('Bob', 'Dolden', 'dr05')

This sort of string manipulation is difficult to get right, especially when the SQL string needs to contain quotes. A good debugging tip is to print the string out to the console to check it – e.g. here we could write

```
System.out.println(writeString);
```

immediately after line 75.

Pressing the **Display database** button lists the **Users** table as in the first program:

```
Fred Bloggs bf01
Don Cowell cd05
Bob Dolden dr03
Dilwyn Edwards ed02
Kate Finney fk05
Joan Knight jk03
```

Self-test exercise

2. Under what circumstances will the `catch` block at lines 77 – 79 be executed?

It is a good idea to place all the fiddly database handling code into an auxiliary class as in the final version of the Usernames and Ids program. Note that now the main program doesn't need to import `java.sql.*`:

DBDemo3.java

```
01  import java.awt.*;
02  import java.awt.event.*;
03  import javax.swing.*;
04  import static javax.swing.JOptionPane.*;
05
06  public class DBDemo3 extends JFrame implements ActionListener {
07
08      JTextField firstName, surName, loginId;
09      JButton writeBtn, displayBtn;
10      DBHandler db = new DBHandler();
11
12      public static void main(String[] args) {
13          new DBDemo3();
14      }
15
16      public DBDemo3() {
…  …  …
45      }
46
47      public void actionPerformed(ActionEvent e) {
48          if (e.getSource() == writeBtn) {
49              String f = firstName.getText();
50              String s = surName.getText();
51              String id = loginId.getText();
52              // if any field is blank, signal an error
53              if (f.equals("") || s.equals("") || id.equals("")) {
54                  showMessageDialog(this, "One or more fields blank");
55                  return;
56              }
57              boolean ok = db.write(f, s, id);
58              loginId.setText("");
59              if (!ok) {
```

```
60                showMessageDialog(this, "Duplicate key " + id);
61            } else {
62                firstName.setText("");
63                surName.setText("");
64            }
65        }
66        if (e.getSource() == displayBtn) {db.displayUsers(System.out);
67        }
68    }
69 }
```

Line 10: This declares a `DBHandler` object to look after the database for us (see listing below).

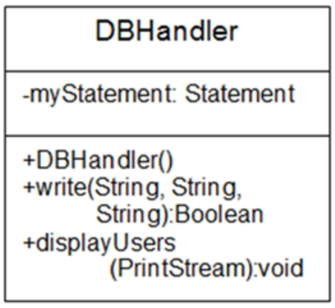
Line 57: We attempt to add a new user to the database. The `write` method of the `DBHandler` class will return `true` if the addition succeeded, `false` otherwise – i.e. if we attempt to add a new user with the same **Id** as an existing one (q.v. Self-test exercise 1). Note that this is how the `add` methods of the various `Collection` classes behave.

Line 66: The `displayUsers` method of our `DBHandler` class will list the contents of the **Users** table to the supplied `PrintStream` `System.out` as in the previous programs.

Here is the auxiliary class:

DBHandler.java

```
import java.io.*;
import static java.lang.System.*;
import java.sql.*;

class DBHandler {

    private Statement myStatement;

    public DBHandler() {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String sourceURL = "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb,
*.accdb)};DBQ="
                    + new File("UserDB.accdb").getAbsolutePath() + ";";
            Connection userDB = DriverManager.getConnection(sourceURL, "admin", "");
            myStatement = userDB.createStatement();
        } // The following exceptions must be caught
        catch (ClassNotFoundException cnfe) {
            out.println(cnfe);
        } catch (SQLException sqle) {
            out.println(sqle);
        }
    }

    public boolean write(String f, String s, String id) {
        String writeString =
                "INSERT INTO Users(Firstname, Surname, Id) VALUES('"
                + f + "', '" + s + "', '" + id + "')";
        try {
            myStatement.executeUpdate(writeString);
        } catch (SQLException sqle) {
            return false; // duplicate key
        }
        return true; // inserted OK
    }

    public void displayUsers(PrintStream outS) {
        try {
            String queryString = "SELECT Firstname, Surname, Id FROM Users ORDER BY Id";
            ResultSet results = myStatement.executeQuery(queryString);
            while (results.next()) {
                outS.print(results.getString(1) + " ");
                outS.print(results.getString(2) + " ");
                outS.println(results.getString(3));
            }
            results.close();
        } catch (SQLException sqle) {
            out.println(sqle);
        }
    }
}
```


DBHandler
-myStatement: Statement
+DBHandler()
+write(String, String, String):Boolean
+displayUsers (PrintStream):void

The only instance variable is a `Statement myStatement`. The constructor (lines 05 – 23) connects this variable to the external database (or raises an exception if it can't). The `write` method takes care of adding new users to the database and the `displayUsers` method lists the contents of the Users table (**Firstname**, **Surname** and **Id** only) to the `PrintStream` supplied as an argument, in **Id** order.

Summary

In this section we have investigated the JDBC-ODBC bridge technique to manipulate an external database in Java, in this case a very simple Access database.

Classes manipulating databases need to import the `java.sql.*` package. Important classes are:

`Connection` – to establish a connection with the external database

`Statement` – to enable the execution of SQL statements passed as strings to the database through the connection. INSERT, DELETE and UPDATE statements are handled using the `executeUpdate` method and SELECT statements using the `executeQuery` method

`ResultSet` – a table-like structure returned by the database as a result of a successful SQL SELECT query. `ResultSet`s can be iterated over using the `next` method. Note that `ResultSet`s are rather complex objects in general, especially if we are using more powerful database engines than MS Access.

Code handling external databases needs to catch `ClassNotFoundException` when trying to establish a connection and `SQLException` both when trying to establish a connection and when manipulating `Statement` objects.

As the code required to access even a simple database is quite fiddly, it is better to isolate it in a "handler" auxiliary class as in the last example (`DBHandler.java`)

It is worth pointing out that in practice a great deal of programming may be done within the database engine itself. Even MS Access has a decent "embedded" OO programming language, namely Visual Basic for Applications (VBA).

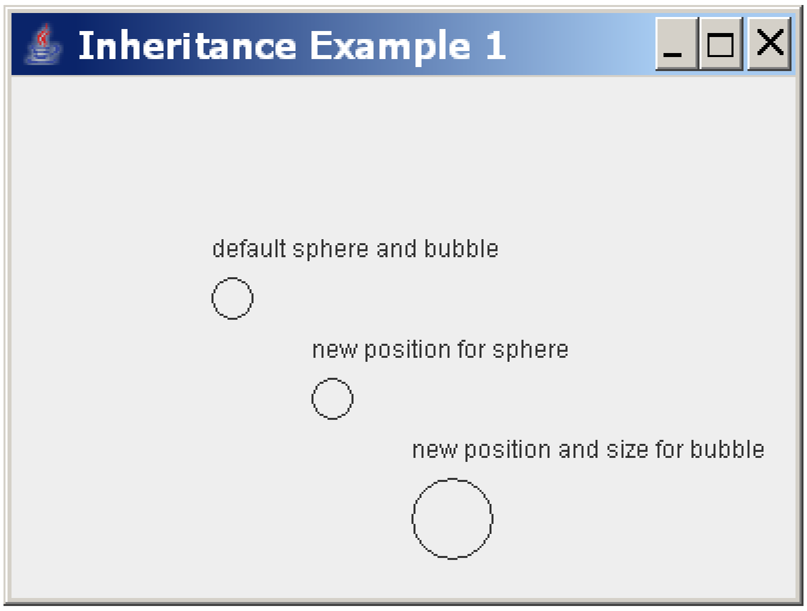
Solutions to self-test exercises

1. If we run the program a second time it tries to enter **Fred Bloggs bf01** twice. As the **Id** field is the *primary key* of the **Users** table it must be unique, hence the second attempt fails. Java reports this as a "general SQL error".
2. This `catch` block will be executed whenever the user attempts to add another record with the same **Id** field as an existing one, for the same reason as in self-test exercise 1.

14. Inheritance

In Java (and most object oriented languages) it is possible for one class to *inherit* the methods and instance variables of another. In this section we see how this is done by means of simple examples. In the next section we look at how inheritance is used in the Java API classes, in particular the various AWT and Swing classes.

Example 1



Here is a simple example, `Example1.java`. This application uses two classes `Sphere` and `Bubble`. A `Sphere` has an `x, y` position with a fixed radius of 10. A `Bubble` has an `x, y` position and a `radius` which can vary. Instead of defining these as completely separate classes, we *extend* class `Sphere` to get class `Bubble`, thereby *re-using* the code in class `Sphere`.

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

public class Example1 extends JFrame {
...
    public void paint(Graphics g) {
        Sphere sphere = new Sphere();
        Bubble bubble = new Bubble();

        g.drawString("default sphere and bubble", 100, 90);
        sphere.display(g); // these display on
        bubble.display(g); // top of each other

        // change settings for sphere and bubble

        sphere.setX(150); sphere.setY(150);
        bubble.setX(200); bubble.setY(200); bubble.setRadius(20);

        g.drawString("new position for sphere", 150, 140);
        g.drawString("new position and size for bubble", 200, 190);
        sphere.display(g); // now they display separately
        bubble.display(g); // bubble can grow, sphere can't
    }
```

The application uses a `Canvas`, whose `paint` method is listed (lines 20 – 37).

Lines 21, 22 declare a `Sphere` object and a `Bubble` object using default no-argument constructors.

Lines 25, 26: both classes have a `display` method to draw the objects on a `Graphics` object. Because of the way the `Sphere` and `Bubble` classes are defined, they are drawn on top of each other.

Lines 30, 31 use various set methods to change the instance variables of the two objects.

Lines 35, 36: the two objects are redisplayed at different x, y locations and the `Bubble` has also had its radius altered (this isn't possible for the `Sphere`).

Here are classes `Sphere` and `Bubble`:

```
01 import java.awt.*;
02
03 class Sphere {
04     protected int x = 100, y = 100;
05
06     public void setX(int newX) { x = newX; }
07
08     public void setY(int newY) { y = newY; }
09
10     public void display(Graphics g) {
11         // radius 10, can't be altered
12         g.drawOval(x, y, 20, 20);
13     }
14 }
```

Line 04: `protected` is like `private` except that `x` and `y` are visible from any class that `extends` this one – like class `Bubble`.

```
01 import java.awt.*;
02
03 class Bubble extends Sphere {
04     protected int radius = 10;
05
06     public void setRadius(int newRadius) { radius = newRadius; }
07
08     public void display(Graphics g) {
09         // overrides display in Sphere - radius can be altered
10         g.drawOval(x, y, 2*radius, 2*radius);
11     }
12 }
```

Line 03: Because `Bubble` extends `Sphere` it *inherits* all the functionality of `Sphere` – it uses instance variables `x` and `y` and methods `setX`, `setY` and `display` just like Sphere.

Lines 08 – 11: We don't want to inherit `display` unchanged – so we *override* or re-write it in this class.

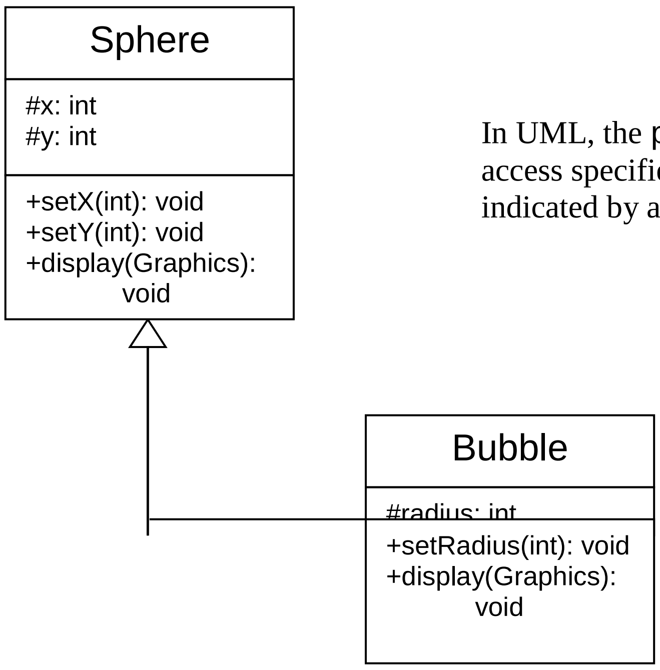
The effect is as if we had written class `Bubble` like this:

```
class Bubble {
   protected int x = 100, y = 100;
   protected int radius = 10;

   public void setX(int newX) { x = newX; }
   public void setY(int newY) { y = newY; }

   public void setRadius(int newRadius) { radius = newRadius; }

   public void display(Graphics g) {
        g.drawOval(x, y, 2*radius, 2*radius);
   }
}
```

so we have *re-used* the code from the `Sphere` class. Another way of putting it is to say that class `Bubble` *inherits* the instance variables `x, y` and methods `setX, setY, display` from class `Sphere` – although in this case it chooses to override the `display` method. Such re-use is one of the key features of object-oriented programming.

You can even extend classes for which you only have the compiled code, not the `.java` source code! This applies to nearly all of the thousands of classes in the Java API. For instance the 14-15 puzzle program introduced in Chapter 6 of the first workbook includes a class `Tile` which extends the `JButton` class, discussed in the next section.

We call `Sphere` the *superclass* or *parent class* of `Bubble` and `Bubble` a *subclass* or *child class* of `Sphere`. In UML inheritance is indicated by an arrow with a large head, thus:



In UML, the `protected` access specifier is indicated by a # sign

protected

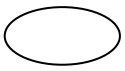
Keywords `public`, `private` and `protected` are *access specifiers*. As we have seen, a `private` instance variable or method may only be accessed from within its defining class, whereas a `public` instance variable or method may be accessed from any other class. Normally, following “information hiding” principles, all instance variables are `private` and the methods are `public` if they are to be used outside the class (by sending a message to an object or to the class itself) or `private` if they are just auxiliary methods to help define the `public` ones.

But with inheritance, there is a need for an intermediate specifier – namely, `protected`. This applies to instance variables and methods and it means that the variable in question *is* visible to code in any *subclass* of its defining class.

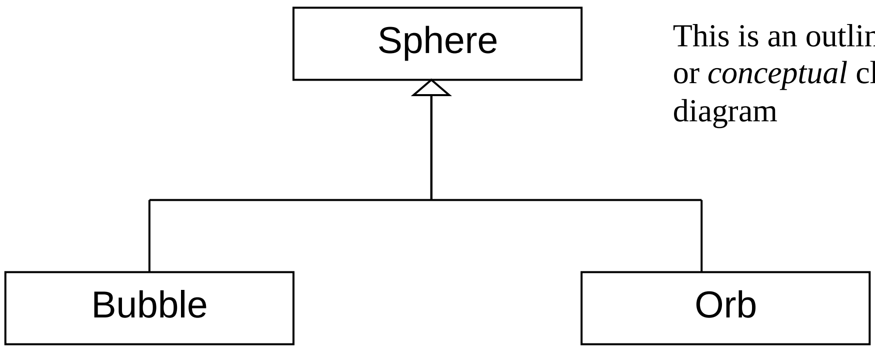
In Java, a class can only have one parent but it can have any number of children. This leads to a tree-like, or *hierarchical* relationship amongst classes.

A class inherits (`protected` or `public`) instance variables and methods not only from its parent, but from the parent of its parent (its grandparent) and so on right up to the top of the hierarchy. In Java every class ultimately inherits from the great- … grandparent of all classes, namely class `Object`, even if you don't state this explicitly. For an example, have a look at the definition of the `JButton` class in the Java documentation. This has 9 public methods by itself, and it inherits well over 200 further methods from its parent class `AbstractButton`, its grandparent `JComponent`, and so on. The classes higher up in the hierarchy have many other child classes, all of which share these methods. (see later in this section).

Self-test exercises

1. An `Orb` is just like a `Bubble` except that it is twice as broad as it is high: 

   Write the class definition for `Orb`. Assume that its height is half its width. (It's just the same as class `Bubble` except for the overriding `display` method.)



This is an outline or *conceptual* class diagram

2. A `Ball` is like a `Bubble`, but with a `color`. Write its class definition, extending class `Bubble`. Set the default `color` to `Color.red`. You will need an extra method `setColor` to allow this to be changed, and you will need to override `display` again (to display the balloon in the chosen color).

   Thus, to create a red `Ball` the programmer would write

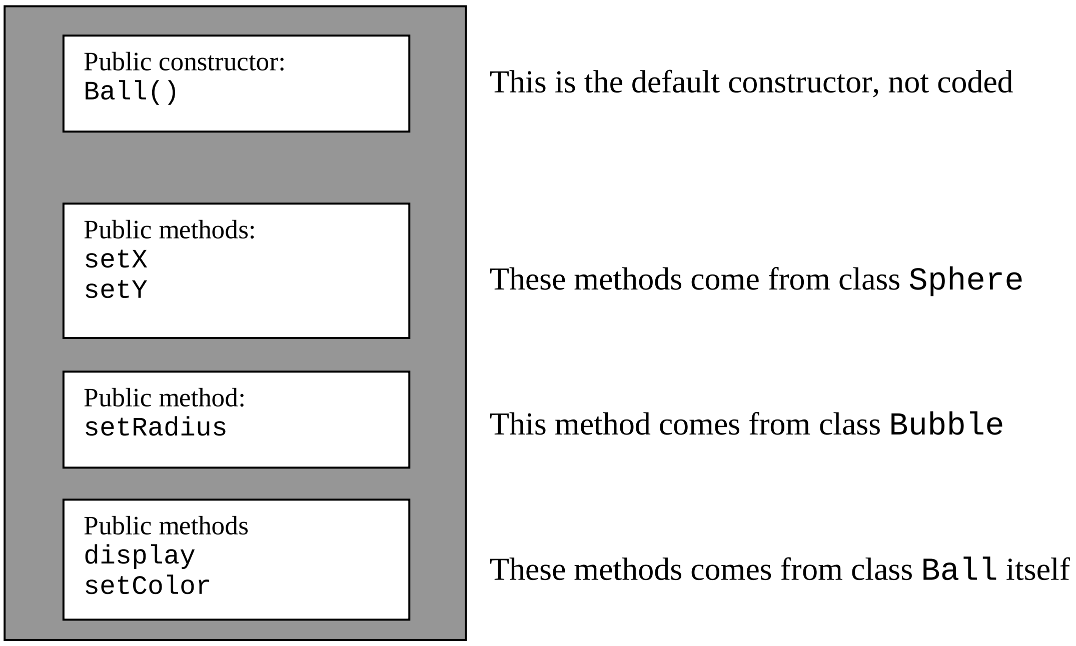
   ```
   Ball redBall = new Ball();
   ```

   and to create a green `Ball`:

   ```
   Ball greenBall = new Ball();
   greenBall.setColor(Color.green);
   ```

   Draw the new hierarchy diagram, showing the relationship between classes `Sphere, Bubble, Ball`.

A Ball object as seen by a program using it


Public constructor:
Ball()
This is the default constructor, not coded
Public methods:
setX
setY
These methods come from class Sphere
Public method:
setRadius
This method comes from class Bubble
Public methods
display
setColor
These methods comes from class Ball itself

super

From within your new class `Ball` any reference to method `display` means the overriding method you defined in the above exercise, and not the original method from the `Bubble` class shown here:

```
public void display(Graphics g) {
   // overrides display in Sphere - radius can be altered
   g.drawOval(x, y, 2*radius, 2*radius);
}
```

If you want to use this original method, you must use keyword `super`, which is always interpreted as a reference to the parent class. `super` can be useful in circumstances where your overriding method adds a bit more functionality to the original one. For instance the definition of method `display` for your `Ball` class could be written like this:

```
public void display(Graphics g) {
   // overrides display in Bubble – must use correct color
     g.setColor(color);
     super.display(g);
}
```

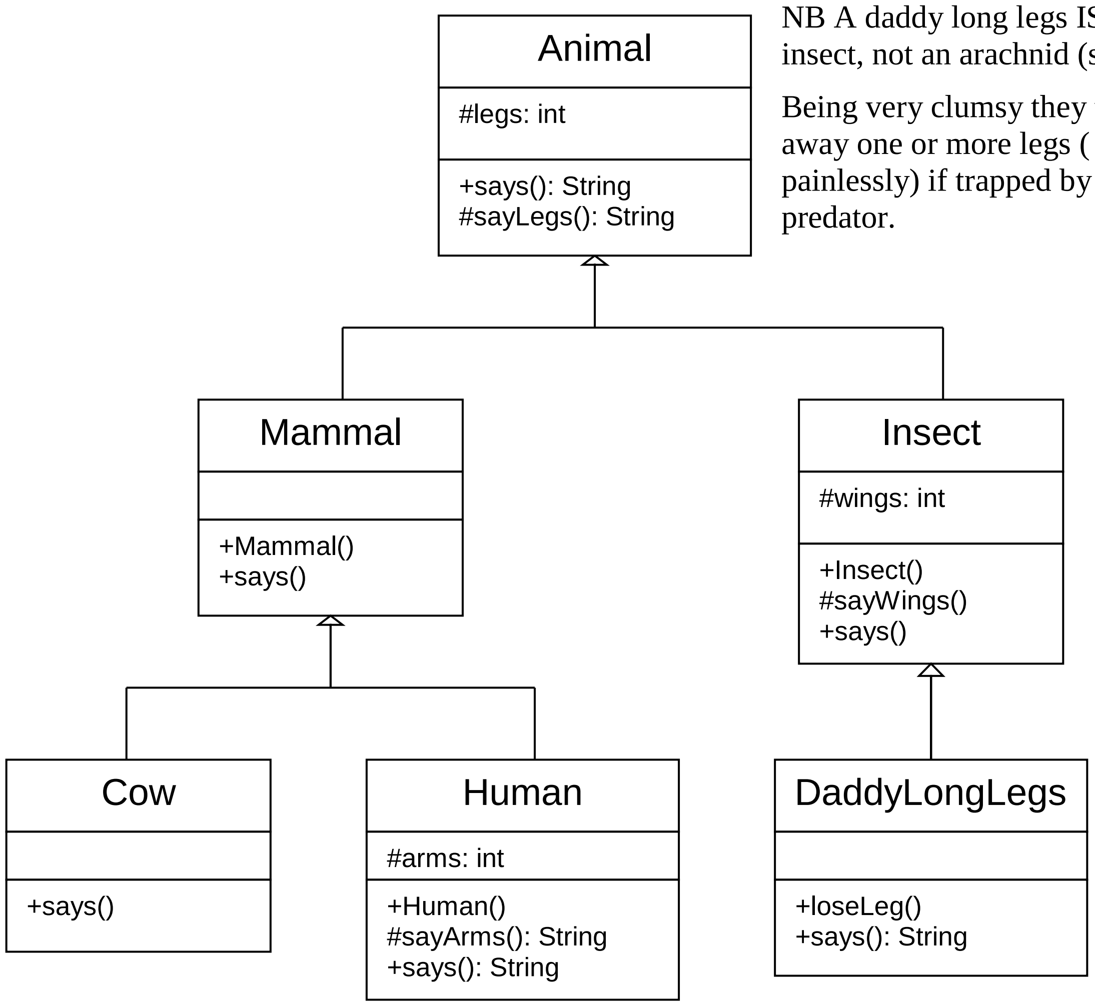
In other words, use the corresponding method from the superclass

Polymorphism

This rather fancy word – it derives from the Greek, meaning “many shaped” – means that objects from different classes can be manipulated using the a common set of methods. We have already seen this in action – for instance, classes `JTextField`, `JButton`, `JLabel` etc all have a void `setText(String s)` method for changing the text shown on them. Our various graphics objects all have a void display`(Graphics g)` to display them.

Animals example

Here is a hierarchy of animal classes:



NB A daddy long legs IS an insect, not an arachnid (spider).

Being very clumsy they throw away one or more legs (quite painlessly) if trapped by a predator.

```
class Animal {
    protected int legs;

    public String says() {
        return "I'm an animal." + sayLegs();
    }

    protected String sayLegs() {
        return " I have " + legs + " legs.";
    }
}
```

```
class Mammal extends Animal {

    Mammal() { legs = 4; }

    public String says() {
        return "I'm a mammal." + sayLegs();
    }
}
```

```
class Cow extends Mammal {

    public String says() {
        return "Moo!" + sayLegs();
    }
}
```

```
class Insect extends Animal {
    protected int wings;

    Insect() { legs = 6; wings = 4; }

    protected String sayWings() {
        return " I have " + wings + " wings.";
    }

    public String says() {
        return "I'm an insect." + sayWings() + sayLegs();
    }
}
```

```
class Human extends Mammal {
    protected int arms;

    Human() { legs = 2; arms = 2; }

    protected String sayArms() {
        return " I have " + arms + " arms.";
    }

    public String says() {
        return "I'm a human being." + sayArms() + sayLegs();
    }
}
```

```
class DaddyLongLegs extends Insect {

    // Daddy long legs frequently lose one
    //  or more legs without apparent harm!
    public void loseLeg() {
        legs--; //ouch
        if (legs < 0) legs = 0; // can't have < 0 legs!
    }

    public String says() {
        return "I'm a daddy long legs." + sayWings() + sayLegs();
    }
}
```

Note that the only classes with explicit constructors are `Mammal`, `Insect` and `Human`. The others use the default constructor or inherit the constructor of their parent.

Here is a harness program to white-box test these classes:

```
class Animals
{
    public static void main(String[] args)
    {
        Animal creature = new Animal();
        Mammal furry = new Mammal();
        Cow daisy = new Cow();
        Human person = new Human();
        Insect buzz = new Insect();
        DaddyLongLegs daddy = new DaddyLongLegs();
        System.out.println(creature.says());
        creature = furry;
        System.out.println(creature.says());
        creature = daisy;
        System.out.println(creature.says());
        creature = person;
        System.out.println(creature.says());
        creature = buzz;
        System.out.println(creature.says());
        creature = daddy;
        System.out.println(creature.says());
        daddy.loseLeg();
        System.out.println(creature.says());
        daddy.loseLeg(); daddy.loseLeg(); daddy.loseLeg();
        daddy.loseLeg(); daddy.loseLeg();
        System.out.println(creature.says());
        daddy.loseLeg();
        System.out.println(creature.says());
    }
}
```

This program produces the following output:

```
I'm an animal. I have 0 legs.
I'm a mammal. I have 4 legs.
Moo! I have 4 legs.
I'm a human being. I have 2 arms. I have 2 legs.
I'm an insect. I have 4 wings. I have 6 legs.
I'm a daddy long legs. I have 4 wings. I have 6 legs.
I'm a daddy long legs. I have 4 wings. I have 5 legs.
I'm a daddy long legs. I have 4 wings. I have 0 legs.
I'm a daddy long legs. I have 4 wings. I have 0 legs.
```

Note that lines 12, 14, 16, 18, 20 are not necessary; instead we could have written

```
13          System.out.println(furry.says());
```

instead (and similarly for lines 15, 17, 19, 21). However this serves to illustrate a point: An assignment between objects `a` and `b`:

```
a = b;
```

is legal *provided* `a` *is an instance of the same class as* `b` *or* `a` *is an instance of a superclass of* `b`.

However you can’t assign a superclass instance to a subclass instance; for example

```
furry = creature;
```

gives a compiler error (Incompatible Types).

Note also that from line 20 on, `creature` and `daddy` refer to the same object. Thus the various statements `daddy.loseLeg()` alter the state of creature as well.

Self-test exercises

3. In the above test harness, could you write `creature.loseLeg()` instead of `daddy.loseLeg()`?

4. Add three extra classes to the Animals hierarchy: `Englishman`, `Scotsman` and `FlyingAnt`. `Englishman` and `Scotsman` are subclasses of `Human`.

   An `Englishman` says “I say old chap, you couldn’t possibly lend me fifty quid until next Thursday?” and says how many arms and legs he has.

   A `Scotsman` says “Sassanach!” and says how many arms and legs he has.

   `FlyingAnt` is a subclass of `Insect`. A flying ant usually loses all its wings quite painlessly after its one and only flight (I think it bites them off) so it needs a method `loseWings()` to set `wings = 0`. A `FlyingAnt` says “I’m a flying ant.” then says how many wings and legs it has (or, if you prefer, “I’m a flying ant” if it has four wings and “I’m a walking ant” if it has no wings).

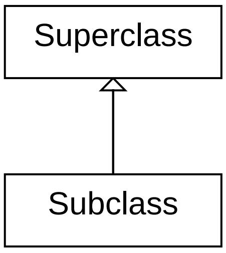
   Test your new classes with a suitable harness program.

Summary

In this section we looked at one of the most powerful features of Java and similar OO languages – *Inheritance*. If class B *extends* class A, B inherits all the public and *protected* instance variables and methods of A (but private variables and methods are not inherited). We say that B is a *subclass* of A and that A is a *superclass* of B.

We also saw the idea of *polymorphism.* This means that we can have two different classes with identically named methods, both having the same return type and parameter lists, but with (usually) different behaviours. For instance we built various simple graphics classes, all having a void display(Graphics g) method to draw an object, and also noted that many Swing components have a void setText(String s) method.

We also saw the UML notation for inheritance between classes – namely an arrow linking them as shown here.



The UML notation for the `protected` access specifier is # (remember + means `public`, - means `private.`

Given any two objects a and b, the assignment

```
a = b;
```

is legal only if `a` and `b` belong to the same class or `a`’s class is a superclass of `b`’s class.

Solutions to self-test exercises

1. Simply replace `Bubble` by `Orb` and in the display method use

   ```
   g.drawOval(x, y, 2*radius, radius);
   ```

2.
   ```
   import java.awt.*;

   class Ball extends Bubble {
       protected Color color = Color.red;

       public void setColor(Color newColor) { color = newColor; }

       public void display(Graphics g) {
           Color oldColor = g.getColor();
           g.setColor(color);
           g.drawOval(x, y, 2*radius, 2*radius);
           g.setColor(oldColor);
       }
   }
   ```

   The highlighted lines make sure that the `Graphics` drawing colour is the same as it was before the `display` method was called.

   It would be a good idea to omit the assignment of `Color.red` to `color` and include constructors

   ```
   Ball() { color = Color.red; }
   Ball(Color c) { color = c; }
   ```

   In all the classes discussed in this section, it would be a good idea to include extra constructors to set all the instance variables to specific values. The more the merrier.

3. No. The compiler would give a “cannot find symbol” error because class `Animal` has no `loseLeg()` method.
4. e.g.

```
class Englishman extends Human {

    public String says() {
        return "I say old chap, you couldn't possibly
lend me fifty quid until next Thursday?"

class Scotsman extends Human {

    public String says() {
        return "Sassanach!"
            + sayArms() + sayLegs();
    }
}

class FlyingAnt extends Insect {

    public void loseWings() { wings = 0; }

    public String says() {
        if (wings > 0)
            return "I'm a flying ant." + sayWings() + sayLegs();
        else
            return "I'm a walking ant." + sayWings() + sayLegs();
    }
}
```

tested with

```
...      ...
         Englishman clarence = new Englishman();
         Scotsman mcTavish = new Scotsman();
         FlyingAnt woody = new FlyingAnt();
         System.out.println(clarence.says());
         System.out.println(mcTavish.says());
         System.out.println(woody.says());
         woody.loseWings();
         System.out.println(woody.says());
...      ...
```

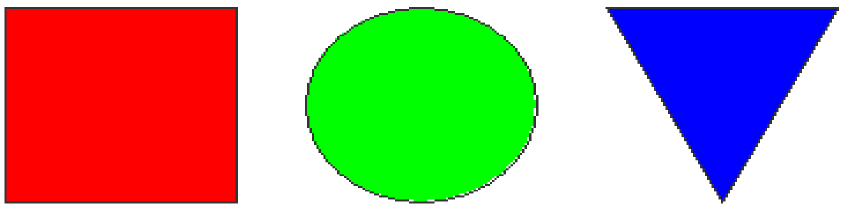
giving output

```
I say old chap, you couldn't possibly lend me fifty quid until next Thursday? I have 2 arms. I have 2 legs.
Sassanach! I have 2 arms. I have 2 legs.
I'm a flying ant. I have 4 wings. I have 6 legs.
I'm a walking ant. I have 0 wings. I have 6 legs.
```

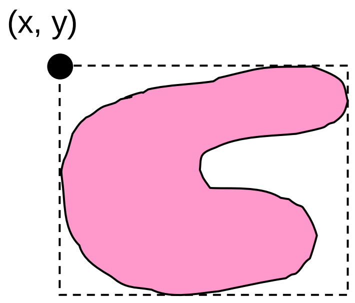
15. Abstract Classes and Interfaces

Case study – Drawing simple shapes

Suppose that we want an application to draw simple two-dimensional shapes on a Canvas – for instance squares, circles and triangles:



(For those of you viewing this in black and white, note that the red rectangle is to the left, and the blue triangle is to the right, of the green oval in the middle.)



It would be a good idea to have a different class for each type of shape. However, these classes have a lot in common. Each shape has:

- a *position* which, following the Java `Graphics` class convention, is given by the `x` and `y` coordinate of the top left of its bounding box (shown dotted here),
- a *width* and *height* (the width and height of the bounding box), and
- a *colour.*

In addition, methods to get and set the x and y position coordinates and the width, height and colour would all be the same.

The only difference between classes is the way in which shapes are to be drawn.

In Java and other OO languages we can express this common functionality in a single parent class `Shape`. This will *specify* a `draw` method but will *not* implement it. This is done by declaring the class, and the `draw` method, to be *abstract*.

Abstract class Shape.java

```
import java.awt.*;
abstract class Shape {

    // What do our shapes have in common?
    // A top left (x, y) position,
    // a width and height
    // and a color, so:
    protected int x, y, width, height;
    protected Color color;

    // Constructor
    Shape(int x, int y, int width, int height, Color color) {
        setX(x);
        setY(y);
        setWidth(width);
        setHeight(height);
        setColor(color);
    }
19
20     // get methods
21     public int getX() { return x; }
22     public int getY() { return y; }
23     public int getWidth() { return width; }
24     public int getHeight() { return height; }
25     public Color getColor() { return color; }
26
27     // set methods
28     public void setX(int x) { this.x = x; }
29     public void setY(int y) { this.y = y; }
30     public void setWidth(int width) {
31         // can't have a negative width so:
32         this.width = (width < 0) ? 0 : width;
33     }
34     public void setHeight(int height) {
35         // can't have a negative height so:
36         this.height = (height < 0) ? 0 : height;
37     }
38     public void setColor(Color color) { this.color = color; }
39
40     // Abstract method to draw the shape
41     public abstract void draw(Graphics g);
42
43     // Note that each subclass MUST implement draw if it is not
44     // itself to be abstract
45 }
```

Line 02: We must declare this class to be `abstract` as it has at least one abstract method.

Line 41: This is how we declare an abstract method. We specify the return type (`void` here), the name of the method (`draw` here) and the list of parameters and their types (`Graphics g` here) just as for an ordinary method definition, BUT there is no method body enclosed between `{` and `}`, just a semicolon.

What does it mean for a class to be abstract? Well, in a client program we can *declare* objects of that type, as in

```
    private Shape firstShape, secondShape, thirdShape;
```

but these objects *can't* be instantiated using the class itself, so:

```
        firstShape = new Shape(0, 0, 50, 50, Color.red);
```

would give a syntax error. Instead we must use a *concrete* subclass (or sub-subclass, etc) of `Shape`:

```
        firstShape = new Rectangle(0, 0, 50, 50, Color.red);
```

(class `Rectangle` is listed below).

Lines 21 – 25, 28 – 38: We declare some useful `get` and `set` methods. These can be re-used by any subclass of `Shape`.

Lines 32, 36: The peculiar **`... ? ... : ...`** syntax is a *conditional expression*.

```
        this.width = (width < 0) ? 0 : width;
```

is equivalent to the more normal but long-winded *conditional statement*

```
        if (width < 0)
```

```
            this.width = 0;
        else
            this.width = width;
```

Bear this in mind when testing – each conditional expression has a branch so will need testing twice.

Lines 12 – 18: It may seem odd to have a constructor for an abstract class as it can't be used directly (see above). However this constructor may be re-used in subclasses using keyword `super`. Here we make use of the five `set` methods for convenience.

Rectangle.java, Oval.java, Triangle.java

These are concrete subclasses of class `Shape` ("concrete" simply means "not abstract").

```
import java.awt.*;
class Rectangle extends Shape {

   Rectangle(int x, int y, int width, int height, Color color) {
      super(x, y, width, height, color);
   }

   public void draw(Graphics g) {
      Color oldColor = g.getColor();
      g.setColor(color);
      g.fillRect(x, y, width, height);
      g.setColor(oldColor);
      g.drawRect(x, y, width, height);
   }
}
```

```
import java.awt.*;
class Oval extends Shape {

   Oval(int x, int y, int width, int height, Color color) {
      super(x, y, width, height, color);
   }

   public void draw(Graphics g) {
      Color oldColor = g.getColor();
      g.setColor(color);
      g.fillOval(x, y, width, height);
      g.setColor(oldColor);
      g.drawOval(x, y, width, height);
   }
}
```

```
import java.awt.*;
class Triangle extends Shape {

   Triangle(int x, int y, int width, int height, Color color) {
      super(x, y, width, height, color);
   }

   public void draw(Graphics g) {
      Color oldColor = g.getColor();
      g.setColor(color);
11        // vertices are (x, y), (x+width, y), (x+width/2, y+height)
12        Polygon t = new Polygon();
13        t.addPoint(x, y);
14        t.addPoint(x+width, y);
15        t.addPoint(x+width/2, y+height);
16        g.fillPolygon(t);
17        g.setColor(oldColor);
18        g.drawPolygon(t);
19    }
20 }
```

Each class extends `Shape` and contains a five-argument constructor and a `draw` method. The constructors re-use the superclass constructor (**line 05** in each listing).

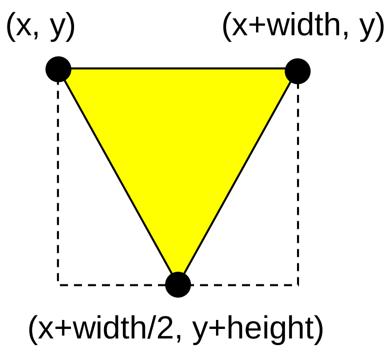
Each `draw` method fills the shape with the chosen colour and then draws the shape using the default colour. The pattern is the same for each:

```
    public void draw(Graphics g) {
        Color oldColor = g.getColor();
        g.setColor(color);
        ... code to fill the shape with the chosen colour
        g.setColor(oldColor);
        ... code to draw the shape's outline with the default colour
    }
```

The `draw` methods for the `Rectangle` and `Oval` classes use the `Graphics` class `fillRect`, `drawRect`, `fillOval` and `drawOval` methods in the obvious way.

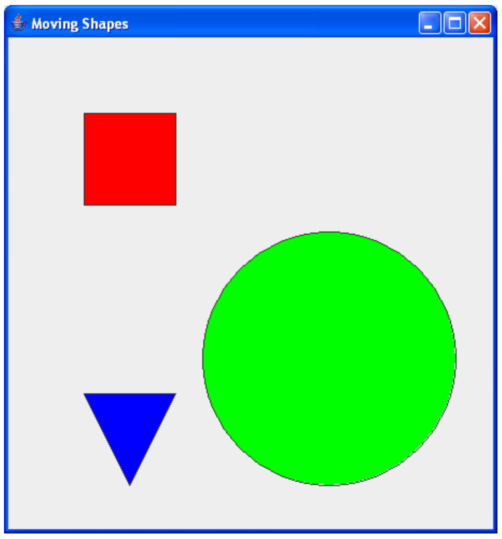
The `draw` method for the `Triangle` class is a bit more complicated. It uses a `Polygon` object and the `Graphics` class `fillPolygon` and `drawPolygon` methods (**lines 12 – 16, 18**).

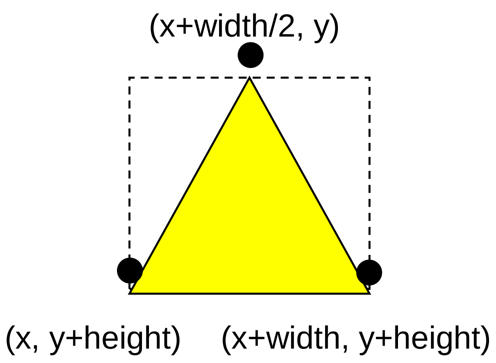
Lines 12 – 15 create a new Polygon and add the three vertices of the triangle to it (see first figure on the right – we want to draw an isosceles triangle with the vertex pointing down).



Self-test exercise

1. Replace lines 13 – 15 so that the triangle is drawn with the vertex pointing up as in the second figure on the right.





Here is a main program to test this Shape hierarchy.

It starts with a red rectangle at the top left, a green circle at the bottom right and a blue triangle at the bottom left. These move diagonally towards the centre under the control of a *Timer.* As they move, the shapes expand / contract.

MovingShapes.java

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

public class MovingShapes extends JFrame
                    implements ActionListener {
    private MyCanvas myCanvas = new MyCanvas();
    private Shape firstShape, secondShape, thirdShape;
    private Timer timer;

    public static void main(String[] args) {
        new MovingShapes();
    }

    public MovingShapes() {
        firstShape = new Rectangle(0, 0, 50, 50, Color.red);
        secondShape = new Oval(250, 250, 200, 200, Color.green);
        thirdShape = new Triangle(0, 400, 50, 50, Color.blue);
        timer = new Timer(100, this); // these statements trigger
        timer.setInitialDelay(0);     // an ActionEvent
        timer.start();                // ten times a second
        setLayout(new BorderLayout());
        setSize(458, 490);
        setTitle("Moving Shapes");
        add("Center", myCanvas);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
         myCanvas.repaint();
    }

    private class MyCanvas extends Canvas {
        public void paint(Graphics g) {
            int x, y, width, height;
            firstShape.draw(g);
            x = firstShape.getX();
            y = firstShape.getY();
            width = firstShape.getWidth();
            firstShape.setWidth(width + 5);
            height = firstShape.getHeight();
            firstShape.setHeight(height + 5);
            firstShape.setX(x + 10);
            firstShape.setY(y + 10);

            secondShape.draw(g);
            x = secondShape.getX();
            y = secondShape.getY();
            width = secondShape.getWidth();
            secondShape.setWidth(width - 5);
            height = secondShape.getHeight();
            secondShape.setHeight(height - 5);
54              secondShape.setX(x - 10);
55              secondShape.setY(y - 10);
56
57              thirdShape.draw(g);
58              x = thirdShape.getX();
59              y = thirdShape.getY();
60              width = thirdShape.getWidth();
61              thirdShape.setWidth(width + 5);
62              height = thirdShape.getHeight();
63              thirdShape.setHeight(height + 5);
64              thirdShape.setX(x + 10);
65              thirdShape.setY(y - 10);
66          }
67      }
68 }
```

Line 08: Three `Shape` variables are declared.

Line 09: A `Timer` is declared. This will control the movement of the shapes.

Lines 16 – 18: The `Shape`s are instantiated to a red square (rectangle with width and height 50), a green circle (oval with width and height 200) and a blue triangle with width and height 50 at the top left, bottom right and bottom left of the canvas respectively (when they are first drawn).

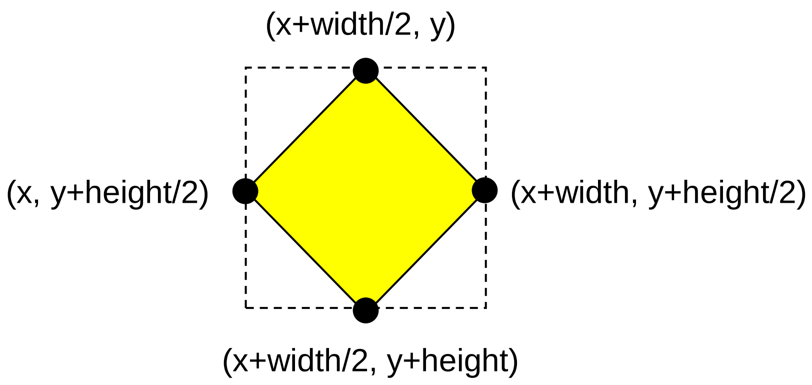
Lines 19 – 21: The `Timer` is 'primed' to trigger an `ActionEvent` ten times a second (the argument 100 refers to 100 milliseconds, i.e. 1/10th of a second).

Lines 30 – 32: The `ActionPerformed` method repaints the canvas ten times a second.

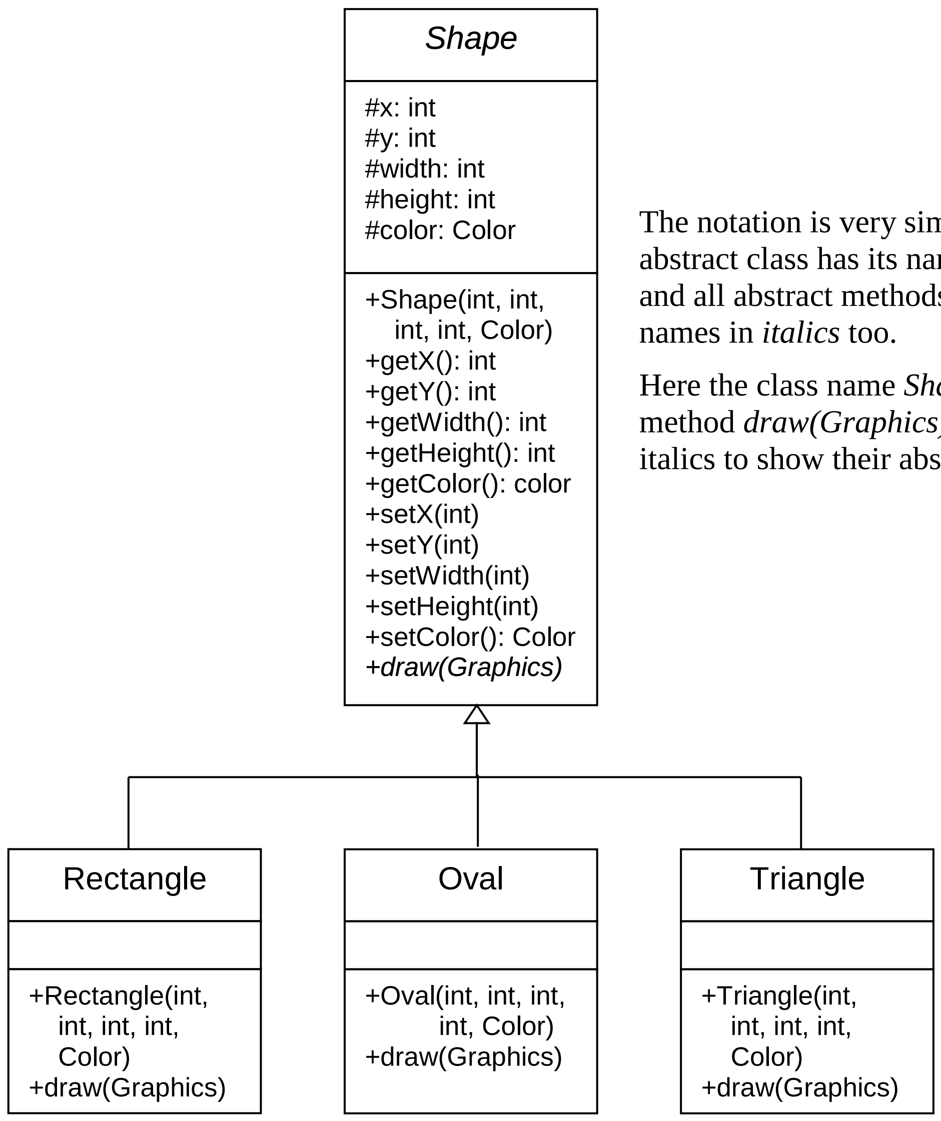
Lines 37 – 45, 47 – 55, 57 – 65: The canvas's `paint` method draws each shape and then resizes and moves it ready for the next call of paint, giving the illusion of the shapes moving and expanding.

Self-test exercise

2. Suppose we have a class `Diamond` as a subclass of `Shape`. It will look much like class `Triangle` listed above, but with a different `Polygon` (lines 12 – 16 of that class). The `Diamond` has four vertices so we must add the four points shown here to the `Polygon`. Write down the four statements that will do this, replacing lines 13 – 15.



UML notation for abstract classes



The notation is very simple. An abstract class has its name in *italics* and all abstract methods have their names in *italics* too.

Here the class name *Shape* and method *draw(Graphics)* are in italics to show their abstract nature.

Note that the concrete subclasses *must* provide implementations for the inherited abstract methods. So `Rectangle`, `Oval` and `Triangle` *must* each have a `public void draw(Graphics g)` method. (If they don't, they must be declared `abstract` in turn).

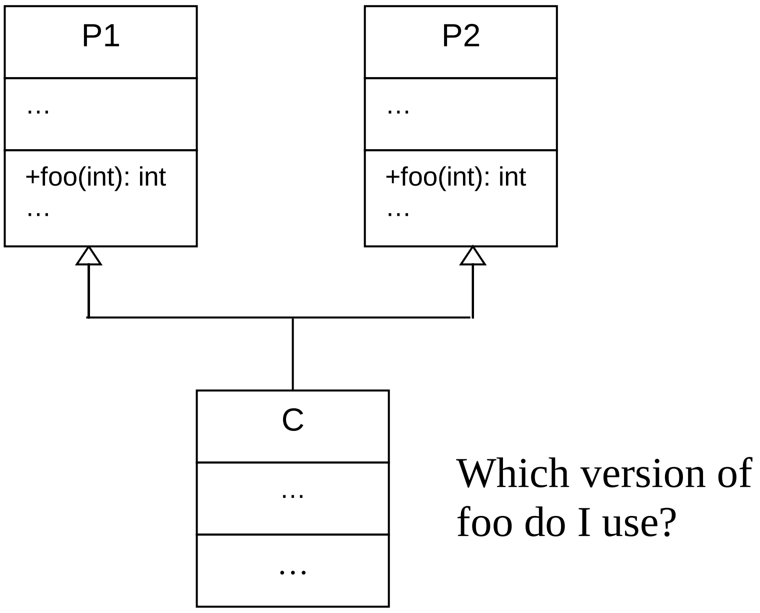
Interfaces

C++ and some other OO languages allow a class to have more that one parent. Suppose that we have classes `P1` and `P2` and a class `C` that extends both of them. C++ uses the following notation for this:

```
class C : public P1, public P2 {
    ...
}
```

Note that this is NOT done in Java (nor in C#)

But there is a major problem with this idea. For instance, what happens if `P1` and `P2` both have a method `public int foo(int x)`, and suppose that these methods do different things. Which one of these will be inherited by `C`?

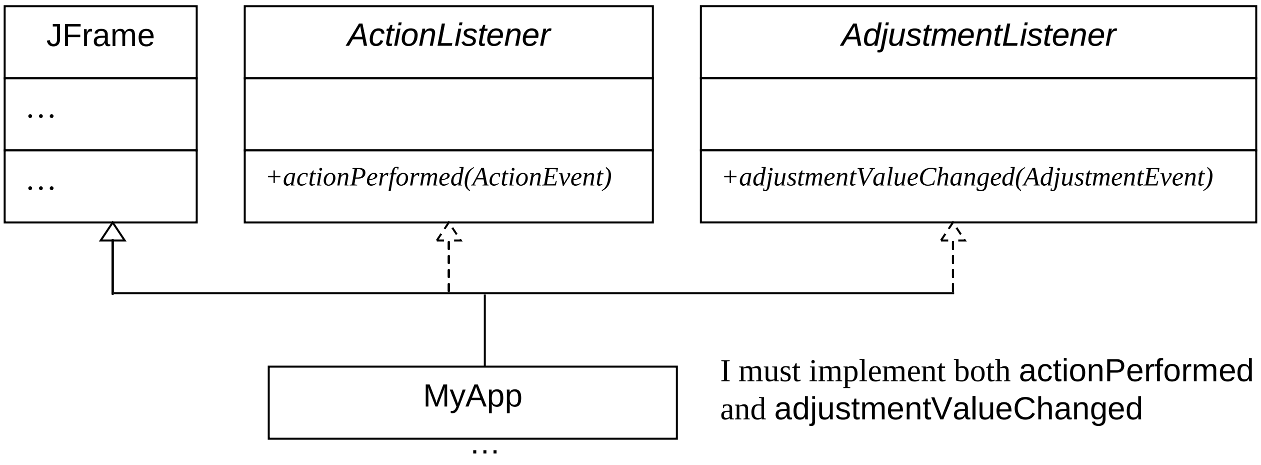


In C++ this is resolved by the rule: Inherit methods and instance variables from the parent classes in a left-to-right order. So it would be the version from superclass P1 that would be used.

Because of the confusion this often leads to (and for other reasons) Java and most modern OO languages don't allow multiple inheritance. But these languages have a trick up their sleeve – *Interfaces*. An interface is like an abstract class in which *all the methods are abstract*.

We have already used many interfaces – for example the event listeners `ActionListener`, `AdjustmentListener` etc and the `Collection` interfaces `List` and `Set`. The simple rule is: A class can *extend* at most one parent superclass, but it may *implement* any number of interfaces as in

```
public class MyApp extends JFrame
                   implements ActionListener, AdjustmentListener {
    ...
}
```


JFrame
…
…
ActionListener
+actionPerformed(ActionEvent)
AdjustmentListener
+adjustmentValueChanged(AdjustmentEvent)
MyApp
…
I must implement both actionPerformed and adjustmentValueChanged

The UML notation for implementing an interface is simply to use a dashed line.

If two interfaces happen to specify the same method name and signature there is no problem as the implementing class satisfies both with a single implementing method.

If you browse the Java API class library documentation you will see many interfaces. They are important as they serve to keep the interaction between different classes (often called the interface between classes, hence the keyword `interface`) to a minimum, which is widely regarded as good software engineering practice.

Interfaces are very simple, for instance here is `ActionListener` (with comments removed)

```
import java.util.EventListener;
public interface ActionListener extends EventListener {
    public void actionPerformed(ActionEvent e);
}
```

`EventListener` is even simpler (with the comments):

```
/**
 * A tagging interface that all event listener interfaces must extend.
 * @since JDK1.1
 */
public interface EventListener {
}
```

This means that `MyApp` is a `JFrame` (and all superclass types), an `ActionListener`, and `AdjustmentListener` and an `EventListener`, so the Java run-time system knows what to do with it.

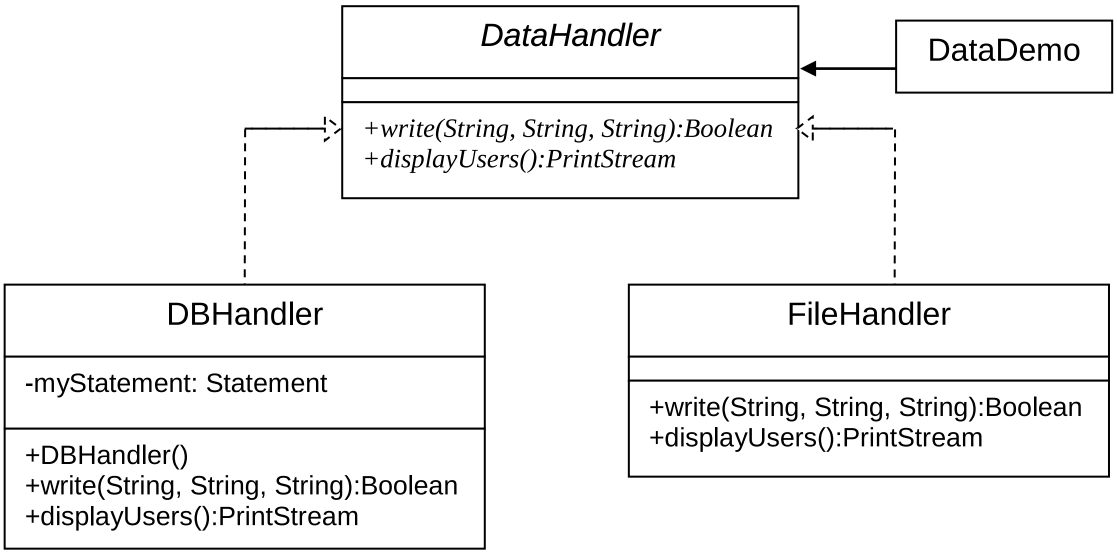
Self-test exercise

3. What is the point of a so-called “tagged” interface like `EventListener` which specifies nothing at all?

NB Normally we omit Java API classes and interfaces when drawing UML class diagrams – although there is a shorthand notation for indicating which classes / interfaces we are extending / implementing.

Data handling demo

In the section on JDBC we saw how to use an auxiliary class to handle communications with a database (Chapter 15, DBDemo3 + DBHandler example). By making use of a simple interface we may redesign this so that the main program can either use a database handler object or a file handler object.



DataHandler
+write(String, String, String):Boolean
+displayUsers():PrintStream
DataDemo
DBHandler
-myStatement: Statement
+DBHandler()
+write(String, String, String):Boolean
+displayUsers():PrintStream
FileHandler
+write(String, String, String):Boolean
+displayUsers():PrintStream

The `DataHandler` interface specifies a `write` method to update the data source and a `displayUsers` method to display the data source. The `DBHandler` and `FileHandler` classes both implement this interface.

The main application `DataDemo` has a `static` variable dh of type `DataHandler` that is either set to a new `DBHandler` object or a new `FileHandler` object depending on the command line argument (which should be either FILE or DB).

We have seen the `DBHandler` class before. It communicates with an Access database UserDB.mdb.

The `FileHandler` class reads from, and writes to, a text file Users.txt.

The point is that the `DataDemo` application doesn't need to know anything about database handling or text I/O. This 'de-coupling' could have been done by using an abstract class for `DataHandler` instead of an interface, but using an interface is more natural and easier.

By using and extra 'Factory' class to deliver either a `DBHandler` or a `FileHandler` the `DataDemo` application could actually be rewritten in such a way that it doesn't even need to know about these two classes. This is an example of the *Factory design pattern.* There are many design patterns for OO software development but their study is beyond the scope of this course.

DataHandler.java

```
public interface DataHandler {
    public boolean write(String f, String s, String id);
    public void displayUsers(java.io.PrintStream out);
}
```

DBHandler.java

```java
import java.io.*;
import java.sql.*;

class DBHandler implements DataHandler {
    private Statement myStatement;

    public DBHandler() {
        try {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            String sourceURL =
                "jdbc:odbc:Driver={Microsoft Access Driver (*.mdb)};DBQ=UserDB.mdb;";
            Connection userDB = DriverManager.getConnection(sourceURL, "admin", "");
            myStatement = userDB.createStatement();
        }
        // The following exceptions must be caught
        catch (ClassNotFoundException cnfe) {
            System.out.println(cnfe);
        }
        catch (SQLException sqle) {
            System.out.println(sqle);
        }
    }

    public boolean write(String f, String s, String id) {
        String writeString =
            "INSERT INTO Users(Firstname, Surname, Id) VALUES('"
            + f + "', '" + s + "', '" + id + "')";
        try {
            myStatement.executeUpdate(writeString);
        }
        catch (SQLException sqle) {
            return false; // duplicate key
        }
        return true; // inserted OK
    }

    public void displayUsers(PrintStream out) {
        try {
            ResultSet results = myStatement.executeQuery
                ("SELECT Firstname, Surname, Id FROM Users");
            while (results.next()) {
                out.print(results.getString(1) + " ");
                out.print(results.getString(2) + " ");
                out.println(results.getString(3));
            }
            results.close();
        }
        catch (SQLException sqle) {
            System.out.println(sqle);
        }
    }
}
```

Apart from implementing the `DataHandler` interface (and not sorting the result set) this is the same as the class listed in Chapter 15.

FileHandler.java

```java
import java.io.*;
import java.util.Date;

class FileHandler implements DataHandler {

    public boolean write(String f, String s, String id) {
        try {
            PrintWriter outFile =
                new PrintWriter(new FileWriter("Users.txt", true));
            outFile.println(new Date() + ": " +
                                f + " " + s + " " + id);
            outFile.close();
13         }
14         catch (IOException ioe) {
15             System.out.println("File Users.txt not found");
16             return false;
17         }
18         return true; // inserted OK
19     }
20
21     public void displayUsers(PrintStream out) {
22         try {
23             BufferedReader inFile =
24                 new BufferedReader(new FileReader("Users.txt"));
25             String line;
26             while ((line = inFile.readLine()) != null)
27                 out.println(line);
28             inFile.close();
29         }
30         catch (IOException ioe) {
31             System.out.println(ioe);
32         }
33     }
34 }
```

Lines 02, 10-11: The `Date` class makes it very easy to "date stamp" the output lines. You may find this technique useful for your coursework.

DataDemo.java

```
01 import java.awt.*;
02 import javax.swing.*;
03 import java.awt.event.*;
04
05 public class DataDemo extends JFrame implements ActionListener {
06     JTextField firstName, surName, loginId;
07     JButton writeBtn, displayBtn;
08     static DataHandler dh;
09
10     public static void main(String[] args) {
11         // will we be using the file or the database?
12         try {
13             String dataType = args[0].toUpperCase();
14             if (dataType.equals("DB"))
15                 dh = new DBHandler();
16             else if (dataType.equals("FILE"))
17                 dh = new FileHandler();
18             else throw new Exception();
19         }
20         catch(Exception e) {
21             System.out.println("Argument must be FILE or DB");
22             System.exit(0);
23         }
24         new DataDemo();
25     }
...    ... as for DBDemo3 Chapter 15
78 }
```

Line 08: we use a variable of type `DataHandler`.

Lines 12 – 23: this code either sets `dh` to a new `DBHandler` or a new `FileHandler`. It halts the program with an error message if the argument is missing or it isn't FILE or DB or file or db, etc (the argument is forced to upper case on input).

Note that line 18 throws an exception if the argument isn't right, which will be caught at lines 20 – 23 (this also traps the "argument missing" exception).

Summary

In this section we have looked at abstract classes and interfaces. Both of these allow us to specify, but not implement, certain methods. These must be implemented by subclasses or implementing classes.

An interface is really just a totally abstract class – it cannot contain any implementation details.

A class may *extend* at most one superclass (abstract or otherwise) and it may also *implement* any number of interfaces.

Abstract classes and interfaces promote the re-use of code in Java and similar languages.

Solutions to self-test exercises

1.

```
13        t.addPoint(x, y);
14        t.addPoint(x+width, y);
15        t.addPoint(x+width/2, y+height);
```

2.

```
13        t.addPoint(x+width/2, y);
14        t.addPoint(x+width, y+height/2);
15        t.addPoint(x+width/2, y+height);
16        t.addPoint(x, y+height/2);
```

3. This *is* puzzling – a tagged interface appears to do nothing useful at all. But it serves to *label* or *tag* a class as being of a certain type. For instance if `ActionListener` wasn't a sub-interface of `EventListener` then we couldn't handle action events in our GUI programs because of the way the Java event handling model works.