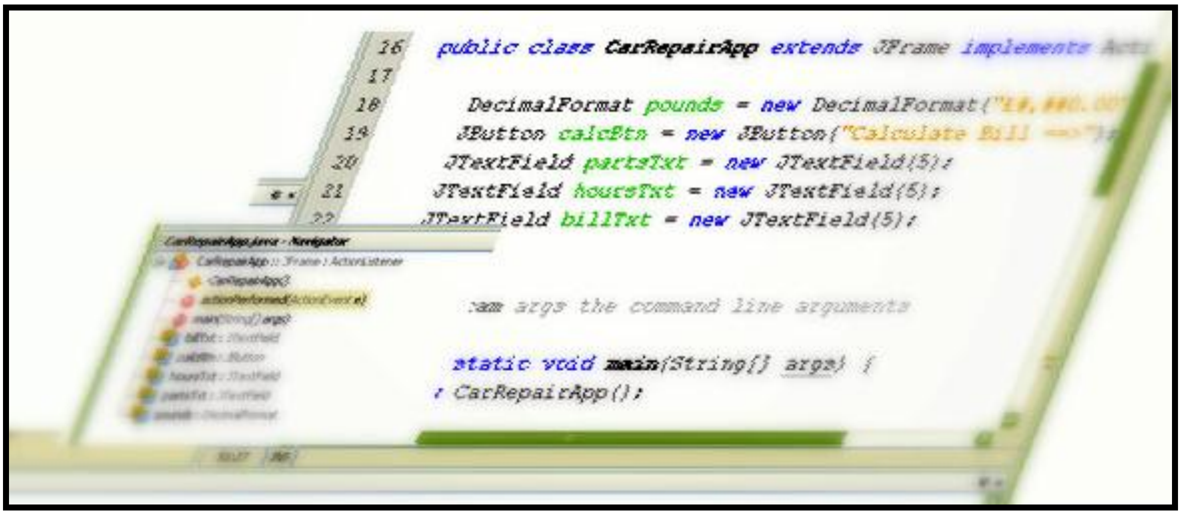
INTRODUCTION TO PROGRAMMING IN JAVA (PART 1)



```
public class CarRepairApp extends JFrame implements Acti

  DecimalFormat pounds = new DecimalFormat(
  JButton calcBtn = new JButton("Calculate Bill -->");
  JTextField partsTxt = new JTextField(5);
  JTextField hoursTxt = new JTextField(5);
  JTextField billTxt = new JTextField(5);

  ram args the command line arguments

  static void main(String[] args) {
  CarRepairApp();
```

School of Computing and Mathematical Sciences

September 2012

UNIVERSITY of GREENWICH

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 mostly written and developed 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

1. Introduction

What is Java?

Java is an OOP (Object-Oriented Programming) language developed by Sun Microsystems in 1990.

Now on Version 6, it is widely used for the development of web-based (and other distributed) software. Its features include architectural neutrality (i.e. it doesn't depend on a specific machine architecture), security, robustness and multi-threading. It offers extensive application programmer interfaces (APIs) including support for graphical user interfaces (GUIs), utilities, database access, networking etc.

What can you do with it?

Amongst many other things, you can

- Produce stand-alone *applications* which run directly on your computer.
- Produce *applets* on a world-wide web server and embed them in your web pages to be run in a browser such as Microsoft Internet Explorer.
- Produce *servlets* on a web server. Unlike applets, these will run on the web server machine; typically these are used to provide dynamic pages by communicating with web databases.
- Produce applications which run on mobile devices. With the exception of the iPhone, Java is nearly ubiquitous on mobile phones.
- Produce things called *java beans* which are roughly software modules for carrying out specific tasks.

We will concentrate on stand-alone *applications*.

How does it work?

Java was created by Sun Microsystems with the aim of producing platform-independence. This means that a program can be written on one type of machine (an IBM PC say) and can be run on any other type of machine (an Apple Macintosh say) without alteration. This is achieved by making compiled Java programs (architecturally neutral *bytecode*) run on a Java Virtual machine (JVM) so that any computer with the JVM installed can run Java programs. The JVM is built into Internet Explorer, Firefox and other web browsers, hence the importance of Java for the Internet.

Java source code (which can be written in any text editor) is passed to the Java Compiler which checks the code for errors and gives a report if it finds any.

When there are no errors it produces a Java class file (bytecode). Depending what it was intended for, this file can either be run as a stand alone *application* or as an *applet* to be added to a HTML document using either the <Applet> or <Object> tag. When a browser comes across one of these tags it downloads from the server the compiled version of the applet (the .class file), along with any other classes the applet needs.

Creating your own Java applications

Software required

As a registered student at the University of Greenwich you have access to a great deal of free software under an academic license. For details, consult the CMS Lab Guide that you were given during week one and, if you want to install it at home, see the information on the student intranet at

http://labs.cms.gre.ac.uk/software/home.asp

The software tools (apart from an editor) that you need to let you develop Java programs are packaged into Sun's Java Software Development Kits (SDKs) or Java Development Kits (JDKs). In the labs you should be able to find J2SDK Version 6 installed (it may be in different places depending on the lab). You will be able to install this on your own PC using the free CD image download provided from the link above. You can also download the latest version of the JDK together with NetBeans directly from http://java.sun.com/ although this is quite a large file.

Editors and IDEs

You can use any text editor for writing Java programs. At the most basic level, NotePad is fine for simple programs, but it provides absolutely no support for Java.

A little better are more advanced text editors such as EditPlus and JEdit which can be configured to help you work with Java.

At the other end of the scale, there are many *Integrated Development Environments (IDEs)* for Java programming; free ones like *NetBeans* and *Eclipse*, and ones that you pay money for, like *JBuilder* and *Together*.

For this course we shall be using NetBeans – this should give you an introduction to good programming techniques and provides an excellent basis for developing your programming skills. It can also be used "straight out of the box", meaning that you should not need to configure it if you want to set it up at home.

However, you should always bear in mind that NetBeans is not part of Java and that Java can be written, compiled and run in a number of different ways (we shall cover some of these later in the course).

Java files (.java, .class, .jar), packages and projects

Whichever editor or IDE you use, when you start to write code you will create a *source program*, e.g. `Hello1.java`.

Note that all Java source programs must have the extension **`.java`*. Also, the program name (e.g. Hello1) should start with an UPPER CASE LETTER.***

The text source program is converted into a bytecode version with the same name but extension `.class` by using the Java compiler `javac`, e.g.

```
javac Hello1.java
```

which will create `Hello1.class` if there are no errors in the source program.

The bytecode version may be:

- a standalone *application*, which could be run in a command window using the `java` command, e.g.

  ```
  java Hello1
  ```

  Note that you do not include the `.class` extension with this command. Also, as Java is CASE SENSITIVE you would need to type this command exactly as shown, with a capital H.

- an *applet*, in which case it is called from a web page and run in a browser, or by the applet viewer which is part of Sun Microsystem's java development kit (jdk).

Apart from the very simplest programs, however, most applications involve several classes and sometimes hundreds or even thousands. Typically a composite application such as this is referred to informally as a *project*.

The word “project” is not a part of the Java specification but it is widely used. To make access to the examples easy we have created two NetBeans projects for you to download – JavaTerm1 and JavaTerm2.

To organise such *projects*, especially those with many classes, it is normal to sort the classes into different folders or *packages* as they are known in the Java context.

For example, a Java game might have all of the classes related to displaying the game in one package and all of the classes related to how the game is controlled in another.

In this course we have organised the example code so that all of the examples for Chapter 1 are in a package called `ch01_intro`, and so on. You should try to keep your own work organised in this way so that it is easier to find things when you are practising your programming, completing your weekly lab work, writing your coursework or revising for tests and exams.

Finally, for ease of distribution and use, multiple class files can be combined together, using the `jar` command, into a single Java archive file with the `.jar` extension.

A .jar file is very like a .zip file and in fact .jar files can be opened by WinZip. We shall not be using them on this course but we will discuss them briefly later on.

Creating your own Java applications in NetBeans

The good news is that NetBeans handles all of these file types automatically. Thus we just need to write the source program and a single button press can compile the code, create a jar file and run it.

Your first application – Hello world

This is just about the simplest program you can write. When run, it displays `"Hello world"`. We follow the steps outlined above.

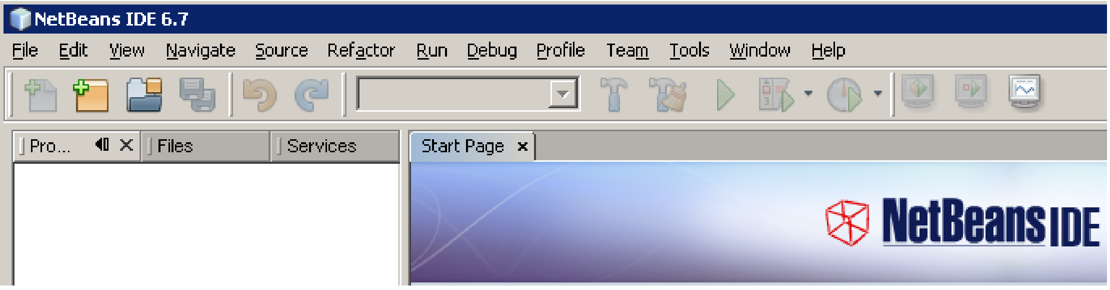
The first step is to create the `.java` (“source”) file. We will use NetBeans[1].

[1] You may have already done an exercise just like this during lab day in week one, but make sure you are able do it now too.

Starting NetBeans

Although it is not difficult to create a new project from scratch you may as well start from the project containing the Term 1 examples. Download JavaTerm1.zip from the COMP1148 TeachMat schedule and unzip it into your J drive.

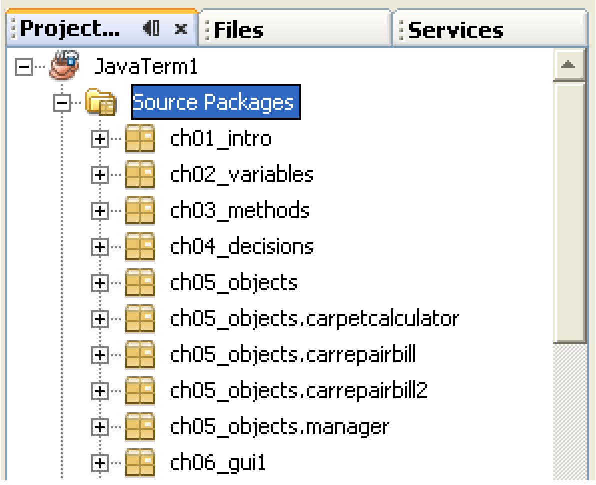
Next open up NetBeans. It is available on all the lab machines but, depending on how it has been installed, it might be in a variety of locations on the Start menu (e.g. **Start | All Programs | Programming | Java | NetBeans** or **Start | All Programs | NetBeans** or **Start | All Programs | Java | NetBeans**).


NetBeans IDE 6.7
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Pro... Files Services
Start Page
NetBeans IDE

This shows the top left corner only.

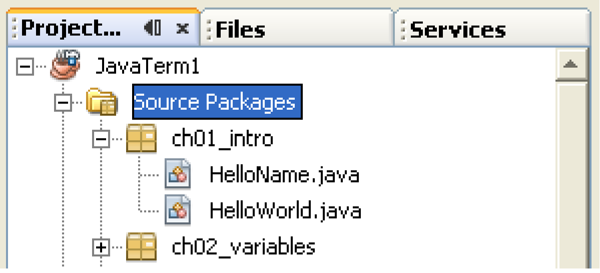
To open the Term 1 examples click on the third icon from the left, or select **File | Open Project...** and browse to its location . You can recognise NetBeans projects by the coffee cup icon.

Once you have opened the project, it should appear in the Projects tab (top left). Double click on it and then double click on "Source Packages". You should see something like this showing all the packages in the project (their names refer to chapters in this workbook):



JavaTerm1
Source Packages
ch01_intro
ch02_variables
ch03_methods
ch04_decisions
ch05_objects
ch05_objects.carpetcalculator
ch05_objects.carrepairbill
ch05_objects.carrepairbill2
ch05_objects.manager
ch06_gui1

Finally double click on `ch01_intro` and you should see this, showing all the classes in this package:

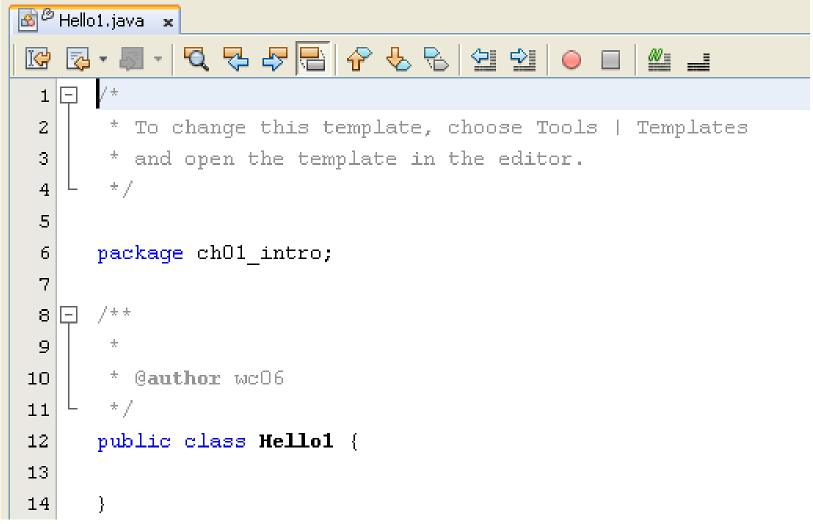


JavaTerm1
Source Packages
ch01_intro
HelloName.java
HelloWorld.java
ch02_variables

Writing your program

There are already a couple of examples there, but we will create our own.

Right click on `ch01_intro` and select **New | Java Class...** Then in the dialog box that pops up, change the class name to Hello1. You should see something like this.



```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package ch01_intro;

/**
 *
 * @author wc06
 */
public class Hello1 {

}
```

As you can see, NetBeans has written a ‘skeleton’ Java application.

Now modify it by deleting the comments (in grey) and typing the rest so that it matches the code below. The finished program should be EXACTLY as follows:

```
package ch01_intro;


// this is my first java program
public class Hello1 {

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

Points to note:

1. Java is *case sensitive*. For example `Hello1` is not the same as `hello1`. Bear this in mind at all times; it can give you a *lot* of trouble!

2. *Spaces are significant*. For example `Hello 1` is not the same as `Hello1`. Bear this in mind at all times; it can give you a *lot* of trouble!

   Forgetting points 1. and 2. probably account for more than half the errors that beginner programmers make.

3. A Java program consists of a number of *classes*, which contain *methods*, which contain *statements*. Simple programs will have just one class (here it is called `Hello1`) with a `main` method as above. Statements in Java always end with a semicolon (;). Moving to a new line does *not* end a statement. Our first program has a single statement

   ```
   System.out.println("Hello World!");
   ```

that prints the *`String`* `"Hello World!"` in the output window – see below (next page).

4. Java uses THREE different types of bracket:

   a. Curly brackets `{` and `}` are used to group statements together. No semicolon is needed after the closing }

   b. Round brackets `(` and `)` are used for arithmetic, e.g. `(a + b) * 2` which can be read as “add a to b, then multiply by 2” and for *parameters* and *arguments* of methods. Here `"Hello world"` is the single *argument* of the *call* to the `System.out.println` method and `args` is the single *parameter* used in the *definition* of the `main` method and.

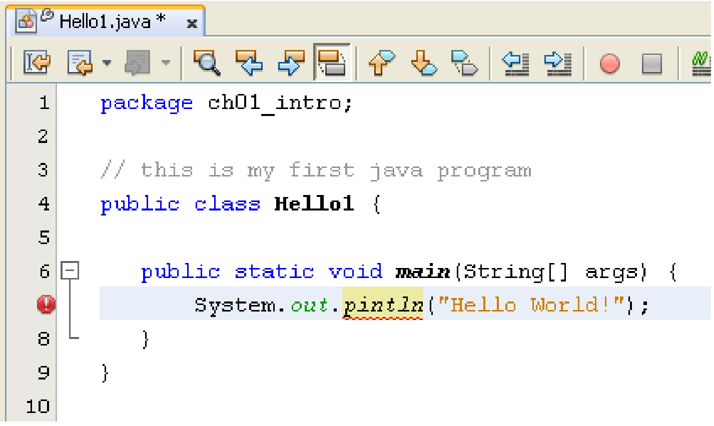
   c. Square brackets `[` and `]` are used in defining and using *arrays*. We won’t be using arrays for some time, *except* for the arguments of the `main` method.

5. The program's file name *must* match (case-sensitive!) the class name. So, as your file is called `Hello1.java`, you *must* call the class `Hello1`. Failure to do so will result in an error.

6. You can put in comments such as
   `// this is my first java program`
   to help explain your code. Everything after the `//` is ignored by Java.

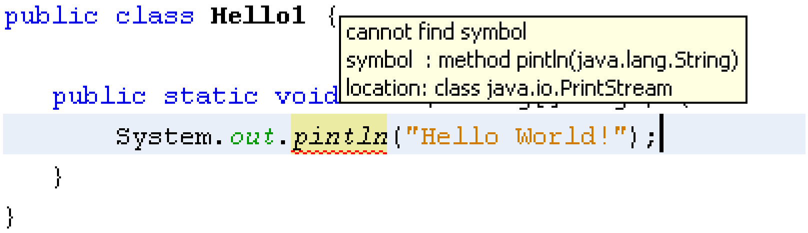
Checking and compiling your program

At this point you may find that you have errors indicated by a red square in the top right corner and a red squiggle under the error(s):



```
package ch01_intro;

// this is my first java program
public class Hello1 {

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

```

You can hover over the squiggle to find out what Java thinks the error is:



```
public class Hello1 {

    public static void
        System.out.pintln("Hello World!");
    }
}
```

cannot find symbol
symbol : method pintln(java.lang.String)
location: class java.io.PrintStream

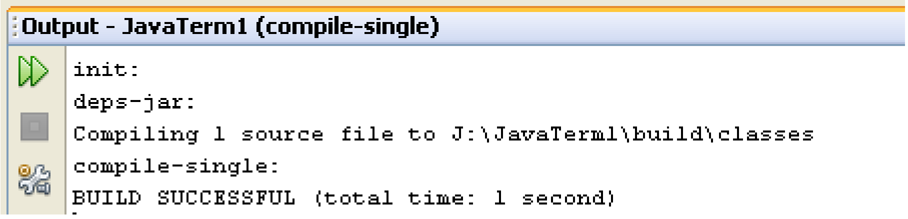
Java and NetBeans do their best to indicate what and where the error is but they don't always get it right. This code looks correct but in fact the user has missed out the letter r. (When typing println, be careful not to confuse the letter l [ell] with the number 1 [one] or the capital I [eye].) If you need to you can change the font size by selecting **Tools | Options** from the menu and choosing the **Fonts & Colors** tab.

Learning how to correct errors is one of the most important skills you will need to develop if you want to master programming.

Once all the errors are correct you should get a green square in the top right. Sometimes, if there are no errors but Java wants to warn you about something you may get a yellow square instead.

Now save and compile your program by right-clicking on Hello1 in the Projects tab (top left) and selecting **Compile File**. Alternatively you can just hit F9.

Check to see whether it has worked by looking in the Output tab (bottom left):



Output - JavaTerm1 (compile-single)

```
init:
deps-jar:
Compiling 1 source file to J:\JavaTerm1\build\classes
compile-single:
BUILD SUCCESSFUL (total time: 1 second)
```

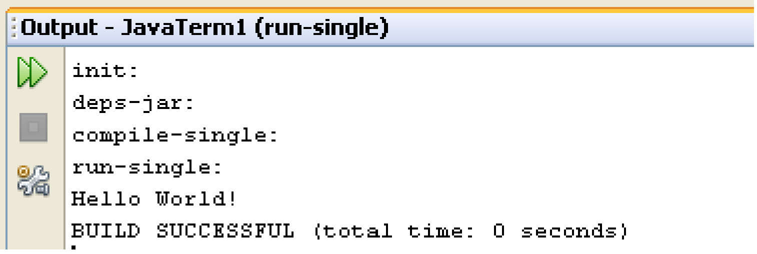
NetBeans will have created a file `Hello1.class` in the `ch01_intro` package of the `build\classes` folder of your project. This file is in *bytecode* and can't be read by humans.

You can check it is there by looking in the folder, but you should never need to access it directly.

Running your program

Now you are ready to run your program. Do this by right-clicking on Hello1 in the Projects tab and selecting **Run File**. Alternatively you can just hit Shift+F6.

If all runs smoothly, you should get something like this:



Output - JavaTerm1 (run-single)

```
init:
deps-jar:
compile-single:
run-single:
Hello World!
BUILD SUCCESSFUL (total time: 0 seconds)
```

The first few lines tell you what NetBeans has done to run the program (note that in fact running the file also saves it and will compile it so strictly speaking there's no need to compile it separately if you are planning to run it).

The actual output "Hello World" follows that. Finally there is a line telling you that the program ran successfully and how long the whole process took.

It is possible that, although the compiler detected no errors, something goes wrong when you run the program. For instance, if you left out the keyword `static` in line 3 the compiler wouldn’t complain but you would get an error message when you try to run the program:

```
Class "ch01_intro.Hello1" does not have a main method.
```

Such errors are called *run-time errors* or *exceptions* as opposed to *compile-time errors*.

> Run-time errors can be more difficult to understand and correct than compile-time errors. For instance it is not obvious what’s wrong here – after all we do have a `main` method. The java runtime system expects this method to be `static` – whatever that is! (We will find out later). You would get exactly the same run-time error if you included the `static` keyword but left out the brackets `[]` after `String`.

Second application – Hello world with *user input*

Your first application produced one line of output (`"Hello world"`) but there was no *input*. Let us change it so that the user can input their name (say `"Chris"`) and the code then outputs `"Hello Chris"`.

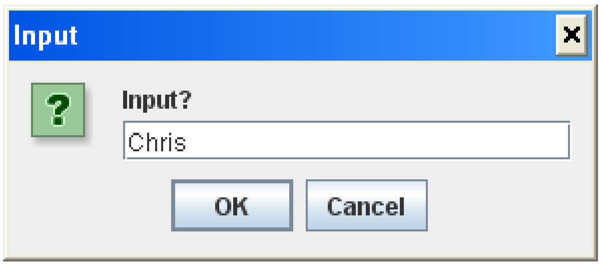
First create a new Java class called Hello2 so that it reads as follows (see page 5 if you have forgotten how to do this):

```
01 package ch01_intro;
02
03 import javax.swing.*;
04
05 class Hello2 {
06
07     public static void main(String[] args) {
08         String name;
09         name = JOptionPane.showInputDialog("Input?");
10         System.out.println("Hello " + name);
11     }
12 }
```

Note that for the moment we will use `JOptionPane.showInputDialog()`, to get text into the program without attempting to understand how it works. However, note that it also requires the addition code at line 03 to be error free.

Compile and run the program right-clicking on Hello2 in the Projects tab and selecting **Run File**. Alternatively you can just hit Shift+F6.

A prompt dialog should appear; type the name Chris into it, like this:



Input

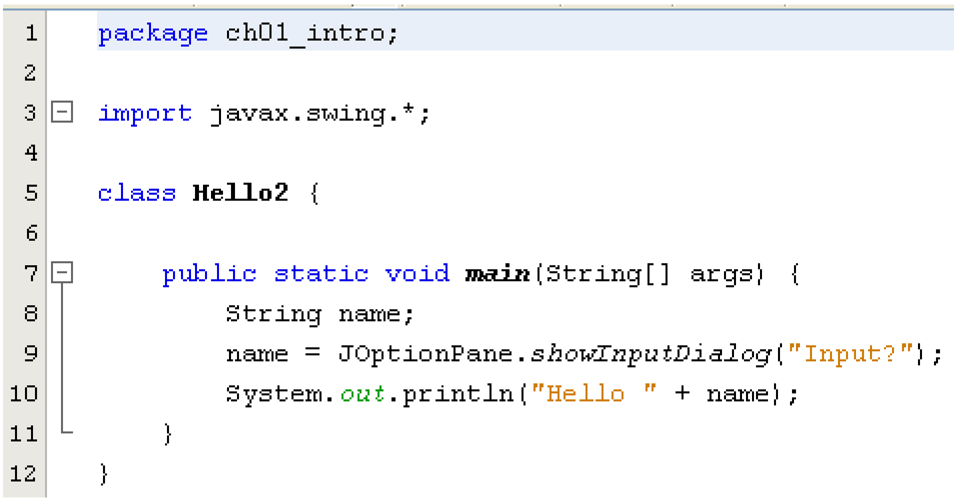
Input?

Chris

Now run it again and type Kate.

Explanation of this code

Here is a screen dump of `Hello2.java`, as it looks in NetBeans:



```
1   package ch01_intro;
2
3   import javax.swing.*;
4
5   class Hello2 {
6
7       public static void main(String[] args) {
8           String name;
9           name = JOptionPane.showInputDialog("Input?");
10          System.out.println("Hello " + name);
11      }
12  }
```

At line 08 we have introduced a `String` *variable* `name`. Variables hold values (in this case a `String` like `"Chris"` or `"Kate"`) within a program, and the values of the variables can change as the program runs.

We look at variables in more detail in the next chapter.

Line 09 is an *assignment* statement which takes whatever the user typed at the prompt and stores it into the variable `name` (i.e. it takes whatever is on the right hand side of the = sign and stores it in the variable on the left hand side).

At line 10 the expression `"Hello " + name` ‘glues’ (or *concatenates*) `String` `"Hello "` and whatever is in the `name` variable. This is then printed out, so that if `name` has the value `"Chris"` the output is

```
Hello Chris
```

If you miss the space at the end of `"Hello "` it will say `HelloChris`.

Self-test exercises

1. What output would you expect if you input the word `Asif`
2. What output would you expect if you input the words `Chris Walshaw`
3. What happens if you hit OK at the prompt without inputting anything? (Try it).
4. If line 09 of the code is changed to

   ```
   System.out.println("Hello " + name + ". How are you?");
   ```

   what output would you expect if the input is `Kate`?

More about NetBeans

NetBeans provides a vast amount of support for Java. First and foremost, it can compile and run Java applications.

In addition, it indicates some of the Java syntax[2]:

- It displays keywords like `class` and `public` in **blue**.
- Text inside string quotes `"  "` is displayed in **brown**.
- Comment text (e.g. after `//` on a line) is displayed in **grey**.
- It underlines errors in **red**.

> This colouring is not just to make your programs look pretty. It helps you to spot mistakes as you type, for instance:
>
> If you type `Public` instead of `public` it won't be coloured blue.
>
> If you forget to type a closing string quote `"` the rest of that line will be brown.

There is masses more to discover about NetBeans which we shall find out throughout the course.

Summary

This chapter has introduced Java with some simple "hello world" applications.

A Java application has at least one *class* with a *main* method that has one or more *statements*.

We have seen that Java applications may be written and modified using NetBeans. They must be written in a file whose name is the same as the class name.

To save and compile a program in NetBeans, right-click on it and select **Compile File**. Alternatively you can just hit F9.

To save, compile and run a program in NetBeans, right-click on it and select **Run File**. Alternatively you can just hit Shift+F6.

You can use System`.out.println(...);` for output.

Solutions to self-test exercises

1. The output will be

   ```
   Hello Asif
   ```

2. The output will be

   ```
   Hello Chris Walshaw
   ```

3. The output will be

   ```
   Hello
   ```

4. The output will be

   ```
   Hello Kate. How are you?
   ```

[2] *Syntax* is the rules and principles that govern how a program is written and structured.

2. Variables and Input/Output (I/O)

Variables

All but the most trivial of programs use *variables*.

A variable is a kind of storage box used to remember values, so that they can be used or altered later on in the program.

Variables are of different kinds and we say that a variable has a particular *data type*.

We have already used a `String` variable called `name` in program `Hello2.java`, reproduced here:

```
01 package ch01_intro;
02
03 import javax.swing.*;
04
05 class Hello2 {
06
07     public static void main(String[] args) {
08         String name;
09         name = JOptionPane.showInputDialog("Input?");
10         System.out.println("Hello " + name);
11     }
12 }
```

To begin to get into good design habits we will list the data that occurs in this class:

Hello2
–name: String

This is a simplified version of the *Universal Modelling Language* (UML) for classes. We will use the full notation later in this course.

The – sign before `name` means that the variable can’t be used outside the class. (In fact it can only be used inside `main`).

Variable `name` is *declared* in line 08 and line 09 is an *assignment statement* that gives it a value, here the value of the user input. Lines 08 and 09 could be combined like this:

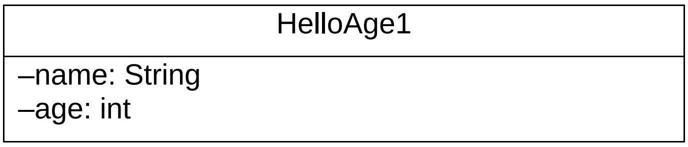
```
     String name = JOptionPane.showInputDialog("Input?");
```

In line 08 `name` is used as part of the *argument* to an *output* statement that prints a line of text to the output window, referred to in Java by `System.out`.

Simple Java programs use numbers and strings. Here is a modification of `Hello2.java` that accepts a person's name and age and prints a suitable message:

```
01 package ch02_variables;
02
03 import javax.swing.*;
04
05 class HelloAge1 {
06
07     public static void main(String[] args) {
08         String name = JOptionPane.showInputDialog("Input?");
09         int age = Integer.parseInt(
                JOptionPane.showInputDialog("Input?"));
10         System.out.println("Hello " + age + " year old " + name);
11     }
12 }
```

We must modify our "class diagram" as we have a second variable `age`:


HelloAge1

–name: String
–age: int

Note that this new version asks the user for two pieces of input, hence `JOptionPane.showInputDialog()` appears twice in the program.

Line 07 declares a whole number `age` (in Java a whole number is declared as an `int`) and assigns it the value of the second argument. Because `JOptionPane.showInputDialog()` always gives us a `String` it must be converted into an `int` using the `parseInt` method of the library class `Integer`. The Java compiler would give an error if line 06 was

```
6       int age = JOptionPane.showInputDialog("Input?");
```

Self-test exercises

1. Suppose line 08 of `HelloAge1.java` was

```
08         System.out.println("Hello " + name + " you are " + age);
```

   instead. What output would you expect for the inputs "`Kate`" and "`16`"?

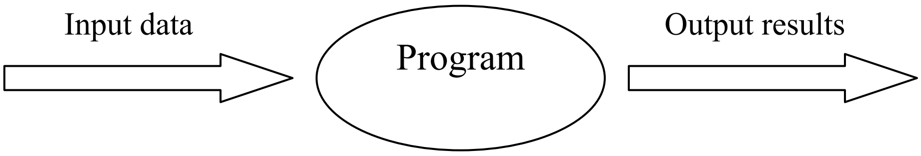
2. Suppose we also inserted the following statement just after line 6 of `HelloAge1.java`:

   `name = "Mr. " + name;`

   What output would you expect for the input "`Asif`" and "`75`"?

Input/Output

Most programs accept some *input data* which is processed to produce *output results*



So far we have seen very simple I/O with the input data being typed in the prompt dialogs and the output results being printed back to the output window.

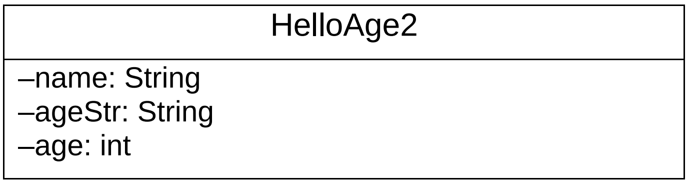
In Java input may be typed in by the program user, read from files or databases or accepted over a communications network. Output may be displayed to the user, written to files or databases or sent over a network.

For most of our programs we will restrict attention to user I/O. Java has many ways of doing this. So far we have used a utility method without knowing how it works and with no control over what it does.

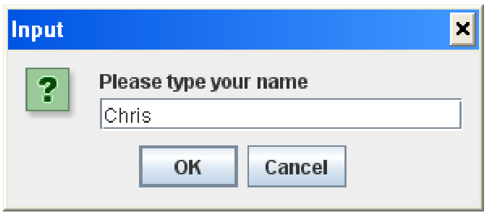
However, it's quite easy to customise the *dialogs* to request input from the user and to display output:

```
01 package ch02_variables;
02
03 import javax.swing.*;
04
05 class HelloAge2 {
06
07     public static void main(String[] args) {
08         String name =
09             JOptionPane.showInputDialog("Please type your name");
10         String ageStr =
11             JOptionPane.showInputDialog("Please type your age");
12         int age = Integer.parseInt(ageStr);
13         JOptionPane.showMessageDialog(null,
14             "Hello " + age + " year old " + name);
15     }
16 }
```

We must modify our class diagram again as we have a third variable `ageStr` .



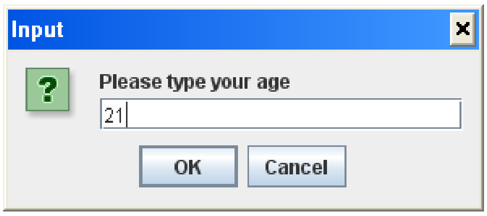
When we run this program, the assignment statement on lines 08-09 pops up the *input box* shown here. The user types some text into the box and clicks OK, and the text is assigned to `String` variable `name`.



Please type your name

Chris

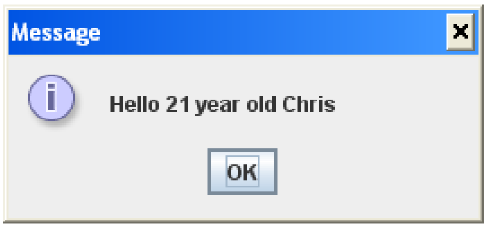
Next the assignment statement on lines 10-11 pops up the second input box shown. The user types some more text (which should be a whole number) and clicks OK, and the text is assigned to `String` variable `ageStr`. This is converted into an `int age` by the assignment at line 12.



Please type your age

21

Finally the statement on lines 13-14 pops up the output *message box* shown here.



Hello 21 year old Chris

Explanation of the code

The input and message *dialogs* come from the Java library class `JOptionPane`. Note the spelling!

`JOptionPane` is part of the Java “Swing” package for writing Windows-like programs. The line

```
03 import javax.swing.*;
```

is needed to tell the Java compiler that you are using this package.

Lines 09 and 11 show the use of the `JOptionPane` `showInputDialog` method. This has a `String` argument to be displayed as a prompt for the user. The user should type something into the box and click OK; whatever is typed is returned as a `String`.

Lines 13-14 displays the output in a message box.

```
13       JOptionPane.showMessageDialog(null,
14           "Hello " + age + " year old " + name);
```

Writing `null` here causes the message box to pop up in the middle of the screen. There are other possibilities which we don’t use.

This expression is evaluated as a single `String` to be displayed

Note that the assignment statement 08-09 is written on two lines just for readability – you could type it just on one line. Likewise for the statements 10-11 and 13-14.

import statements

Java comes with a huge library of re-usable code called the *Application Programmer’s Interface* or API. We cover this in the second workbook but for now, note that you must include `import` statements at the beginning of your code to use portions of this library. For instance in the `HelloAge2` program we needed the statement

```
import javax.swing.*;
```

to use the Swing class `JOptionPane`.

> **Note**: The `*` in the `import` statement is a *wildcard* which tells the Java compiler that we may be using any of the classes in the `javax.swing` package. As we are only using the `JOptionPane` class here we could have used
>
> ```
> import javax.swing.JOptionPane;
> ```
>
> instead.

It is quite common to have several such import statements. However some of the API classes are regarded as so important that they don’t need imports. These include classes `System`, `String` and `Math`. Such classes actually belong to the `java.lang` package.

Static import statements

Java version 5 and later introduced a convenient shorthand facility when using *static*[†] methods and fields from library classes. Have a look at these versions of the two earlier `HelloAge` programs:

```
01 package ch02_variables;
02
03 import static java.lang.System.*;
04
05 class HelloAge3 {
06
07     public static void main(String[] args) {
08         String name = args[0];
09         int age = Integer.parseInt(args[1]);
10         out.println("Hello " + age + " year old " + name);
11     }
12 }
```

In this version of `HelloAge1.java` we have added a *static import* statement for all the static members of the `System` class at line 03 and this means that we can omit the `System.` prefix when referring to any of the static fields (such as `out`) of that class. Compare line 10 of this program with line 09 of `HelloAge1` listed above. There isn't much point here but it will save a lot of typing if we have a lot of references to `System` fields in our program.

We can use the same technique with the Swing version:

```
01 package ch02_variables;
02
03 import static javax.swing.JOptionPane.*;
04
05 class HelloAge4 {
06
07     public static void main(String[] args) {
08         String name =
09                 showInputDialog("Please type your name");
10         String ageStr =
11                 showInputDialog("Please type your age");
12         int age = Integer.parseInt(ageStr);
13         showMessageDialog(null,
14                 "Hello " + age + " year old " + name);
15     }
16 }
```

Here we have replaced the import statement for the Swing package with a static import for all the static members of the `JOptionPane` class instead at line 03. Now we can omit the `JOptionPane.` prefix when referring to any of the static methods of that class. Compare lines 09, 11 and 13 of this program with the corresponding lines of `HelloAge2` listed above. This version is more readable and convenient to type.

You can have more than one such static import in a program and it will work provided that there are no name clashes involved.

Let's look at two more substantial programs now.

[†] We explain what 'static' means a little later. For now, note that the static methods and fields are just the elements of these library classes that we can use directly.

Car repair bill calculator

When you take your car to the garage to be repaired, the bill is usually calculated from the cost of the new parts + the labour cost, and then VAT is added. The labour cost in turn is calculated from the cost per hour multiplied by the number of hours worked.

Suppose that the labour cost per hour is £20 and that VAT is 17.5%.

A program to calculate the bill needs two inputs: the parts cost, and the number of man-hours worked. There is a single output, namely the bill value. For instance if the parts cost is £30 and there are 3 man-hours then the bill before VAT is 30 + 3 * 20 = £90 which (according to my arithmetic) comes to £90 * 1.175 = £105.75.

Note that we use * to denote multiplication in Java (as in all programming languages).

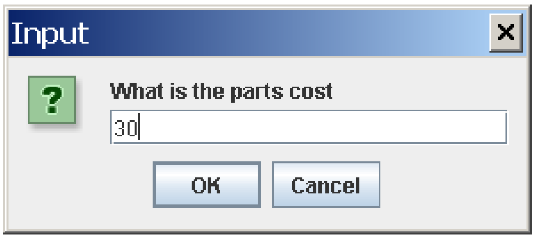
The parts cost, hours worked and bill may have fractional values so we can't use `int`. For numbers with a fractional part we use `double`.

If we are taking in the cost of parts and the hours of labour from the user they will be read in as `String` and then converted to `int.` We therefore need two `String` variables for the inputs and two `double`s for their converted values. In addition we need a double for the output. The data for the class is:

CarRepair
–partsStr: String –hoursStr: String –parts: double –hours: double –bill: double

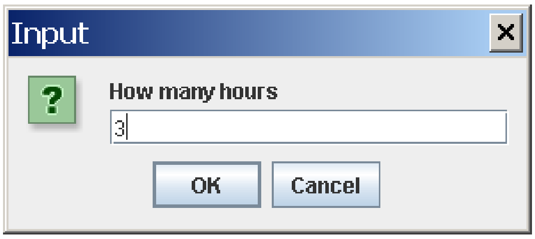
Here is the program, with typical inputs and outputs:

```
package ch02_variables;

import static javax.swing.JOptionPane.*;

class CarRepair {

    public static void main(String[] args) {
      String partsStr = showInputDialog("What is the parts cost");
      String hoursStr = showInputDialog("How many hours");
      double parts = Double.parseDouble(partsStr);
      double hours = Double.parseDouble(hoursStr);
      // calculate bill before VAT
      double bill = parts + hours * 20;
      // add VAT
      bill = bill * 1.175;
      showMessageDialog(null, "Your bill is £" + bill);
    }
}
```



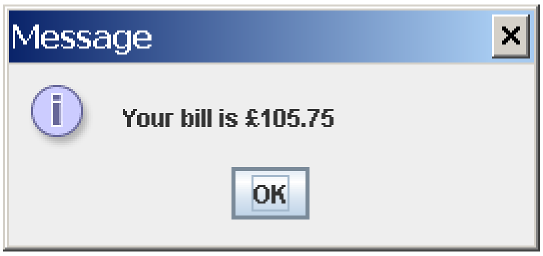
What is the parts cost

30



How many hours

3



Your bill is £105.75

Note that at lines 10 and 11 we use `Double.parseDouble( ... )` to convert from a `String` to a `double` value.

The statement at line 15 may confuse you. Read it as saying “The new value of bill is its old value multiplied by 1.175”. Another way of writing this in Java is

```
15       bill *= 1.175;
```

which you can read as “multiply bill by 1.175”.

Self-test exercises

3. Modify the Car repair bill calculator so that the user can enter the cost per hour instead of assuming that it is £20. Add a new `String` variable `costStr` and a new `double` variable `cost`.

4. Modify the Car repair bill calculator again so that the user can enter the VAT rate instead of assuming that it is 17.5%. Add a new `String` variable `vatStr` and a new `double` variable `vat`.

   Make sure you get the VAT calculation right. In place of

   ```
   bill = bill * 1.175;
   ```

   you will need

   ```
   bill = bill * (1 + vat / 100);
   ```

   `/` is used in Java for division. Note that the brackets are needed to make sure that the addition `1 + vat / 100` is carried out before the multiplication. This is because `*` and / take *precedence* over + (and over `-`).

   A neater way of writing this is

   ```
   bill *= 1 + vat / 100;
   ```

Carpet calculator

Let us design and write a program to calculate the cost of putting a new carpet in a single room. The size of the room is:

- Length: 4 metres
- Width: 3 metres

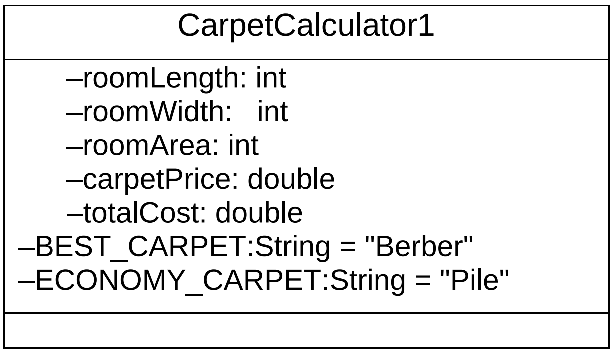
There is a choice of two types of carpet:

- Berber (best) @ £27.95 per square metre
- Pile (economy) @ £15.95 per square metre

Analyze the problem

There are no program inputs – that may come later – but the program uses data. What types of data do we need?

- The room dimensions are given in metres, both whole numbers, so for the moment we will use `int` for these and for the room area. If we were thinking ahead, we might have used `double` so that the user could input a fractional figure (e.g. 4.5 metres).
- The carpet prices are given as a price per square metre. This number has a fractional part so we should use `double` for these.
- We also need to have a `double` variable for the total cost.



```
CarpetCalculator1
    –roomLength: int
    –roomWidth:   int
    –roomArea: int
    –carpetPrice: double
    –totalCost: double
–BEST_CARPET:String = "Berber"
–ECONOMY_CARPET:String = "Pile"
```

So we could use the following data variables:

Length of room in metres
Width of room in metres
Area of room in square metres
Carpet price per square metre
Total cost of carpet

The last two are some useful *constant values*

Constants

In this application the name of the best carpet is “Berber” and the name of the economy carpet is “Pile”. Let’s assume that these are constant and will never change.

The dimensions of the room and the carpet prices are given to us so they could also be constants. However we might want to change the program to allow for input of these quantities.

In Java constant values are specified by the keyword `final` so we shall use

```
final String BEST_CARPET = "Berber";
final String ECONOMY_CARPET = "Pile";
```

Now we can write the Java code:

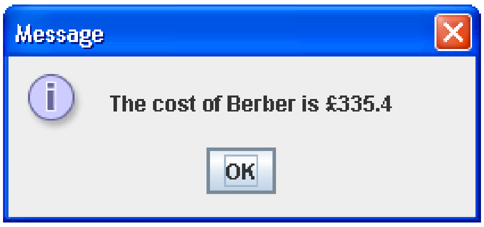
```
01 package ch02_variables;
02
03 import static javax.swing.JOptionPane.*;
04
05 class CarpetCalculator1 {
06
07     public static void main(String[] args) {
08         final String BEST_CARPET = "Berber";
09         final String ECONOMY_CARPET = "Pile";
10
11         int roomLength = 4;
12         int roomWidth = 3;
13         int roomArea = roomLength * roomWidth;
14
15         double carpetPrice, totalCost;
16
17         // best carpet
18         carpetPrice = 27.95;
19         totalCost = roomArea * carpetPrice;
20         showMessageDialog(null,
21            "The cost of " + BEST_CARPET + " is £" + totalCost);
22
23         // economy carpet
24         carpetPrice = 15.95;
25         totalCost = roomArea * carpetPrice;
26         showMessageDialog(null,
27            "The cost of " + ECONOMY_CARPET + " is £" + totalCost);
28     }
29 }
```

Lines 08 – 09 define the two constant values. Lines 11 – 12 declare and set the `int` variables for the room dimensions. (We may want to input these instead). Line 15 declares the two `double` variables used. This sort of declaration is shorthand for two separate declarations

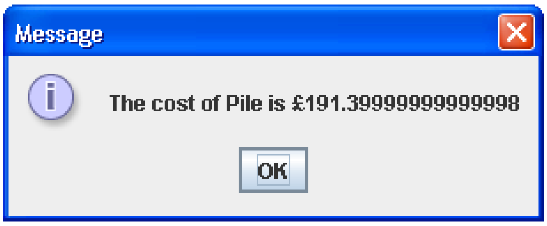
```
double carpetPrice;
double totalCost;
```

Lines 13 declares the variable for the room area and calculates it at the same time. Lines 17 – 21 (23 – 27) calculate and display the total cost for best carpet (economy carpet).

When the application runs, the following message boxes are displayed one after another.



The cost of Berber is £335.4



The cost of Pile is £191.39999999999998

Unfortunately neither cost is displayed as we would like. It would be better to round the costs to two decimal places, so that the actual costs displayed are £335.40 and £191.40 respectively. A tutorial exercise shows how to do this by using the Java library class `DecimalFormat`.

Self-test exercise

5. How would you modify the Carpet Calculator so that the user can enter the room length and width?

Variable types in Java

So far we have used three types of data variables – `String` for text items, `int` for whole numbers and `double` for numbers with a fractional part. `int` and `double` are *primitive* types which are understood directly by the Java run-time system. The `String` type is *not* primitive, being defined by the Java API class `String`.

There are eight primitive data types in Java. All other data types are defined by Java *classes*, which are ultimately derived from these primitive types, as we shall see. These primitive types are described below for the sake of completeness, `int` and `double` being the most commonly used. `int` is OK for whole numbers in the range roughly + or – 2000000000 and `double` gives an accuracy of 17 significant figures.

Primitive Data Types

Keyword	**Description**	**Size**
(whole numbers – integers)		
`byte`	Byte-length integer	8-bit
`short`	Short integer	16-bit
`int`	**Integer**	**32-bit**
`long`	Long integer	64-bit
(fractional numbers – reals)		
`float`	Single-precision floating point	32-bit
`double`	**Double-precision floating point**	**64-bit**
(other types)		
`char`	A single character	16-bit
`boolean`	A boolean value (`true` or `false`)	true or false

Naming things in Java

Java programs deal with *classes*, *methods*, *variables* and *constants*. Each of these items has a name.

As mentioned in the first chapter, it is VERY IMPORTANT to realise that Java is *case sensitive* so that `A` is different from `a`. Java names:

- must start with a letter, _ or `$` character
- must not contain any spaces
- can contain any number of letters, digits, _ and `$` characters

In addition, there is a widely accepted Java *style* that says:

- Begin every *variable* or *method* name with a *lower-case* letter
- Begin every *class* name with an *UPPER-CASE* letter
- Use meaningful names
- For names using more than one word, begin each word after the first with an *upperCaseLetter* (this is sometimes known as *camel* notation)
- For constants (`final` variables) use all UPPER_CASE letters with _ characters between words

Examples:

```
roomLength
salaryOfEmployee
main
println
showInputDialog
```

Variable and method names

```
VAT
INCHES_PER_FOOT
```

Constant names

```
CarRepair
Hello1
CarpetCalculator1
String
System
JOptionPane
DecimalFormat
```

Class names

There are lots of keyword names reserved by Java and if you use one accidentally you will quite likely get a confusing error message. If you use NetBeans such a name appears in blue so you know right away that you can't use it.

The Java API has literally thousands of classes. So far we have used

- `String` for declaring text variables
- `System`. This class has an object called `out` which is Java's standard output stream. We can send this object `print` or `println` messages to write data as in

```
System.out.println("Hello world");
```

(`print` is just like `println` except that it doesn't move on to the next line.)

It has other useful objects and methods. For instance

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

may be used to stop a program running at any time.

- `JOptionPane`. As we have seen, this class has useful methods to pop up input and message dialog boxes.

Self-test exercise

6. Which of the following variable names are allowed in Java, and which have the correct style?

```
volume          AREA            Length          3sides
side1           getFirst        lenth           mysalary
your salary     screenSize      double
```

Java arithmetic

We have seen that we can use *operators* such as `+` (addition) `*` (multiplication) and `/` (division) in Java. Here is a more complete list of operators:

Operator	**Meaning**	**Note**
`*`	Multiplication	In complex expressions, `*` `/` and `%` take precedence over `+` and `-` . This can be overridden using brackets. For example the value of `1 + 2 * 3` is `7` but the value of `(1 + 2) * 3` is `9`
`/`	Division	
`%`	Remainder	
`+`	Addition	
`-`	Subtraction	
`++`	Add 1	`i++` has the same effect as `i = i + 1`
`--`	Subtract 1	`i--` has the same effect as `i = i - 1`
`*=, /=, %=, +=, -=`	Multiply by, etc	`x *= 2` has same effect as `x = x * 2,` etc

You must be careful with division. This is because `x / y` has an `int` result if both `x` and `y` are `int`s. So if `x` has the value `2` and `y` has the value `3`, `x / y` is `0` and *not* `0.6666666666666667`. Java truncates (chops off) any decimal places.

`%` is the *remainder* operator. `x % y` has an `int` result only if both `x` and `y` are `int`s. For instance,

`7 % 4` is `3`,   `8.6 % 2` is `0.6`   and   `8.7 % 1.5` is `1.2`

`%` is hardly ever used for `double`s.

Casting

Sometimes you need an `int` to turn into a `double`, or vice-versa. If the conversion will lose some information, or if you want to make sure the calculation is done using a particular type, this usually needs a *cast*.

For example, in the car repair application, if the customer is going to pay cash you might want to round down the bill to the nearest pound (the nearest whole number). The information lost is the “pence” part of the bill.

Consider:

```
1    int ivalue = 33;
2    double dvalue = 3.9;
3    int i;
4    double x;
5    x = ivalue;            // x becomes 33.0
6    i = (int) dvalue;      // i becomes 3 using (int) cast
7    x = (double) (10+11)/2; // x becomes 10.5 using (double) cast
```

- The first assignment on line 5 doesn't need a cast as no information is lost.
- The second assignment needs the cast `(int)` as information will be lost. Leaving the cast out would give a compiler error.
- In the third assignment the expression `(10+11)` is evaluated to give `21` which is then cast to `21.0` and the division by `2` gives `10.5`. If the cast `(double)` was left out the result would be `10.0`.

Comments in Java

You can insert comments into Java programs in two ways:

- Everything on a line after `//` is ignored.
- For multi-line comments you can start with `/*` and then everything up to `*/` will be ignored.
- Remember that NetBeans colours all commented text grey.

Self-test exercises

7. Suppose we have two `double` variables `length` and `breadth`. Write an assignment statement to set the `double` variable `area` to the product of `length` and `breadth`.

8. Using `double radius = 7.5;` write assignment statements to calculate the circumference and area of a circle and the volume of a sphere, based on the same radius. The formulae are:

   Circumference = $2\pi r$, area = πr^2, volume = $4\pi r^3/3$

   Use `double circumference = 2.0*Math.PI*radius;` etc. `Math` is a Java class containing lots of mathematical constants and methods.

9. Two students take a Java exam and the marks are assigned to two variables:

   ```
   int mark1 = 44, mark2 = 51;
   ```

   Write a statement to assign the average of these two marks to a `double` variable `average`.

10. Suppose we have

    ```
    int totalSeconds = 307;
    ```

    Add two more assignment statements to split `totalSeconds` into two variables `minutes` and `seconds`. Use the `/` and `%` operators.

11. What is the value of `n` after the following sequence of statements?

```
int n = 1;
n++;
n *= 2;
n *= 3;
n--;
```

12. Suppose we have `int x = 22, y = 33;` What is output by

    `System.out.println("The sum is " + x + y);`

    (Warning: this is a trick question!)

Summary

This chapter has introduced Java *variables* and *input/output*.

Programs use `int` variables for whole numbers, `double` variables for numbers with a fractional part and `String` variables (note the capital `S`) for text.

You may input text data to a program by using the method

`JOptionPane.showInputDialog("[prompt text]");`

You may output information from a program using

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

or

`JOptionPane.showMessageDialog(null, ...);`

You can use `import static` to import static library methods – saves on typing.

There are strict rules for naming variables, constants, classes and methods in Java and there is also a widely used naming convention.

The rules for Java arithmetic were discussed.

Solutions to self-test exercises

1. Hello Kate you are 16
2. Hello Mr. Asif you are 75
3. See 4.
4. Here is the complete program with the changes highlighted in bold

```
package ch02_variables;

import static javax.swing.JOptionPane.*;

class CarRepair2 {

    public static void main(String[] args) {
        String partsStr = showInputDialog("What is the parts cost");
        String hoursStr = showInputDialog("How many hours");
        String costStr =
                showInputDialog("What is the labour cost per hour");
        String vatStr =
                showInputDialog("What is the current VAT rate");
        double parts = Double.parseDouble(partsStr);
        double hours = Double.parseDouble(hoursStr);
        double cost = Double.parseDouble(costStr);
        double vat = Double.parseDouble(vatStr);
        // calculate bill before VAT
        double bill = parts + hours * cost;
        // add VAT. Note how it is calculated
        bill = bill * (1 + vat / 100);
        showMessageDialog(null, "Your bill is Â£" + bill);
    }
}
```

5. Here is the complete program with the changes highlighted in bold

```
package ch02_variables;

import static javax.swing.JOptionPane.*;

class CarpetCalculator2 {

    public static void main(String[] args) {
        final String BEST_CARPET = "Berber";
        final String ECONOMY_CARPET = "Pile";

        String roomLengthStr =
                showInputDialog("What is the room length (m)");
        String roomWidthStr =
                showInputDialog("What is the room width (m)");

        int roomLength = Integer.parseInt(roomLengthStr);
        int roomWidth = Integer.parseInt(roomWidthStr);
        int roomArea = roomLength * roomWidth;

        double carpetPrice, totalCost;

        // best carpet
        carpetPrice = 27.95;
        totalCost = roomArea * carpetPrice;
        showMessageDialog(null,
               "The cost of " + BEST_CARPET + " is £" + totalCost);

        // economy carpet
        carpetPrice = 15.95;
        totalCost = roomArea * carpetPrice;
        showMessageDialog(null,
               "The cost of " + ECONOMY_CARPET + " is £" + totalCost);
    }
}
```

6. Not allowed:

   `3sides` doesn't start with a letter
   `your salary` has a space
   `double` is a Java keyword

   Wrong style:

   AREA should be `area` (AREA *is* ok for a constant)
   `Length` starts with a capital letter
   `mysalary` should be `mySalary`
   `lenth` is OK but is incorrectly spelt. Java programmers can't spell.

7. `area = length * breadth`;

8. `double area = Math.PI * radius * radius`;
   `double volume = 4 * Math.PI * radius * radius * radius / 3`;

9.
```
average = (mark1 + mark2) / 2.0; // note 2.0 NOT 2
```

10.
```
int totalSeconds = 307;
int minutes = totalSeconds / 60;
int seconds = totalSeconds % 60;
```

11.
```
int n = 1;      // n is 1
n++;            // n is 2
n *= 2;         // n is 4
n *= 3;         // n is 12
n--;            // n is 11
```

12. The output is

```
The sum is 2233
```

and not as you might expect

```
The sum is 55
```

This is because the first + produces the `String "The sum is 22"` and then the second + is also interpreted as joining `String`s together. To get the second answer the statement should have been

```
System.out.println("The sum is " + (x + y));
```

The extra pair of brackets force Java to calculate `x + y` first.

3. Methods

Methods are groups of statements placed together under a single name. All Java applications have a class which includes a `main` method

```
class MyClass {

   public static void main(String[] args) {
      ... statements
   }

}
```

So far all our applications just have this `main` method. However many of these statements invoked, or called, other library methods such as `println`, `parseInt` and `showInputDialog`. In this chapter we show how to write and call our own methods.

One reason for writing methods additional to the `main` method is to simplify the code in the `main` method (and elsewhere).

Car repair bill calculator revisited

```
package ch03_methods;

import static javax.swing.JOptionPane.*;

class CarRepair {

    public static void main(String[] args) {
        String partsStr = read("What is the parts cost");
        String hoursStr = read("How many hours");
        double parts = Double.parseDouble(partsStr);
        double hours = Double.parseDouble(hoursStr);
        // calculate bill before VAT
        double bill = parts + hours * 20;
        // add VAT
        bill *= 1.175; // same as bill = bill * 1.175;
        display("Your bill is £" + bill);
    }

    private static void display(String s) {
        showMessageDialog(null, s);
    }

    private static String read(String prompt) {
        return showInputDialog(prompt);
    }
}
```

This application behaves in exactly the same way as the version in the previous chapter. What we have done is to tidy up the I/O by defining and calling extra methods `display` and `read` in `main` instead of invoking `JOptionPane.showMessageDialog` and `JOptionPane.showInputDialog` directly. Here's how it works:

- At line 08 the `read` method is called. The *argument* `"What is the parts cost"` is passed to set the value of the *parameter* `prompt` of the `read` method
- Line 24 calls the `showInputDialog` method with that prompt value. The user types in a string and clicks OK and that string is passed back to line 24
- Line 24 then immediately `return`s it to the call at line 08 where it is assigned to the variable `partsStr`.

The same sequence of events happens at line 09, the user input being assigned to variable `hoursStr`.

Let us assume that the user typed input 30 for the parts cost and 3 for the number of hours worked.

- At line 16 the value of `bill` would then be (30 + 3 * 20) * 1.175 = 105.75 and the argument `"Your bill is £" + bill` would evaluate to `"Your bill is £105.75"`.
- This value is passed to the parameter `s` of the `display` method and line 20 calls the `showMessageDialog` method with that message value.
- When the user clicks OK in the message box control passes back to the point after the call of `display` at line 16. As this is the last statement in `main` the application ends.

This may seem quite complicated (and probably unnecessary for this very simple example) but the method definitions at lines 19 – 21 and 23 – 25 can be thought of as *recipes* or *methods* for displaying an output string and reading an input string with a given prompt. When we write the `main` method we can forget about the `JOptionPane` methods and use the simpler `display` and `read` methods instead.

There are two sorts of method:

- *Subroutine* methods that just do something useful without returning a value. Examples of these are the `main` method of an application and the library methods `println` and `showMessageDialog`. These use the keyword `void` in their header as in

  ```
  private static void display(String s)
  ...
  ```

- *Function* methods that return a value. Examples of these are the library methods `showInputDialog` (which returns a `String` value), `parseInt` (which returns an `int` value) and `parseDouble` (which returns a `double` value). These use a type (primitive or class) in their header as in

  ```
  private static String read(String prompt)
  ...
  ```

Function methods *must* include at least one `return` statement to return the required value as in

```
return JOptionPane.showInputDialog(prompt);
```

and the value returned must match the type specified in their header (here a `String` is returned). Subroutine methods don't need `return` statements but may include statements of the form

```
return;
```

in which case the rest of the method is ignored and control passed back to the calling method.

Finally, in our design diagram for the class we can now add another box for the methods.

CarRepair
–partsStr: String –hoursStr: String –parts: double –hours: double –bill: double
–display(String) –read(String) : String

To summarise:

- When a method is *called* the *arguments* are evaluated, if necessary – as at line 16 above:

  ```
  display("Your bill is £" + bill);
  ```

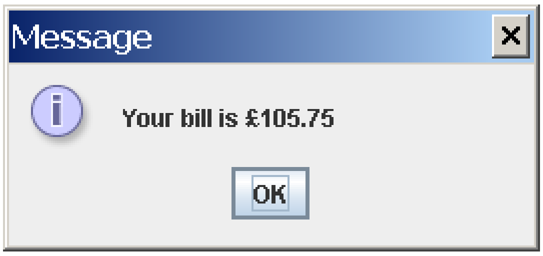
  Here the argument is the `String` **`"Your bill is £" + bill`**. Supposing that the value of `bill` is 105.75, the value of the argument becomes **`"Your bill is £105.75"`**

- The values of the *arguments* are passed across to the corresponding *parameters* of the method and then the code in the body of the method executes. Parameters are just like variables defined in the method, but with a starting value set by the arguments of the call.

  Hence the value passed across to the parameter `s` of `display` at line 17 is

  ```
  "Your bill is £105.75"
  ```

  and at line 20 `s` is passed on to the `showMessageDialog` method resulting in

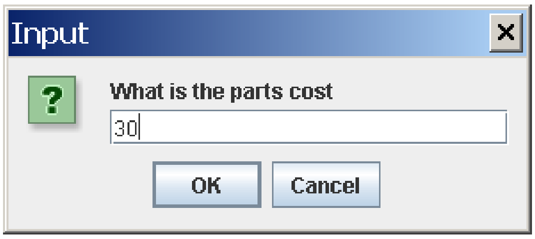
  

  Your bill is £105.75

  For a subroutine method such as `display`, control passes back to the point of call when the end of the method's code body is reached, or when a `return;` statement is executed. Subroutines don't actually need a `return;` statement.

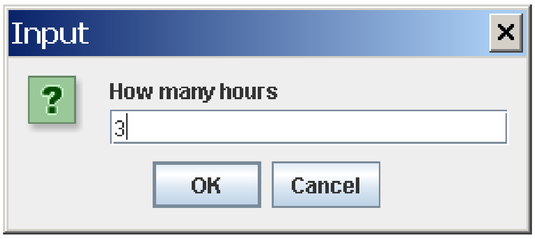
One exception is that when the end of the code body for `main` is reached, or when a `return;` statement is executed in `main`, the program simply stops running.

- For a function method, control passes back to the point of call when a `return <some value>;` statement is executed, and this value is (normally) used at the point of call. Functions *must* have at least one such `return` statement, and the type of the value returned *must* match the type specified in the header.
- In this example, at line 24 the `return` statement returns whatever `String` the user typed into the `showInputDialog` box. This happens twice. Suppose that the user typed in the `String "30"` for the parts cost (at line 08) and the `String "3"` for the number of hours (at line 09):



What is the parts cost

30



How many hours

3

This would result in `"30"` being assigned to `partsStr` at line 08 and `"3"` being assigned to `hoursStr` at line 09. Then lines 10 – 15 are executed and at line 16 the `display` method does its stuff as discussed above.

Modifiers

The keywords `public`, `private` and `static` are method *modifiers*.

- `public` methods may be called from outside the classes in which they are defined.
- `private` methods may only be called from methods defined in the same class. Their job is just to simplify the code in other methods and make the code easier to understand.
- `static` methods belong to the class itself, and not to the objects created by the class. Non-`static` (or *dynamic*[†]) methods have access to the inner state of class objects. (This will become clearer in the next chapter). `static` methods may be `public` or `private`, although they are nearly always `public`.

The `main` method of an application *must* be `public` because the Java run-time system needs to call it to run the application. It must be `static` because it is called before objects (if any are needed) are created. If you leave out the keyword `static` in the definition of main:

[†] Note that there is no keyword `dynamic`. All methods and variables are dynamic by default unless the keyword `static` is used.

```
public void main(String[] args) {
    ...
}
```

the program will compile but an exception occurs when you run it (as we saw in Chapter 1).

Library methods

The Java API contains thousands of methods, both `static` and dynamic. We have already used some of these:

`static` methods `showInputDialog`, `showMessageDialog` of the `JOptionPane` class as in

```
String name =
   JOptionPane.showInputDialog("Please type your name");
```

```
JOptionPane.showMessageDialog(null,
   "Hello " + age + " year old " + name);
```

`static` methods `parseInt` and `parseDouble` of the `Integer` and `Double` classes respectively as in:

```
int age = Integer.parseInt(ageStr);
```

```
double parts = Double.parseDouble(partsStr);
```

When calling a `static` method we always use

```
ClassName.methodName(arguments)
```

unless the calling and called methods are in the same class or we have used a static import.

For a dynamic method to be called an object of the corresponding class must exist. For instance the `print` and `println` methods are dynamic methods of the `PrintStream` class as in

```
System.out.println("Hello world");
```

(Note: `out` is a `PrintStream` object defined the `System` class to produce text in the output window.)

There are further examples of this in Tutorial 2:

Self-test exercises

1. Why do all the API library methods have to be `public`?
2. In the car repair bill calculator listed above, if we left out the keyword `static` in the definitions of the `display` and `read` methods at lines 19 and 23, the compiler would give error messages. Why?
3. Assume that we have taken the `display` and `read` methods as above and included them in the `HelloAge2` application (see Chapter 2). Replace the calls of the `JOptionPane` methods at lines 08 – 11 and 13 – 14 with equivalent calls of `display` and `read` instead.

We could use another method in the car repair program to perform the calculation. This method – `calculateBill` – has two parameters. In general methods may have any number of parameters, including zero. The class diagram now reads

CarRepair2
–partsStr: String –hoursStr: String –parts: double –hours: double –bill: double
–display(String) –read(String) : String –calculateBill(double, double) : double

Although slightly longer, this version makes the `main` method simpler:

```
01 package ch03_methods;
02
03 import static javax.swing.JOptionPane.*;
04
05 class CarRepair2 {
06
07     public static void main(String[] args) {
08         String partsStr = read("What is the parts cost");
09         String hoursStr = read("How many hours");
10         double parts = Double.parseDouble(partsStr);
11         double hours = Double.parseDouble(hoursStr);
12         double bill = calculateBill(parts, hours);
13         display("Your bill is £" + bill);
14     }
15
16     private static void display(String s) {
17         showMessageDialog(null, s);
18     }
19
20     private static String read(String prompt) {
21         return showInputDialog(prompt);
22     }
23
24     private static double calculateBill(double p, double h) {
25         double b = p + h * 20;
26         return b * 1.175;
27     }
28 }
```

At line 12 the values of `parts` and `hours` are passed to the corresponding parameters `p` and `h` of the `calculateBill` method. This has a variable `b` which is set at line 25 to the value of the bill before VAT, then line 26 returns this value multiplied by 1.175, i.e. with VAT added. The return value is then assigned to `bill` at line 12.

Scope of variables

So far we have used *local* variables. The *scope* of a local variable is the code where it may be referred to, and is simply the method in which it is defined. This applies to the method's *parameters* too. For instance the scope of the variable `parts` declared at line 10 is the `main` method lines 10 – 13 (i.e. anywhere after it has been declared). Any reference to `parts` outside this scope would be an error in this program (but see below).

Within a particular method we have complete freedom in naming the variables and parameters. That is, we don't have to worry about any use of the same names elsewhere. The `calculateBill` method could have been defined like this:

```
24    static double calculateBill(double parts, double hours) {
25       double bill = parts + hours * 20;
26       return bill * 1.175;
27    }
```

with no difference to the compiled program. There are now two variables named `parts`; one in the `main` program declared at line 10 and one which is a parameter of the `calculateBill` method at line 24. For each reference to `parts` the Java compiler decides which of these two is intended; the references at lines 10 and 12 are to the variable in the `main` program, the reference at line 25 is to the parameter of `calculateBill`.

If you find this a bit confusing, then make sure you use different names throughout. Using the same names does make the program easier to read – provided that you do it consistently. We could have defined `calculateBill` in a *very* confusing manner like this:

```
24    static double calculateBill(double hours, double parts) {
25       double fish = hours + parts * 20;
26       return fish * 1.175;
27    }
```

As far as the Java compiler is concerned, this makes no difference (even though we are now referring to the parts as `hours` and vice-versa). But it makes the program much more difficult to understand!

Summary

This chapter has introduced Java *methods*. One reason for using methods is to simplify an application by breaking its code down into smaller chunks.

There are two types of method, *Subroutine* and *Function* methods, e.g.

```
private static void display(String s)
...
private static String read(String prompt)
...
```

Subroutine methods include the keyword `void` in their header, Function methods use a type or class name in their header. Function methods must include at least one `return` statement to return their value.

`public` methods may be called from outside the defining class, `private` methods can only be called from other methods in the same class. All Java library methods are `public`. Unless you intend a method to be re-used by other classes, it should be `private`

`static` methods belong to the class itself, and not to the objects created by the class. More on this in the next chapter.

Solutions to self-test exercises

1. Making an API library method `private` would be pointless as you couldn't use it!
2. This is rather puzzling. The answer is that non-`static` (or *dynamic*) methods need an object of the class to be defined and there is no defined object in the Car repair bill calculator. We will find out more about objects later on.
3.
```
String name = read("Please type your name");
display("Hello " + age + " year old " + name);
```

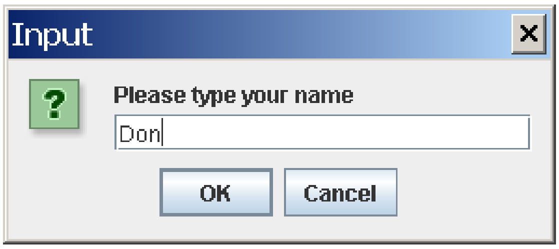
4. Decisions – `if` and `switch`

This chapter shows how to make decisions in Java using the `if` and `switch` statements.

Decisions are a fundamental part of any programming language and allow a program to react differently according to different conditions or input.

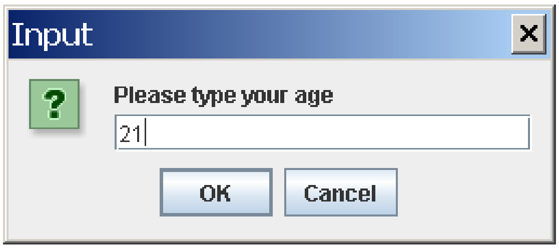
Consider the `HelloAge` program we looked at earlier. Let us modify it so that it displays a different message if the user's age is 60 or over.

First run



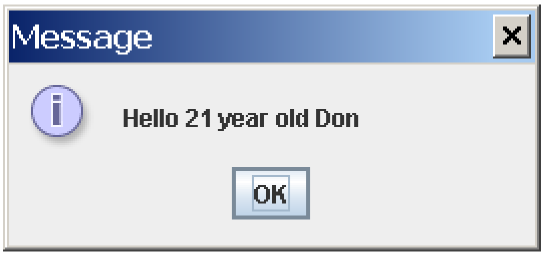
Please type your name

Don



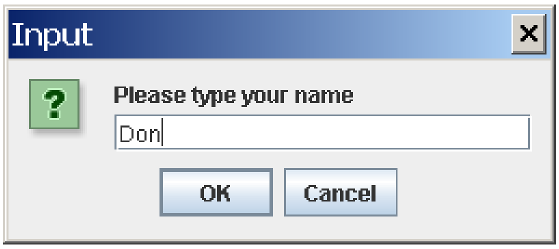
Please type your age

21



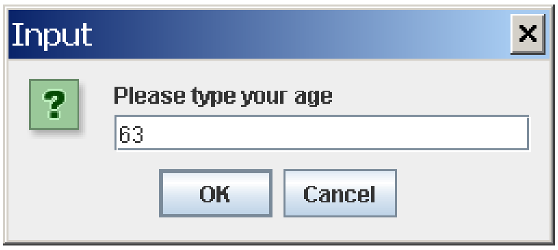
Hello 21 year old Don

Second run



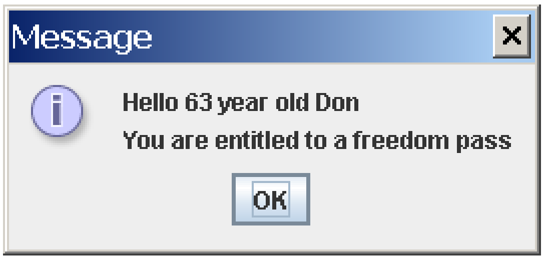
Please type your name

Don



Please type your age

63



Hello 63 year old Don
You are entitled to a freedom pass

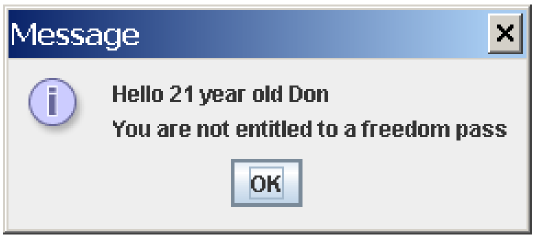
To achieve this behaviour we need an `if` statement:

```
01 package ch04_decisions;
02
03 import static javax.swing.JOptionPane.*;
04
05 class HelloAge {
06
07     public static void main(String[] args) {
08         String name = showInputDialog("Please type your name");
09         String ageStr = showInputDialog("Please type your age");
10         int age = Integer.parseInt(ageStr);
11
12         String freedom = "";
13         if (age >= 60) {
14             freedom = "\nYou are entitled to a freedom pass";
15         }
16         showMessageDialog(null,
17                 "Hello " + age + " year old " + name + freedom);
18     }
19 }
```

At line 12 we have declared a `String` variable `freedom` and given it the value `""`. The `if` statement at lines 13, 14 makes a decision on eligibility for a freedom travel pass. Line 14, which assigns a different value to `freedom`, will only be executed if the *test* `(age >= 60)` succeeds. (Putting `\n` at the front of this string forces a new line).

Very important note: there is no `;` after the condition on line 13. That is because, strictly speaking, it is not a *statement*.

We could display a different message for users under 60. Suppose Don lied about his age:



Hello 21 year old Don
You are not entitled to a freedom pass

To achieve this we add an `else` after the `if`:

```
13       if (age >= 60) {
14          freedom = "\nYou are entitled to a freedom pass";
15       } else {
16          freedom = "\nYou are not entitled to a freedom pass";
17       }
```

Self-test exercise (in lab)

1. Load `HelloAge.java`, add the highlighted lines, recompile and rerun the program to check this.

Points to note:

- The test *must* be enclosed in brackets `(` and `)`.

- Unlike many other languages, there is no keyword `then` in Java. Instead, the statement immediately following the test is executed if the test is true.
- There is no closing keyword such as `end if`. Java uses curly braces, `{` and `}`, to enclose the statements to be executed when a test evaluates as true. They are unnecessary if there is only one statement but when there is more than one, they *must* be enclosed in `{` and `}`.

Definitions

`if` statements without an `else` part are called ***one-armed conditionals***.

`if` statements with an `else` part are called ***two-armed conditionals***.

General format for a one-armed conditional

Regardless of how many statements are to be executed (0, 1, 2, ...):

```
if (test) {
   statement 1;
   statement 2;
   ...
   statement n;
}
```

Abbreviated form if there is only a single statement to be conditionally executed:

```
if (test)
   statement;
```

Even if there is only a single statement to be executed, many programmers prefer to enclose it in `{` and `}`, e.g.

```
if (age >= 60) {
   freedom += "\nYou are entitled to a freedom pass";
}
```

as it makes the meaning of the code clearer.

NetBeans encourages this style too. If you select **Source | Format** from the menus (or Alt+Shift+F) it will reformat your code (including indenting it nicely) and insert braces around all your conditionally executed statements.

Tip: You should get into the habit of pressing Alt+Shift+F when you have finished writing or editing a piece of code. It *always* makes it easier to spot errors.

General format for a two-armed conditional

```
if (test) {
   statements to be executed if test is true
} else {
   statements to be executed if test is false
}
```

Comparison operators

Java has six comparison operators:

Operator	Meaning	Note
>	Greater than	
<	Less than	
==	Equals	NOT a single =
!=	Not equal to	NOT <>
<=	Less than or equal to	
>=	Greater than or equal to	

Note the format of the equals operator! If you tried to use = you would get a syntax error. In Java, = is reserved strictly for the assignment operator.

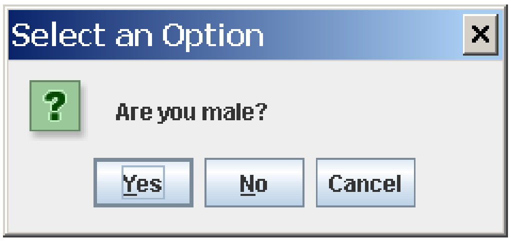
Confirm Dialogs

We can ask the user simple yes / no questions using `JOptionPane.showConfirmDialog`

Program `HelloAge2` demonstrates this. It is like `HelloAge`, but it also asks the user what sex they are, and it addresses the user as **Mr.** or **Ms.** accordingly:

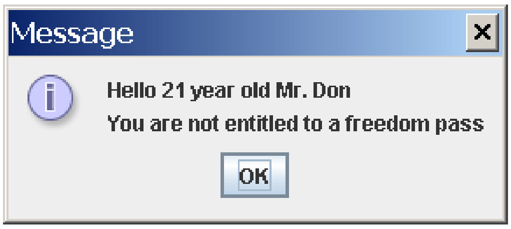
```
package ch04_decisions;

import static javax.swing.JOptionPane.*;

class HelloAge2 {

    public static void main(String[] args) {
        String name = showInputDialog("Please type your name");
        String ageStr = showInputDialog("Please type your age");
        int age = Integer.parseInt(ageStr);

        int sex = showConfirmDialog(null, "Are you male?");

        String title;
        if (sex == 0) {
             // user selected Yes
            title = "Mr. ";
        } else {
            title = "Ms. ";
        }

        String freedom;
        if (age >= 60) {
            freedom = "\nYou are entitled to a freedom pass";
        } else {
            freedom = "\nYou are not entitled to a freedom pass";
        }

        showMessageDialog(null,
          "Hello " + age + " year old " + title + name + freedom);
    }
}
```

If you run this program with the same input as before (name **Don**, age **21**) and also selected **Yes** from the confirm dialog produced at lines 13, 14:



Are you male?

the value of `sex` will be 0 so the `if` statement at lines 15 – 20 sets title to **`"Mr. "`** and the message dialog (lines 28 – 29) becomes



Hello 21 year old Mr. Don
You are not entitled to a freedom pass

Selecting **No** will return a the value 1, and selecting **Cancel** will return the value 2. Strictly speaking, you should not use 0, 1, 2 but `JOptionPane.YES_OPTION`, `JOptionPane.NO_OPTION`, `JOptionPane.CANCEL_OPTION` instead, i.e. line 15 would be safer as:

```
        if (sex == JOptionPane.YES_OPTION) {
```

Self-test exercise

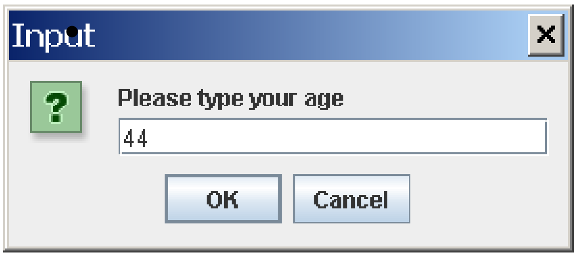
2. `JOptionPane.showInputDialog` has a Cancel button as well. What do you think will be returned if you click this instead of OK?

Nested `if`s

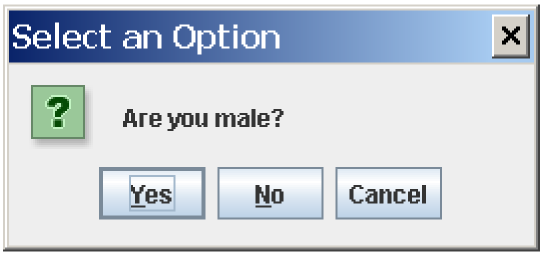
The conditional parts of an `if` statement may also be `if` statements. This is called using *nested ifs*. To illustrate, suppose we want the user's title to be

- **Mr.** if male and 18 or over
- **Ms.** if female and 18 or over
- **Master** if male and under 18
- **Miss** if female and under 18

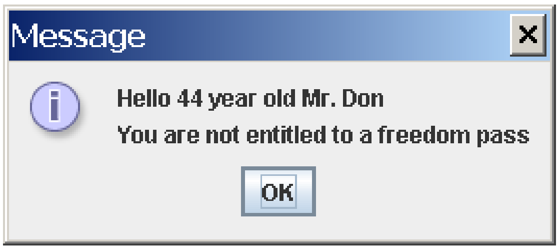
e.g. age 44, male:



Please type your age
44

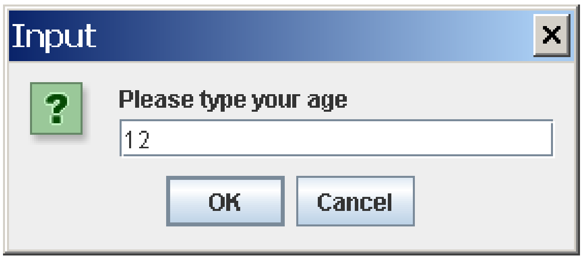


Are you male?

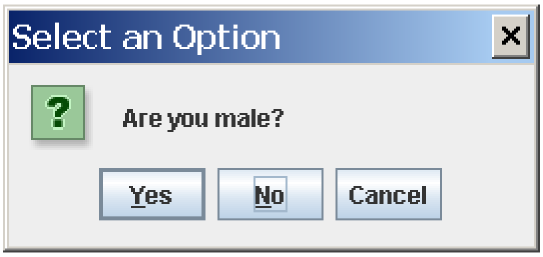


Hello 44 year old Mr. Don
You are not entitled to a freedom pass

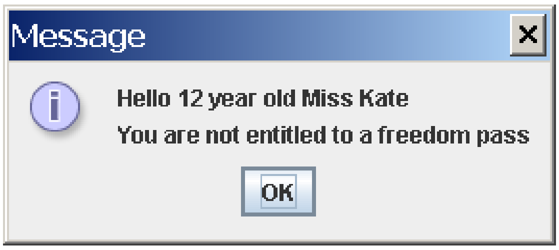
or age 12, female:



Please type your age
12



Are you male?



Hello 12 year old Miss Kate
You are not entitled to a freedom pass

The nested `if` to achieve this looks like this (from `HelloAge3`):

```
15          if (sex == 0) {
16              // user selected Yes
17              if (age >= 18) {
18                  title = "Mr. ";
19              } else {
20                  title = "Master ";
21              }
22          } else {
23              // user selected No or Cancel
24              if (age >= 18) {
25                  title = "Ms. ";
26              } else {
27                  title = "Miss ";
28              }
29          }
```

Understanding this type of nested `if` can get quite tricky. Here the overall conditional statement is on lines 15 – 29.

If the overall test `(sex == 0)` is true, the nested `if` on lines 17 – 21 is executed and the nested `if` on lines 24 – 28 is ignored.

If the overall test `(sex == 0)` is false, the nested `if` on lines 17 – 21 is ignored and the nested `if` on lines 24 – 28 is executed instead.

It is crucial to make clear what `else` goes with what `if` – hence the importance, as here, of using indentation and braces { and }for program layout.

As a contrast, the same code could be written as below – which version do you think is easiest to understand?

```
15 if (sex == 0)
16 if (age >= 18)
17 title = "Mr. ";
18 else
19 title = "Master ";
20 else
21 if (age >= 18)
22 title = "Ms. ";
23 else
24 title = "Miss ";
```

Here the `else` at line 20 goes with the `if` at line 15, but this is not apparent from the layout.

Tip (once again): Fortunately, NetBeans will insert all the indentation and braces for you ... but only if you remember to use Alt+Shift+F.

Alternative conditions and combined tests

It is often easier to understand nested `if`s if you replace them with combinations of conditions. We can achieve this by representing alternative conditions, using the `else if` keywords, and by using operators such as and (**&&**), or (**||**) and not (**!**).

Here is another version (from `HelloAge4`):

```
if (sex == 0 && age >= 18) {
    title = "Mr. ";
} else if (sex == 0 && age < 18) {
    title = "Master ";
} else if (sex != 0 && age >= 18) {
    title = "Ms. ";
} else {
    title = "Miss ";
}
```

The general form is:

```
if (test1) {
     statement to execute if test1 is true
} else if (test2) {
     statement to execute if test2 is true
...
} else if (testN) {
     statement to execute if testN is true
} else {
     statement to execute if all the above tests are false
}
```

The `boolean` type

`boolean` variables can have just two values, `true` or `false`, and they can be used to remember a condition. For instance we could make the nested `if` easier to read by using two `booleans`, `adult` and `male`. Here is a new version (`HelloAge5`):

```
14         boolean adult = (age >= 18);
15         boolean male = (sex == 0);
16         if (male && adult) {
17             title = "Mr. ";
18         } else if (male && !adult) {
19             title = "Master ";
20         } else if (!male && adult) {
21             title = "Ms. ";
22         } else {
23             title = "Miss ";
24         }
```

Line 14 sets `adult` to `true` if `age` is greater or equal to 18, `false` otherwise. Line 15 sets `male` to `true` if sex is 0, `false` otherwise. These boolean values are used to make the nested `if` clearer. (Recall that ! means “not”.)

Self-test exercises

3. What is displayed as a result of the following nested `if`

```
if (age < 18) {
   if (age > 0) {
      out.println("Too young");
   } else {
      out.println("Impossible! ");
   }
} else if (age > 64) {
   if (age > 130) {
      out.println("Preposterous! ");
   } else {
      out.println("Too old");
   }
} else {
   out.println("OK");
}
```

when

(a) `age = -2`,
(b) `age = 0`,
(c) `age = 15`,
(d) `age = 17`,
(e) `age = 18`,
(f) `age = 22`,
(g) `age = 64`,
(h) `age = 65`,
(i) `age = 100`,
(j) `age= 130`,
(k) `age = 200`?

Can you rewrite it in a way that is easier to understand?

4. Consider the following nested `if`:

```
if (reply == "no") {
   if (choice <= 6) {
      message = "reply 1";
   } else {
      message = "not in range";
      }
   } else if (choice == 7 || choice == 8) {
      message = "reply 2";
   } else {
      message = "reply 3";
   }
```

Predict the value that will be given to `message` given the following:

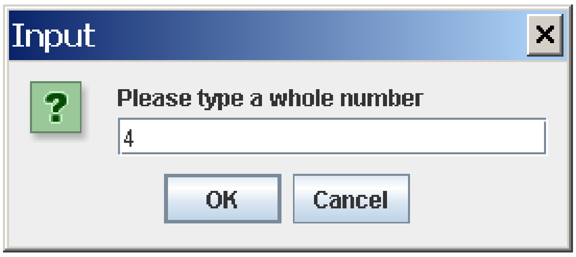
(i) `reply = "no", choice = 2`
(ii) `reply = "no", choice = 7`
(iii) `reply = "yes", choice = 5`
(iv) `reply = "yes", choice = 8`
(v) `reply = "maybe", choice = 4`

The `switch` statement

If a choice depends only on a small range of values, e.g. days of the week, the `switch` statement can be used instead of a nested `if`. The following example asks the user to input a whole number, and displays the corresponding day of the week using 0 for Sunday, 1 for Monday ... 6 for Saturday. If the number is not between 0 and 6 it displays “Not a day” instead

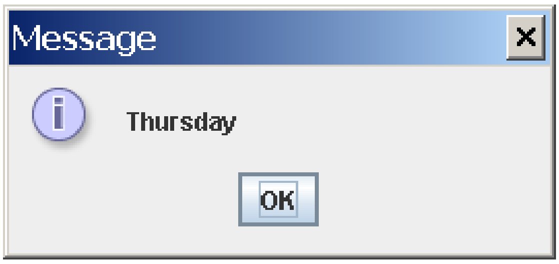
```
package ch04_decisions;

import static javax.swing.JOptionPane.*;

class SwitchDemo {

    public static void main(String[] args) {
        String numStr = showInputDialog("Please type a whole number");
        int dayNum = Integer.parseInt(numStr);
        String dayName;
        switch (dayNum) {
            case 0:
                dayName = "Sunday";
                break;
            case 1:
                dayName = "Monday";
                break;
            case 2:
                dayName = "Tuesday";
                break;
            case 3:
                dayName = "Wednesday";
                break;
            case 4:
                dayName = "Thursday";
                break;
            case 5:
                dayName = "Friday";
                break;
            case 6:
                dayName = "Saturday";
                break;
            default:
                dayName = "Not a day";
        }
        showMessageDialog(null, dayName);
    }
}
```

First run



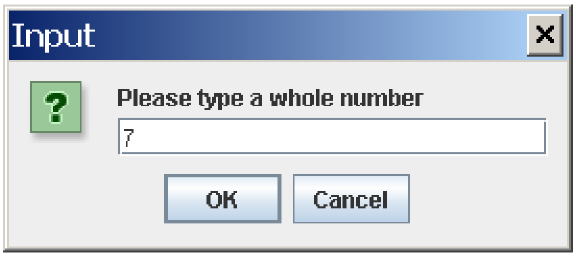
Please type a whole number

4



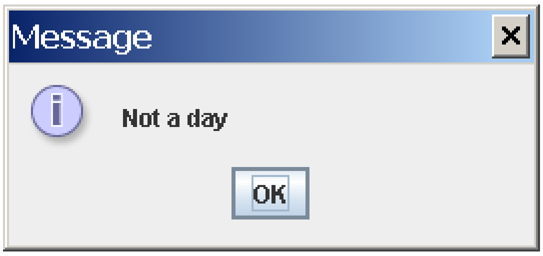
Thursday

Second run



Please type a whole number

7



Not a day

The value of `dayNum` is fed into the switch at line 11. Its value is checked against the case labels – 0 to 6 here. If a match is found, the corresponding statements are executed. The `break;` statements are important as they prevent control running on to the next case – instead control passes to the point after the `switch` (line 36).

The `default` case is chosen when the switch variable doesn't match any of the other cases. If it is left out and no match is found, the whole switch statement does nothing.

Self-test exercise

5. What would the output be in the above program if all the `break;` statements were omitted? Why?

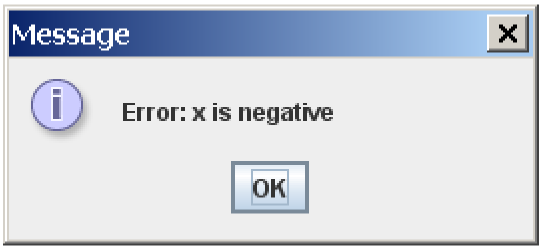
And Finally – The Dreaded Extra Semicolon Error

New Java programmers often get confused about the use of the semicolon ; Consider the following application.

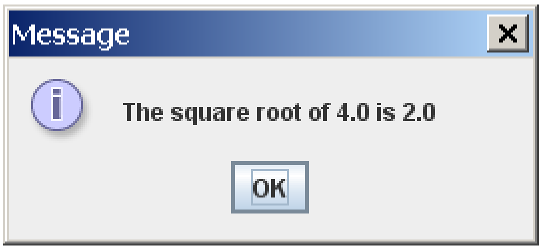
The user inputs a `double` value, and if this is negative an error message should be displayed and `x` should be set to `–x`. If not, the program should just continue to calculate and display the square root of the input

```
import static javax.swing.JOptionPane.*;
class SquareRoot
{
    public static void main(String[] args)
    {
        String xStr = showInputDialog("Please type the value of x");
        double x = Double.parseDouble(xStr);
        if (x < 0);
        {
            showMessageDialog(null, "Error: x is negative");
            x = -x;
        }
        // x is non-negative so we can find and display its square root
        showMessageDialog(null, "The square root of " +
                                x + " is " + Math.sqrt(x));
    }
}
```

The program compiled OK. When tested with input `-4`, it behaved as expected:

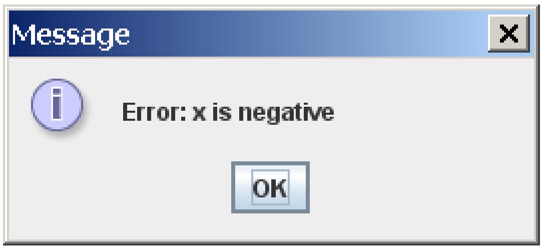


Error: x is negative

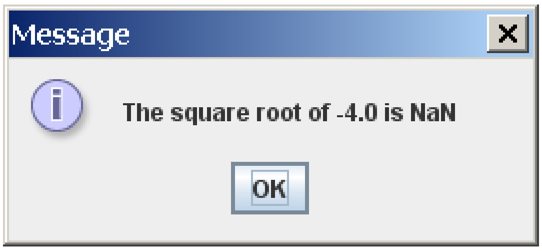


The square root of 4.0 is 2.0

However, this is what happened when it was tested with input `4`:



Error: x is negative



The square root of -4.0 is NaN

The program unexpectedly displayed the error message and set `x` to `-4`. The Java square root function returns the value NaN (which stands for "not a number") for negative numbers, hence the second message display.

What went wrong?

```
if (x < 0);
```

The problem is this semicolon. In Java, semicolons terminate statements. The Java compiler thinks there is a statement between **`if (x < 0)`** and **`;`** so it doesn't give an error. This is called the *null* statement. It gets executed – it does nothing at all – and the program continues to the **`showMessageDialog`** box no matter what the value of **`x`**. In effect the **`;`** terminates the **`if`** statement prematurely and the following code will *always* be executed.

If you make this mistake for an **`if ... else`** statement the compiler will give an error message.

Summary

- `if` statements allow the programmer to control the sequence of actions by making the program carry out a test. Following the test, the computer carries out one of two actions.
- The test must be enclosed in brackets and there is no `then` keyword or `endif` keyword.
- There are two varieties of the `if` statement, one with an `else` part and one without.
- `if` statements can be nested for more complex decision making
- We can also allow for multiple alternatives by using `else if` constructions
- We can combine tests using operators such as && and `||`
- `boolean` variables take on the values `true` or `false` for use in tests.
- The `switch` statement carries out a test, but provides for an unlimited set of outcomes. It is restricted to tests on `byte`, `short`, `int`, `long` or `char` variables. (nearly always `int` variables).
- Beware of the "extra semicolon" mistake with `if` statements.
- NetBeans can lay out your code nicely and insert braces automatically

Tip (third time lucky): NetBeans can insert all the indentation and braces for you – remember to use Alt+Shift+F, it is your friend!

Solutions to self-test exercises

1. Do this in the lab.

2. If the user selects Cancel it really meant they don't want to do this just now so the code should really handle this. Instead the test `sex == 0` fails and `title` is set to `"Ms. "`.

3. (a) Impossible! (b) Impossible! (c) Too young (d) Too young (e) OK (f) OK (g) OK (h) Too old (i) Too old (j) Too old (k) Preposterous!

   The test could be written much more simply:

```
if (age <= 0) {
    out.println("Impossible!");
} else if (age < 18) {
    out.println("Too young");
} else if (age <= 64) {
    out.println("OK");
} else if (age <= 130) {
    out.println("Too old");
} else {
   out.println("Preposterous!");
}
```

4. (i) reply 1 (ii) not in range (iii) reply 3 (iv) reply 2 (v) reply 3. For the last answer, note that `"yes"` and `"maybe"` are equivalent – they are both not `"no"`

5. No matter what whole number was input, the output would always be Not a day. Because the breaks are omitted, every case statement after the one chosen would be executed in turn, always finishing with the default case.

   Leaving out `break`s is a common mistake when using a switch.

5. Objects and Classes

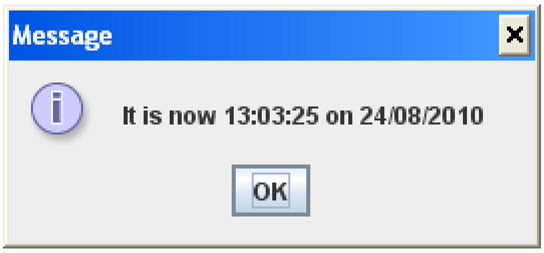
So far we have written applications with a single class. These applications have made use of various Java library classes such as `System` and `JOptionPane`. So far, the only *objects* we have used have been `String`s and the standard output stream `System.out`.

Date demonstration application

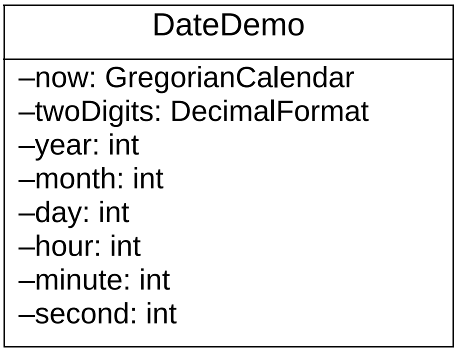
Here is a DateDemo program (see also Tutorial 2):

```
package ch05_objects;

import static javax.swing.JOptionPane.*;
import java.util.*;
import java.text.DecimalFormat;

class DateDemo {

    public static void main(String[] args) {
        GregorianCalendar now = new GregorianCalendar();
        DecimalFormat twoDigits = new DecimalFormat("00");
        int year = now.get(Calendar.YEAR);
        int month = now.get(Calendar.MONTH) + 1;
        int day = now.get(Calendar.DATE);
        int hour = now.get(Calendar.HOUR_OF_DAY);
        int minute = now.get(Calendar.MINUTE);
        int second = now.get(Calendar.SECOND);
        showMessageDialog(null,
                "It is now " + twoDigits.format(hour) + ":" +
                twoDigits.format(minute) + ":" +
                twoDigits.format(second) + " on " +
                twoDigits.format(day) + "/" +
                twoDigits.format(month) + "/" +
                year);
    }
}
```

Here is a typical output message from the program (when it was run on Monday afternoon, 24th August 2010):

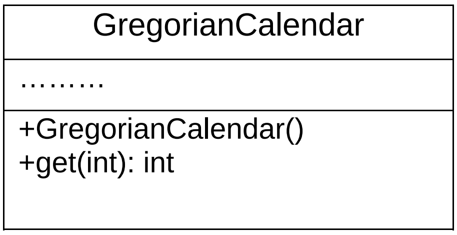


It is now 13:03:25 on 24/08/2010



DateDemo

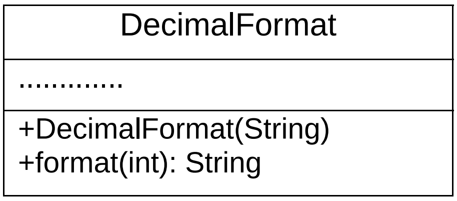
–now: GregorianCalendar
–twoDigits: DecimalFormat
–year: int
–month: int
–day: int
–hour: int
–minute: int
–second: int



GregorianCalendar

………

+GregorianCalendar()
+get(int): int



DecimalFormat

....

+DecimalFormat(String)
+format(int): String

Note that no methods are put in the class diagram as they are not part of `DateDemo` class. The methods used are methods of the `GregorianCalendar` and `DecimalFormat` classes.

We don't usually document API classes as they are fully documented in the Java Application Programmer's Interface (API) – see Chapter 7 below. The documentation given here is purely for illustrative purposes. It ignores all the other public methods (shown with a + sign) of the two library classes.

Explanation of the code

Lines 03 – 05 are needed to import the Java library classes `JOptionPane`, `Calendar, GregorianCalendar` and `DecimalFormat`.

Line 10 declares a `Calendar` variable `now` and assigns to it a *new* `GregorianCalendar` object. `GregorianCalendar()` is the no-argument *constructor* of the `GregorianCalendar` class [note that you still need to write the `(` and `)` brackets] which returns the current date and time (to the nearest second). The `GregorianCalendar` class has many other constructors for creating `GregorianCalendar` objects.

Line 11 declares a `DecimalFormat` variable `twoDigits` and assigns to it a *new* `DecimalFormat` object. `DecimalFormat(String pattern)` is the *constructor* of the `DecimalFormat` class that has a `String` formatting argument. These format strings can be quite complex; here we simply want to be able to format an `int` using exactly two digits (lines 19 – 23), so that `9` becomes `09` etc.

The `GregorianCalendar` class has many dynamic (i.e. non-`static`) methods. This application uses the `get` method on lines 12 – 17 to obtain various elements of the object `now`. Its meaning is quite intuitive but note:

- `get(Calendar.DATE)` returns the day of the month, between 1 and 31.
- `get(Calendar.MONTH)` returns the month of the year with January represented by 0 and December by 11. This is why line 12 adds 1 to get the more normal representation.
- `get(Calendar.HOUR_OF_DAY)` returns the hour in the range 0 to 23 as in a 24 hour clock. If we used `get(Calendar.HOUR)` instead the hour would be in the range 0 (12 o'clock) to 11 (11 o'clock) and we would need something else to tell whether it was am or pm.

get and **set** methods

API classes often have methods named `getXXX` to return part of the internal state of an object. Many API classes also provide `setXXX` methods to alter part of the internal state of an object. For instance the `GregorianCalendar` class has nineteen `getXXX` methods and ten `setXXX` methods to get and set various properties of a `GregorianCalendar` object . Note that we have no idea exactly how the information is represented internally, nor do we need to know this.

Calling dynamic methods

In Java we call a method which belongs to an object using

```
objectName.methodName(arguments)
```

This is also called sending a *message* to the object.

Self-test exercises

1. What would the output from this application be if we added the following statements immediately after line 08 (assuming that the program was run at 1.03 pm and 25 seconds on 24th August 2010):

```
int h = now.get(Calendar.HOUR) + 1;
now.set(Calendar.HOUR, h);
```

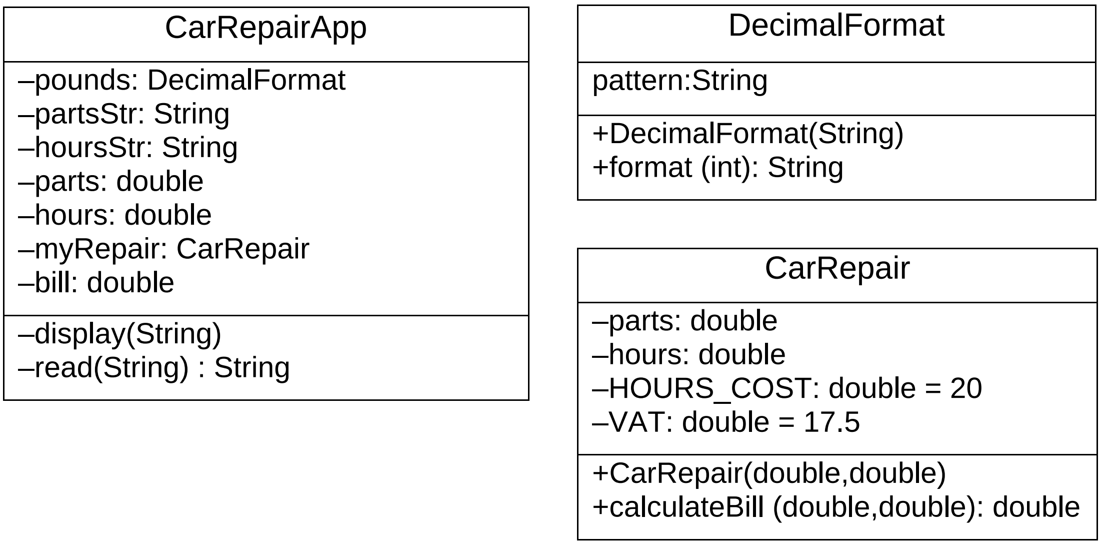
2. What if we added the following statements immediately after line 10 instead? (The application doesn't "crash" – it still produces a message box).

```
int h = now.get(Calendar.HOUR) + 24;
now.set(Calendar.HOUR, h);
```

Car repair bill calculator using a separate class

We are going to use this next example to show how you can separate the I/O and presentation in the main method of the application class `CarRepairApp` leaving the calculation of the bill to an *auxiliary* class called `CarRepair`.

Here is the main application class and the auxiliary classes it uses



Aside: because we want to keep all the code for this application together and not confuse it with other programs we have created a separate package called `ch05_objects.carrepairbill`. You can see this by looking in the projects tab of NetBeans or, if you look at the `JavaTerm1\src\ch05_objects` folder (e.g. using Windows explorer), you will see it contains a subfolder called `carrepairbill`. Packages like this are a fundamental part of the Java language and allow us to have more than one class with the same name, provided they are in different packages (e.g. there is another version of this program in `ch05_objects.carrepairbill2`).

In fact, it's possible to create a hierarchy of packages (such as `java.awt.event` where `event` is effectively a folder/package inside `awt` which is inside `java`). Each package normally contains a specific set of classes to do certain tasks (and, strictly speaking, they don't have to bear any relation to the classes in the parent folders).

Here is the `CarRepairApp` class:

```
package ch05_objects.carrepairbill;

import static javax.swing.JOptionPane.*;
import java.text.DecimalFormat;

class CarRepairApp {

    public static void main(String[] args) {
        DecimalFormat pounds = new DecimalFormat("?#,###.00");
        String partsStr = read("What is the parts cost");
        String hoursStr = read("How many hours");
        double parts = Double.parseDouble(partsStr);
        double hours = Double.parseDouble(hoursStr);
        CarRepair myRepair = new CarRepair(parts, hours);
        double bill = myRepair.calculateBill();
        display("Your bill is " + pounds.format(bill));
    }

    private static void display(String s) {
        showMessageDialog(null, s);
    }

    private static String read(String prompt) {
        return showInputDialog(prompt);
    }
}
```

Here is the `CarRepair` class:

```
package ch05_objects.carrepairbill;

class CarRepair {

    private double parts;
    private double hours;
    private static final double HOURS_COST = 20;
    private static final double VAT = 17.5;

    CarRepair(double p, double h) {
        parts = p;
        hours = h;
    }

    public double calculateBill() {
        double bill = parts + hours * HOURS_COST;
        return bill * (1 + VAT / 100);
    }
}
```

The `CarRepair` class has:

- `double` *instance variables* `parts` and `hours`. These are declared on lines 05 and 06, separate from any of the methods used.
- a *constructor* `CarRepair(double p, double h)` for creating a new `CarRepair` object with `p` assigned to `parts` and `h` to `hours`
- a `public double calculateBill()` method to calculate the bill

When we use the `CarRepair` class we create an instance of it in the main application called `myRepair` at line 12. This is created by the *constructor* at lines 10 – 13 in the CarRepair class. The values passed across to parameters `p` and `h` are the values of `parts` and `hours` in the main program. Remember that there is no need to use the same variable names in the main application and the `CarRepair` class, but it is convenient to do so.

Lines 07 and 08 define two useful constant values for use in the `calculateBill` method. As they are `static` they belong to the `CarRepair` class itself, and not to any object of the class (such as `myRepair` defined on line 14 of the `CarRepairApp` class).

Carpet calculator with a separate class

In a similar way let us revisit the Carpet calculator, using a separate class to do the calculations.

First, here is the application class `CarpetCalculatorApp`. By writing this first we get an idea of what the auxiliary `CarpetCalculator` class must do. We will also use two instances, `pounds` and `twoDP` of the `DecimalFormat` class.

```
package ch05_objects.carpetcalculator;

import static javax.swing.JOptionPane.*;
import java.text.DecimalFormat;

class CarpetCalculatorApp {

    public static void main(String[] args) {
        DecimalFormat pounds = new DecimalFormat("£###,##0.00");
        DecimalFormat twoDP = new DecimalFormat("##0.00");
        String roomLengthStr =
                showInputDialog("What is the room length (m)?");
        String roomWidthStr =
                showInputDialog("What is the room width (m)?");
        double roomLength = Double.parseDouble(roomLengthStr);
        double roomWidth = Double.parseDouble(roomWidthStr);
        String carpetName;
        double squareMeters;
        double totalCost;

        CarpetCalculator luxury = new CarpetCalculator("Berber",
                27.95, roomLength, roomWidth);

        CarpetCalculator economy = new CarpetCalculator("Pile",
                15.95, roomLength, roomWidth);

        // luxury carpet
28          carpetName = luxury.getCarpetName();
29          squareMeters = luxury.getAreaSquareMeters();
30          totalCost = luxury.determineTotalCost();
31          showMessageDialog(null,
32                  "The cost of " + twoDP.format(squareMeters) +
33                  " square meters of " + carpetName +
34                  " is " + pounds.format(totalCost));
35
36          // economy carpet - same square metres calculation
37          carpetName = economy.getCarpetName();
38          totalCost = economy.determineTotalCost();
39          showMessageDialog(null,
40                  "The cost of " + twoDP.format(squareMeters) +
41                  " square meters of " + carpetName +
42                  " is " + pounds.format(totalCost));
43      }
44 }
```

Explanation of the code

We use the two `DecimalFormat` objects (lines 09, 10) so that we can format an area in square metres and an amount in pounds.

This version asks for the room dimensions directly as `double`s (lines 11 – 14). We want to print out the carpet name, area in square metres and total cost for the two different types of carpet, hence the variables at lines 15 – 17.

Lines 21 – 25 create two `CarpetCalculator` objects `luxury` and `economy`.

Lines 28 – 30 and similarly 37 – 38 use methods of the `CarpetCalculator` class to get the carpet name and area in square metres, and to determine the total cost. The `showMessageDialog` calls use these values, formatted as needed, to display the results.

The lines of code interacting with the auxiliary class `CarpetCalculator` are shown in bold.

Instance variables

When designing a class to create objects, we need first to consider what data types make up each individual object. These are called the *instance variables* of the class. They have the *entire class* as their scope so they are available to all the methods. They are nearly always `private`. What do we need for the `CarpetCalculator` class?

- A `String carpetName` for the carpet name
- A `double pricePerSquareMetre` for the carpet cost per square metre

These two are fairly obvious. But what else do we need? For instance, do we need the room length and width? Lines 29 and 30 of the above application show that we need to be able to calculate the area in square metres, and the overall carpet cost. We can do this if we just hold the area in square metres, so:

- A `double areaSquareMetres` for the room area

Constructors and dynamic methods

All classes should have at least one *constructor*. We'll see what a constructor looks like below, but basically it's a special method whose job is to create new objects belonging to the class. Lines 21 – 25 show that a constructor needs to be defined with a header like this:

-
  ```
  CarpetCalculator(String name, double price,
              double roomLength, double roomWidth)
  ```

Lines 28 – 30 show the dynamic methods needed:

- `public String getCarpetName()`
- `public double getAreaSquareMetres()`
- `public double determineTotalCost()`

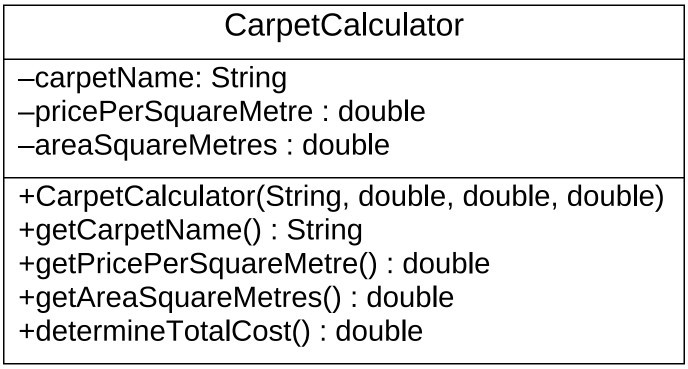
and while we are about it we may as well have another `get` method

- `public double getPricePerSquareMetre()`

Note that these dynamic methods have no arguments. This is quite common as they have access to the instance variables to do their work.

UML Class Diagrams

There is a concise notation for documenting a class design, namely a *class diagram*. We have already been using class diagrams informally. This notation comes from the *Universal Modelling Language* (UML) which has widespread use in modern software design. Here is the class diagram for `CarpetCalculator`:



CarpetCalculator

–carpetName: String
–pricePerSquareMetre : double
–areaSquareMetres : double

+CarpetCalculator(String, double, double, double)
+getCarpetName() : String
+getPricePerSquareMetre() : double
+getAreaSquareMetres() : double
+determineTotalCost() : double

The class name is written in the top box.

The instance variables needed are written in the second box. They are usually `private`. The UML notation for `private` is a – sign.

The constructor(s) and methods are written in the third box. Constructors have the same name as the class and are used to create objects in conjunction with the keyword `new`. It is common for a class to have many constructors, i.e. many ways of building a new object. This is OK provided they have different parameter lists.

Constructors are nearly always `public`, and so are most methods. The UML notation for `public` is a + sign.

The actual parameter names used by methods are irrelevant, but their types *are* relevant. For instance we specify that the constructor takes a `String` and three `double`s.

Before going ahead and coding the class, it's worth writing down what the constructor and each method has to do:

- The three `get` methods are straightforward, they must simply return the values of the instance variables `carpetName`, `pricePerSquareMetre` and `areaSquareMetres`.
- `determineTotalCost()` – clearly all this has to do is return the product of `pricePerSquareMetre` and `areaSquareMetres`.
- The constructor

  ```
  CarpetCalculator(String name, double price,
          double roomLength, double roomWidth)
  ```

  is the most complicated method. It should assign `name` to `carpetName`, `price` to `pricePerSquareMetre` and calculate and assign `areaSquareMetres` from `roomLength` and `roomWidth`.

Here is the code for our auxiliary class:

```
package ch05_objects.carpetcalculator;

class CarpetCalculator {

    private String carpetName;
    private double pricePerSquareMeter;
    private double areaSquareMeters;

    // constructor - roomLength and roomWidth given in meters
    CarpetCalculator(String name, double price,
            double roomLength, double roomWidth) {
        carpetName = name;
        pricePerSquareMeter = price;
        areaSquareMeters = roomLength * roomWidth;
    }

    // get methods
    public String getCarpetName() {
        return carpetName;
    }

    public double getPricePerSquareMeter() {
        return pricePerSquareMeter;
    }

    public double getAreaSquareMeters() {
        return areaSquareMeters;
    }

    // method to calculate total cost
    public double determineTotalCost() {
        return pricePerSquareMeter * areaSquareMeters;
    }
}
```

Using an extra class like this enables the programmer to concentrate on I/O and presentation in the application class (the one with the `main` method), leaving all the calculation details to the auxiliary class.

Self-test exercise

3. Describe how you would rewrite `CarpetCalculatorApp.java` using methods `read` and `display` as in `CarRepair2.java` (see Chapter 3).

The Default Constructor

It is possible to define a class without specifying a constructor. For instance the main application classes we have seem so far have had no constructors because we don't use them to create objects.

If we have a class `MyClass.java` without a constructor, Java assumes there is a *default* constructor which does nothing other than creating an object:

```
MyClass(){
}
```

and then the programmer can write

```
MyClass m = new MyClass();
```

to create an object. In such a case, the object has a possibly unsafe state so you should provide `set` methods to give new objects a sensible state. To illustrate this, we rewrite the `CarRepair` class with `set` methods and modify the application `CarRepairApp` class accordingly. The changes are highlighted in bold:

```
package ch05_objects.carrepairbill2;

import static javax.swing.JOptionPane.*;
import java.text.DecimalFormat;

class CarRepairApp {

    public static void main(String[] args) {
        DecimalFormat pounds = new DecimalFormat("?#,###.00");
        String partsStr = read("What is the parts cost");
        String hoursStr = read("How many hours");
        double parts = Double.parseDouble(partsStr);
        double hours = Double.parseDouble(hoursStr);
        CarRepair myRepair = new CarRepair();
        myRepair.setParts(parts);
        myRepair.setHours(hours);
        double bill = myRepair.calculateBill();
        display("Your bill is " + pounds.format(bill));
    }

    private static void display(String s) {
        showMessageDialog(null, s);
    }

    private static String read(String prompt) {
        return showInputDialog(prompt);
    }
}
```

```
package ch05_objects.carrepairbill2;

class CarRepair { // no constructor so default is assumed

    private double parts;
    private double hours;
    private static final double HOURS_COST = 20;
    private static final double VAT = 17.5;

    public void setParts(double p) {
        parts = p;
    }

    public void setHours(double h) {
        hours = h;
    }

    public double calculateBill() {
        double bill = parts + hours * HOURS_COST;
        return bill * (1 + VAT / 100);
    }
}
```

Relying on the default constructor is regarded as bad practice! If you provide any other constructor, Java does not recognize the default.

Actually, it would be better to keep the original constructor in the CarRepair class (see pp. 49-) in which case Java assumes that there is no default constructor, so we must add one:

```
public CarRepair() {
}
```

In this case the `CarRepair()` class could be used for *both* versions of `CarRepairApp`.

Self-test exercises

4. What happens to the above version of the Car repair bill calculator if the programmer omits lines 15 and 16 in `CarRepairApp.java`?
5. Instance variables may *not* be used by `static` methods. Why not?

Summary

This chapter has focused on *objects*. Apart from simple main applications and some library classes, classes are designed to create objects.

Objects have a *state*, for instance a `Date` object holds year, month, day, hour, minute, second information.

When defining a class the state of each object is held in *instance variables* which are usually `private` and declared outside any method. For instance the `CarpetCalculator` class has three private instance variables, a `String` for the carpet name, a `double` for the price per square metre and a `double` for the area in square metres. The scope of each instance variable is the whole class so all instance variables may be used by all dynamic methods

Objects are created using a *constructor* which is a method with the same name as the class. Classes may have more than one constructor. If no constructor is provided then a default no-argument constructor is assumed, which does nothing.

Dynamic methods which use and possibly change an object's state are declared `public`. They are used outside the class (e.g. in a main application class) using the notation

```
objectName.methodName(arguments)
```

and this is called sending a *message* to the object.

A UML class diagram is a concise way of documenting a class design.

We shall revisit this important material in Chapter 8.

Solutions to self-test exercises

1. The output would be

   ```
   It is now 14:03:25 on 24/08/2010
   ```

2. The output would be

   ```
   It is now 13:03:25 on 25/08/2010
   ```

   The `Date` class is quite clever in maintaining a consistent internal state. It takes care of leap years and leap seconds (what are they?) too.

3. Include the `read` and `display` methods exactly as on pp. 29 and 30 in `CarpetCalculatorApp.java` and use

   ```
   String roomLengthStr =
         read("What is the room length (m)?");
   ```

   instead of lines 09-10 and

   ```
   display("The cost of " + twoDP.format(squareMetres) +
           " square metres of " + carpetName +
           " is " + pounds.format(totalCost));
   ```

   instead of lines 29 – 32 etc.

4. The bill is displayed as £0.00 no matter what the input was. The default value for numeric variables is 0.

5. Because `static` methods belong to the *class* itself they have no access to individual objects. Indeed `static` methods may be used without ever creating objects of the class – for example the `main` method of a Java application, and various methods of the API class `Math`.

6. Graphical User Interface (GUI) programming

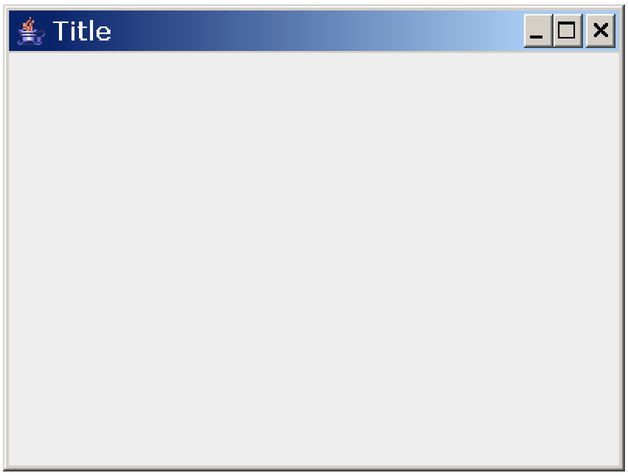
This chapter shows how to write Java application programs that run in windows, using *controls* such as buttons, text fields, combo boxes, check boxes, radio buttons and so on. All of these (and many more) are defined in the *Swing* package `javax.swing.*` which must be imported into your main class.

Most GUI applications are *subclasses* of the `JFrame` class and as such they have a lot in common. It would be very boring to have to type this common stuff into NetBeans all the time. Instead you will find a `JFrameBasic` class in the `ch06_gui1` package which you can use as a basis for new GUI applications.

If you open this up now you will see the following code in the editor:

```
package ch06_gui1;

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

public class JFrameBasic
        extends JFrame implements ActionListener {

    public static void main(String[] args) {
        JFrameBasic jf = new JFrameBasic();
    }

    public JFrameBasic() {
        setLayout(new FlowLayout());
        setSize(400, 300);
        setTitle("Title");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        // add your event handling code here
    }
}
```

If you run this as it stands (Shift+F6), you will see this window:



This can be dragged around, maximized, minimized and closed down just like any other Windows form. The grey area (called the *content pane*) is where the various controls would be placed – there are none in this bare template.

Explanation of the template code

The imports at lines 03 – 04 are needed for all Swing applications. The import at line 05 is needed if we want to handle *events* such as the user clicking buttons (we usually need this!)

Lines 07 – 08 declare your class and the words `extends JFrame` means that our application is a *subclass* of the Swing `JFrame` class. We discuss subclasses later on in the course.

The words `implements ActionListener` says that we want our application to “listen” for *events* such as the user pressing a button. Writing this means that the class must have an `actionPerformed` method which will be called by the Java runtime system when an action event happens. A skeleton is provided at lines 23 – 25.

The `main` method is needed because this is an application. All it needs to do is create a new object of this class, which is done at line 11. This will call the constructor at lines 14 – 21.

There are different ways of *laying out* the controls on the content pane. The simplest is *flow layout* which adds controls from top to bottom and left to right in a manner similar to writing text (centrally justified) in a word processor. Line 15 chooses flow layout.

Line 16 sets the size of the frame in pixels. Change the numbers to suit your own application. Note that the content pane is smaller than this: its width is 8 pixels less than the frame width because there is a border 4 pixels wide. Its height is 35 pixels less (to make room for the title bar). So the content pane of the above default frame is 392 by 265 pixels.

Line 17 sets the text to be displayed in the title bar. Change the string to suit your own application.

Line 18 allows the user to close the application by clicking the close window button ☒
Line 19 places the window in the middle of the screen when it opens (if you delete this line is will appear at the top left). Line 20 is needed to make the frame visible.

If you want to stop the user resizing the frame add the statement

```
setResizable(false);
```

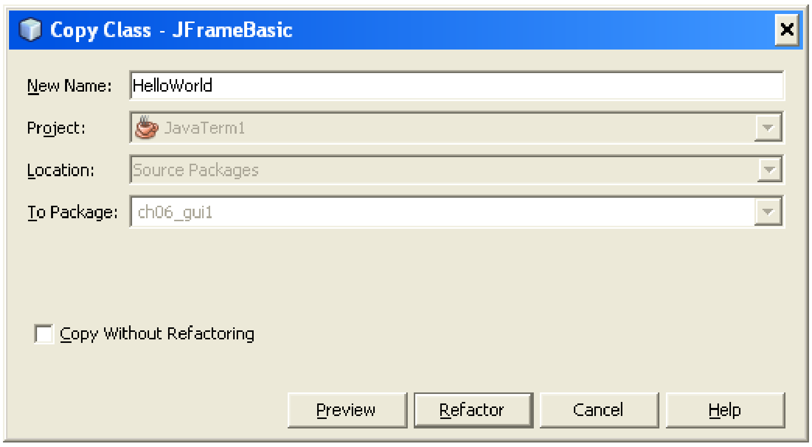
before this line.

If your frame has buttons, you write the code to be executed when the user clicks them in the `actionPerformed` method replacing line 25.

Self-test Lab Exercise

1. *Hello World as a `JFrame`*. In the NetBeans Projects tab, right-click on `JFrameBasic` and select Copy (Ctrl+C). Then right-click on the package name `ch06_gui1` and select Paste (Ctrl+V). NetBeans will bring up a dialog

window as follows. Set the New Name to HelloWorld and click on the Refactor button.



Copy Class - JFrameBasic

New Name: HelloWorld

Project: JavaTerm1

Location: Source Packages

To Package: ch06_gui1

□ Copy Without Refactoring

This will create a new class called HelloWorld in the package, with all of the JFrameBasics changed to HelloWorld. Make the following changes (indicated in bold):

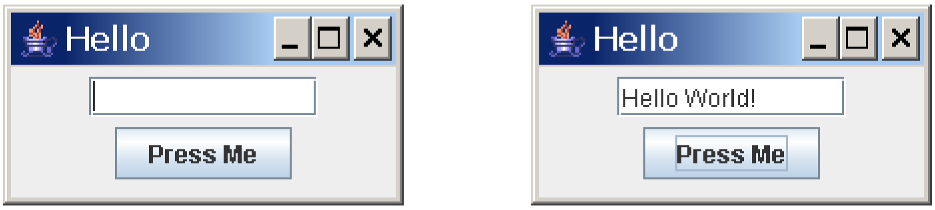
```
01 package ch06_gui1;
02
03 import java.awt.*;
04 import javax.swing.*;
05 import java.awt.event.*;
06
07 public class HelloWorld
08         extends JFrame implements ActionListener {
09
10     JTextField helloText = new JTextField(10);
11     JButton helloButton = new JButton("Press Me");
12
13     public static void main(String[] args) {
14         HelloWorld jf = new HelloWorld();
15     }
16
17     public HelloWorld() {
18         setLayout(new FlowLayout());
19         setSize(200, 100);
20         setTitle("Hello");
21         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22         setLocationRelativeTo(null);
23
24         add(helloText);
25         add(helloButton);
26         helloButton.addActionListener(this);
27
28         setVisible(true);
29     }
30
31     public void actionPerformed(ActionEvent e) {
32         helloText.setText("Hello World!");
33     }
34 }
```

Lines 10 and 11 declare a *text field* 10 columns wide and a *button* with text Press Me They are *instance variables* of the frame. Lines 24 and 25 add them to the frame's content pane. Line 26 is important. It *registers* the button with the action listener for the frame (identified by this). When the user clicks the button, the `actionPerformed` method is called automatically and the text in the text field is set to `Hello World!`

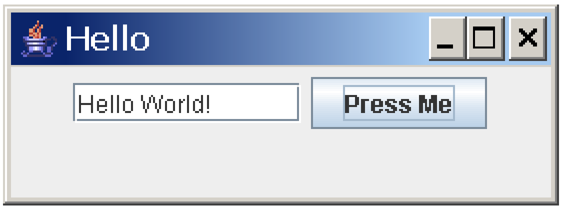
If there was more than one button, the parameter `e` of the `actionPerformed` method could be used to determine which control caused the event.

If you leave out line 26 the application compiles, but clicking the button has no effect.

Here is the appearance of the frame before and after clicking the button:



If you use the mouse to make the frame wider, the button "flows" up alongside the text field:

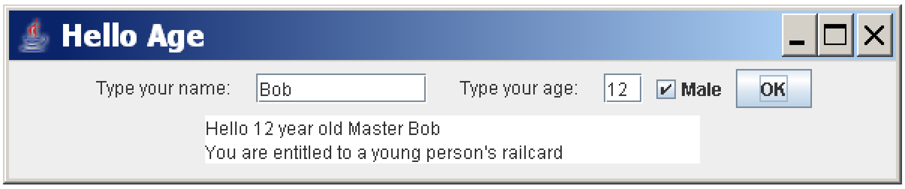


If you cannot get it to work you will find a completed version in `HelloWorldDone`.

HelloAge as a frame application

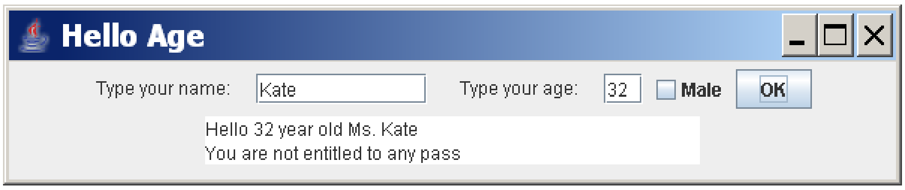
Let us rewrite the `HelloAge` application as a JFrame, introducing ing a few extra controls – labels, a check box, a text field and a text area.

```
package ch06_gui1;

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

public class HelloAge
        extends JFrame implements ActionListener {

    JTextField nameTxt = new JTextField(10);
    JTextField ageTxt = new JTextField(2);
    JCheckBox male = new JCheckBox("Male");
    JTextArea output = new JTextArea(2, 30);
    JButton ok = new JButton("OK");

    public static void main(String[] args) {
        HelloAge jf = new HelloAge();
    }

    public HelloAge() {
        setLayout(new FlowLayout());
        setSize(600, 120);
        setTitle("Hello Age");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);

        add(new Label("Type your name:"));
        add(nameTxt);
        add(new Label("     Type your age:"));
        add(ageTxt);
        add(male);
        add(ok);
        ok.addActionListener(this);
        add(output);
        output.setEditable(false);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        String name = nameTxt.getText();
        int age = Integer.parseInt(ageTxt.getText());

        String title;
        if (male.isSelected() && age >= 18) {
            title = "Mr. ";
        } else if (male.isSelected() && age < 18) {
            title = "Master ";
        } else if (!male.isSelected() && age >= 18) {
            title = "Ms. ";
        } else {
            title = "Miss ";
        }

        String pass;
        if (age <= 25) {
            pass = "\nYou are entitled to a young person's railcard";
        } else if (age >= 60) {
            pass = "\nYou are entitled to a freedom pass";
        } else {
            pass = "\nYou are not entitled to any pass";
        }

        String message = "Hello " + age + " year old " +
                title + name + pass;
        output.setText(message);
    }
}
```



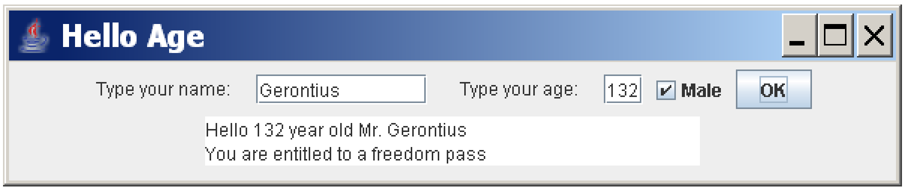
Bob ☑ Male 12

Hello 12 year old Master Bob
You are entitled to a young person's railcard



Kate ☐ Male 32

Hello 32 year old Ms. Kate
You are not entitled to any pass



Gerontius ☑ Male 132

Hello 132 year old Mr. Gerontius
You are entitled to a freedom pass

Lines 10 – 14 declare two text fields, a button, a *check box* and a *text area*. Check boxes have two states, *selected* (a tick is displayed) and *unselected* (no tick is displayed). Text areas can have multiple lines.

Lines 27 – 35 place these controls on the content pane. Line 35 sets the "editable" property of the text area to `false`, which means that the text it contains can't be changed by the user.

Lines 40 and 41 demonstrate the `getText()` method of the `JTextField` class, used to read the text in the field. (All controls with a text part understand the `getText()` method).

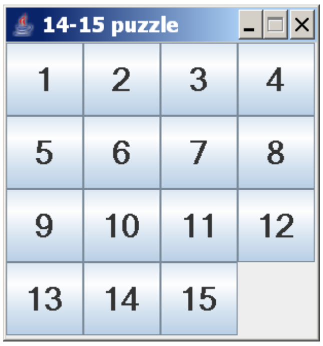
Lines 44, 46 and 48 demonstrate the `isSelected()` method of the `JCheckBox` class, which tells if the check box is selected or not. (Radio buttons also understand the `isSelected()` method).

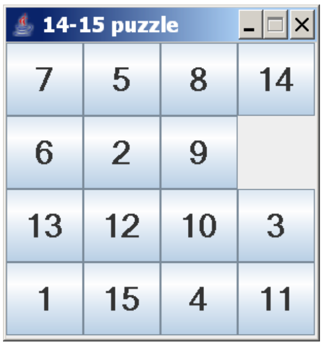
The layout of this application isn't very nice. And if you run it and resize the frame, the controls get horribly jumbled up.

Layout managers

`JFrame`s can have other layout managers to help make the GUI appearance more pleasing. We illustrate this with three demo applications: the 14-15 sliding tile puzzle, which uses *grid layout*, an application to play with various shapes, which uses *border layout* and flow layout together, and a quiz application which uses all three layouts. These programs are quite complex so they are not listed here. If you want to run them and view the code they are in their own separate packages, `ch06_gui1.puzzle`, `ch06_gui1.shapes` and `ch06_gui1.quiz` respectively.

14-15 puzzle





This is in a non-resizable frame. It has 16 buttons as shown here in the solved state. The button with text 16 is made invisible using `setVisible(false).`

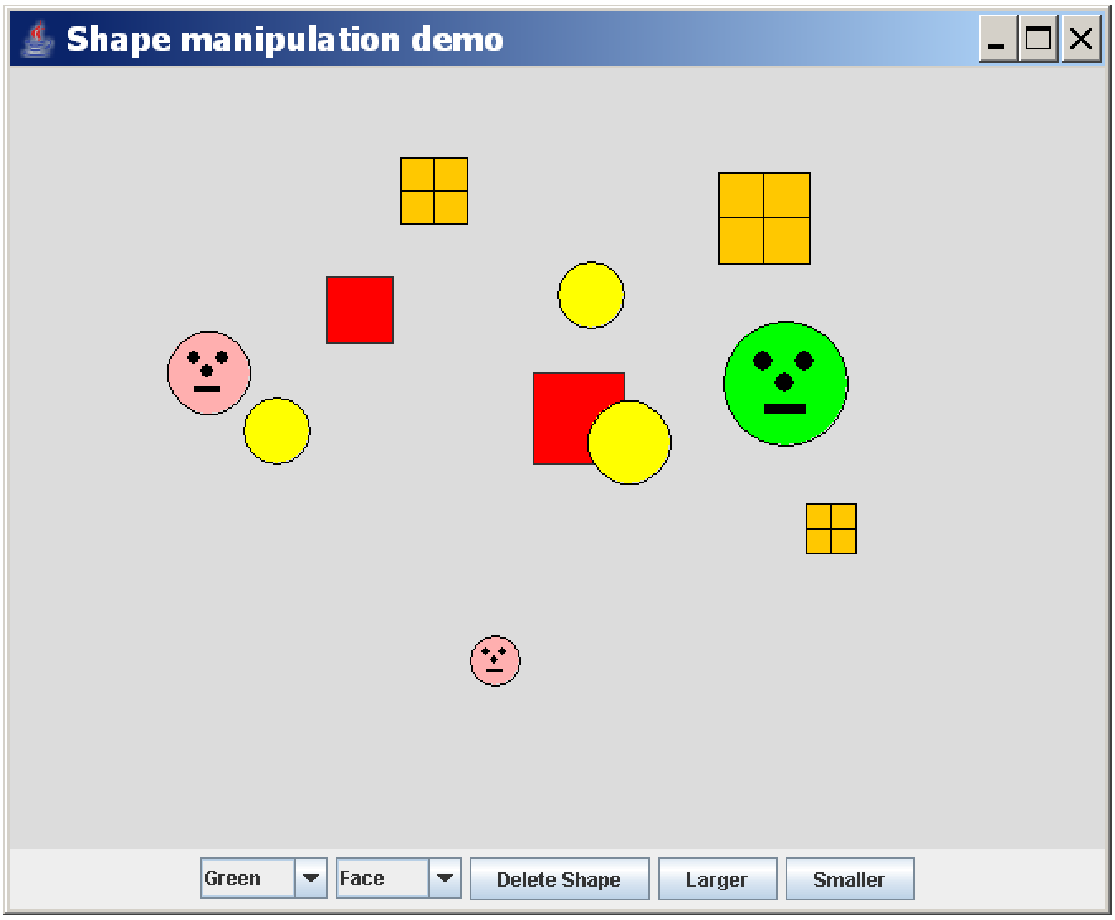
The frame uses the grid layout manager set up in this case with

```
setLayout(new GridLayout(4, 4));
```

As the buttons are added to the frame they go into the grid positions in a left-to-right, top-to-bottom fashion, filling the available space.

In this state, when any button is clicked the text on the buttons is randomly shuffled as shown in the example here. Now, as the user clicks on the buttons they appear to move into the vacant space if adjacent (actually the texts and visibility of the buttons are adjusted). When the puzzle reaches the solved state again a message box pops up saying how many moves it took (less than 100 is good going) and the buttons are reshuffled.

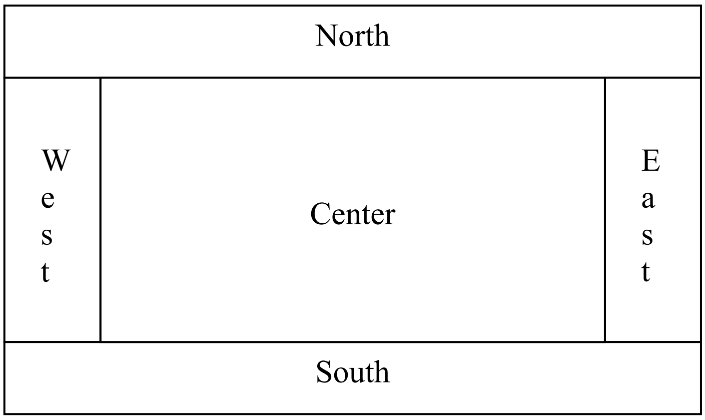
Shapes demo



This application uses the border layout manager set up with

```
setLayout(new BorderLayout());
```

Border layout splits the content pane (or other container such as a *panel*) into up to five regions North, South, East, West and Center (note the American spelling)

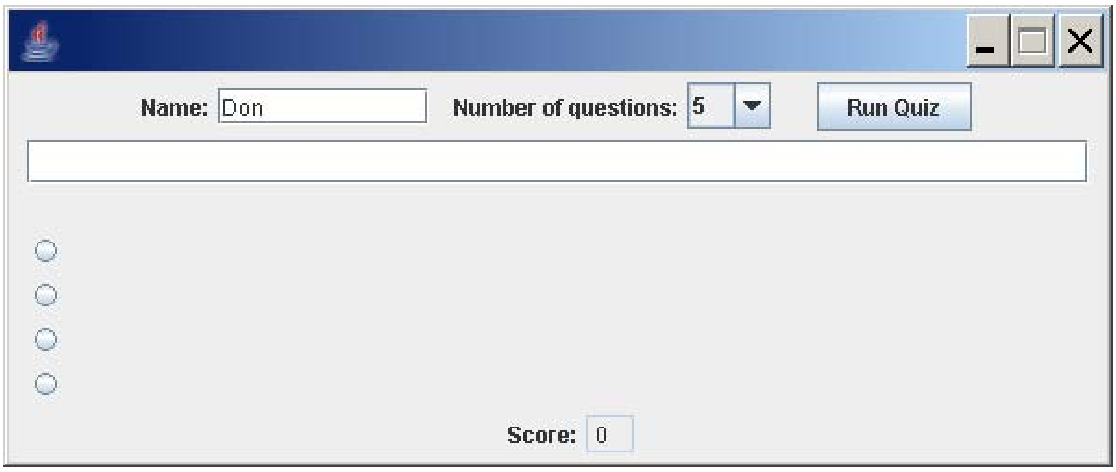


In the Shapes application the South region is occupied by a *panel* that contains two *combo boxes* and three buttons. This panel uses flow layout as you can see. The North, East and West regions are not used and the Center region takes up all the rest of the available space. A *canvas* is added here. Canvases can be drawn on and they can listen out for mouse events.

In this application the user selects a colour and a shape from the combo boxes. Right clicking with the mouse on the canvas draws the selected shape. Shapes can be moved by dragging and dropping with the mouse, and the selected shape can be made larger, smaller or deleted by using the three buttons.

The frame can be resized; the panel still occupies the entire South region and the canvas occupies the rest of the content pane.

Quiz



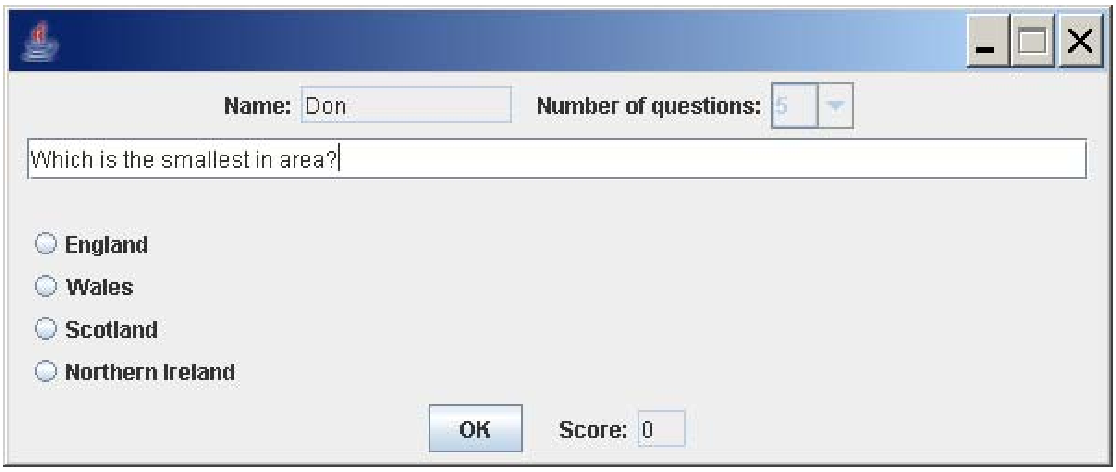
Name: Don

Number of questions: 5

○
○
○
○

Score: 0

This is the startup state. The user types his/her name and selects the number of questions (from 1 to 10) using the combo box, then clicks the Run Quiz button



Name: Don

Number of questions: 5

Which is the smallest in area?

○ England
○ Wales
○ Scotland
○ Northern Ireland

Score: 0

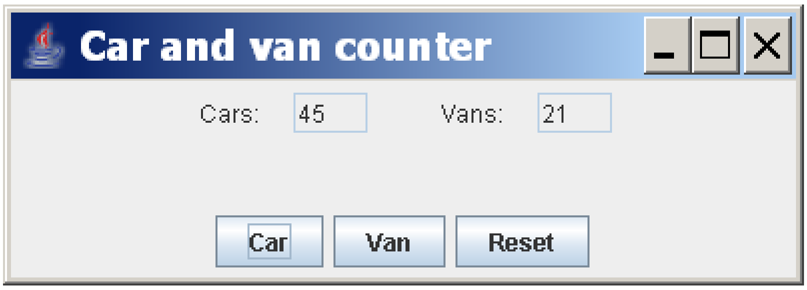
In this case five questions are selected at random from a data bank of about 100 questions stored in a file in the resources folder of your the JavaTerm1 project, `res\ch06_gui1\QA.txt`. (We cover files later on). Note that the Run Quiz button is now invisible, the name textbox is not editable and the combo box is disabled. An OK button for the user to submit his/her selected answer has become visible. When the quiz has finished, the score is displayed on a message box and the state reverts to the start state.

The frame uses border layout with panels in each of the five regions. The North and South panels use flow layout, the Center panel uses grid layout with 6 rows and 1 column. This grid is filled with a text field for the question, a dummy control for padding and four radio buttons in a button group (so that at most one may be selected at any time). The East and West regions are occupied by dummy panels for padding.

Simple demos

The puzzle, shapes and quiz applications show what may be done with Swing controls and layout managers, but they are rather complex. Here are some simpler examples.

Car and van counter



Car and van counter

Cars: 45 Vans: 21

This application simulates an observer at the roadside counting off the numbers of cars and vans that pass by. It starts off with Cars and Vans set to 0. Every time the Car button is clicked, one is added to the number of cars and every time the Van button is clicked, one is added to the number of vans. The Reset button sets Cars and Vans back to 0.

```
package ch06_gui1;

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

public class Vehicles extends JFrame implements ActionListener {

    JTextField carsTxt = new JTextField(3);
    JTextField vansTxt = new JTextField(3);
    JButton carsBtn = new JButton("Car");
    JButton vansBtn = new JButton("Van");
    JButton reset = new JButton("Reset");
    int cars = 0, vans = 0; // counters

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

    public Vehicles() {
        setLayout(new BorderLayout());
        setSize(400, 140);
        setTitle("Car and van counter");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        JPanel top = new JPanel();
        top.setLayout(new FlowLayout());
        top.add(new Label("Cars:"));
        top.add(carsTxt);
        carsTxt.setEditable(false);
31         top.add(new Label("         Vans:"));
32         top.add(vansTxt);
33         vansTxt.setEditable(false);
34         carsTxt.setText("0");
35         vansTxt.setText("0");
36         add("North", top);
37         JPanel bottom = new JPanel();
38         bottom.setLayout(new FlowLayout());
39         bottom.add(carsBtn);
40         bottom.add(vansBtn);
41         bottom.add(reset);
42         carsBtn.addActionListener(this);
43         vansBtn.addActionListener(this);
44         reset.addActionListener(this);
45         add("South", bottom);
46         setVisible(true);
47     }
48
49     public void actionPerformed(ActionEvent e) {
50         if (e.getSource() == carsBtn) {
51             cars++;
52         } else if (e.getSource() == vansBtn) {
53             vans++;
54         } else if (e.getSource() == reset) {
55             cars = vans = 0;
56         }
57         carsTxt.setText("" + cars);
58         vansTxt.setText("" + vans);
59     }
60 }
```

This uses two `int` counters declared at line 14 to keep a count of the number of cars and vans passing by.
The frame uses border layout with a panel in each of the North and South regions. These panels use flow layout.
Lines 42 – 44 add the action listener for the frame to each of the three buttons.

The `actionPerformed` method is called each time any of the three buttons is clicked. The `ActionEvent` parameter `e` is used to find out which button was clicked, using `e.getSource()` to determine which button was clicked as the source of the event. This is common practice.

At line 57 the `setText` method needs a `String` hence `"" + cars`. This is a shortcut – more formally we could use

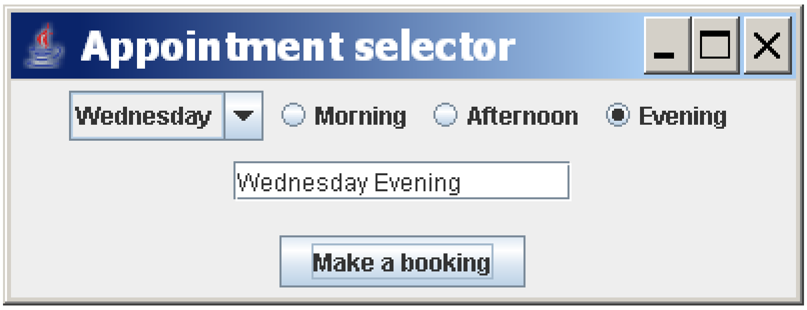
```
carsTxt.setText(Integer.toString(cars));
```

Similarly for line 58.

Self-test exercise

2. How would you alter this application to count cars, vans *and lorries* passing by?

Appointment selector



Wednesday Evening

This application illustrates the use of combo boxes and radio buttons after the user has made some selections. It is rather pointless as it stands, but if it were connected to a database could be used as the basis for a calendar type application.

```
package ch06_gui1;

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

public class Appointment
        extends JFrame implements ActionListener {

    JComboBox day = new JComboBox();
    JRadioButton am = new JRadioButton("Morning", true);
    JRadioButton pm = new JRadioButton("Afternoon", false);
    JRadioButton eve = new JRadioButton("Evening", false);
    ButtonGroup timeSlot = new ButtonGroup();
    JTextField bookTxt = new JTextField(15);
    JButton bookBtn = new JButton("Make a booking");

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

    public Appointment() {
        setLayout(new BorderLayout());
        setSize(400, 150);
        setTitle("Appointment selector");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        JPanel top = new JPanel();
        top.setLayout(new FlowLayout());
        day.addItem("Monday");
        day.addItem("Tuesday");
        day.addItem("Wednesday");
        day.addItem("Thursday");
        day.addItem("Friday");
        top.add(day);
        timeSlot.add(am);
        timeSlot.add(pm);
        timeSlot.add(eve);
        top.add(am);
        top.add(pm);
        top.add(eve);
        add("North", top);
        JPanel middle = new JPanel();
        middle.setLayout(new FlowLayout());
        middle.add(bookTxt);
46         add("Center", middle);
47         JPanel bottom = new JPanel();
48         bottom.setLayout(new FlowLayout());
49         bottom.add(bookBtn);
50         add("South", bottom);
51         bookBtn.addActionListener(this);
52         setVisible(true);
53     }
54 
55     public void actionPerformed(ActionEvent e) {
56         String time;
57         if (am.isSelected()) {
58             time = " Morning";
59         } else if (pm.isSelected()) {
60             time = " Afternoon";
61         } else {
62             time = " Evening";
63         }
64         bookTxt.setText(day.getSelectedItem() + time);
65     }
66 }
```

At line 10 a combo box is declared, initially empty.

Lines 11 – 13 declare three radio buttons with a text and initial state. If they are all to go in the same button group exactly one should be set to true, the others to false. Radio buttons should always belong to a button group (when a button is selected it will switch all of the others to false). One is declared at line 14.

At lines 36 – 41 the radio buttons are added to both the button group and the panel.

At lines 57, 59 we use `isSelected()` for radio buttons just as we do for check boxes.

At line 64 the `getSelectedItem()` method returns the selected item in the list. You can also use the `getSelectedIndex()` method that returns the position of the selected item in the list, starting at 0. Thus if the selected item is `"Thursday"` `day.getSelectedIndex()` returns the value 3.

Self-test exercise

3. How would you rewrite `HelloAge.java` using radio buttons for the user's sex instead of a check box?

Summary

This chapter has been a very brief introduction to GUI programming using Swing. We investigated the following Swing controls:

- JFrames – with a useful template file
- JButtons and the ActionListener technique for event handling
- JLabels, JTextFields and JTextAreas for text handling
- JCheckBoxes, JRadioButtons + ButtonGroups and JComboBoxes for making choices

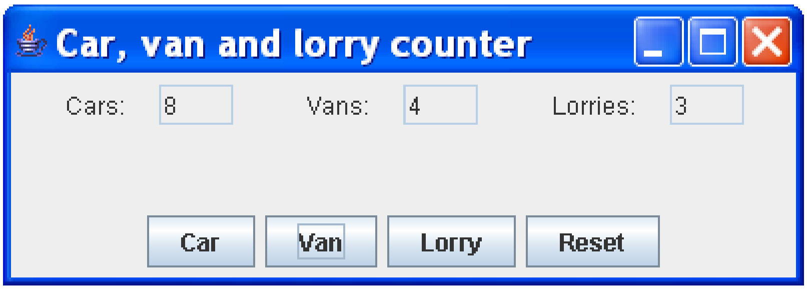
We saw that all button click events are handled by a single ActionPerformed method. This has an ActionEvent parameter that can be tested to discover which button caused the event to happen.

We also saw that there are layout managers to control the appearance of the various controls on a form – flow layout, border layout, grid layout.

There are many other types of control, layout managers and event listeners in Java but these are the fundamental ones.

Solutions to self-test exercises

1. Do this in the lab!
2. Include another `JTextField lorriesTxt = new JTextField(3)` and `JButton lorriesBtn = new JButton("Lorry");` and another `int lorries = 0;` Add `lorriesTxt` with a suitable label to the top and `lorriesBtn` to the bottom and add this button to the action listener. In `actionPerformed` add a clause `else if (e.getSource() == lorriesBtn) lorries++;` and to reset write `cars = vans = lorries = 0;` Finally add `lorriesTxt.setText("" + lorries);` and change the title:


Car, van and lorry counter

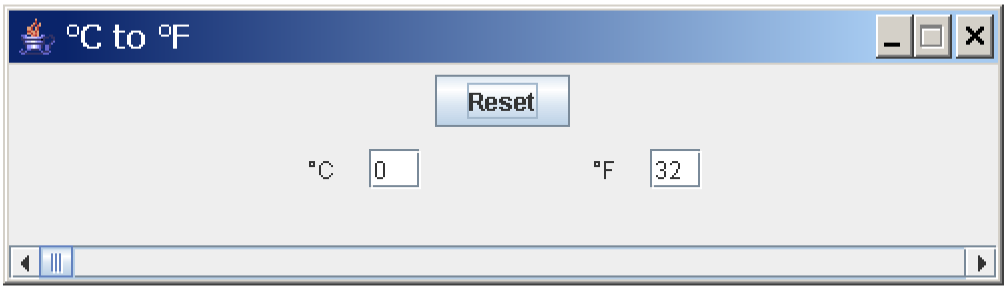
Cars: 8 Vans: 4 Lorries: 3

3. Get rid of the check box `male` and add two radio buttons `male` and `female` in a button group. Set one to `true`, the other to `false`. Add both buttons to the top panel.

7. GUI programming – Scrollbars and Canvases

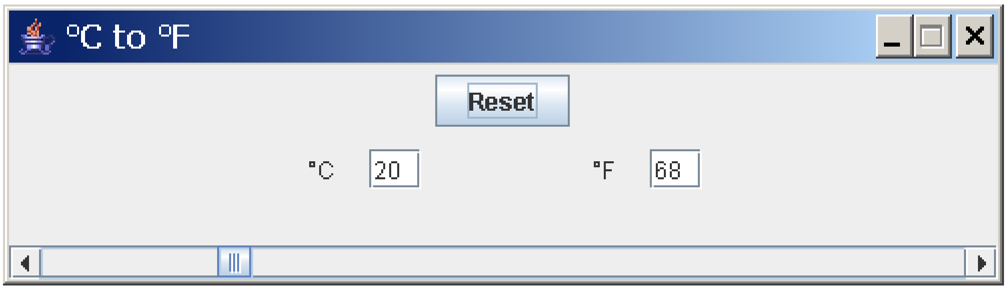
Scrollbars and sliders

The following application converts °C to °F. The user adjusts a *scrollbar* for °C between 0 and 100 and this is instantaneously converted into °F. Pressing the Reset button sets the scrollbar and the °C and °F values back to the startup position shown:



°C 0 °F 32

The conversion formula used is °F = °C * 9/5 + 32. Thus 20 °C = 68 °F



°C 20 °F 68

```
package ch07_gui2;

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

public class CtoF extends JFrame
        implements ActionListener, AdjustmentListener {

    JTextField centigradeTxt = new JTextField(2);
    JTextField fahrenheitTxt = new JTextField(2);
    JScrollBar centigradeScr =
            new JScrollBar(JScrollBar.HORIZONTAL, 0, 0, 0, 100);
    JButton resetBtn = new JButton("Reset");
    JPanel top = new JPanel();
    JPanel middle = new JPanel();

    public static void main(String[] args) {
        CtoF jf = new CtoF();
    }

    public CtoF() {
        reset();
        setLayout(new BorderLayout());
        setSize(500, 140);
        setTitle("°C to °F");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLocationRelativeTo(null);
        top.setLayout(new FlowLayout());
30         top.add(resetBtn);
31         resetBtn.addActionListener(this);
32         middle.setLayout(new FlowLayout());
33         middle.add(new Label("°C"));
34         middle.add(centigradeTxt);
35         middle.add(new Label("                          °F"));
36         middle.add(fahrenheitTxt);
37         add("South", centigradeScr);
38         centigradeScr.setBlockIncrement(5);
39         centigradeScr.addAdjustmentListener(this);
40         add("Center", middle);
41         add("North", top);
42         setResizable(false);
43         setVisible(true);
44     }
45
46     public void actionPerformed(ActionEvent e) {
47         reset();
48     }
49
50     public void adjustmentValueChanged(AdjustmentEvent e) {
51         int centigrade = centigradeScr.getValue();
52         // convert to the nearest ?F
53         int fahrenheit = 32 + Math.round(centigrade * 1.8f);
54         centigradeTxt.setText("" + centigrade);
55         fahrenheitTxt.setText("" + fahrenheit);
56     }
57
58     private void reset() {
59         centigradeTxt.setText("0");
60         fahrenheitTxt.setText("32");
61         centigradeScr.setValue(0);
62     }
63 }
```

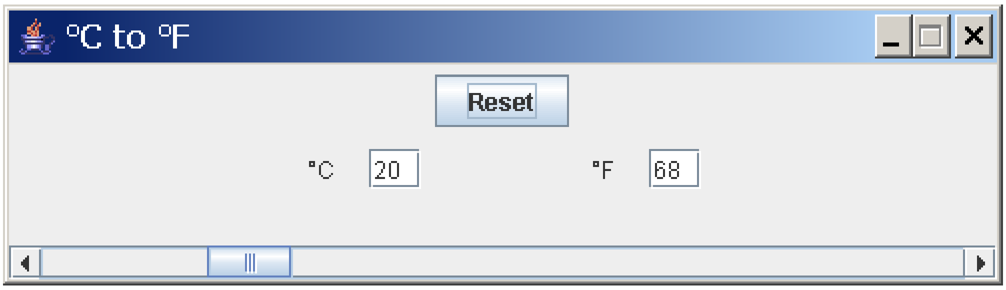
Line 08: **`AdjustmentListener`** is used for scrollbar events so this is implemented as well as **`ActionListener`**.

Lines 12 – 13: This sets up a horizontal scrollbar with minimum value 0, maximum value 100, initial value 0. In general we would use

```
new Scrollbar(JScrollBar.HORIZONTAL, value, extent, min, max)
```

where `value` is the initial value, `extent` is the thickness of the "handle", `min` is the minimum value and the maximum value is `max – extent`. It's easier to let `extent = 0` when a minimum handle thickness is used. We could have obtained a thicker handle with the same value range by writing

```
12     JScrollBar centigradeScr =
13         new JScrollBar(JScrollBar.HORIZONTAL, 0, 10, 0, 110);
```



If we want a vertical scrollbar the first argument would be **`JScrollBar.VERTICAL.`**

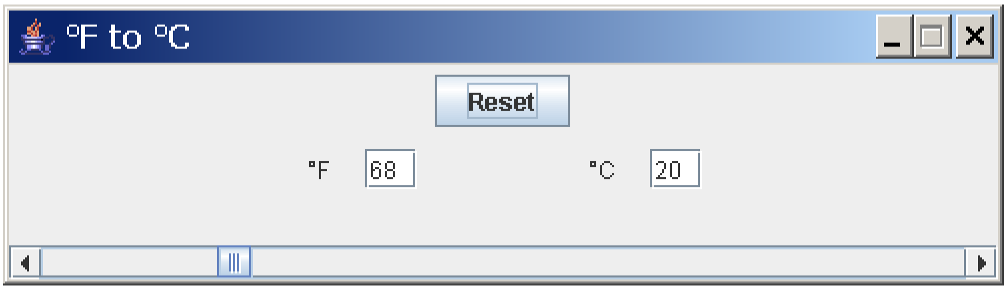
Lines 37 – 39: The scrollbar is added to the south of the frame, its *block increment* is set to 5 and it is added to the adjustment listener for the frame so that adjustment events can be handled. The block increment is the amount the scrollbar's value changes if the user clicks the region either side of the handle.

Lines 50 – 56: The **`adjustmentValueChanged`** method must be implemented for the **`AdjustmentListener`** interface, just as **`actionPerformed`** must be implemented for **`ActionListener`**. This method is called every time the scrollbar changes value. Line 51 uses the **`getValue()`** method to obtain the value of the scrollbar for processing.

Line 61 uses the **`setValue(int value)`** method to set the scrollbar's value to 0.

Self-test exercise

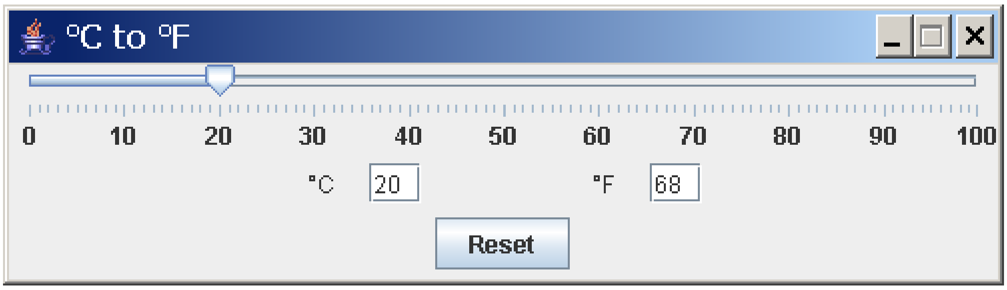
1. Informally describe the changes needed to write an application to convert °F in the range 32 to 212 to °C.



°F 68 °C 20

Scrollbars are normally used along the edges of a frame or panel using border layout.

The Java Swing API introduced a scrollbar variant called a *slider*. Here is a variant of the CtoF converter using a slider:



°C 20 °F 68

The lines of code relating to slider handling are:

```
...
06 import javax.swing.event.*;
...
08 public class CtoFSlider extends JFrame
09                   implements ActionListener, ChangeListener
...
13     JSlider centigradeSl = new JSlider(0, 100, 0);
...
37         centigradeSl.setMajorTickSpacing(10);
38         centigradeSl.setMinorTickSpacing(1);
39         centigradeSl.setPaintTicks(true);
40         centigradeSl.setPaintLabels(true);
41         add("North", centigradeSl);
42         centigradeSl.addChangeListener(this);
...
53     public void stateChanged(ChangeEvent e) {
```

```
54          int centigrade = centigradeSl.getValue();
55          // convert to the nearest ?F
56          int fahrenheit = 32 + Math.round(centigrade * 1.8f);
57          centigradeTxt.setText("" + centigrade);
58          fahrenheitTxt.setText("" + fahrenheit);
59      }
```

For the complete listing, see `CtoFSlider.java` in the package `ch07_gui2`.

Line 06: we need an extra import for slider event handling.

Lines 08 – 09: sliders need the `ChangeListener` interface.

Line 13: the commonest constructor is `JSlider(int min, int max, int value)`

This produces a horizontal slider as shown.

Lines 37 – 40: these are used to paint the numbers and tick marks shown.

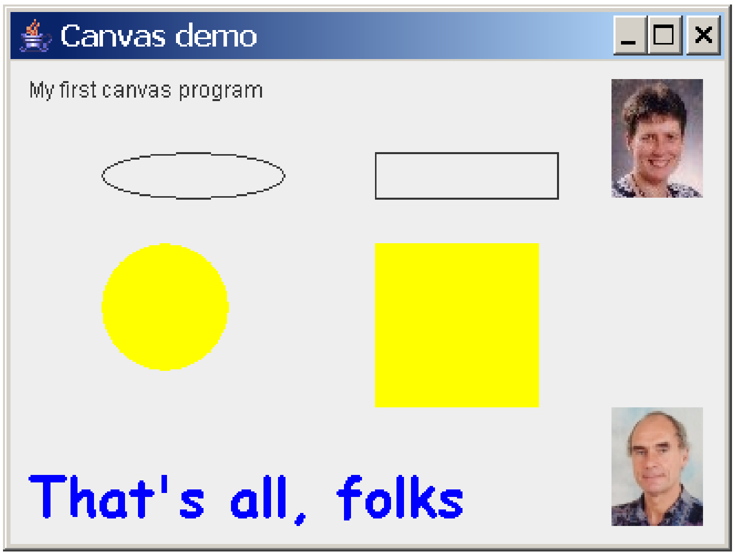
Line 42: we use `addChangeListener`, not `addAdjustmentListener`.

Lines 53 – 59: The `stateChanged` method must be implemented for the `ChangeListener` interface, just as `actionPerformed` must be implemented for `ActionListener`. This method is called every time the slider changes value. Line 56 uses the `getValue()` method to obtain the value of the slider for processing.

For a more comprehensive illustration see `SliderDemo.java`. Note that this demo takes its images from the `res\ch07_gui2\doggy` folder.

Canvases

A *canvas* is a rectangular region upon which graphics and text may be drawn (using the canvas's *graphics context*) and which may accept mouse events for processing. The following silly application demonstrates some of the graphics facilities available (here showing pictures of Kate and Don):



My first canvas program

That's all, folks

This program was written with the aid of a second template file called `CanvasBasic.java` in `ch07_gui2`.

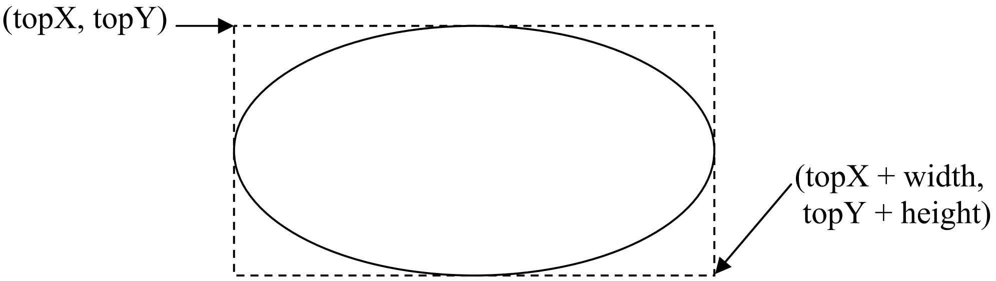
```
01 package ch07_gui2;
02
03 import java.awt.*;
```

```
04 import javax.swing.*;
05 import java.awt.event.*;
06
07 public class CanvasDemo extends JFrame
08         implements ActionListener {
09
10     private MyCanvas canvas = new MyCanvas();
11
12     public static void main(String[] args) {
13         CanvasDemo jf = new CanvasDemo();
14     }
15
16     public CanvasDemo() {
17         setLayout(new BorderLayout());
18         setSize(400, 300);
19         setTitle("Canvas demo");
20         add("Center", canvas);
21         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22         setLocationRelativeTo(null);
23         setVisible(true);
24     }
25
26     public void actionPerformed(ActionEvent e) {
27         // add your event handling code here
28     }
29
30     private class MyCanvas extends Canvas {
31
32         @Override
33         public void paint(Graphics g) {
34             g.drawString("My first canvas program", 10, 20);
35             g.drawOval(50, 50, 100, 25);
36             g.drawRect(200, 50, 100, 25);
37             g.setColor(Color.yellow);
38             g.fillOval(50, 100, 70, 70);
39             g.fillRect(200, 100, 90, 90);
40             Image kate =
41                new ImageIcon("res/ch07_gui2/kate.jpg").getImage();
42             Image don =
43                new ImageIcon("res/ch07_gui2/don.jpg").getImage();
44             g.drawImage(kate, 330, 10, this);
45             g.drawImage(don, 330, 190, this);
46             g.setColor(Color.blue);
47             g.setFont(new Font("Comic Sans MS", Font.BOLD, 30));
48             g.drawString("That's all, folks", 10, 250);
49         }
50     }
51 }
```

To use a canvas you must define a class extending library class **`Canvas`**. This is *inside* the `JFrame` class (lines 30 – 50). Such classes are called *nested* or inner classes and are usually declared private. The `public void paint(Graphics g)` method is called when the application first runs and whenever the canvas needs refreshing, e.g. whenever the frame containing the canvas is obscured by another window. The parameter `g` is the canvas's *graphics context*. You draw on the canvas by sending messages to this object, as at lines 34 – 48.

Line 34: `drawString(String s, int x, int y)` draws a string `s` on the canvas starting at pixel position (`x, y`) – the top left of the string's first character.

Lines 35, 36, 38, 39: the `Graphics` class has several methods to draw and fill various shapes. Each shape other than a line is drawn in a conceptual “bounding box” specified by its top left and bottom right corners. For instance, here is the effect of `g.drawOval(topX, topY, width, height)`:



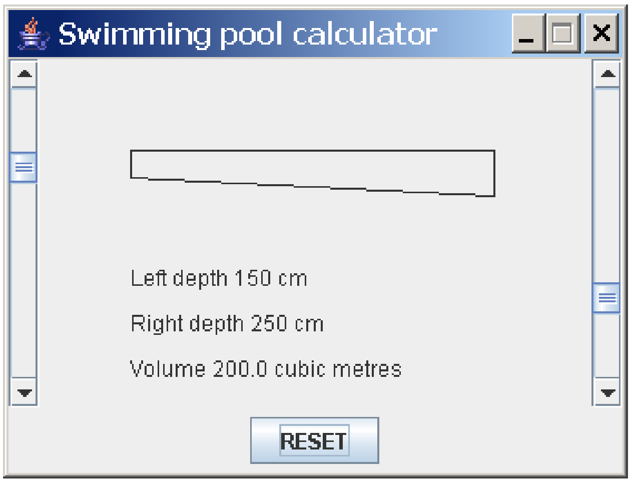
Lines 37, 46: the `setColor(Color c)` method changes the drawing colour of the canvas. (You can also use `getColor()` to find out the current colour). Note the American spelling: `Color`, not `Colour`.

Lines 40 – 45 show how you can draw images. (Images are better used as icons on other Swing components. For details see the online Java documentation described at the end of this chapter.)

Line 47 shows how you change the drawing font. You can also change the font for labels and text fields, etc, in a similar fashion. The argument is a font, usually constructed on the fly as here. The three parameters of the Font constructor are the font name (which must match a system or Java defined font), the font style and the point size. Fonts may be system dependent, see the Java documentation for details.

Line 05 and lines 26 – 28 are unnecessary as this application doesn’t handle any events. They are included by the template file and they do no harm!

Swimming pool calculator



Left depth 150 cm

Right depth 250 cm

Volume 200.0 cubic metres

A landscape gardener wants an application to calculate the volume of a swimming pool 20 metres long by 5 metres wide, where the depth at each end may vary between 1 metre and 3 metres in centimetre increments

The formula for calculating the volume is

Volume = (left + right) * 0.5

where left and right are in cm and Volume is in cubic metres.

```
01 package ch07_gui2;
02
03 import java.awt.*;
04 import java.awt.event.*;
05 import javax.swing.*;
06
07 class Pool extends JFrame implements ActionListener, AdjustmentListener {
08
09     private JScrollBar leftScr =
10             new JScrollBar(JScrollBar.VERTICAL, 100, 0, 100, 300);
11     private JScrollBar rightScr =
12             new JScrollBar(JScrollBar.VERTICAL, 300, 0, 100, 300);
13     private int left,  right; // depth of pool at each end in cm
14     private JPanel bottom = new JPanel();
15     private JButton reset = new JButton("RESET");
16     private PoolCanvas poolCanvas = new PoolCanvas();
17
18     public static void main(String[] args) {
19         Pool pool = new Pool();
20     }
21
22     Pool() {
23         setLayout(new BorderLayout());
24         setTitle("Swimming pool calculator");
25         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
26         setLocationRelativeTo(null);
27         setSize(344, 261); // so the canvas is 300 by 200
28         leftScr.setBlockIncrement(5);
29         rightScr.setBlockIncrement(5);
30         leftScr.addAdjustmentListener(this);
31         rightScr.addAdjustmentListener(this);
32         reset.addActionListener(this);
33         left = 100;
34         right = 300;
35         bottom.setLayout(new FlowLayout());
36         bottom.add(reset);
37         add("West", leftScr);
38         add("East", rightScr);
39         add("South", bottom);
40         add("Center", poolCanvas);
41         setResizable(false);
42         setVisible(true);
43     }
44
45     public void actionPerformed(ActionEvent e) {
46         left = 100;
47         right = 300;
48         leftScr.setValue(left);
49         rightScr.setValue(right);
50         poolCanvas.repaint();
51     }
52
53     public void adjustmentValueChanged(AdjustmentEvent e) {
54         if (e.getSource() == leftScr) {
55             left = leftScr.getValue();
56         }
57         if (e.getSource() == rightScr) {
58             right = rightScr.getValue();
59         }
60         poolCanvas.repaint();
61     }
62
63     class PoolCanvas extends Canvas {
64
65         @Override
66         public void paint(Graphics g) {
67             g.drawLine(50, 50, 250, 50);                          // top
68             g.drawLine(250, 50, 250, 50 + right / 10);            // right
69             g.drawLine(250, 50 + right / 10, 50, 50 + left / 10); // bottom
70             g.drawLine(50, 50 + left / 10, 50, 50);               // left
71             g.drawString("Left depth " + left + " cm", 50, 125);
72             g.drawString("Right depth " + right + " cm", 50, 150);
73             g.drawString("Volume " + ((left + right) * 0.5) +
74                     " cubic metres", 50, 175);
75         }
76     }
77 }
```

Lines 45 – 51: The `actionPerformed` method (for the RESET button) sets the instance variables `left` and `right` to their initial values and the two scrollbars to their initial state. It then repaints the canvas, which invokes the `paint` method of the nested `poolCanvas` class.

Lines 53 – 61: The `adjustmentValueChanged` method (for the two scrollbars) determines which scrollbar was adjusted and updates `left` or `right` accordingly, and repaints the canvas.

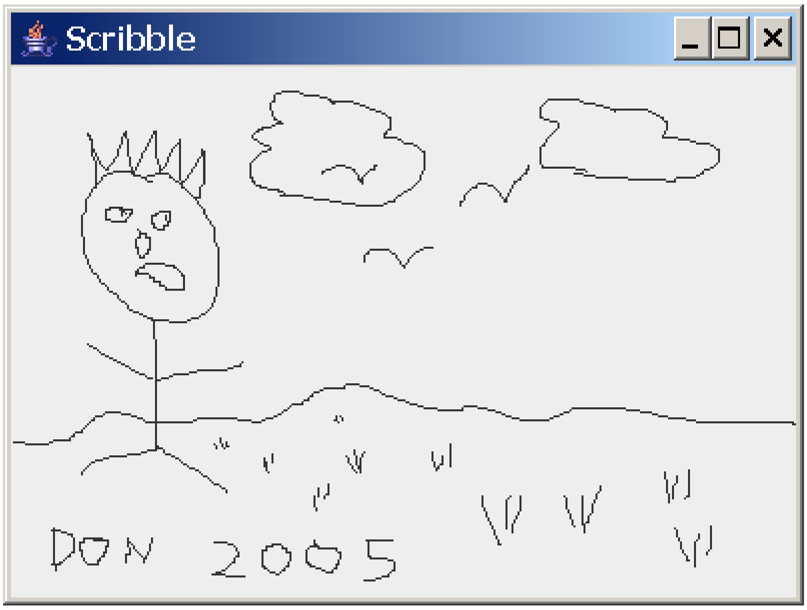
Lines 66 – 75: The nested class's `paint` method draws four lines to depict the pool and displays the left depth, right depth and volume. Note that `drawLine(x1, y1, x2, y2)` draws a straight line between points (`x1, y1`) and (`x2, y2`).

Self-test exercise

2. If you remove the statement `poolCanvas.repaint();` at line 50 the application runs exactly as before. Why?

Handling mouse events

A canvas can be made to recognize and handle various mouse events. The following little program uses both the `mousePressed` event (a mouse key is pressed down) and the `mouseDragged` event (the mouse is being dragged with a key pressed down) to let the user draw a picture (this masterpiece was created by Don some years ago).



```
01 package ch07_gui2;
02
03 import java.awt.*;
04 import javax.swing.*;
05 import java.awt.event.*;
06
07 public class Scribble extends JFrame {
08
09     private ScribbleCanvas canvas = new ScribbleCanvas();
10
11     public static void main(String[] args) {
12         Scribble jf = new Scribble();
13     }
14
15     public Scribble() {
```

```
16          setLayout(new BorderLayout());
17          setSize(400, 300);
18          setTitle("Scribble");
19          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
20          setLocationRelativeTo(null);
21          add("Center", canvas);
22          setVisible(true);
23      }
24
25      private class ScribbleCanvas extends Canvas
26              implements MouseListener, MouseMotionListener {
27
28          int lastX, lastY; // mouse position within canvas
29
30          public ScribbleCanvas() {
31              addMouseListener(this);
32              addMouseMotionListener(this);
33          }
34
35          public void mousePressed(MouseEvent e) {
36              lastX = e.getX();
37              lastY = e.getY();
38          }
39
40          public void mouseDragged(MouseEvent e) {
41              int x = e.getX(), y = e.getY();
42              Graphics g = getGraphics();
43              g.drawLine(lastX, lastY, x, y);
44              lastX = x;
45              lastY = y;
46          }
47
48          // additional event handlers required by the listeners
49          public void mouseMoved(MouseEvent e) {
50          }
51
52          public void mouseClicked(MouseEvent e) {
53          }
54
55          public void mouseReleased(MouseEvent e) {
56          }
57
58          public void mouseEntered(MouseEvent e) {
59          }
60
61          public void mouseExited(MouseEvent e) {
62          }
63      }
64  }
```

Line 26: It's the *inner* or *nested canvas* class that implements the mouse listeners, not the enclosing `JFrame`.

Lines 30 – 33: The `ScribbleCanvas` constructor enables the mouse listeners.

Lines 35 – 38: The `mousePressed` event handler remembers the point (`lastX, lastY`) where the mouse was pressed.

Lines 40 – 46: The `mouseDragged` event is repeatedly fired as the mouse is dragged. Each time, it remembers the new mouse position (`x, y`), draws a line from (`lastX, lastY`) to (`x, y`) and finally updates (`lastX, lastY`) for the next drag event. In effect a series of connected straight lines is drawn on the canvas.

Lines 49 – 62: These dummy handlers are required to satisfy the interfaces. `mouseMoved` is required by the `MouseMotionListener` interface and the others are required to satisfy the `MouseListener` interface.

The drawing is done in the `mouseDragged` handler by getting the `Graphics` context directly (line 43). If you move the application frame off-screen or move another window over it, part or all of the scribble will be erased. This is because there is no `paint` method to refresh the display.

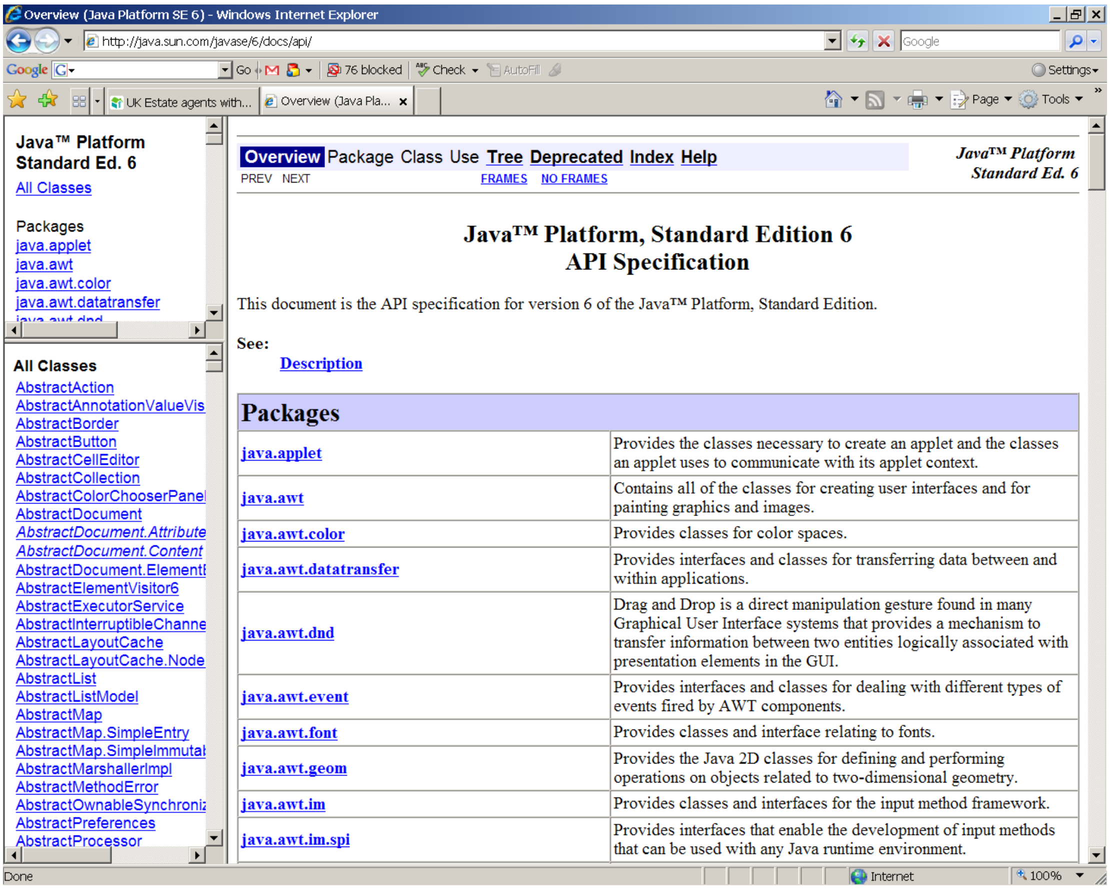
Brief guide to using the online Java documentation

Complete documentation and tutorial support may be found by visiting Sun's website

http://java.sun.com/javase/6/docs/

The most important resource linked from there is the Java Application Programmer's Interface (API) documentation

http://java.sun.com/javase/6/docs/api/



Java™ Platform
Standard Ed. 6

All Classes

Packages
java.applet
java.awt
java.awt.color
java.awt.datatransfer

All Classes
AbstractAction
AbstractAnnotationValueVis
AbstractBorder
AbstractButton
AbstractCellEditor
AbstractCollection
AbstractColorChooserPanel
AbstractDocument
AbstractDocument.Attribute
AbstractDocument.Content
AbstractDocument.Element
AbstractElementVisitor6
AbstractExecutorService
AbstractInterruptibleChanne
AbstractLayoutCache
AbstractLayoutCache.Node
AbstractList
AbstractListModel
AbstractMap
AbstractMap.SimpleEntry
AbstractMap.SimpleImmutal
AbstractMarshallerImpl
AbstractMethodError
AbstractOwnableSynchroniz
AbstractPreferences
AbstractProcessor

Overview Package Class Use **Tree** **Deprecated** **Index** **Help**
PREV NEXT FRAMES NO FRAMES

Java™ Platform
Standard Ed. 6

Java™ Platform, Standard Edition 6
API Specification

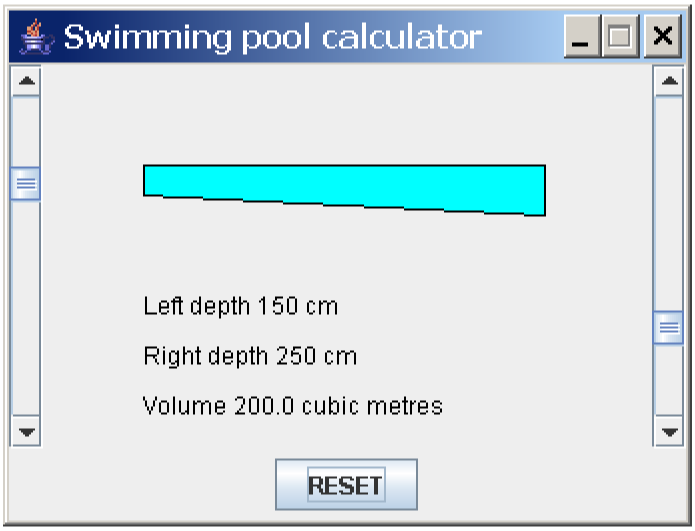
This document is the API specification for version 6 of the Java™ Platform, Standard Edition.

See:
Description

Packages	
java.applet	Provides the classes necessary to create an applet and the classes an applet uses to communicate with its applet context.
java.awt	Contains all of the classes for creating user interfaces and for painting graphics and images.
java.awt.color	Provides classes for color spaces.
java.awt.datatransfer	Provides interfaces and classes for transferring data between and within applications.
java.awt.dnd	Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between two entities logically associated with presentation elements in the GUI.
java.awt.event	Provides interfaces and classes for dealing with different types of events fired by AWT components.
java.awt.font	Provides classes and interface relating to fonts.
java.awt.geom	Provides the Java 2D classes for defining and performing operations on objects related to two-dimensional geometry.
java.awt.im	Provides classes and interfaces for the input method framework.
java.awt.im.spi	Provides interfaces that enable the development of input methods that can be used with any Java runtime environment.

This is arranged in three frames as shown. The bottom left frame lists all of the (thousands of) classes in the API. By clicking on a link in the top left pane you can select just the classes in a particular package. the classes You can browse this to find details of all of the thousands of classes in the library. For instance there are many more methods in the `Graphics` class. When you select a class the documentation for it appears in the pane on the right. This usually consists of a brief description of the class, summaries of the fields, constructors and methods of the class linked to more detailed information, and links to fields and methods *inherited* from superclasses (classes that this class extends).

Suppose we want to enhance the swimming pool example to show the pool filled in blue.



Left depth 150 cm

Right depth 250 cm

Volume 200.0 cubic metres

The pool isn't rectangular so we can't use `drawRect` and `fillRect`. Browsing the documentation we find that there *are* methods of the `Graphics` class called `drawPolygon(Polygon p)` and `fillPolygon(Polygon p)`:

drawPolygon

```
public void drawPolygon(Polygon p)
```

Draws the outline of a polygon defined by the specified `Polygon` object.

Parameters:

`p` - the polygon to draw.

fillPolygon

```
public void fillPolygon(Polygon p)
```

Fills the polygon defined by the specified Polygon object with the graphics context's current color.

The area inside the polygon is defined using an even-odd fill rule, also known as the alternating rule.

Parameters:

`p` - the polygon to fill.

The pool is a polygon with four sides defined by the four points (50, 50), (250, 50), (250, 50 + right/10), (50, 50 + left/10).

The description of these methods contains a link to the `Polygon` class. There we find that it's quite easy to create a polygon from a set of points. For our pool we need

```
Polygon p = new Polygon();
p.addPoint(50, 50);
p.addPoint(250, 50);
p.addPoint(250, 50+right/10);
p.addPoint(50, 50+left/10);
```

Having created the polygon we can fill it in a light blue colour, then draw an outline around it in black:

```
g.setColor(Color.cyan);
g.fillPolygon(p);
g.setColor(Color.black);
g.drawPolygon(p);
```

Rewriting the code in the paint method will have the desired effect. If you don't like cyan, browse the `Color` class documentation for an alternative.

You can download the full documentation from Sun's site and install it on your own PC. But note that it has many links to internet locations (particularly for tutorial support). However the full API documentation is self-contained.

Chapter 18 of the second workbook gives more information on the API.

Summary

This chapter has probed more deeply into the GUI features of Java. We have investigated:

- `JScrollBar`s – with the `AdjustmentListener` technique for event handling
- `JSlider`s – with the `ChangeListener` technique for event handling
- `Canvas`es – with the `MouseListener` and `MouseMotionListener` techniques for event handling
- Some features of the `Graphics` class
- Inner classes and anonymous inner classes for event handling (optional)
- Using Sun's on-line Java documentation

Solutions to self-test exercises

1. Copy the program `FtoC` and make sure that NetBeans changes all occurrences of `CtoF` to `FtoC`. Rename the scrollbar to `fahrenheitScr` and change its min to 32 and max to 212. Swap the positions of `centigradeTxt` and `fahrenheitTxt` and their labels. Rewrite `adjustmentValueChanged` using the formula

   ```
   int centigrade = Math.round((fahrenheit-32) * 5/9.0f);
   ```

   and you're done.

2. Tricky! If you look carefully at the code, and think about what's going on, you will notice that `actionPerformed` changes the scrollbar values and so triggers `adjustmentValueChanged`, which calls `poolCanvas.repaint()`

8. Using Auxiliary Classes

Separating Presentation from Processing

In the last two chapters we looked at ways of using the GUI features of Java to improve the appearance of programs. You will have noted how much code it takes to present even quite simple user interfaces. This code is quite repetitive and, with practice, easy to write. However it does tend to get in the way of the processing required to solve the problem in hand.

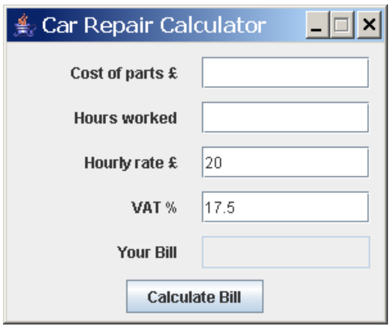
In Chapter 5 we saw how it is possible to separate the presentation from the processing by using separate classes. In this chapter we revisit this idea, looking again at the Car Repair and Carpet Calculator problems and at the Ruritanian Income Tax Calculator from a previous Tutorial.

In each case we have an application class with a `main` method for the presentation, plus a separate auxiliary class to do the processing required.

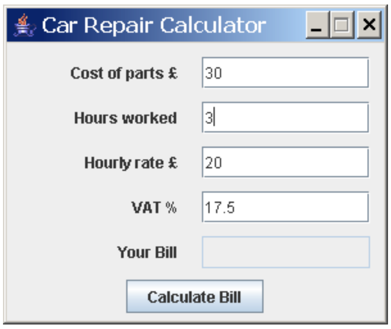
For running simple applications it goes something like this:

1. The user supplies some input and triggers an *event* such as clicking a button.
2. The event handler, e.g. `actionPerformed`, takes this input and creates an object from the auxiliary class using a *constructor*.
3. The event handler then requests some processing to be done on this object by using methods of the auxuiliary class.
4. The event handler uses the result of this processing to update the user interface.

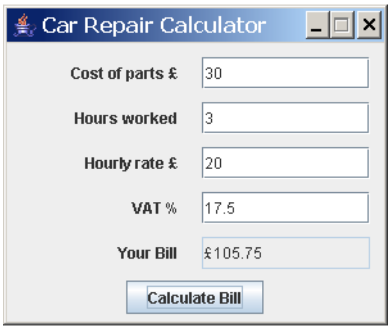
Car Repair Bill Calculator Revisited



Cost of parts £:
Hours worked:
Hourly rate £: 20
VAT %: 17.5
Your Bill:



Cost of parts £: 30
Hours worked: 3
Hourly rate £: 20
VAT %: 17.5
Your Bill:



Cost of parts £: 30
Hours worked: 3
Hourly rate £: 20
VAT %: 17.5
Your Bill: £105.75

The first screenshot shows the initial state of the application (`CarRepairApp1.java`). In the second screenshot the user has typed some figures into the Cost and Hours worked fields. In the third screenshot the user has clicked the Calculate Bill button and the bill is calculated according to steps 1 – 4 above. Here is the relevant code from the `actionPerformed` method:

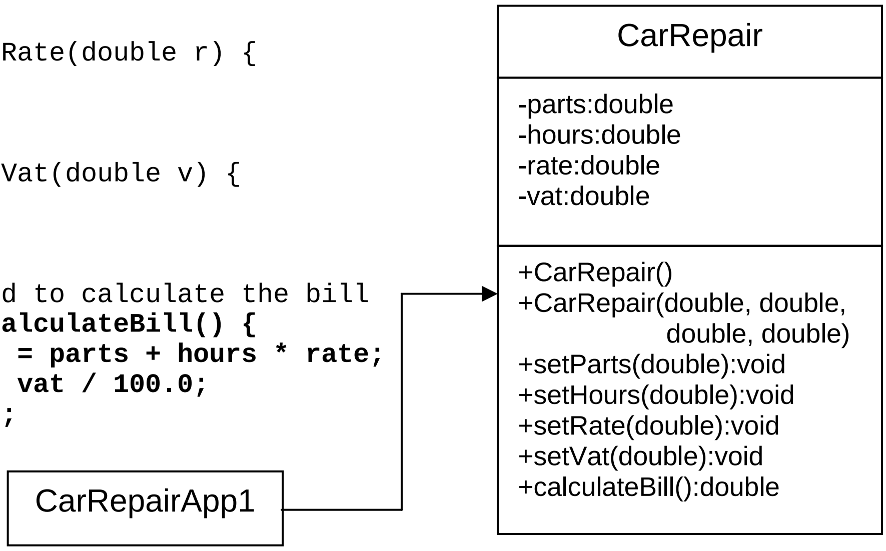
```
// Create a new CarRepair object to calculate the bill
CarRepair c = new CarRepair(parts, hours, rate, vat);
double bill = c.calculateBill();
billTxt.setText(pounds.format(bill));
```

1. When the user clicked the button the actionPerformed method read the four user inputs into double variables parts, hours, rate and vat.
2. The auxiliary class is called CarRepair. At line 68 a constructor of this class is used to create a CarRepair object c.
3. At line 69 the bill is calculated using a method calculateBill() of c.
4. At line 70 the returned bill value is formatted and placed in the output text box.

For a full code listing of CarRepairApp1.java see the ch08_auxiliaryclasses.carrepairbill package. It includes some *exception handling* to stop the program from crashing if the user doesn't type anything, or types non-numeric data, into the input text fields.

Here is a listing of the auxiliary class CarRepair.java and its UML diagram:

```
package ch08_auxiliaryclasses.carrepairbill;

class CarRepair {
    // private instance variables

    private double parts,  hours,  rate,  vat;

    // public no-argument constructor
    public CarRepair() {
    }

    // public constructor with four arguments
    public CarRepair(double p, double h, double r, double v) {
        parts = p;
        hours = h;
        rate = r;
        vat = v;
    }

    // public 'set' methods
    public void setParts(double p) {
        parts = p;
    }

    public void setHours(double h) {
        hours = h;
    }

    public void setRate(double r) {
        rate = r;
    }

    public void setVat(double v) {
        vat = v;
    }

    // public method to calculate the bill
    public double calculateBill() {
        double bill = parts + hours * rate;
        bill *= 1 + vat / 100.0;
        return bill;
    }
}
```



The link joining the `CarRepairApp1` and `CarRepair` classes indicates an *association* between them. `CarRepairApp1` can "see" `CarRepair` but not vice-versa. The lines in bold show the relevant code for this example. But note that there is a second no-argument constructor and four methods to set the values of the instance variables. These are used by a second application class `CarRepairApp2.java` which uses different code in `actionPerformed` to achieve the same effect:

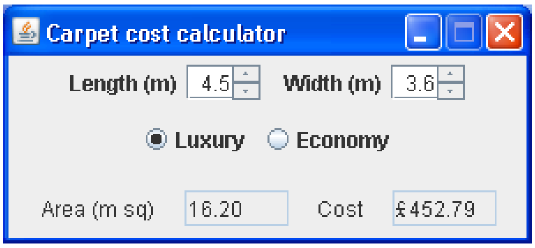
```
        // Create a new CarRepair object to calculate the bill
        // using the no-argument constructor and the set methods
        CarRepair c = new CarRepair();
        c.setParts(parts);
        c.setHours(hours);
        c.setRate(rate);
        c.setVat(vat);
        double bill = c.calculateBill();
        billTxt.setText(pounds.format(bill));
```

Note that this version uses the auxiliary class without that class needing to be recompiled.

Carpet Calculator revisited

Here we have two upgraded versions of the Carpet Calculator. The first is just a GUI version of our previous program. However the second extends the functionality a little by allowing the user to input the room dimensions in metres and centimetres.

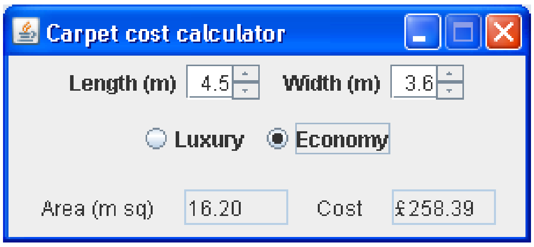
`CarpetCalculatorApp1:`


Carpet cost calculator

Length (m) 4.5 Width (m) 3.6

● Luxury ○ Economy

Area (m sq) 16.20 Cost £452.79

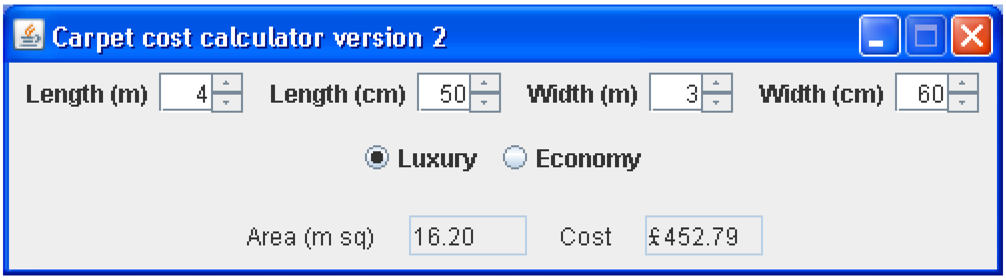

Carpet cost calculator

Length (m) 4.5 Width (m) 3.6

○ Luxury ● Economy

Area (m sq) 16.20 Cost £258.39

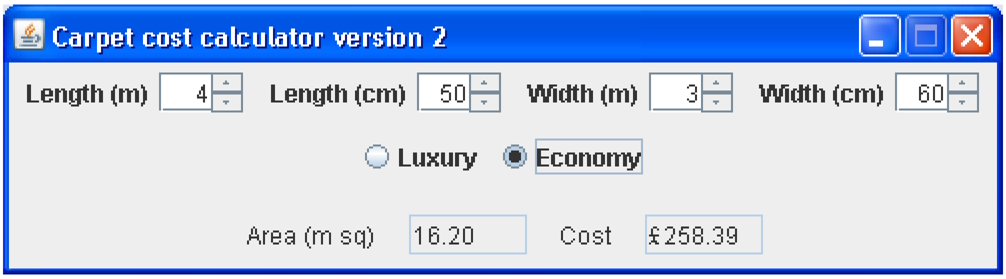
`CarpetCalculatorApp2:`


Carpet cost calculator version 2

Length (m) 4 Length (cm) 50 Width (m) 3 Width (cm) 60

● Luxury ○ Economy

Area (m sq) 16.20 Cost £452.79


Carpet cost calculator version 2

Length (m) 4 Length (cm) 50 Width (m) 3 Width (cm) 60

○ Luxury ● Economy

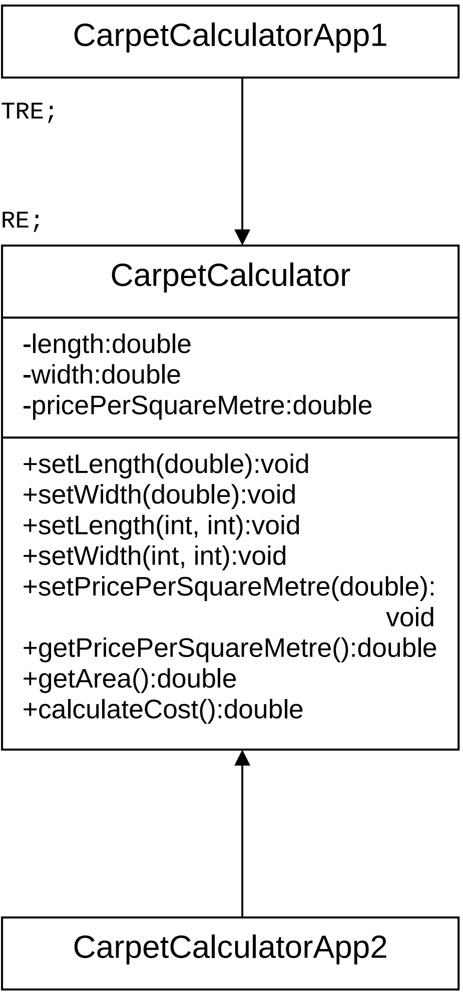
Area (m sq) 16.20 Cost £258.39

Both applications respond to `ItemListener` (for the radio buttons) and `ChangeListener` (for the *spinner* controls) events using the `itemStateChanged` and `stateChanged` event handlers respectively.

In both cases a `CarpetCalculator` object is created at startup and its `pricePerSquareMetre` instance variable initially set to 27.95 (`LUXURY_COST`). The `itemStateChanged` handler resets this variable depending on the state of the radio buttons, and updates the area and cost text fields using the `getArea` and `calculateCost` methods of `CarpetCalculator`.

In each application the `stateChanged` handlers update the `length` and `width` instance variables of the `CarpetCalculator` object using `setLength` and `setWidth` methods. There are two each of these, one with a `double` argument for the length (width) in metres, one with two `int` arguments for metres / centimetres. Having updated these, the area and cost text fields are updated as before.

```
package ch08_auxiliaryclasses.carpetcalculator;

class CarpetCalculator {

    private double length; // metre
    private double width;  // metre
    private double pricePerSquareMetre;

    // some useful constants
    static final double CENTIMETRES_PER_METRE = 100.0;

    // the default constructor is used so we need set methods
    public void setLength(double l) {
        length = l;
    }

    public void setWidth(double w) {
        width = w;
    }

    public void setLength(int metres, int centimetres) {
        length = metres + centimetres / CENTIMETRES_PER_METRE;
    }

    public void setWidth(int metres, int centimetres) {
        width = metres + centimetres / CENTIMETRES_PER_METRE;
    }

    public void setPricePerSquareMetre(double price) {
        pricePerSquareMetre = price;
    }

    // get methods
    public double getPricePerSquareMetre() {
        return pricePerSquareMetre;
    }

    public double getArea() {
        return length * width;
    }

    // method to calculate the cost of the carpet
    public double calculateCost() {
        return pricePerSquareMetre * getArea();
    }
}
```



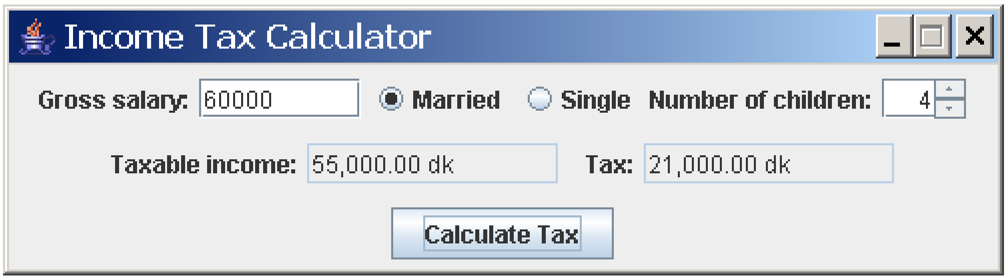
Remember that if the default constructor (with no arguments) is used, it creates an object with a default initial state (here all the instance variables will have value 0.0).

Just as we may have several different constructors, we may also have different methods with the same name. `CarpetCalculatorApp1` uses `setLength(double):void` and `CarpetCalculatorApp2` uses `setLength(int, int):void` (and similarly for the width calculations).

For a full listing of these two applications (they are rather long, having a lot of GUI logic) see the relevant packages in your NetBeans project. They contain details of how to set up and use spinners – these are a bit fiddly but well worth the effort.

TaxCalculator

This is an implementation of a previous tutorial exercise. Separating the GUI application and the tax calculation into different classes is useful here as calculating the tax needs a bit of thought, best uncluttered with Swing widgets.



Income Tax Calculator

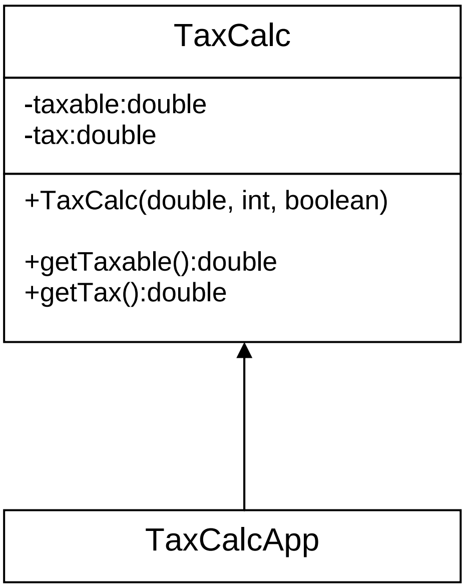
Gross salary: 60000 ● Married ○ Single Number of children: 4

Taxable income: 55,000.00 dk Tax: 21,000.00 dk

The user types their gross salary into the text field, selects Married or Single and spins to their number of children. Just as in the Car Repair example, the sequence of events is:

- The user clicks the button
- `ActionPerformed` reads the input from the controls and uses this to create a `TaxCalc` object `tc` to do the calculation
- the `getTaxable()` and `getTax()` methods of the `TaxCalc` class perform the necessary calculations
- the correct values are formatted and placed on the two non-editable text fields (dk stands for *doshky*)

```
package ch08_auxiliaryclasses.taxcalculator;

class TaxCalc {
    // instance variables

    private double taxable;
    private double tax;

    // constructor, which does the tax calculation
    TaxCalc(double gross, int nKids, boolean married) {
        // calculate taxable income first
        // this is gross - personal allowance - child allowance

        // calculate personal allowance
        double personalAllowance, childAllowance;
        if (married) {
            personalAllowance = 3000;
        } else {
            personalAllowance = 2000;
        }

        // calculate child allowance
        childAllowance = nKids * 500;

        // calculate taxable. If negative, set it to zero
        taxable = gross - personalAllowance - childAllowance;
        if (taxable < 0) {
            taxable = 0;
        }

        // calculate tax. There are three bands:
        //                taxable <= 20,000
        //                tax is 20% of taxable
        //                20,000 < taxable <= 40,000
        //                tax is 20% of first 20,000 +
        //                           40% of remainder
        //                taxable > 40,000
        //                tax is 20% of first 20,000 +
        //                           40% of next 20,000 +
        //                           60% of remainder
        double remainder;
        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;
        }
    } //  end of constructor

    // get methods to return taxable and tax which
    // have been calculated by the constructor
    public double getTaxable() {
        return taxable;
    }

    public double getTax() {
        return tax;
    }
}
```



This is one way of getting the job done – the constructor performs the tax calculations, setting the instance variables `taxable` and `tax` to their correct values (see the comments for an explanation), then the `getTaxable` and `getTax` methods are trivial.

Summary

In this chapter we have revisited the idea set forward in Chapter 4 of using separate classes to do the calculation. This is a good idea when writing GUI applications as the problems involved with getting the GUI sorted out are involved enough without doing the calculations in the same class. The basic idea is that the GUI class creates an object with a constructor of the calculating class, then interrogates it using suitable methods of the same class. We can change the GUI class without needing to change (or re-compile) the calculation class, and vice-versa. This is what Object Oriented programming is all about.

9. Repetition – while, for and do

This chapter shows how to implement loops (repetition) in Java using the **`while`**, **`for`** and **`do`** constructs.

Some of the programs illustrating loops use a JFrame with a canvas. Only the relevant code is shown for these. Others output to System.out.

`while` loops

These are the simplest kind. The first example (**`Stars.java`**) prompts the user for a whole number and prints out that number of stars:

```
01 package ch09_loops;
02
03 import javax.swing.JOptionPane;
04 import static java.lang.System.*;
05
06 class Stars {
07
08     public static void main(String[] args) {
09         String numberStr =
10                 JOptionPane.showInputDialog("how many?");
11         int n = Integer.parseInt(numberStr);
12         int counter = 0;
13         while (counter < n) {
14             out.print("*");
15             counter++;
16         }
17         out.println();
18     }
19 }
```

The loop is on lines 13 – 16. The *body* of the loop between `{` and `}` is executed repeatedly as long as the *condition* `counter < n` remains `true`.

Suppose that the user types in the number 8. In this case variable `n` has value 8 at line 11 and `counter` starts off with the value 0 at line 12. The condition `counter < n` is `true` so the body is executed, an asterisk is printed and 1 is added to `counter` which now has value 1. The condition `counter < n` is still `true` so the body is executed again, another asterisk is printed and 1 added to `counter` which now has value 2. And so on. The body is executed 8 times in all. At the end of the 8th iteration[†], `counter` has value 8 and the condition `counter < n` is `false` so the loop stops iterating and the program continues with the statement after the loop at line 17, which prints a new line.

Self-test exercises

1. What will be output if we run this program with the input 10.
2. What will be output if we run this program with the input 0.

[†] *iteration* is the technical word for looping. We say that the loop body is iterated many times.

The answer to the last exercise is: No asterisks are printed.

The loop body is never executed because the condition `counter < n` is `false` right at the start.

It is *very important* to realise that sometimes the body of a `while` loop is not executed at all. For example, we may write code to process an input text file a line at a time, which would look like this (we cover file handling later on):

```
open the file;
while ( there is still something to read ) {
    read a line from the file;
    process the line;
}
```

so the loop body gets executed once for each line in the file. But maybe the file is empty – i.e. it has no lines at all. (This is possible in Windows and other systems). This code still works as there is nothing to read at the start and the body is never executed, thus avoiding the error of trying to read a line that isn't there.

General format for a `while` loop

When there are multiple statements in the loop body:

```
while (test) {
   statement 1;
   statement 2;
   ...
   statement n;
}
```

Short form if the loop body is a single statement:

```
while (test)
   statement;
```

Most programmers prefer to use `{` and `}` even if there is only one statement in the loop body. NetBeans will insert them if you ask it to format the code.

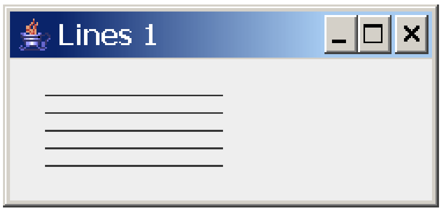
Tip (reminder): Are you remembering to use Alt+Shift+F?

The body is executed repeatedly as long as the test condition remains `true`. If the test is `false` right at the start, the body is not executed at all.

Points to note:

- The test *must* be enclosed in brackets `(` and `)`. There is no semi-colon (like `if`).

There is no closing `end while` or `loop` keyword. When more than one statement is to be executed in the loop body, they must be enclosed by `{` and `}`.



The second example (`Lines1.java`) uses a `JFrame` with a canvas. It's not very exciting. It just draws five lines. There are no controls other than the canvas, and no event handling. The paint method of the canvas runs when the program starts, giving the result shown here.

```
...
26          public void paint(Graphics g) {
27              int n = 0;
28              int x = 20;
29              int y = 20;
30              while (n < 5) {
31                  g.drawLine(x, y, x + 100, y);
32                  y += 10;
33                  n++;
34              }
35          }
...
```

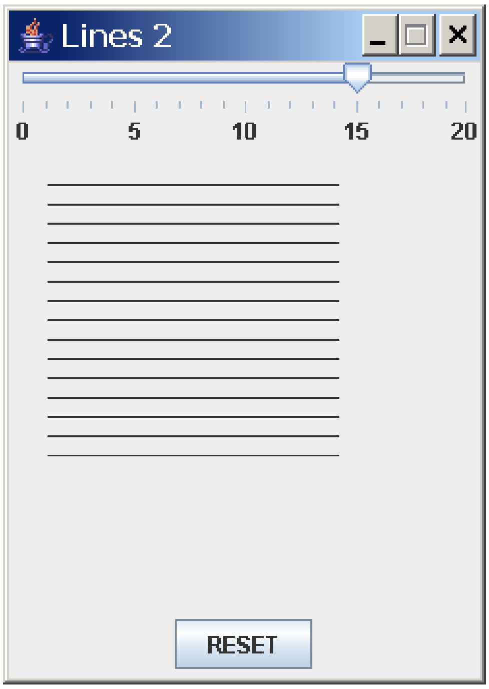
For a full code listing, see the ch09_loops package in your NetBeans project.

The while loop at lines 30 – 34 iterates 5 times with n taking the values 0, 1, 2, 3, 4. Each time around a horizontal line 101 pixels long is drawn, the y pixel positions for these lines being 20 at the start (line 29), then 30, 40, 50, 60 respectively as 10 is added after each line is drawn (line 32).

Self-test exercise

3. What is the value of y at the end of the above paint method?

The third example (Lines2.java) extends this by allowing the user to choose the number of lines to be drawn, using a JSlider control.



The user selects the number of lines (here 15 have been selected) using the slider. The length of each line in pixels is 10 * the number of lines + 1 so the lines roughly make a square.

Each time a new value is selected, the entire canvas is repainted with the chosen number of lines.

The RESET button sets the slider value back to 0 and 0 lines are drawn.

The actionPerfomed method (for the RESET button) and stateChanged method (for the slider) are shown below, along with the nested Canvas class.

actionPeformed simply sets the slider value to 0. This will trigger a ChangeEvent if the slider had a different value, and this in turn calls stateChanged which repaints the canvas.

The MyCanvas class's paint method obtains the number of lines nLines from the slider, and then the while loop draws that number of lines. Note that if nLines = 0 (as it will be after a RESET or when the program is first run) the body of the while loop does not run at all as the condition n < nLines is false at the start.

For a full code listing, see `Lines2.java` in the `ch09_loops` package.

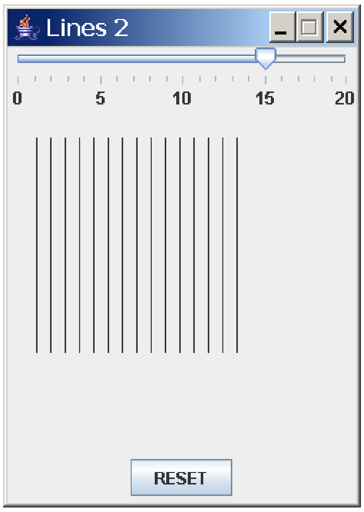
```
        public void paint(Graphics g) {
            int n = 0;
            int x = 20;
            int y = 20;
            while (n < nLines) {
                g.drawLine(x, y, x + nLines * 10, y);
                y += 10;
                n++;
            }
        }
```

The JFrame has instance variables

```
JSlider nLinesSl
   = new JSlider(0, 20, 0);
```

and

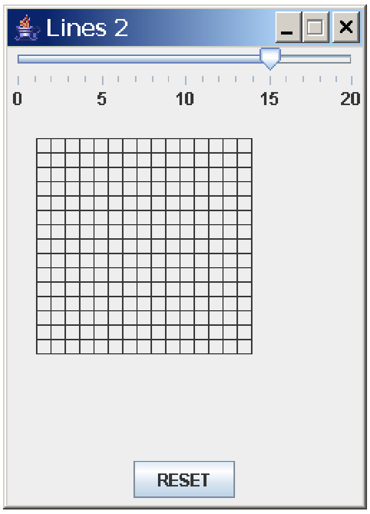
```
MyCanvas canvas
    = new MyCanvas();
```



Self-test exercises

4. What changes are required in the above while loop for the program to display vertical lines instead, as shown on the left?

5. What changes are required to display both vertical and horizontal lines, as shown on the right? (Note that there is one extra horizontal and one extra vertical line!)



A mathematical problem

So far all our loops have been used to step through a sequence of values. Most loops are like this and Java has a special type of loop – the `for` loop – for stepping through a sequence. But before we leave `while` loops, here's a simple sounding problem.

Start with a positive whole number n

1. If n is even divide by 2
2. If n is odd multiply by 3 and add 1
3. Repeat from step 1 until n = 1

For instance if we start with n = 6, n takes the values

6, 3, 10, 5, 16, 8, 4, 2, 1

And if we start with n = 7, n takes the values

7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

As far as anyone knows, no matter what number n you take to start with, this process eventually stops with n = 1.

Here is a Java application (`NumberTest.java`) which accepts input `n > 0` and which calculates and prints out the sequence of numbers given by rules 1 – 3 above

```
package ch09_loops;

import javax.swing.JOptionPane;
import static java.lang.System.*;
```

```
05
06 class NumberTest {
07
08     public static void main(String[] args) {
09         String numberStr =
10                 JOptionPane.showInputDialog("think of a number");
11         int n = Integer.parseInt(numberStr);
12         while (n > 1) {
13             out.print(n + ", ");
14             if (n % 2 == 0) {
15                 n /= 2; // divide by 2
16             } else {
17                 n = 3 * n + 1;
18             }
19         }
20         out.println(1);
21     }
22 }
```

The while loop at lines 12 – 19 implements the given rules. Note that if `n % 2 == 0` then `n` is even, otherwise `n` is odd.

Try running this program with input 27. You get a sequence of 112 numbers, the highest being 9232, eventually ending with 1.

Self-test exercise

6. Write a program like `NumberTest.java` above to implement the following rules:

   a. If n is even divide by 2

   b. If n is a multiple of 3 divide by 3

   c. Otherwise, multiply by 3 and add 1

   In the lab, try running your program with input 31. You should get a sequence of 107 numbers, the highest again being 9232.

`for` loops

`for` loops are best when stepping through a sequence of values. Here is a variant of `Stars.java`, this time using a for loop:

```
package ch09_loops;

import javax.swing.JOptionPane;
import static java.lang.System.*;

class StarsFor {

    public static void main(String[] args) {
        String numberStr =
                JOptionPane.showInputDialog("how many?");
        int n = Integer.parseInt(numberStr);
        for (int counter = 0; counter < n; counter++) {
            out.print("*");
        }
        out.println();
    }
}
```

The `for` loop at lines 12 – 14 does the same job as the `while` loop previously. Variable `counter` takes on values `0, 1, ... n-1` as before.

General format of a `for` loop

```
for (initial statement; test; final statement) {
   statement 1;
   statement 2;
   ...
   statement n;
}
```

The `initial statement` is done once before the loop is started.
It is usually a simple assignment setting a "control variable" to a starting value, e.g. `counter = 0.`

The `test` is carried out prior to any execution of the loop. If it fails, the loop finishes straight away.
It is usually a simple test on the value of the control variable used in the `initial statement`, e.g. `counter < n.`

The `final statement` is carried out just before the end of each repetition of the loop.
It usually just increments or decrements the control variable by + or – 1, e.g. `counter++.`

If the loop body is a single statement, the `{` and `}` brackets may be omitted (but most programmers prefer to keep them).

Tip: Alt+Shift+F (enough said?).

The `for` loop may be regarded as shorthand for the equivalent code using `while`:

```
initial statement;
while (test) {
   statement 1;
   statement 2;
   ...
   statement n;
   final statement;
}
```

Points to note:

- There is no closing `endfor` or `next` keyword. When more than one statement is to be executed in the loop body, they must be enclosed by { and }.
- As most for loops use a single control variable, Java allows the initial statement to declare that variable, as in the `StarsFor` program above.
- The initial and final statements can in fact combine several statements separated by commas, as in:

```
for (int i = 0, y = 20; i < 10; i++, y += 20) {
    System.out.println("i = " + i + ", y = " + y);
}
```

However, most sensible Java programmers like to keep their `for` loops simpler than this.

Self-test exercises

7. What output is produced by the above `for` loop?

8. Rewrite the `while` loops in `Lines1.java` and `Lines2.java` as for loops.

do-while loops

`do-while` loops are a variant of `while` loops where the test comes at the *end* of the loop, not the *beginning*. For instance we might have written

```
12          do {
13              out.print(n + ", ");
14              if (n % 2 == 0) {
15                  n /= 2; // divide by 2
16              } else {
17                  n = 3 * n + 1;
18              }
19          } while (n > 1);
```

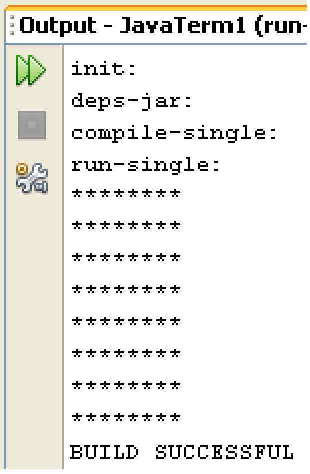
instead of lines 12 – 19 in `NumberTest.java`.

`do-while` loops are mentioned here for the sake of completeness; they aren't used much. The body of the loop will *always* be executed at least once. This can be useful but could lead to errors in situations where there is actually nothing to do.

Nested loops

It is quite common to have a loop within a loop (and occasionally a loop within a loop within a loop, etc). Here is a simple example (`SquareStars.java`):

```
01 package ch09_loops;
02
03 import javax.swing.JOptionPane;
04 import static java.lang.System.*;
05
06 class SquareStars {
07
08     public static void main(String[] args) {
09         String numberStr =
10                 JOptionPane.showInputDialog("how many?");
11         int n = Integer.parseInt(numberStr);
12         for (int i = 0; i < n; i++) {
13             for (int j = 0; j < n; j++) {
14                 out.print("*");
15             }
16             out.println();
17         }
18     }
19 }
```



```
init:
deps-jar:
compile-single:
run-single:
********
********
********
********
********
********
********
********
BUILD SUCCESSFUL
```

The *inner* for loop at lines 13 – 15 is *nested* inside the *outer* for loop at lines 12 – 17. The inner loop runs to completion for each iteration of the outer loop. So the inner loop prints a row of `n` asterisks and the outer loop uses this to print `n` rows of asterisks.

If we replace line 13 with

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

then for the first iteration `i = 0` so a single asterisk is printed, for the second iteration `i = 1` so two asterisks are printed, and so on. The output is now a triangle of asterisks as shown here.

```
*
**
***
****
*****
******
*******
********
```

Self-test exercise

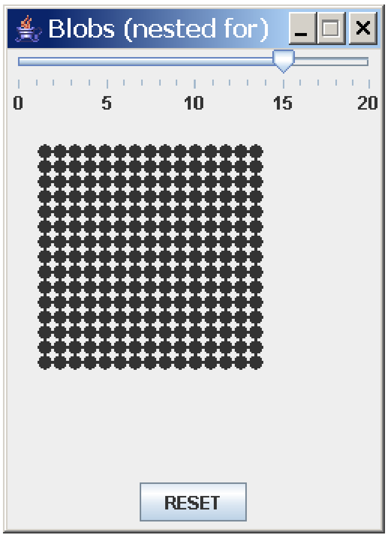
9. What simple change to the original line 13 would produce an upside down triangle as shown here ?

```
********
*******
******
*****
****
***
**
*
```

Blobs

This program (`Blobs.java`) is a variation on the `Lines2.java` program discussed above. It uses nested `for` loops to draw a square grid of blobs (filled circles)

```
53          public void paint(Graphics g) {
54              int x, y;
55              for (int i = 0; i < size; i++) {
56                  for (int j = 0; j < size; j++) {
57                      x = 20 + 10 * i;
58                      y = 20 + 10 * j;
59                      g.fillOval(x, y, 10, 10);
60                  }
61              }
62          }
```



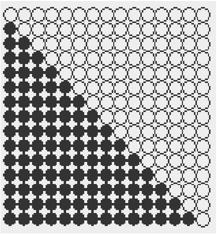
The inner loop at lines 56 – 60 draws a column of filled circles (`size` is an instance variable of the `JFrame` class set by the slider). The outer loop iterates the inner loop `size` times to draw several columns.

We can achieve a nice effect by alternating filled circles with drawn circles simply by replacing line 59 with

```
59                      if ((i + j) % 2 == 0) { // i+j even
60                          g.fillOval(x, y, 10, 10);
61                      } else {
61                          g.drawOval(x, y, 10, 10);
62                      }
```

Self-test exercises

10. What change to line 59 would produce the alternative output shown here?



11. If line 59 was

```
59                      if (i % 2 == 0 || j % 2 == 0) {
```

what would the output look like? What if we replace `||` with &&?

Stopping a loop early

Sometimes we might want to "jump out" of a loop before it has finished. For instance in `NumberTest.java` we might want to jump out as soon as n reaches the value 9232. We do this by using keyword `break` (remember the `switch` statement?)

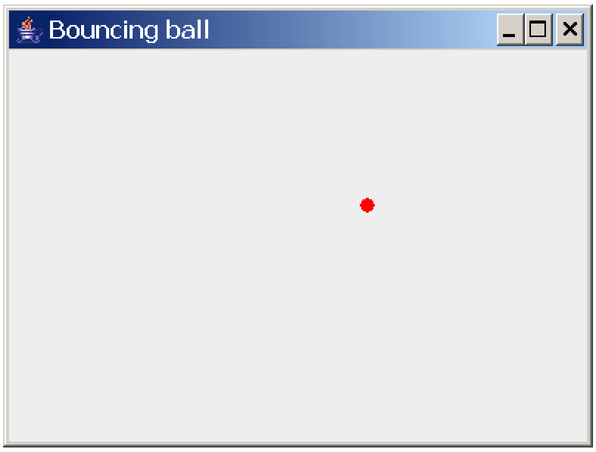
```
12          while (n > 1) {
13              if (n == 9232) {
14                  break;
15              }
16              out.print(n + ", ");
17              if (n % 2 == 0) {
18                  n /= 2; // divide by 2
19              } else {
20                  n = 3 * n + 1;
21              }
22          }
```

Stop the loop and continue execution after line 22

In fact `break;` can be used to "escape" from any block of code. In practice it isn't used much as it can make code hard to understand.

Often you may want to return from a method in the middle of a loop (or perhaps a nested `if` or a `switch`). It is then much simpler to use the `return;` or `return aValue;` statement – which will effectively stop the loop running..

Combining control structures – an animation example



This program (`Bouncer.java`) is a very crude animation of a ball bouncing around a canvas. It serves to illustrate how we can combine loops and tests. See the notes for details

You can write nice animation programs in Java but you really need to use *timers* or be able to understand *threads*.

```
    public void paint(Graphics g) {
        int leftX = 0, topY = 0;
        int rightX = getWidth()-10, bottomY = getHeight()-10;
        int bounces = 0;
        while (bounces < 20) {
            wasteTime();
            Color backgroundColour = getBackground();
            g.setColor(backgroundColour);
            g.fillOval(x, y, diameter, diameter);
            if (x <= leftX) {
                xChange = -xChange;
                bounces++;
```

```
47                 }
48                 if (x >= rightX) {
49                     xChange = -xChange;
50                     bounces++;
51                 }
52                 if (y <= topY) {
53                     yChange = -yChange;
54                     bounces++;
55                 }
56                 if (y >= bottomY) {
57                     yChange = -yChange;
58                     bounces++;
59                 }
60                 x += xChange;
61                 y += yChange;
62                 g.setColor(Color.red);
63                 g.fillOval(x, y, diameter, diameter);
64             }
65         }
66
67         private void wasteTime() {
68             for (int i = 1; i < 10000000; i++);
69         }
```

Variables `x`, `xChange`, `y` and `yChange` are instance variables of the enclosing frame with default values 7, 7, 2, 2 respectively. They determine the position of the ball, and the distance the ball moves for each iteration of the loop. You could add controls to change them.

The `while` loop (lines 39 – 64) counts 20 bounces. It could have been written

```
while (true) {
    ...   ...
}
```

(i.e. an “infinite loop”) with a conditional statement

```
if (!stillGoing) {
    break;
}
```

inside it where `stillGoing` is a `boolean` variable which could be controlled by the user (e.g. with `JButtons` to set it to `true` or `false`).

The `wasteTime()` method is needed to slow the animation down (effectively by doing nothing one million times!) – it would be better to use a timer.

Each of the four `if` statements at lines 44 – 56 detects the ball hitting the edge of the canvas, changes its direction and adds 1 to the count of bounces.

Lines 60 – 63 move and draw the ball.

Lines 41 – 43 wipe the old position of the ball out – try removing them to see why they are needed.

Line 37 determines the extent of the canvas (using the `Canvas` class `getHeight()` and `getWidth()` methods).

For the rest of the program see `Bouncer.java` in the `ch09_loops` package.

Summary

In this chapter we have looked at *repetition* or *looping* in Java, using `while` and `for` loops.

We looked briefly at `do-while` loops, but these aren't used much.

We have seen several examples of the use of loops, including nested loops (loops inside loops) and the use of selection (`if`) constructs in loops.

Solutions to self-test exercises

1. **********

2. No output

3. y starts at 20, and each time around the loop 10 is added. The loop iterates 5 times so the final value is 70. The last value isn't used in the loop.

4. Replace lines 58 – 59 with

```
58          g.drawLine(x, y, x, y + nLines * 10);
59          x += 10
```

5. Use two loops, one for horizontal lines and one for vertical lines. Same as in the original code and qe 4. above, BUT use <= instead of < in the tests as one extra vertical and horizontal line are required.

6. Replace the loop with

```
12          while (n > 1) {
13              out.print(n + ", ");
14              if (n % 2 == 0) {
15                  n /= 2;
16              } else if (n % 3 == 0) {
17                  n /= 3;
18              } else {
19                  n = 3 * n + 1;
20              }
21          }
```

7.

```
i = 0, y = 20
i = 1, y = 40
...   ...
i = 9, y = 220
```

Don't write such loops. They are much too clever and hard to understand.

8. For instance the loop in answer 4 becomes

```
56              for (n = 0; n < nLines; n++) {
57                  g.drawLine(x, y, x, y + nLines * 10);
58                  x += 10;
59              }
```

It is a common mistake to increment the counter inside the loop as well as in the `for` bit. For instance if we left in line 60 (`n++;`) then `n` would go up in steps of 2!

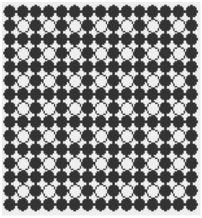
9. Replace with

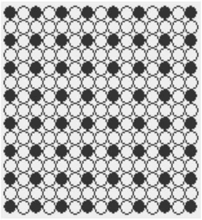
```
13              for (int j = i; j < n; j++) {
```

10. Replace with

```
59                     if (i < j)
```

11. It would look like this (||) or this (&&). Pretty, aren't they ?





10. Arrays

You may have seen arrays in other languages. In Java arrays are *objects* created by an API class `Array` and they are declared like this:

```
int[] ages;
String[] band;
JButton[] digit;
```

Arrays can hold values of primitive types (`int`, `float` etc) or of any object type such as `String` or `JButton`. Before use, arrays must be *instantiated* like any other type of object, e.g.:

```
ages = new int[6];
band = new String[4];
digit = new JButton[10];
```

Array indices in Java start at 0 and finish at one less than the declared length. So after the above statements we have six `int`s

```
ages[0], ages[1], ... ages[5]
```

four `String`s

```
band[0], band[1], band[2], band[3]
```

and ten `JButton`s

```
digit[0], digit[1], ... digit[9]
```

The length of an array is obtained via its `length` attribute, thus `ages.length` is 6.

> *It is important to note that, for arrays of objects, the elements have to be instantiated separately – they will start off with a null value.*

For small arrays of primitive types or strings, there is a convenient shorthand notation:

```
int[] ages = {23, 64, 96, 13, 7, 32};
```

is equivalent to

```
int[] ages = new int[6];
ages[0] = 23;
ages[1] = 64;
ages[2] = 96;
ages[3] = 13;
ages[4] = 7;
ages[5] = 32;
```

Quick Exercise

1. Suppose we have

   ```
   String band[] = {"John", "Paul", "George", "Ringo"};
   ```

   What is the equivalent code? What is `band[2]`? What is `band.length`?

Arrays are normally processed with `for` loops. In fact loops and arrays almost always go together – arrays contain similar types of data and loops are used to repeatedly process that data.

For example, to set all the ages to `0`:

```
10          for (int i = 0; i < ages.length; i++) {
11              ages[i] = 0;
12          }
```

As the index variable `i` changes, a different array element is being referred to. For instance when `i` has the value 3 the statement `ages[i] = 0;` sets `ages[3]` to 0.

Note the use of `ages.length` here. Much better than using `6` – if you change the declaration to have 10 ages, this line of code doesn't need to change as well.

Arrays1.java

This program demonstrates simple array processing

```
01 package ch10_arrays;
02
03 import static java.lang.System.*;
04
05 class Arrays1 {
06
07     public static void main(String[] args) {
08         String[] band = {"John", "Paul", "George", "Ringo"};
09         out.println("band array:");
10         for (int i = 0; i < band.length; i++) {
11             out.println(band[i]);
12         }
13
14         String s = band[0];
15         for (int i = 0; i < band.length - 1; i++) {
16             band[i] = band[i + 1];
17         }
18         band[band.length - 1] = s;
19
20         out.println();
21         out.println("new contents of band array:");
22         for (int i = 0; i < band.length; i++) {
23             out.println(band[i]);
24         }
25     }
26 }
```

If you compile and run this program the output is:

```
band array:
John
Paul
George
Ringo
```

Output produced by the `for` loop at lines 10 – 12

```
new contents of band array:
Paul
George
Ringo
John
```

Output produced by the `for` loop at lines 22 – 24

The contents of the array apart from the 0th element have all been shifted down one place and the 0th element moved to the end. (This is called a *left circular shift*). How does this work?

At the start array `band` looks like this:

band:

John	Paul	George	Ringo

After the statement at line 10 we have the following state:

band:

John	Paul	George	Ringo

s:

John

the 0th element having been copied to the `String` variable `s`.
The `for` loop at line 11 steps through the array *except for the last element* overwriting the current element with the next one. It is as if we had written

```
band[0] = band[1];
band[1] = band[2];
band[2] = band[3];
```

and the state now looks like this:

band:

Paul	George	Ringo	Ringo

s:

John

Finally the statement at line 12 copies the 0th element saved in `s` to the last element of the array:

band:

Paul	George	Ringo	John

s:

John

Quick Exercise

2. Suppose we had a different (rather obscure) band of musicians at line 08:

```
String[] band = {"Dave Dee", "Dozy", "Beaky", "Mick", "Tich"};
```

What would the output be for this band?

ArrayIndexOutOfBounds exception

It is very common when working with arrays to make the mistake of stepping outside the bounds of the array. For instance if lines 15 – 17 had been written

```
15          for (int i = 0; i < band.length - 1; i++) {
16              band[i] = band[i + 1];
17          }
```

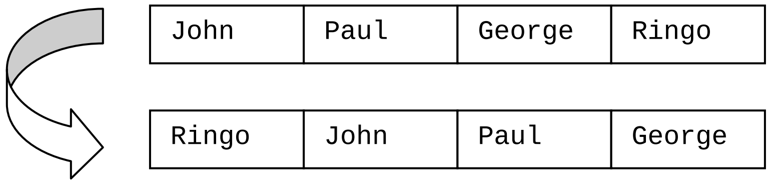
then the last time around the loop `i` would have the value 3 and `band[i+1]` would refer to the non-existent element `band[4]`. Whenever this happens the Java run-time system “throws” an `ArrayIndexOutOfBounds` exception and the program terminates:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
    at Arrays1.main(Arrays1.java:11)
```

(You can in fact “trap” this and many other types of exception by using a `try ... catch` block, as we shall see in the next workbook.)

Quick Exercises

3. The above program performs a left circular shift on the `band` array. A *right circular shift* moves all but the last element up one place, and moves the last element to the beginning, e.g.



John	Paul	George	Ringo

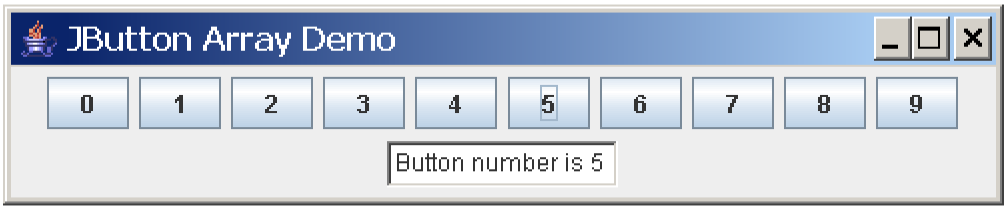
Ringo	John	Paul	George

Rewrite lines 15 – 17 to perform a right circular shift instead.

Arrays of controls

Arrays may contain any type of object. For instance it is often convenient to have arrays of Swing objects such as `JButton`s, `JTextField`s, etc. Here are two similar examples using arrays of `JButton`s.

JButtons.java



Button number is 5

When the user clicks a button the event handler determines which one it is by reading its caption.

```
package ch10_arrays;

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

public class JButtons extends JFrame implements ActionListener {

    private TextField message = new TextField(13);

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

    public JButtons() {
        setLayout(new BorderLayout());
        setSize(500, 100);
        JPanel top = new JPanel();
        JButton[] digit = new JButton[10];
        for (int b = 0; b < 10; b++) {
            digit[b] = new JButton("" + b);
            top.add(digit[b]);
            digit[b].addActionListener(this);
        }
        add("North", top);
        JPanel bottom = new JPanel();
```

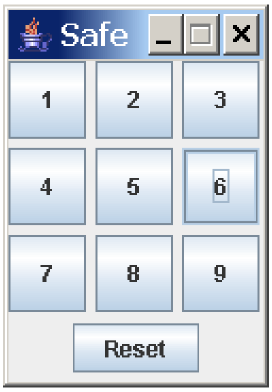
This instantiates the `JButton` array to have length 10.

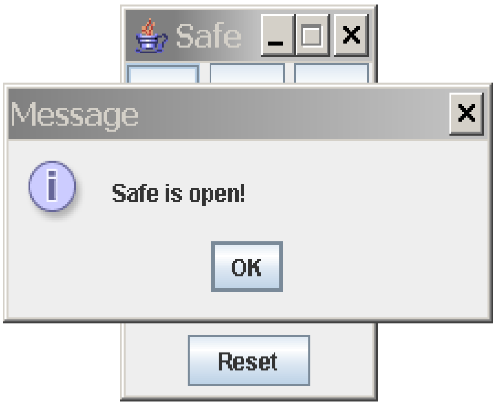
Each `JButton` in the array must be instantiated separately.

```
        bottom.add(message);
        add("South", bottom);
        setTitle("JButton Array Demo");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
      message.setText("Button number is " + e.getActionCommand());
    }
}
```

getActionCommand() returns the text on the JButton causing the event

Safe.java





Safe is open!

The user tries to open the safe by keying in the combination using the buttons labelled 1 to 9. If she guesses the correct combination 634921 a message box pops up saying the safe is open. The Reset button resets the safe to its initial state:

```
package ch10_arrays;

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

public class Safe extends JFrame implements ActionListener {

    static final int combination = 634921;
    int guess = 0;
    JButton reset = new JButton("Reset");

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

    public Safe() {
        setLayout(new BorderLayout());
        setSize(222, 303);
        JPanel middle = new JPanel();
        middle.setLayout(new GridLayout(3, 3, 5, 5));
        JButton[] digit = new JButton[10];
        // ignore 0th JButton
        for (int b = 1; b < 10; b++) {
            digit[b] = new JButton("" + b);
            middle.add(digit[b]);
            digit[b].addActionListener(this);
        }
        add("Center", middle);
```

Remember that this creates a 3 by 3 grid with a 5 pixel gap between grid rows and columns.

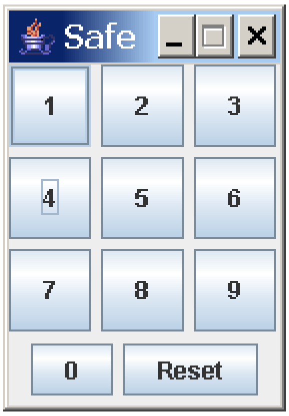
The 0th JButton isn't used. This loop adds the JButtons labelled 1 to 9 to the grid

```
30          JPanel bottom = new JPanel();
31          bottom.add(reset);
32          reset.addActionListener(this);
33          add("South", bottom);
34          setTitle("Safe");
35          setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
36          //setResizable(false);
37          setVisible(true);
38      }
39
40      public void actionPerformed(ActionEvent e) {
41          if (e.getSource() == reset) {
42              guess = 0;
43          } else {
44              guess = guess * 10 +
45                      Integer.parseInt(e.getActionCommand());
46          }
47          if (guess == combination) {
48              JOptionPane.showMessageDialog(this, "Safe is open!");
49          }
50      }
51 }
```

To see how this works, suppose the user enters the correct combination by clicking the buttons labelled 6, 3, 4, 9, 2, 1 in turn. At the start guess is 0. After clicking the 6 button `guess` is multiplied by 10 (at line 44) and 6 (the value of the expression at line 41) is added, so `guess` is now 6. After the 3 button is clicked `guess` is multiplied by 10 and 3 is added, so `guess` is now 63. This carries on with `guess` taking successive values 634, 6349, 63492, 634921 at which stage the conditional statement at lines 42, 43 executes, showing the message box.

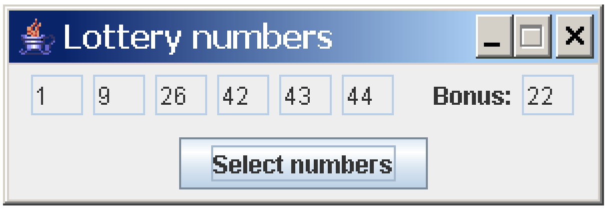
Quick Exercise

4. There is no 0 button in the Safe application. What changes are required to add a 0 button as shown, with the combination altered to 406827?



Here is a more substantial example, using an array of text fields, to generate UK lottery numbers. (We shall develop several versions of this application.)

Lottery.java



1 9 26 42 43 44 Bonus: 22

When this application runs it generates seven random numbers, all different, between 1 and 49. Six are displayed in a `JTextField` array, sorted in ascending order, and the seventh (the bonus ball) in a separate `JTextField`. Each time the user clicks the Select numbers button a fresh set of lottery numbers is generated and displayed.

As the generation and sorting of the lottery numbers is quite complicated, this process is done by an auxiliary class `LotteryNumbers.java`. Here is the GUI class:

```
package ch10_arrays.lottery1;

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

public class Lottery extends JFrame implements ActionListener {

    private JTextField[] txtNumber = new JTextField[6]; // numbers
    private JTextField txtBonus; // bonus ball
    private JButton select;
    private LotteryNumbers nums;

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

    Lottery() {
        setLayout(new BorderLayout());
        JPanel top = new JPanel();
        for (int i = 0; i < txtNumber.length; i++) {
            txtNumber[i] = new JTextField(2);
            top.add(txtNumber[i]);
            txtNumber[i].setEditable(false);
        }
        top.add(new JLabel("     Bonus:"));
        txtBonus = new JTextField(2);
        top.add(txtBonus);
        txtBonus.setEditable(false);
        add("North", top);
        JPanel bottom = new JPanel();
        select = new JButton("Select numbers");
        bottom.add(select);
        select.addActionListener(this);
        add("South", bottom);
        setTitle("Lottery numbers");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 100);
        setResizable(false);
        setVisible(true);
        // create a LotteryNumbers object and get
        // the first set of numbers and bonus ball
        nums = new LotteryNumbers(txtNumber.length);
        update(nums.getNumbers(), nums.getBonus());
    }

    // get a new set of numbers and bonus ball
    public void actionPerformed(ActionEvent e) {
        nums.generate();
        update(nums.getNumbers(), nums.getBonus());
    }

    // update the text fields
    private void update(int[] number, int bonus) {
        for (int i = 0; i < number.length; i++) {
            txtNumber[i].setText("" + number[i]);
        }
        txtBonus.setText("" + bonus);
    }
}
```

Most of this code is standard GUI stuff. The interesting code is highlighted in bold.

At line 12 a `LotteryNumbers` object `nums` is declared.

At line 43 `nums` is instantiated, using a constructor specifying the number of lottery balls required. By writing `txtNumber.length` here instead of 6 we make the application more general. The only place where we need to indicate that there are six lottery numbers is at line 09.

Lines 54 – 59 show a private method to update the text fields. This takes an `int` array of numbers (the six lottery numbers) and a separate `int` (the bonus number) and uses these to set the texts on the `txtNumber` array and `txtBonus`.

This method is called as the application is loaded, at line 44 in the `Lottery` class constructor. The arguments to this call are `nums.getNumbers()` and `nums.getBonus()`.

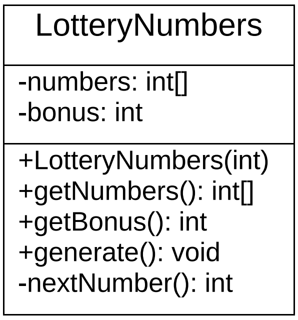
We can deduce from this that the `LotteryNumbers` class must have public methods `getNumbers()` to return an `int` array of lottery numbers, and `getBonus()` to return a single `int` lottery number.

It is worth noting that the value returned by a function method can be an array object. This is very useful and the Java class libraries contain many methods which return arrays.

As these methods are called at line 44 just after `nums` has been instantiated, we can also deduce that the `LotteryNumbers` constructor must generate the first set of numbers.

The `actionPerformed` method at lines 48 – 51, called whenever the `select` button is clicked, tells us that the `LotteryNumbers` class must also have a public method `generate()` to generate a new set of numbers (line 49). Line 50 calls the private `update` method again to refresh the text fields.

By designing and writing the GUI class first, we have a clear idea of what the auxiliary class must do. Here is its UML class diagram:



LotteryNumbers

-numbers: int[]
-bonus: int

+LotteryNumbers(int)
+getNumbers(): int[]
+getBonus(): int
+generate(): void
-nextNumber(): int

The `LotteryNumbers` class clearly needs `private` instance variables to hold the lottery numbers and the bonus number.

As we can see from the GUI code the `LotteryNumbers` class needs a constructor with a single argument specifying the number of lottery numbers, two `get` methods to return the values of the instance variables, and a void `generate()` method.

We have also included a private int method nextNumber() to help generate the lottery numbers at random.

We may need another method to sort the lottery numbers into ascending order, but let's leave this for now.

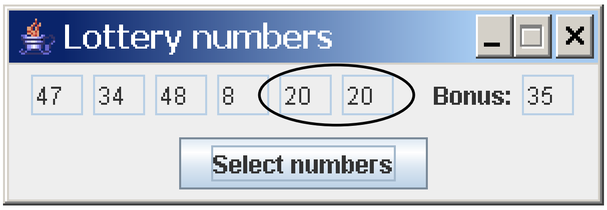
Here is the first attempt at writing the auxiliary class:

LotteryNumbers.java (version 1)

```
01 package ch10_arrays.lottery1;
02
03 class LotteryNumbers {
04
05     private int[] numbers;
06     private int bonus;
07
08     LotteryNumbers(int size) {
09         numbers = new int[size];
10         generate();
11     }
12
13     public int[] getNumbers() {
14         return numbers;
15     }
16
17     public int getBonus() {
18         return bonus;
19     }
20
21     // generate lottery numbers and bonus ball
22     // first attempt
23     public void generate() {
24         // generate random numbers between 1 and 49
25         for (int i = 0; i < numbers.length; i++) {
26             numbers[i] = nextNumber();
27         }
28         // generate the bonus ball
29         bonus = nextNumber();
30     }
31
32     // returns a random int between 1 and 49
33     private int nextNumber() {
34         int number;
35         number = (int) (Math.random() * 49) + 1;
36         return number;
37     }
38 }
```

At line 35 we use the library function Math.random(). This generates a random double between 0.0 and 1.0, but never equal to 1.0. Multiplying by 49 and using the (int) cast gives a random int between 0 and 48, so adding 1 gives a random int between 1 and 49 which is what we want for a random lottery number.

If we run the application, it should generate seven random numbers between 1 and 49 and display them, but without sorting the six lottery numbers. However there is a nasty bug, as shown here:


Lottery numbers

47 34 48 8 20 20 Bonus: 35

The same number 20 has been chosen twice! This was noticed only after clicking the `select` button a number of times. Dealing with random numbers is always tricky as programs behave differently each time they are run.

We need a way of remembering the numbers we have already generated so that we can reject them if they come up twice. Clearly this has nothing to do with the GUI class, just with the `LotteryNumbers` class. This is one major advantage of using auxiliary classes – we can focus our attention exactly where it is needed.

One way to fix this bug is to use a `boolean` array `chosen` (say). Each time a new number is chosen we can remember this by setting the corresponding element of this array to `true`. So if we have already chosen 23, 14 and 35 the array will look like this:

0	1	...	13	14	15	...	22	23	24	...	34	35	36	...	48	49
✗	✗	...	✗	✓	✗	...	✗	✓	✗	...	✗	✓	✗	...	✗	✗

Here we have used ✗ to represent `false` and ✓ to represent `true`.

Here is the revised version of the auxiliary class:

LotteryNumbers.java (version 2)

Only the relevant code and revised class diagram are shown here.

```
package ch10_arrays.lottery2;

class LotteryNumbers {
    ...
    public void generate() {
        // set up a boolean array to remember the numbers chosen
        // initially all values will be false
        boolean[] chosen = new boolean[50];
        for (int i = 0; i < numbers.length; i++) {
            numbers[i] = nextNumber(chosen);
        }
        // generate the bonus ball
        bonus = nextNumber(chosen);
    }

    // returns a random int between 1 and 49
    private int nextNumber(boolean[] chosen) {
        int number;
        do {
            number = (int) (Math.random() * 49) + 1;
        } while (chosen[number]);
        chosen[number] = true;
        return number;
    }
}
```

At line 29 we declare the local `boolean` array `chosen`. Initially all its elements will have value `false`.
At lines 29 and 32 we create the lottery numbers as before, but this time private method `nextNumber` takes this `boolean` array as an argument.

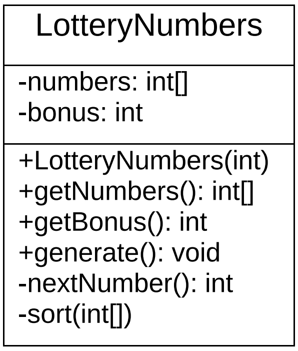
The key code is lines 36 – 43 in the method definition. The `do ... while` loop repeatedly generates a random `int` between 1 and 49 until a number is found that hasn't already been generated (i.e. for which `chosen[number]` is `false`). Line 41 then "remembers" that this number has been chosen for the next time the method is called.

Now we need to sort the six lottery numbers into ascending order. Again this only needs a modification to the auxiliary class, which now has an extra private method `sort`:

LotteryNumbers.java (version 3)

Only the relevant code and revised class diagram are shown here.

```
01 package ch10_arrays.lottery3;
02
03 class LotteryNumbers {
...   ...
23    public void generate() {
...   ...
33        sort(numbers);
34    }
35
...   ...
46    // sort int array a using bubblesort
47    private void sort(int[] a) {
48        int t;
49        for (int i = a.length - 1; i > 0; i--){
50            for (int j = 0; j < i; j++) {
51                if (a[j] > a[j + 1]) {
52                    t = a[j];
53                    a[j] = a[j + 1];
54                    a[j + 1] = t;
55                }
56            }
57        }
58    }
59 }
```



The sort method used here, called Bubble sort, deserves some comment. It works with a number of passes of the array. Each pass (`for` loop at lines 49 – 57) compares successive pairs of adjacent values `a[j]` and `a[j + 1]`, swapping them (lines 52 – 54) if they are in the wrong order. In this way the first pass "bubbles" the largest number to the end of the array, the second pass "bubbles" the next largest number to the position before this, and so on. Note that it will work for an array of any length.

It's worth going through the bubble sort method in detail to convince yourself that it really does work. Let us suppose that the generated lottery numbers are

22, 18, 35, 47, 14, 28

The outer loop (line 49) causes `i` to take on successive values 5, 4, 3, 2, 1 as `a.length` is 6.

The inner loop (line 50) causes `j` to take successive values from 0 to `i - 1`.

For each value of `j` in the table below, the adjacent pair `a[j], a[j + 1]` have been highlighted. If they were swapped as a result of being out of order a ✓ has been put in the Swap? column, otherwise a ✗ has been put there.

i	j	Swap?	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	
			22	18	35	47	14	28	
5	0	✓	**18**	**22**	35	47	14	28	
5	1	✗	18	**22**	**35**	47	14	28	
5	2	✗	18	22	**35**	**47**	14	28	
5	3	✓	18	22	35	**14**	**47**	28	
5	4	✓	18	22	35	14	**28**	**47**	End of 1st pass
4	0	✗	**18**	**22**	35	14	28	47	
4	1	✗	18	**22**	**35**	14	28	47	
4	2	✓	18	22	**14**	**35**	28	47	
4	3	✓	18	22	14	**28**	**35**	47	End of 2nd pass
3	0	✗	**18**	**22**	14	28	35	47	
3	1	✓	18	**14**	**22**	28	35	47	
3	2	✗	18	14	**22**	**28**	35	47	End of 3rd pass
2	0	✓	**14**	**18**	22	28	35	47	
2	1	✗	14	**18**	**22**	28	35	47	End of 4th pass
1	0	✗	**14**	**18**	22	28	35	47	End of 5th pass

Quick Exercises

5. The Ruritanian lottery consists of just four numbers (from 1 to 49) plus a bonus ball. What change(s) is (are) required to `Lottery.java` and / or `LotteryNumbers.java` to satisfy the requirements of the Ruritanian lottery?

6. Convince yourself that the three statements `t = x; x = y; y = t;` actually do swap the values of `x` and `y` by completing the following trace table:

	x	y	t
	4	2	
`t = x;`			
`x = y;`			
`y = t;`			

Bubble sort is just about the simplest, and least efficient, sort available. More complex and efficient sorts are used in spreadsheet and database applications, etc. Actually, with a modern language like Java, programmers seldom needs to design and implement their own sort method, there's usually a good one available in the API library.

So why re-invent the wheel[†]? Java library class `Arrays` contains a number of utility methods for processing arrays, including a variety of sort algorithms. We could delete the bubble sort method (lines 46 – 58) and instead replace line 33 with

```
33        java.util.Arrays.sort(numbers);
```

On the other hand it's good programming practice getting to grips with such algorithms!

Arrays with more than one dimension

This is beyond the scope of our introductory course, but you might be interested in a very brief introduction. If we declare

```
int [][] table = {{1, 2, 3}, {10, 20, 30}};
```

we have a structure like this:

1	2	3
10	20	30

and we refer to `table[0][1]` which is 2, `table[1][2]` which is 30, and so on. Note the peculiar syntax. Such arrays can be very useful – they are typically processed using nested `for` loops – and in Java you can manipulate 2D arrays of buttons, for example. For some examples, see the Puzzle application from section 6 and the TicTacToe (noughts and crosses) application from this section on the web site.

Not So Quick Exercises

7. (Finding the largest and smallest elements of an array of numbers)

   The following method returns the largest (maximum) element of a `double` array:

```
public static double max(double[] a) {
    double largest = Double.NEGATIVE_INFINITY;
    for (int i = 0; i < a.length; i++)
        if (a[i] > largest) largest = a[i];
    return largest;
}
```

[†] You might not know that such a library class / method exists. Or it might take you longer to find out about it than to simply sit down and write your own. But it's more professional to use library code wherever possible to reduce the risk of bugs in your code.

Write a similar method (called `min`) to return the smallest (minimum) element of a `double` array.

N.B. `Double.NEGATIVE_INFINITY` is a notional value for -∞ and `Double.POSITIVE_INFINITY` is a notional value for +∞. For any actual `double` value `d` it is always the case that

```
Double.NEGATIVE_INFINITY < d < Double.POSITIVE_INFINITY
```

If the array is empty (i.e. if its length is 0) it makes mathematical sense to return these values for the maximum and minimum respectively.

8. (Finding the sum and average of an array of numbers)

   Write `public static` methods `sum` and `average` to find the sum and average of a `double` array. If the array has length 0 the sum is 0 and the average is `Double.NaN` which is a notional value for “not a number”. (N.B. you get this value if you try to divide 0 by 0.)

9. (Reversing an array of `String`s)

   Write a `public static String[]` method `reverse` to reverse an array of `String`s. So if we have

   ```
   String band[] = {"John", "Paul", "George", "Ringo"};
   String dnab[] = reverse(band);
   ```

   then `dnab[0]` is `"Ringo"`, `dnab[1]` is `"George"`, `dnab[2]` is `"Paul"` and `dnab[3]` is `"John"`.

Summary

- Arrays in Java are objects holding a collection of data. The whole structure has a single name. All the items in an array are of the same type. Individual components are referred to by means of an index as in `table[2]` or `data[n]` and so on.
- The length of an array is fixed once the array has been created, and can be found by referring to `arrayName.length`.
- The lower index of an array is always `0` and the upper index `length – 1`. Any attempt to go beyond these bounds will result in a run-time error.
- An array may be declared and instantiated in one go like this:

  ```
  int[] harry = new int[24];
  ```
- It is a common mistake to declare and instantiate an array of objects (such as `JButtons`) but to forget that each individual object must be instantiated separately. (See for example lines 16 and 18 of `JButtons.java` above)
- It is common practice to use `for` loops to process arrays.

Solutions to quick and not so quick exercises

1. The equivalent code is

```
String[] band = new String[4];
band[0] = "John";
band[1] = "Paul";
band[2] = "George";
band[3] = "Ringo";
```

`band[2]` is `"George"` and `band.length` is `4`.

2. For this rather obscure 60's Merseybeat group the output would be

```
band array:
Dave Dee
Dozy
Beaky
Mick
Tich

new contents of band array:
Dozy
Beaky
Mick
Tich
Dave Dee
```

3. We need to work from the *end* of the array to the *beginning*:

```
String s = band[band.length–1];
for (int i = band.length-1; i > 0; i--) {
    band[i] = band[i–1];
}
band[0] = s;
```

so for the Beatles this becomes in effect

```
String s = band[3];
band[3] = band[2];
band[2] = band[1];
band[1] = band[0];
band[0] = s;
```

4. Line 09 should be

```
09    static final int combination = 406827;
```

and we need to add three lines to the constructor:

```
31        digit[0] = new JButton("0");
32        bottom.add(digit[0]);
33        digit[0].addActionListener(this);
```

Nothing else is required.

5. All you need do is replace 6 with 4 in line 06 of `Lottery.java`!

```
private JTextField[] txtNumber = new JTextField[4]; // numbers
```

6. The complete trace table is

	x	y	t
	4	2	
`t = x;`	4	2	**4**
`x = y;`	**2**	2	4
`y = t;`	2	**4**	4

7.
```
public static double min(double[] a) {
    double smallest = Double.POSITIVE_INFINITY;
    for (int i = 0; i < a.length; i++)
        if (a[i] < smallest) smallest = a[i];
    return smallest;
}
```

8.
```
public static double sum(double[] a) {
    double total = 0;
    for (int i = 0; i < a.length; i++) total += a[i];
    return total;
}

public static double average(double[] a) {
    double total = 0;
    for (int i = 0; i < a.length; i++) total += a[i];
    return total / a.length; // may return NaN
}
```

or, more briefly

```
public static double average(double[] a) {
    return sum(a) / a.length; // may return NaN
}
```

9.
```
public static String[] reverse(String[] a) {
    String[] answer = new String[a.length];
    for (int i = 0; i < a.length; i++)
        answer[i] = a[a.length - i - 1];
    return answer;
}
```

If we write this as a `void` method to reverse the argument string directly we need to swap the last with the first, the one from last with the second, etc. The code is a little more fiddly – we only go halfway through the array:

```
public static void reverse1(String[] a) {
    String swap;
    for (int i = 0; i < a.length/2; i++) {
        int j = a.length - i - 1;
        swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }
}
```