Getting started with Java

Barry Cornelius
Computing Services, University of Oxford
Date: last updated 5th June 2004; first created in 1996
http://users.ox.ac.uk/~barry/papers/
mailto:barry.cornelius@oucs.ox.ac.uk

1 Introduction

1.1 What is Java?

- Java is an object-oriented programming language developed by Sun Microsystems.
- It has: garbage collection, multithreading and exception handling.
- It does not have: architecture-dependent constructs, structs, unions, pointer arithmetic, operator-overloading and multiple inheritance.
- It is accompanied by many *packages* (each of which is a group of related classes). Each package forms an *Application Programming Interface* or *API*. One of the key packages is one for building GUIs (graphical user interfaces).
- The definition of the Java language and its accompanying APIs is known as the *Java 2 Platform*. The current version is Version 1.4.2.

1.2 How is it executed?

- Unlike most programming languages, Java source code is not compiled into native code. Instead, a Java compiler translates Java source code into an architecturally-neutral intermediate form known as *bytecode*.
- Instructions in this bytecode are interpreted by a Java interpreter.
- Sun provide a Software Development Kit for the Java 2 Platform (Java 2 SDK). This includes a compiler and an interpreter. Previous versions of the SDK were known as the *Java Development Kit* (JDK).
- The Java 2 SDK was first released in December 1998. The latest version is 1.4.2. However, some people still use versions of the JDK such as JDK 1.1.x and JDK 1.0.2 (which dates back to 1996). Whereever possible, new systems should be written using the latest version of the Java 2 SDK.
- Sun provide implementations of the Java 2 SDK for Solaris 2.x, for Windows and for Linux. All of these products can be downloaded (free of charge) from: http://java.sun.com/products/.
- Other companies provide rival products to Sun's SDK.
- Programs written for an early version of Java can be compiled by a more recent compiler. Bytecodes produced by an old compiler can be interpreted by a more recent interpreter.

1.3 What are Java applications?

- A Java application is a conventional program. It must have a 'method' (i.e., a function) called `main`.
- Suppose the file `HWTion.java` contains the Java application:

```
0001: public class HWTion {                                          // HWTion.java
0002:    public static void main(String[ ] args) {
0003:       System.out.println("Hello World!");
0004:    }
0005: }
```

Note: the line numbers should be ignored: they do not form part of the program.

- The application can be compiled by using the Unix/MS-DOS command:

  ```
  javac HWTion.java
  ```

  This produces a file of bytecodes in the file `HWTion.class`.

- The file `HWTion.class` can be interpreted (i.e., executed) by using the command:

  ```
  java HWTion
  ```

- At the University of Durham, the commands `javac` and `java` currently run the compiler and interpreter of the Java 2 SDK v 1.4.1-01. All the commands of JDK 1.1.x can be accessed by following a command name with a minus sign, and those of JDK 1.0.2 can be accessed by using two minus signs. Here are some examples:

  ```
  javac-- HWTion.java
  java-- HWTion
  ```

1.4 What are Java applets?

- A Java applet is Java source code whose bytecodes will be executed as part of viewing a WWW page. The applet's author compiles the Java source code into bytecodes.
- These bytecodes will be downloaded from their author's site by a WWW browser when the WWW page is visited. So, the browser needs a Java interpreter to interpret the bytecodes.
- This is true for browsers that are *Java-aware*, e.g., Microsoft's *Internet Explorer*, Netscape's *Navigator* or Mozilla. The earlier versions of these browsers only understood JDK 1.0.2. Later versions of these browsers understand JDK 1.1.x or Java 2 SDK bytecode files that use the facilities of JDK 1.1.x. Examples are Version 4.0x (and later versions) of Netscape's Navigator (which supports most aspects of JDK 1.1.x) and Version 4.x (and later versions) of Microsoft's Internet Explorer (which supports some aspects of JDK 1.1.x).
- Some versions of WWW browsers support Java 2 Platform v 1.2 and later. For more details see ITS Guide 108 *Advanced Java*.
- Sun's Java 2 SDK (and the earlier JDKs) also include an *appletviewer* that can be used if you do not have a Java-aware browser.
- Suppose the file `HWLet.java` contains the Java applet:

  ```
  0006: import java.applet.Applet;                                      // HWLet.java
  0007: import java.awt.Graphics;
  0008: public class HWLet extends Applet {
  0009:    public void paint(Graphics rGraphics) {
  0010:       rGraphics.drawString("Hello World!", 50, 25);
  0011:    }
  0012: }
  ```

- The applet can be compiled by using the Unix/MS-DOS command:

  ```
  javac HWLet.java
  ```

  This produces a file of bytecodes in the file `HWLet.class`.

- When a browser reads the WWW page given below, it finds that it has to retrieve the file `HWLet.class`. When the bytecodes in this file arrive, the browser can interpret them.

  ```
  0013: <HTML>
  0014: <HEAD>
  0015: <TITLE> HWLet example </TITLE>
  0016: </HEAD>
  0017: <BODY>
  0018: Before the output from the applet.
  0019: <APPLET CODE="HWLet.class" WIDTH=150 HEIGHT=25>
  0020: </APPLET>
  0021: After the output from the applet.
  0022: </BODY>
  0023: </HTML>
  ```

- You can access a WWW page containing the above HTML instructions by using the URL http://www.dur.ac.uk/barry.cornelius/papers/a.taste.of.java/code/HWLet.html.

1.5 APIs

- Related classes can be grouped together in a *package*. Together the classes of a package form a way of programming in a particular area: together they define what is called an *application programming interface* (an *API*).
- Java has a large number of APIs already defined. Examples include APIs:
  - for file I/O,
  - for doing 2D and 3D graphics,
  - communicating with databases (*JDBC*),
  - for sending an e-mail message,
  - for enabling access to WWW pages,
  - for accessing objects of Java programs running on other computers,
  - for supporting the writing of applets.
- Some of these APIs are considered to be crucial, called the *Core APIs*.
- There are WWW pages documenting the Core APIs: http://java.sun.com/j2se/1.4.2/docs/api. These WWW pages can also be downloaded to filespace on your own computer.
- This Guide uses the notation `javaapi:java/lang/String.html` to refer to the WWW page http://java.sun.com/j2se/1.4.2/docs/api/java/lang/String.html.

1.6 A digression: what is JavaScript?

- Here is a WWW page containing some source code that is written in *JavaScript*:

```
<HTML> <HEAD> <TITLE> Square demo </TITLE> </HEAD>
<BODY>
<P> Start. </P>
<SCRIPT LANGUAGE="JavaScript">
<!--  hide this script from some browsers
function mysquare(myarg) {
     document.write("<P> Hello once more </P>");
     document.write("<P> <code>myarg</code> is: ", myarg, "</P>");
     return myarg*myarg;
} ;
document.write("<P> Value returned is: ", mysquare(7), "</P>");
//    end of hide -->
</SCRIPT>
<P> Finish. </P>
</BODY> </HTML>
```

- JavaScript is another programming language.
- You put JavaScript code in HTML documents with a `<SCRIPT>` tag.
- The JavaScript code is not compiled: instead, it is interpreted by a JavaScript-aware WWW browser. Unlike Java, JavaScript is understood by all browsers, including most of the early versions of Microsoft's Internet Explorer and Netscape's Navigator.
- You can access a WWW page containing the above HTML instructions by using the URL http://www.dur.ac.uk/barry.cornelius/papers/a.taste.of.java/code/square.html.
- JavaScript has most of Java's expression syntax and basic control flow constructs, but it does not have Java's strong type checking and static typing. You cannot write your own classes.
- So, JavaScript is not as powerful as Java. And it is confusing for *Java* to be included as part of the name *JavaScript*.

2 Declarations, statements, input and output

2.1 A simple Java program

```
import java.io.BufferedReader;                               // Convert.java
import java.io.InputStreamReader; import java.io.IOException;
public class Convert {                                 /* BJC 960603 */
   public static void main(String[ ] args) throws IOException {
      System.out.println("type in the lowest Fahrenheit value");
      BufferedReader input = new BufferedReader(new InputStreamReader(System.in));
      int lower = Integer.parseInt(input.readLine());
      System.out.println("type in the number of lines: ");
      int numOfLines = Integer.parseInt(input.readLine());
      int upper = lower + numOfLines - 1;
      for (int fahrenheit = lower; fahrenheit <= upper; fahrenheit++ ) {
         float celsius = to_celsius(fahrenheit);
         System.out.println(fahrenheit + " F is " + celsius + " C");
      }
   }
   // function to convert a temperature from degrees Fahrenheit
   // to degrees Celsius
   private static float to_celsius(float fahr) {
      return (fahr - 32.0F)*5.0F/9.0F;
   }
}
```

2.2 Primitive types

name	purpose of the type	default value	examples of literal values
`boolean`	logical values	false	`false`, `true`
`char`	Unicode characters	\u0000	`’ ’`, `’A’`, `’\101’`, `’\u0041’`, `’\’’`, `’\t’`
`byte`	signed integers	0	use `int` literal values
`short`	signed integers	0	use `int` literal values
`int`	signed integers	0	`0`, `42`, `2147483647`
`long`	signed integers	0	`0L`, `42L`, `9223372036854775807L`
`float`	IEEE 754 floating pt. values	0.0	`0.0F`, `9.81F`, `2.9979E8F`, `6.6252e-34F`
`double`	IEEE 754 floating pt. values	0.0	`0.0`, `9.81`, `1.0e100`, `0.5E-100`

Later we will see that the *field*s of a *class declaration* are initially given the *default value* given above, whereas a variable declared in a *method declaration* has no default value. However, a Java compiler will generate a compilation error for most attempts to use an uninitialized variable.

In Java, each of these types has a *range* that is defined by the language. So the range does not change as you move your Java source code from one platform to another. Here are the ranges:

name	size (bits	smallest value of type	largest value of type
`boolean`	1	N/A	N/A
`char`	16	\u0000	\uFFFF
`byte`	8	-128	$+127$
`short`	16	-32768	$+32767$
`int`	32	-2147483648	$+2147483647$
`long`	64	-9223372036854775808	$+9223372036854775807$
`float`	32	$-3.40282347E+38$	$+3.40282347E+38$
`double`	64	$-1.79769313486231570E+308$	$+1.79769313486231570E+308$

Values of the types `float` and `double` that are small in magnitude are considered to be zero:

name	smallest negative value	smallest positive value
`float`	$-1.40239846E-45$	$+1.40239846E-45$
`double`	$-4.94065645841246544E-324$	$+4.94065645841246544E-324$

2.3 Declarations and initializers

In Java, a *local variable declaration* is a form of statement, and so such declarations may appear at any point in a block. Here is an example:

```
char ch, separator, delimiter;
```

When a variable is declared, you can include an *initializer* that gives the variable its initial value. However, often the initial value of a variable can only be established after the execution of a few statements. In these cases, it is sometimes argued that less errors occur if the declaration is left until an appropriate initial value is known.

The initializer may be an expression that is calculated at runtime as is the case with the `lower`, `numOfLines` and `upper` variables in the program `Convert` given above.

If the value of a variable is never changed after it has been initialized, this can be (and should be) documented by using the `final` keyword. In the program `Convert`, this change could be made for the declarations of `input`, `lower`, `numOfLines`, `upper` and `celsius`, e.g.:

```
final int upper = lower + numOfLines - 1;
```

Note that this particular use of `final` was not permitted in JDK 1.0.x.

2.4 Expressions

Java's operators are similar to those of C. For example, Java has the operators `+`, `-`, `*`, `/`, and `%` for performing arithmetic. Each of these has an associated assignment operator. For example, the statement: `a += b;` has the same meaning as the statement: `a = a + b;`

The operator ++ is a unary operator that is used to increment its operand which must be a variable. If it is used as a prefix operator, the value of the expression is the value of the variable after it has been incremented, whereas if it is used as a postfix operator, the value of the expression is the value of the variable before it has been incremented. The operator – behaves like the operator ++ except that the variable is decremented instead of incremented.

Java also has the relational operators ==, `!`=, <, <=, >, >=. It has the two operators `&&` and `||`, that can be used to *and* and *or* `boolean` values. Like C, these two operators do *short-circuit evaluation*. Unlike C, *full evaluation* can be performed by using the operators `&` and `|` instead.

An expression that is preceded by the name of a type enclosed in parentheses is called a *cast expression*. The value of the expression is converted to a value of the type. Here are two examples:

```
double speedOfLight = ...;
int roughSpeedOfLight = (int)speedOfLight;
int someInt = ...;
char someChar = (char)someInt;
```

2.5 Statements

2.5.1 Blocks

There are many situations where the syntax of Java requires a single statement. A sequence of statements can be considered to be a single statement if they are turned into a *block* (which is called a *compound statement* in some other languages). This is done by surrounding the sequence of statements by and . Examples of this appear in the following section.

2.5.2 Conditional statements

Java has 2 kinds of conditional statements. Here are some examples of an `if` statement:

```
if (a < 0)           if (a > b)             if (a > b) {
   a = -a;              larger = a;            larger = a;
                     else                      System.out.println("a>b");
                        larger = b;         }
                                            else {
                                               larger = b;
                                               System.out.println("a<=b");
                                            }
```

Although C (and C++) allow the condition after the `if` to have an arithmetic value, this is not permitted in Java: the condition must be a `boolean` expression.

Here are two examples of a `switch` statement:

```
switch ( dayNumber ) {                   switch ( dayNumber ) {
   case 2: case 3: case 4:                  case 1:
   case 5: case 6:                             readRatherHeavyNewspaper();
      gotoWork();                              break;
      doWork();                             case 7:
      goHome();                                break;
      watchTV();                            default:
      gotoPub();                               gotoWork();
      break;                                   doWork();
   case 7:                                     goHome();
      break;                                   watchTV();
   case 1:                                     gotoPub();
      readRatherHeavyNewspaper();        }
}
```

Following the symbol `switch`, there should be an expression which is enclosed by parentheses. In the above examples, this expression is on the line:

```
switch ( dayNumber ) {                   switch ( dayNumber ) {
```

and it just consists of the variable `dayNumber`. The expression should be of type `char`, `byte`, `short`, or `int`.

When the `switch` statement is executed, the expression is evaluated and then control is passed to the statement whose associated `case` label has a value equal to that of the expression. If there is no such statement, then control is passed to the statement associated with the `default` label if there is one; otherwise, control is passed to the statement following the `switch` statement.

A `break` statement must be executed if you wish to leave the `switch` statement before the last statement of the `switch` statement. So, normally, there will be a `break` statement just before each `case` label and before the `default` label (if there is one).

2.5.3 Looping statements

Java has 3 kinds of looping statements. Here is an example of a `for` statement:

```
int numMonths = Integer.parseInt(input.readLine());
int rainfallSum = 0;
for ( int monthNum = 0; monthNum < numMonths; monthNum++ ) {
   int figureForMonth = Integer.parseInt(input.readLine());
   rainfallSum += figureForMonth;
}
System.out.println(rainfallSum);
```

The above example has the variable `monthNum` declared in the `for` statement itself. If you do this, then this variable can only be used within the `for` statement. If you leave out the type, then the variable must be declared elsewhere and the variable is similar to any other variable of the block containing the `for` statement.

Here is an example of a `while` statement:

```
int rainfallSum = 0;
int figureForMonth = Integer.parseInt(input.readLine());
while ( figureForMonth >= 0 ) {
   rainfallSum += figureForMonth;
   figureForMonth = Integer.parseInt(input.readLine());
}
System.out.println(rainfallSum);
```

Here is an example of a `do` statement:

```
int rainfallSum = 0;
do {
   int figureForMonth = Integer.parseInt(input.readLine());
   if ( figureForMonth >= 0 )
      rainfallSum += figureForMonth;
} while ( figureForMonth >= 0 ) ;
System.out.println(rainfallSum);
```

Although C (and C++) allow the condition of a `while` statement or a `do` statement to have an arithmetic value, this is not permitted in Java: the condition must be a `boolean` expression.

2.5.4 Other control statements

A `break` statement terminates the execution of a `for`, `do`, `while` or `switch` statement, and transfers control to the statement following that statement. A `break` statement may include a label, and this label indicates that it is the statement with that label that is to be terminated.

A `continue` statement transfers control to end of the current iteration of a `for`, `do` or `while` statement. A `continue` statement may include a label, and, if this is the case, control skips to the end of the loop that has this label.

Java also has `try` statements, `catch` clauses, `finally` clauses, and `throw` statements. These are all used for *exception handling*, and details about these will be given later.

2.6 Objects

2.6.1 Reference variables

Besides the primitive types that can be used for simple values, we often want to represent structured values. For example, we might want to represent a date in history, a point in two-dimensional space, and so on. In Java, a variable that is of a *reference type* is used to *refer* to a structured value.

For example, in order to represent a point in two-dimensional space, the package `java.awt` provides a ‘class declaration’ called `Point`. Such a class declaration automatically provides a reference type called `Point`, and we can declare a variable called `myPoint` to be of this reference type by the declaration:

```
java.awt.Point myPoint;
```

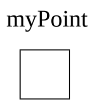
To avoid having to repeat the package name every time we want to use `Point`, we can use an `import` declaration at the start of the file containing the Java source code:

```
import java.awt.Point;
```

Having done this, we can declare the variable `myPoint` by:

```
Point myPoint;
```

Such a declaration only introduces a *reference variable*, a variable that can refer to an object that contains the details about the point.



In some ways, a reference variable is like a *pointer variable* in Pascal, C or C++.

2.6.2 Creating an object

Having declared the reference variable, we ought to get it to refer to a `Point` object. This is done by using an assignment statement where the RHS contains a *class instance creation expression*:

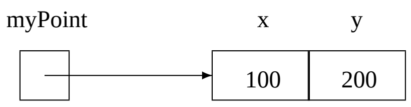
```
myPoint = new Point(100,200);
```

The creation expression `new Point(100,200)` uses a ‘constructor’ for the class `Point` to create an object of that class with x and y *field*s of 100 and 200. We will see later that we can do this because this kind of constructor has been provided by the designers of the `java.awt` package. Often a class provides several different constructors, e.g., as well as a constructor that has two parameters which are the x and y coordinates, the designers of the class `Point` could also have provided a constructor to construct a `Point` from a `String`:

```
myPoint = new Point("100:200");
```

but they chose not to do this.

So we now have a variable called `myPoint` that refers to a point that has the x and y coordinates 100 and 200.



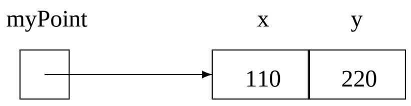
The above declaration of `myPoint` together with the above assignment statement can be shortened to a declaration that has an initializer:

```
Point myPoint = new Point(100,200);
```

2.6.3 Referring to the fields of an object

We can use the *dot notation* to refer to the fields of an object, e.g., we can use `myPoint.x` and `myPoint.y`. For example, we could change the point being represented by 10 units in the x direction and 20 units in the y direction by the assignment statements:

```
myPoint.x += 10;
myPoint.y += 20;
```



Note that this is a little different to the languages Pascal, C and C++ where some extra syntax is used to say that we are dereferencing a pointer.

2.6.4 Applying methods to an object

The designers of the `java.awt` package have thought that we may want to move an existing point to a new point in space, and so they have provided a *method* to do this. A method is what would be called a *function* or *procedure* in other programming languages. The method that they have provided is called `translate`. So, instead of the above two assignment statements, we could write:

```
myPoint.translate(10,20);
```

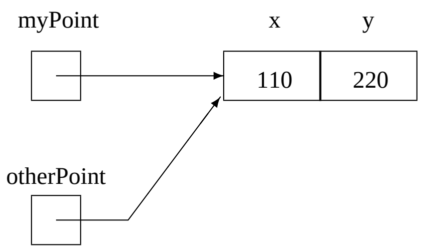
Note that the dot notation that we used above to refer to the two fields of a `Point` object is also used in the call of a method. You should look at this call in the following way: ‘apply the `translate` method with arguments 10 and 20 to the `myPoint` object’. Note: you would have written something like: `translate(myPoint,10,20)` in languages like Pascal and C.

2.6.5 Copying objects

Suppose we have:

```
0118: Point otherPoint;
0119: otherPoint = myPoint;
```

The assignment statement causes `otherPoint` to refer to the same object that `myPoint` refers to.

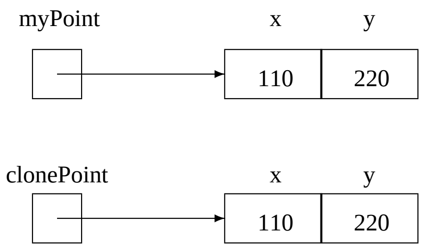


So, the above assignment statement does not produce a clone. The classes of the Core APIs use two different ways of enabling you to produce a clone of an object:

- a class sometimes provides a method called `clone`;
- a class sometimes provides a suitable constructor.

Although `Point` does not provide a `clone` method, it does provide a suitable constructor:

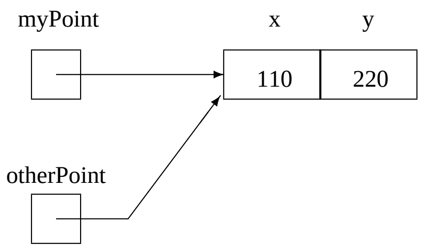
```
0120: Point clonePoint;
0121: clonePoint = new Point(myPoint);
```



2.6.6 Comparing objects

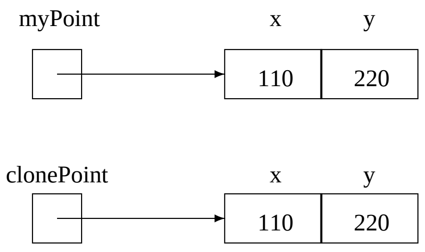
The == operator in the following condition is asking whether the two reference variables refer to the same object:

```
0122: if ( myPoint == otherPoint ) { ... } else { ... }
```



If, instead, you want to ask whether the two objects referred to by two reference variables have the same value, you can often use a method called `equals`:

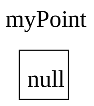
```
0123: if ( myPoint.equals(clonePoint) ) { ... } else { ... }
```



2.6.7 The value `null`

If a reference variable has the value `null`, then this means that the variable does not currently refer to any object. An assignment statement can be used to indicate this:

```
myPoint = null;
```



And you can test whether a reference variable does not refer to an object:

```
if ( myPoint == null ) { ... } else { ... }
```

Note: whilst `null` appears to be a keyword of the language, it is technically the *null literal*.

2.6.8 Garbage collection

Pascal/C/C++ programs inadvertently `dispose`/`free`/`delete` objects which are still in use:

```
var p, q:^integer;    int *p, *q;                 int *p, *q;
new(p);               p = malloc(sizeof(int));    p = new int;
p^ := 27;             *p = 27;                    *p = 27;
q := p;               q = p;                      q = p;
dispose(p);           free(p);                    delete p;
writeln(q^);          printf("%d\n", *q);         cout << *q << endl;
```

And programs often cause memory leaks by *not* using `dispose`/`free`/`delete` on unwanted objects.

In Java, you do not delete objects: instead, Java has *garbage collection*. The garbage collector detects objects no longer in use, and reuses their space. Also, unlike C++, you do not have to provide destructors for classes.

2.7 Arrays

2.7.1 Introduction

In Java, an *array* is a collection of values that are of the same primitive type or of the same reference type. Since an array type is itself a reference type, arrays of arrays can be constructed.

2.7.2 An array of integers

The numbers of the days in a non-leap year on which each of the twelve months start are 1, 32, 60, 91, 121, 152, 182, 213, 244, 274, 305 and 335. For example, March 1st is the 60th day of the year. Suppose we want to provide an array called `monthStarts` that contains this information.

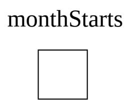
In Java, there are two syntaxes for an array declaration. To keep C programmers happy, an array declaration can be written using the following syntax:

```
int monthStarts[ ];
```

However, the following syntax is better:

```
int[ ] monthStarts;
```

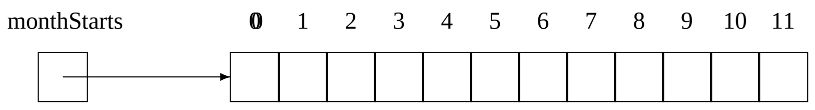
Note that the number of elements in the array is not included in this declaration. This is because this declaration only declares a reference variable that can be used to refer to an array object.



In order to create the actual array object we need to use an assignment statement that contains an *array creation expression* on its RHS:

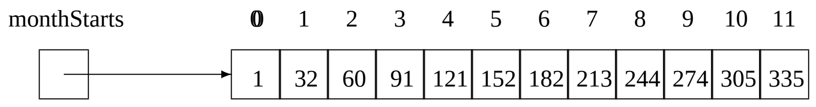
```
monthStarts = new int[12];
```

So this has set up `monthStarts` to be a reference variable that refers to an array of 12 integers, with indexes from 0 to 11.



You can access each individual element using the usual notation:

```
monthStarts[0] = 1;
monthStarts[1] = 32;
...
monthStarts[11] = 335;
```



If an index is out of bounds, the exception `ArrayIndexOutOfBoundsException` will be thrown: details about ‘exception handling’ are given later.

As before, the creation expression can be used as an initializer:

```
int[ ] monthStarts = new int[12];
monthStarts[0] = 1;
monthStarts[1] = 32;
...
monthStarts[11] = 335;
```

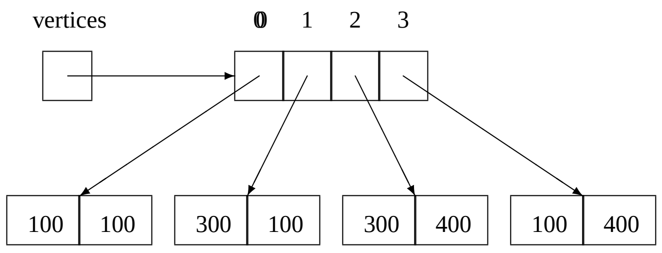
This code can be abbreviated to:

```
int[ ] monthStarts = { 1,32,60,91,121,152,182,213,244,274,305,335 };
```

2.7.3 An array of points

Suppose we want an array where each element is an object of class `java.awt.Point`. Perhaps we want an array to represent the four vertices of the rectangle (100,100), (300,100), (300,400) and (100,400). We can do this as follows:

```
Point[ ] vertices = new Point[4];
vertices[0] = new Point(100,100);
vertices[1] = new Point(300,100);
vertices[2] = new Point(300,400);
vertices[3] = new Point(100,400);
```



Once again, this code can be abbreviated:

```
Point[ ] vertices = { new Point(100,100), new Point(300,100),
                      new Point(300,400), new Point(100,400) };
```

2.7.4 Flexible arrays

Within the square brackets of an array creation expression, there needs to be an expression indicating the number of elements that are required in the array object. This expression may be one whose value is not known until runtime. For example:

```
BufferedReader input = ... ;
int size = Integer.parseInt(input.readLine());
int[ ] monthStarts = new int[size];
```

2.7.5 Even more flexible arrays

Having created an array object, the size of the array object is fixed. Suppose you are storing details about a collection of people, and suppose the size of the collection changes during the course of the execution of a program. It may be that you have no idea what the maximum size of the collection will be. Although you could arbitrarily choose a large value, this is wasteful of space, and no matter what value you choose, your program will fail if the value you choose is too small. In such situations, it is probably better to use a `List`, a `Set` or a `Map`. These are facilities that are provided by the Collections API of the Java 2 Platform. For more details, see ITS Guide 108 *Advanced Java*.

2.8 Methods

In Java, the word *method* is used instead of *function*, *procedure* or *subroutine*. The argument to `println` in:

```
System.out.println(convertToCelsius(82.0));
```

is `convertToCelsius(82.0)`. This is an example of a call of a method such as:

```
private static double convertToCelsius(double fahr) {
   return (fahr - 32.0)*5.0/9.0;
}
```

Unlike C and C++, there is no default return type: you must specify it. If the method does not return a result, `void` should be used as the return type.

In Java, a parameter of a method behaves like a local variable of the method. It gets its initial value from the argument passed in the call. Any assignment to the parameter within the method only affects the value of the local variable. If a method does not assign a value to the parameter, this can be (and should be) documented by using the `final` keyword. However, note that this use of `final` was not permitted in JDK 1.0.x. Here is an example:

```
private static double convertToCelsius(final double fahr) {
   return (fahr - 32.0)*5.0/9.0;
}
```

Unlike other languages, a method cannot change the value of the variable that is passed as an argument. So given:

```
private static void silly(double p) {
   p = p + 4.2;
   System.out.println(p);
}
```

the following code will not alter the value of the variable `a`:

```
a = 2.7;
silly(a);
System.out.println(a);
```

If a method has no parameters, then it is declared with an empty parameter list:

```
private static void m() { ... }
```

and a call has an empty argument list:

```
m();
```

It is possible to declare several methods having the same name provided that they can be distinguished by the types of their parameters. This is called *method overloading*. Here is an example where the name `min` is declared twice:

```
private static long min(long a, long b) { return a<b ? a : b; }
private static Date min(Date a, Date b) { return a.before(b) ? a : b; }
```

At a call of `min`, the compiler can look at the arguments to see which `min` is required.

Note: the modifiers `private` and `static` will be discussed later.

2.9 Output and input

2.9.1 Attaching an output stream

To write values to a file called `results`, you can use a variable of the type `PrintWriter`. Suppose you want to use a variable called `filout`:

```
PrintWriter filout =
      new PrintWriter(new BufferedWriter(new FileWriter("results")));
```

This declaration assumes that the following imports appear at the start of the file of source code:

```
import java.io.BufferedWriter;
import java.io.FileWriter;
import java.io.PrintWriter;
```

If you want some output to be sent to the *standard output*, which is usually the screen, you can use `System.out` as an output stream. The variable `out` is a variable (of the type `java.io.PrintStream`) that is a ‘class variable’ of the class `System` which is defined in the package `java.lang`. Any class of this package is automatically available to a program without the need for any `import` declaration.

2.9.2 Outputting values to the output stream

In order to output a textual representation of a value, the `print` method should be applied to an object of the class `PrintWriter` or `PrintStream`. Here are two examples:

```
filout.print("Hello World!");
System.out.print("Hello World!");
```

The `print` method can be passed:

- an argument of any primitive type;
- an argument of any reference type for which the method `toString` is defined;
- an expression that uses the *string concatenation operator*, e.g.:

  ```
  int first = 42;
  System.out.print("first has the value " + first);
  ```

If you want the output to move on to the next line after the value has been printed, use `println` instead of `print`.

2.9.3 Closing a file

As output to a `BufferedWriter` stream is buffered, the stream will need to be *close*d when you have finished using it:

```
filout.close();
```

2.9.4 Attaching an input stream

To read values, you will need a variable of the type `BufferedReader` that is in the package `java.io`. To read from the *standard input*, which is normally the keyboard, you can use `System.in` as an input stream. So, if you want to use a variable called `input` for this input stream, you can use the following declaration:

```
BufferedReader input =
      new BufferedReader(new InputStreamReader(System.in));
```

This declaration assumes that the following imports appear at the start of the file of source code:

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

If, instead, you want to read values from a file called `data`, you can use:

```
BufferedReader filin = new BufferedReader(new FileReader("data"));
```

This declaration assumes that the following imports appear at the start of the file of source code:

```
import java.io.BufferedReader;
import java.io.FileReader;
```

2.9.5 Reading a line of characters

The method `readLine` can be used to read in a line of characters from an input stream. It returns a value of type `String`. Here are two examples:

```
String inputLine = input.readLine();
String filinLine = filin.readLine();
```

The class `String` is defined in the package `java.lang`, and so it can be used in a program without the need for an import declaration.

2.9.6 Reading a value

If you would like to read a value into a variable whose type is one of the primitive types, you first need to call `readLine` to read in a line of characters and then call an appropriate method to parse the string. Here is an example where a value of type `int` is obtained from the keyboard:

```
String line = input.readLine();
int intVal = Integer.parseInt(line);
```

This can be abbreviated to:

```
int intVal = Integer.parseInt(input.readLine());
```

Given a variable called `line` containing a `String`:

```
String line = input.readLine();
```

values of the other primitive types can be obtained using the following statements:

```
long longVal = Long.parseLong(line);
float floatVal = Float.parseFloat(line);
double doubleVal = Double.parseDouble(line);
boolean booleanVal = new Boolean(line).booleanValue();
```

Note that the methods `parseFloat` and `parseDouble` were introduced into Java when the Java 2 Platform was released, and so, if you are using JDK 1.0.2 or JDK 1.1.x, you will have to use methods called `floatValue` or `doubleValue` instead (in a similar way in which `booleanValue` is used above).

The classes `Integer`, `Long`, `Float`, `Double` and `Boolean` are defined in the package `java.lang`, and so they can be used in a program without the need for any import declarations.

2.9.7 Handling more than one data item per line

You can use the class `java.util.StringTokenizer` if you want more than one data item per line. Suppose a line contains an `int`, followed by a `float`, followed by another `int`. You could use:

```
String line = input.readLine();
StringTokenizer tokens = new StringTokenizer(line);
String token = tokens.nextToken();
int firstInt = Integer.parseInt(token);
token = tokens.nextToken();
float theFloatVal = Float.parseFloat(token);
token = tokens.nextToken();
int secondInt = Integer.parseInt(token);
```

2.9.8 Flushing the output

If you want the user to type on the same line as a prompt, you will need to *flush* the output stream after outputting the prompt:

```
BufferedReader input =
      new BufferedReader(new InputStreamReader(System.in));
System.out.print("Type in an integer: ");
System.out.flush();
String line = input.readLine();
int value = Integer.parseInt(line);
```

2.9.9 Dealing with `java.io.IOException`

If you are going to use the classes and methods from the `java.io` package, you will find that you are unable to compile your program unless it indicates what you want to happen if an *exception* called `java.io.IOException` occurs. Details about 'exception handling' are given later. So, to begin with, you may be happy for your program to crash if an IO exception occurs. This can be done by adding the clause `throws IOException` to the heading of any method that does IO. For example:

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

This code assumes that the following import appears at the start of the file of source code:

```
import java.io.IOException;
```

2.9.10 JDK Version 1.0.x

Many of the classes given above are not available if you use Version 1.0.x of the JDK. And the input-output facilities provided by Version 1.0.x can only handle *byte streams*. Those of JDK Version 1.1.x (and later) include support for *character streams*, i.e., streams containing 16-bit Unicode characters rather than just 8-bit bytes.

3 Handling strings

3.1 Creating an object of the class `String`

Although:

```
String tName = new String("James Gosling");
```

is the obvious way of creating a string object and making `tName` point to it, for strings there is an alternative syntax for the *class instance creation expression*. You can use `"James Gosling"` instead of using `new String("James Gosling")` as in:

```
String tName = "James Gosling";
```

So you have a choice here: both forms of syntax can be used to create new string objects.

A string literal can include characters that are *non-graphic character*s. This is done by using an *escape sequence*. An escape sequence is also necessary for putting a single quote, a double quote or a backslash in a string:

```
System.out.println("Lister glared at Rimmer.  \"You really are a smeghead\", he said.");
```

We will sometimes need to represent a string that has no characters. The string literal `""` or the expression `new String("")` can be used. Such a string is called the *empty string*.

3.2 Applying methods to a `String` object

The class `java.lang.String` comes with a large number of methods for manipulating strings. A list of these methods is documented in the *Method Detail* section of `javaapi:java/lang/String.html`.

For example, if you want to access an individual character of a string, you can use a method called `charAt`. The value that is returned is of type `char`. You use an argument that is an `int` value to indicate the position of the character which you want to be returned. However, its value needs to be one less than the position of the character. So if you want the first character of the string to be returned, you need an argument with the value 0:

```
String tName = new String("James Gosling");
char tFirstChar = tName.charAt(0);
System.out.println("The first character of the name is: " + tFirstChar);
```

The `println` statement will output the line:

```
The first character of the name is: J
```

There is also a method that can be used to find out how many characters there are in a string:

```
String tName = new String("James Gosling");
int tNameLength = tName.length();
char tLastChar = tName.charAt(tNameLength - 1);
System.out.println("The last character of the name is: " + tLastChar);
```

This will output:

```
The last character of the name is: g
```

3.3 The exception `StringIndexOutOfBoundsException`

Many of the methods of the class `String` have an argument that is an integer that is the position of a character within a string. If you pass an argument that is invalid, the method will signify that it cannot handle this situation, by *throw*ing an *exception* called `StringIndexOutOfBoundsException`. An *exception* is an occurrence of an exceptional circumstance, a situation that does not normally occur.

For example, if you call `charAt` with the value 5 when a string has 5 characters, your program will crash displaying lines like:

```
java.lang.StringIndexOutOfBoundsException: String index out of range: 5
        at java.lang.String.charAt(String.java)
        at StringIndexTest.main(StringIndexTest.java:6)
```

Instead of letting the program crash like this, we can include code in our program that will be executed when an exception occurs. Java has a statement called a *try statement* that is used to *handle exceptions*, and we will look at *try statement*s later.

Java divides exceptions into two categories: *checked exception*s and *unchecked exception*s. A `StringIndexOutOfBoundsException` is an unchecked exception, and Java says that a program does not have to say what it wants to happen when an unchecked exception occurs.

3.4 Changing a `String` object

The class `java.lang.String` is rather unusual: none of its methods alter the object to which the method is being applied. The objects of the class are said to be *immutable*.

Instead of a method altering the value of a string object, it will produce a new string object. For example, consider:

```
String tToday = new String("1999-07-11");
tToday = tToday.replace('-', ':');
System.out.println(tToday);
```

First, a string object containing the string `"1999-07-11"` is created and `tToday` is made to point to it. Then the method `replace` is applied to the string object that is pointed to by `tToday`. This does not change that string object, but instead creates a new string object in which any occurrences of the `'-'` character are replaced by a `':'` character. Then the value of `tToday` is changed. It is currently pointing to the first string object, and it is now altered to point to the new string object. There is now no variable pointing to the first string object: it is lost. Finally, the string that `tToday` points to is output by the call of the `println` method:

```
1999:07:11
```

3.5 Copying `String` objects

As was shown with the class `java.awt.Point`, you can make another variable refer to the same string by an assignment statement:

```
String tName = new String("James Gosling");
String tSameName = tName;
```

Both reference variables refer to the same object.

Earlier, when we used the `String` constructor, we passed a string literal as an argument. If you want a clone of a `String` object, then you can pass that `String` object as the argument of a `String` constructor:

```
String tName = new String("James Gosling");
String cloneName = new String(tName);
```

3.6 String concatenation

The class `java.lang.String` is unusual because an operator is defined in the language specifically for the concatenation of the values of two objects of this class:

```
String tFirstName = new String("James");
String tSurname = new String("Gosling");
String tName = tFirstName + tSurname;
```

The variable `tName` now points to a string object containing the string `"JamesGosling"`. Perhaps that is not what we were after. So use this instead:

```
String tName = tFirstName + " " + tSurname;
```

The string concatenation operator is very flexible in that it will convert any operand (that is permitted) into a string. Here is an example:

```
Point tFirstPoint = new Point(100, 200);
String tLine = "The point has the value " + tFirstPoint;
System.out.println(tLine);
```

This will output:

```
The point has the value java.awt.Point[x=100,y=200]
```

If you have a long string literal, the string concatenation operator can be used to help in the layout of the text. For example, the statement:

```
System.out.println("Lister glared at Rimmer.  \"You really are a smeghead\", he said.");
```

can instead be written as:

```
System.out.println("Lister glared at Rimmer." +
                   "  \"You really are a smeghead\", he said.");
```

3.7 A program that uses these ideas about `String`s

Suppose we want a program that takes a person's name arranged as *FirstName Surname* and outputs it in the format *Surname, Initial* where *Initial* is the first letter of the *FirstName*. We will also suppose that the output must be displayed in *upper-case*. Here is a program that does this for the name `"James Gosling"`:

```
public class SimpleString {                              // SimpleString.java
   public static void main(final String[] pArgs) {
      final String tName = new String("James Gosling");
      System.out.println(tName);
      final char tFirstChar = tName.charAt(0);
      final int tPositionOfSpace = tName.indexOf(" ");
      final String tSurname = tName.substring(tPositionOfSpace + 1);
      String tLabel = tSurname + ", " + tFirstChar;
      tLabel = tLabel.toUpperCase();
      System.out.println(tLabel);
   }
}
```

The two `println`s of this program produce the following output:

```
James Gosling
GOSLING, J
```

3.8 The class `StringBuffer`

Besides the class `String`, there is another class called `StringBuffer` (which is also in the `java.lang` package). When you wish to build up a string gradually by performing a lot of string manipulation, it is more efficient to use a `StringBuffer` rather than create a lot of `String` objects. If you have a `StringBuffer` variable called `tStringBuffer`, you can apply `toString` method to the variable in order to create a `String` from `tStringBuffer`:

```
public static String reverse(String source) {
   int charNum;
   int numChars = source.length();
   StringBuffer temp = new StringBuffer(numChars);
   for (charNum = numChars-1; charNum>=0; charNum--) {
      temp.append(source.charAt(charNum));
   }
   return temp.toString();
}
```

4 Using classes for data abstraction

4.1 Introduction

Typically a program has to maintain several data structures each of which is manipulated in many different ways. It is best for the pieces of code that manipulate a particular data structure to be located in a small number of functions. And it would be desirable if the program could be written so that each data structure can only be accessed from its associated functions, i.e., it is not directly accessible to the rest of the program. In this way, we would then prevent a data structure from accidentally being misused. What we want is a way of building a wall around a data structure and the functions that manipulate it, and only allowing some of these functions to be accessible from outside the wall. Modern programming languages have a construct to do this: for example, Ada has packages, Fortran90 and Modula-2 both have modules, and C++ and Java both have classes.

In this section, we look at how to write our own class declarations.

4.2 Using a class declaration to define your own type

There are two main characteristics to a *type*:

- a type has a set of values associated with it;
- a type has a set of operations that are permitted on these values.

For example, the type `int` refers to the set of integer values from some large negative value to some large positive value, together with operations such as addition and subtraction (denoted by `+` and `-`).

One approach to writing a program is to identify the objects of the problem that you need to represent in the program. Each object can be in a number of states (i.e., may possess one of a number of different values) and has a set of operations that can be performed on it.

Although some of these objects can be realised in your program by a variable of a type that is pre-defined in the programming language you are using, it would be useful to be able to define your own types to represent the other objects. The process of identifying the types needed for these objects is referred to as *data abstraction*.

4.3 A class called `Date`

In Java, it is possible to use a *class declaration* to define your own type. For example, suppose that it is necessary to manipulate some dates in a program. We can think of dates as being composed of three parts, the day, month and year. Operations that are performed on dates include constructing dates, copying dates, comparing two dates, getting the day, month and year parts of a date, and performing input-output for values that are dates.

4.4 Stage A: providing a primitive version of the class `Date`

To begin with, we will produce a class declaration that is just able to represent values that are dates: it provides no operations, and so there will be little that we can do with these date values.

Within each object of this class, three `int`s will be used to represent the year, month and day parts of a date. Here is the class declaration:

```
// A class for representing values that are dates.              // Date.java
public class Date
{
   public int year;
   public int month;
   public int day;
}
```

This class declaration for the class `Date` needs to be stored in the file `Date.java`.

Here is a program that uses the class `Date`. It is called `NoelProg`, and so these lines need to be stored in the file `NoelProg.java`:

```
// This program creates an object of the class Date         // NoelProg.java
// and then sets its fields to represent Christmas Day 1999.
public class NoelProg
{
   public static void main(final String[] pArgs)
   {
      final Date tNoelDate = new Date();
      tNoelDate.year = 1999;
      tNoelDate.month = 12;
      tNoelDate.day = 25;
      System.out.println(tNoelDate.year + "-" +
                         tNoelDate.month + "-" + tNoelDate.day);
   }
}
```

When we want to execute the `NoelProg` program, we first have to compile the two pieces of Java source code:

```
javac Date.java
javac NoelProg.java
```

This produces the files `Date.class` and `NoelProg.class`. Since it is the file `NoelProg.java` that contains the `main` method, we can execute the program by typing:

```
java NoelProg
```

What does the `NoelProg` program do? The first statement:

```
final Date tNoelDate = new Date();
```

is a declaration. The left-hand side establishes a reference variable called `tNoelDate`. The initializer on the right-hand side is a *class instance creation expression*:

```
new Date()
```

This creates an object that is just big enough to hold the fields of the class, i.e., the three fields called `year`, `month` and `day`.

Each field will be initialized to a value which depends on the type of the field. The default values for fields were given earlier. As the three fields of the class `Date` have the type `int`, they will be initialized to zero. The initializer causes `tNoelDate` to be assigned a value that points to this object.

Following the declaration of `tNoelDate`, there are three assignment statements that assign values to each of these three fields. For example:

```
tNoelDate.year = 1999;
```

puts a value in the `year` field of `tNoelDate`. Here the *dot notation* introduced earlier is being used. Although it is possible to assign values to the year, month and day fields that do not represent a date, we will ignore this deficiency.

The last statement of the program outputs the line:

```
1999-12-25
```

Here is some jargon: a piece of code that uses another class is said to be a *client* of the class. So the program `NoelProg` is a *client* of the class `Date`.

Although this is exciting because we have declared this class ourselves, there is nothing new about the way in which we are using the class. It is much like what we did with the class `Point` earlier.

4.5 Stage B: adding a constructor and a method declaration

4.5.1 Stage B1: adding a constructor declaration

The three assignment statements in the `NoelProg` program (given above) ensure that the `Date` object has the values that we want it to have. When we create an object, we will frequently want to assign values to all of the fields of the object. For this reason, Java allows class declarations to have *constructor*s.

With a class to represent dates, an obvious constructor is one that creates a date object from three integers:

```
final Date tNoelDate = new Date(1999, 12, 25);
```

Here the *class instance creation expression* uses a constructor that has three `int` arguments. This is only possible if the class declaration for `Date` has a *constructor declaration* that has three `int` arguments:

```
public Date(final int pYear, final int pMonth, final int pDay)
{
   year = pYear;
   month = pMonth;
   day = pDay;
}
```

In many ways, a constructor looks like a method declaration. However, there are two differences: there is no result type and the declaration has the same name as the class.

So, when the declaration:

```
final Date tNoelDate = new Date(1999, 12, 25);
```

is executed, first the object is constructed with default initial values, and then the constructor is executed. So, the values 1999, 12, 25 are assigned to `pYear`, `pMonth` and `pDay`, and then the block of the constructor leads to the following statements being executed:

```
year = 1999;
month = 12;
day = 25;
```

A constructor can refer to the fields of the object being initialized by using the names of the fields. So these statements result in the fields of the object having their values changed. The final act of the declaration is to make `tNoelDate` refer to the object that has just been created by the *class instance creation expression*.

After this declaration, the `NoelProg` program executes:

```
tNoelDate.day++;
```

This statement increases the value of the `day` field of this object by 1.

4.5.2 Stage B2: using a method to display the value of an object

The `NoelProg` program that was given earlier outputs the value of a `Date` object by using:

```
System.out.println(tNoelDate.year + "-" +
                   tNoelDate.month + "-" + tNoelDate.day);
```

Displaying the value of an object is a common task and:

- to save us from writing the above code each time we want to output a date;
- to ensure that we get consistent output;

it is useful to put the code for outputting a date into a method.

The following class declaration for `Date` includes a method declaration for a method called `display`:

```
// A class for representing values that are dates.              // Date.java
public class Date
{
   public int year;
   public int month;
   public int day;
   public Date(final int pYear, final int pMonth, final int pDay)
   {
      year = pYear;
      month = pMonth;
      day = pDay;
   }
   public void display()
   {
      System.out.println(year + "-" + month/10 + month%10 +
                                "-" + day/10 + day%10);
   }
}
```

Its use is illustrated by this version of the `NoelProg` program:

```
// This program creates an object of class Date            // NoelProg.java
// representing Christmas Day 1999, then moves
// the day field on by 1, and then outputs the new date.
public class NoelProg
{
   public static void main(final String[] pArgs)
   {
      final Date tNoelDate = new Date(1999, 12, 25);
      tNoelDate.day++;
      tNoelDate.display();
   }
}
```

When the `NoelProg` program executes the statement:

```
tNoelDate.display();
```

the method called `display` will get called, and it will be applied to the object pointed to by the `tNoelDate` variable. When the block of `display` is executed, i.e., when the statement:

```
System.out.println(year + "-" + month/10 + month%10 + "-" + day/10 + day%10);
```

is executed, the references to `year`, `month` and `day` are references to the `year`, `month` and `day` fields of `tNoelDate`. The uses of `/10` and `%10` ensure that two digits are always output for the month and day values.

The call of this method will output the line:

```
1999-12-26
```

4.6 Grouping fields and methods together to implement a type

The above class declaration for `Date` not only has the declaration of three fields (`year`, `month` and `day`): it also has the declaration of a method (`display`). Earlier, it was suggested that the two main characteristics of a type are a set of values and some operations to perform on those values. So, one of the major attractions of a class declaration is that it allows us to group together:

- fields to implement the values of a type;
- methods to implement the operations of a type.

The fields and methods are sometimes referred to as the *member*s of the class.

4.7 Stage C: hiding fields, providing access methods and `toString`

4.7.1 Stage C1: hiding the fields and accessing them using methods

With the previous class declaration for a date, the fields of an object are directly accessible from a client, i.e., a program like `NoelProg` can refer to the `day` field of the object pointed to by `tNoelDate` by using `tNoelDate.day`. It can do this because, in the class declaration, the fields have a `public` modifier, e.g.:

```
public int day;
```

Back in the real world, when you want to get off a bus, you usually indicate this by signalling to the bus driver in some way, e.g., by pressing a button that rings a bell. Giving everyone a brake pedal would not be a good idea! In the same way, it is unusual to expose the fields of an object to a client. Instead of making a field `public`, we will make it `private` and usually we will provide some methods to allow access to the field. Such methods are called *access method*s.

So in the following class declaration for `Date` the three fields for year, month and day have been made `private`:

```
private int iYear;
private int iMonth;
private int iDay;
```

At the same time, the names of these fields have been changed. In this Guide, the `i` prefix will be used for entities that are *internal* to a class. You can also remember the meaning of `i` because it is also a letter of the words h*i*dden and pr*i*vate.

```
// A class for representing values that are dates.              // Date.java
public class Date
{
   private int iYear;
   private int iMonth;
   private int iDay;
   public Date(final int pYear, final int pMonth, final int pDay)
   {
      iYear = pYear;
      iMonth = pMonth;
      iDay = pDay;
   }
   public int getYear()
   {
      return iYear;
   }
   public int getMonth()
   {
      return iMonth;
   }
   public int getDay()
   {
      return iDay;
   }
   public void setYear(final int pYear)
   {
      iYear = pYear;
   }
   public void setMonth(final int pMonth)
   {
      iMonth = pMonth;
   }
   public void setDay(final int pDay)
   {
      iDay = pDay;
   }
   public String toString()
   {
      return iYear + "-" + iMonth/10 + iMonth%10 + "-" + iDay/10 + iDay%10;
   }
}
```

The above class declaration also provides six access methods called `getYear`, `getMonth`, `getDay`, `setYear`, `setMonth` and `setDay`. Two of these are used by the following version of the `NoelProg` program:

```
// This program creates an object of class Date             // NoelProg.java
// representing Christmas Day 1999, then moves the
// day component on by 1, and then outputs the new date.
public class NoelProg
{
   public static void main(final String[] pArgs)
   {
      final Date tNoelDate = new Date(1999, 12, 25);
      final int tDay = tNoelDate.getDay();
      tNoelDate.setDay(tDay + 1);
      System.out.println(tNoelDate.toString());
      System.out.println(tNoelDate);
   }
}

```

When the program calls `getDay` as in `tNoelDate.getDay()`, the block of `getDay` will execute the statement:

```
return iDay;
```

So the value of the `iDay` field of the object that is the target of the method invocation is returned. In the case of the `NoelProg` program, `getDay` is being applied to the object pointed to by `tNoelDate`, and the method returns the value of its `iDay` field. So, 25 gets returned.

In the next statement, the program executes:

```
tNoelDate.setDay(tDay + 1);
```

So `pDay` (the parameter of `setDay`) is assigned the value 26 and this value is used in the block of `setDay` to change `iDay` to 26. Once again, the object pointed to by `tNoelDate` is the target of this call and so it is this object's `iDay` field that is changed to 26.

The technique of hiding fields behind access methods is an important one. It is called *data encapsulation* (or *information hiding*). If you look at the WWW pages for the Core APIs you will find very few classes that have public fields. Possibly the only ones are in the classes `java.awt.Point` and `java.awt.Rectangle`.

4.7.2 Stage C2: using `toString` instead of `display`

Although in Stage B2, we found it useful to introduce a method (`display`) which uses `println` to display the value of a `Date` object, in Java it is more usual:

- for a class to declare a method (called `toString`) that returns a string that is some textual representation of the value;
- for a client to do whatever it wants with the string, e.g., one possibility being to call `print` or `println` to output the string.

So, instead of having a method called `display` that calls `println`, the version of the class `Date` given above declares a method called `toString` that just returns a string that `NoelProg` passes as an argument to `println`:

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

When `toString` gets called, it just executes:

```
return iYear + "-" + iMonth/10 + iMonth%10 + "-" + iDay/10 + iDay%10;
```

You can see that the result type of `toString` is the type `String`. So the execution of this return statement forms a string consisting of the concatenation of the `iYear` field of `tNoelDate`, a hyphen, the two digits of the `iMonth` field of `tNoelDate`, another hyphen, and then the two digits of the `iDay` field of `tNoelDate`, e.g., `"1999-12-25"`. This is the string that `NoelProg` passes as an argument to `println`.

It is usual to call this method `toString`. The reason for this is that the statement:

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

can be abbreviated to:

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

This is because both `print` and `println` are defined so that if a variable of a reference type is passed as an argument then that type's `toString` method is called.

4.8 Using the default version of `toString`

If you provide a class declaration but fail to provide a `toString` method, it is still possible for a program to apply the `toString` method to an object of the class. For example, if the `toString` declaration of `Date`'s declaration is removed, the `NoelProg` program is still a valid program. When it is run, the program will execute the `toString` method of a class called `Object`. The two calls of `println` would then produce output that is something like:

```
Date@80cb419
Date@80cb419
```

This is the name of the class, followed by an `@`, followed by the *hashcode* of the object (given in the hexadecimal notation).

One of the key aspects of an object-oriented programming language such as Java is *inheritance*. This is a topic which will be described later. What we need to know at this stage is that a class is *derived*, by default, from a class called `Object` (belonging to the package `java.lang`). It is said to be a *subclass* of the class `Object`. This means that, if a program applies a method to an object, and the class of the object does not provide the method, but it is provided by the class `Object`, then `Object`'s method will be called. The WWW page `javaapi:java/lang/Object.html` contains a list of the methods provided by the class `Object`: they are `clone`, `equals`, `finalize`, `getClass`, `hashCode`, `notify`, `notifyAll`, `toString` and `wait`.

On the WWW page `javaapi:java/lang/Object.html#toString()`, it says: ‘ In general, the `toString` method returns a string that *textually represents* this object. The result should be a concise but informative representation that is easy for a person to read. It is recommended that all subclasses override this method. ’ And this is exactly what is happening above: the definition of `toString` given in the class declaration for `Date` *overrides* the one given in `Object`.

4.9 Stage D: providing class variables, class methods and class constants

If a class declaration includes a field, then every object that is of this class will include this field. Such a field is called an *instance variable*. It is also possible to have a field that is associated with the class rather than with each object of the class. Such a field is called a *class variable*, and it is indicated by using a `static` modifier.

A superficial example would be a class declaration that has a field that is used to count how many times methods of the class have been called:

```
private static int tNumberOfCalls = 0;
```

In order for this to work, we would need to add the statement:

```
tNumberOfCalls++:
```

to each of the methods of the class. Here is such a class declaration:

```
// A class for representing values that are dates.
// Barry Cornelius, 20th September 1999
import java.util. StringTokenizer;
public class Date
{
   private static int iNumberOfCalls = 0;
   private int iYear;
   private int iMonth;
   private int iDay;
   public static int getNumberOfCalls()
   {
      return iNumberOfCalls;
   }
   public Date(final int pYear, final int pMonth, final int pDay)
   {
      iNumberOfCalls++;
      iYear = pYear;
      iMonth = pMonth;
      iDay = pDay;
   }
   public int getYear()
   {
      iNumberOfCalls++;
      return iYear;
   }
   ...
   public void setYear(final int pYear)
   {
      iNumberOfCalls++;
      iYear = pYear;
   }
   ...
   public String toString()
   {
      iNumberOfCalls++;
      return iYear + "-" + iMonth/10 + iMonth%10 + "-" + iDay/10 + iDay%10;
   }
}
```

A method that can be applied to any object of its class is called an *instance method*. It is also possible for a class to have standalone methods: such a method is called a *class method*.

So we could add to the `Date` declaration a method that returns the value of the class variable `iNumberOfCalls`, i.e., that returns the number of times methods of the class have been called. Such a method declaration appears in the above class declaration. Note that it is easy to detect a class method because its declaration includes `static`.

A class method is called by putting the name of the class on the left of the dot. An example of the call of `getNumberOfCalls()` is shown in the following version of the `NoelProg` program. It outputs the value 7:

```
// This program creates objects of the version of the       // NoelProg.java
// class Date that keeps track of the number of calls of its methods.
public class NoelProg
{
   public static void main(final String[] pArgs)
   {
      final Date tNoelDate = new Date(1999, 12, 25);      // 1
      final int tDay = tNoelDate.getDay();                // 2
      tNoelDate.setDay(tDay + 1);                         // 3
      System.out.println(tNoelDate.toString());           // 4
      System.out.println(tNoelDate);                      // 5
      final Date tAnotherDate = new Date(2000, 12, 25);   // 6
      System.out.println(tAnotherDate);                   // 7
      System.out.println("number of calls is: " + Date.getNumberOfCalls());
   }
}

```

As a class method is not applied to an instance of a class, it does not make sense to refer to non-static members (e.g., `iDay` and `toString`) in the block of the method of a class method (e.g., `getNumberOfCalls`). Any attempt to do this produces a compilation error like *Can't make a static reference to nonstatic variable iDay in class Date*.

If it is appropriate for a class to have a constant associated with it, then you can use a class variable whose declaration includes the `final` modifier. For example, the class `java.lang.Math` includes:

```
public static final double PI = 3.14159265358979323846;
```

4.10 Stage E: the final version of the `Date` class

4.10.1 Stage E1: the text of the final version of the `Date` class

The final version of the class declaration for `Date` is given below. It contains a number of new features.

```
// A class for representing values that are dates.             // Date.java
import java.util. StringTokenizer;
public class Date
{
   private int iYear;
   private int iMonth;
   private int iDay;
   public Date()
   {
      this(0, 0, 0);
   }
   public Date(final Date pDate)
   {
      this(pDate.iYear, pDate.iMonth, pDate.iDay);
   }
   public Date(final int pYear, final int pMonth, final int pDay)
   {
      iYear = pYear;  iMonth = pMonth;  iDay = pDay;
   }
   public Date(final String pDateString)
   {
      try
      {
         final StringTokenizer tTokens = new StringTokenizer(pDateString, "-");
         final String tYearString = tTokens.nextToken();
         iYear = Integer.parseInt(tYearString);
         final String tMonthString = tTokens.nextToken();
         iMonth = Integer.parseInt(tMonthString);
         final String tDayString = tTokens.nextToken();
         iDay = Integer.parseInt(tDayString);
      }
      catch(Exception pException)
      {
         iYear = 0;  iMonth = 0;  iDay = 0;
      }
   }
   public int getYear()
   {
      return iYear;
   }
   public int getMonth()
   {
      return iMonth;
   }
   public int getDay()
   {
      return iDay;
   }
   public void setYear(final int pYear)
   {
      iYear = pYear;
   }
   public void setMonth(final int pMonth)
   {
      iMonth = pMonth;
   }
   public void setDay(final int pDay)
   {
      iDay = pDay;
   }
   public boolean equals(final Object pObject)
   {
      if ( ! (pObject instanceof Date) )
      {
         return false;
      }
      return iYear==((Date)pObject).iYear &&
             iMonth==((Date)pObject).iMonth &&
             iDay==((Date)pObject).iDay;
   }
   public int hashCode()
   {
      return iYear*416 + iMonth*32 + iDay;
   }
   public String toString()
   {
      return iYear + "-" + iMonth/10 + iMonth%10 + "-" + iDay/10 + iDay%10;
   }
}
```

4.10.2 Stage E2: providing other constructors

Earlier, we saw that a class declaration can have several method declarations each having the same name provided the types of the parameters of each declaration are different (*method overloading*). In the same way, a class declaration can provide several constructors so long as the types of the parameters of each constructor are different.

Although we have a means of outputting the value of a date object, we currently have no means of reading a textual representation of a date from the keyboard or from a file. Obviously, we could use `readLine` to read a textual representation of a date and store it in a string. What we then need is a way of parsing the string and forming an appropriate `Date` object. The above class declaration contains a constructor that can be used to initialize a new `Date` object from a string (as well as one for initializing a date from three integers). It could be used as follows:

```
Date tTodaysDate = new Date("1999-09-26");
```

The above class declaration for `Date` also contains the constructor:

```
public Date()
{
   this(0, 0, 0);
}
```

This constructor is one which has no parameters, and so it would get used for the following declaration:

```
Date tDate = new Date();
```

The body of this constructor contains some magic: the `this` means ‘use the constructor that you will find elsewhere in this class declaration that matches the arguments following the `this`’. So since the `0, 0, 0` are three `int`s then the `this(0, 0, 0)` leads to using the constructor that has three `int`s as parameters. If you use `this` in this way, the `this` statement must appear as the first statement of the constructor.

By this means, you can provide a constructor that has no arguments in order to generate an object with default values (that you can choose).

Earlier it was mentioned that the classes of the Core APIs use two different ways of producing a copy of an object:

- a class sometimes provides a method called `clone`;
- a class sometimes provides a suitable constructor.

Getting the code of a `clone` method completely right is difficult: instead, the above class declaration provides `Date` with a constructor that can be used for cloning.

Here is an example of how this constructor can be used. If we first do something like:

```
final Date tNoelDate = new Date(1999, 12, 25);
```

we can later pass this reference variable as an argument to the new constructor:

```
final Date tHappyDate = new Date(tNoelDate);
```

We finish up with two reference variables pointing to two different objects that have the same value.

4.10.3 Stage E3: defining a method called `equals`

If a client uses `Date` variables, then using == on these variables only determines whether they are pointing to the same object. However, if a class declares an appropriate method called `equals`, then a client can determine whether the dates are the same. So, when you declare a class, it is important to declare a method called `equals`. This is done in the class declaration for `Date` that is given above.

If we declare a class and fail to declare a method called `equals`, then `equals` can still be applied to an object of the class because the class `java.lang.Object` has a method called `equals`. However, `Object`'s `equals` will just tell you whether the target and the argument point to the same objects (i.e., it does the same as ==): it will not compare the values of the two objects.

The above declaration of `equals` has the following header:

```
public boolean equals(final Object pObject);
```

rather than:

```
public boolean equals(final Date pDate);
```

So why would you want to work with a parameter of type `Object`? Well, one important reason will occur when we look at forming collections of data: for example, we may be wanting to represent a collection of dates, a collection of strings, a collection of points, and so on. Java has a number of useful classes that can be used to manipulate collections. So as to make these classes generally useful the methods of these classes are written in terms of the type `Object`. For example:

```
public boolean add(Object pObject);
public    void add(int pIndex, Object pObject);
public boolean contains(Object pObject);
public  Object get(int pIndex);
public boolean remove(Object pObject);
public  Object remove(int pIndex);
```

are methods that can be used to perform operations on one kind of collection (a *list*). These collection classes are discussed in ITS Guide 108 *Advanced Java*.

These collection classes are wonderful because they allow us to create dynamically growing collections of objects. But the person who wrote the code of the methods of these collection classes was not in a position to know what sort of objects you would be storing in a collection. When you call a method like `contains` (that finds out whether an object is in the collection), behind the scenes `contains` will call a method with the header:

```
public boolean equals(Object pObject);
```

Now, if you are storing `Date`s in the collection, and if in the class `Date` you have declared `equals` with a parameter of type `Date`:

```
public boolean equals(final Date pDate);
```

then this `equals` will not be called (because the type of the parameter is different). Instead, the method called `equals` from the class `Object` will be called: as explained earlier this returns true if and only if the target and the argument point to the same object (and not if the two objects have the same values). This would be an inappropriate method to be used by `contains`. So instead we declare `equals` with a parameter of type `Object`.

Such a method declaration *overrides* the `public boolean equals(Object pObject)` that is declared in the class `java.lang.Object`.

4.10.4 Stage E4: adding `hashCode` (to help with using collections)

You can imagine that when checking whether a collection contains a particular object it can be quite time-consuming to use `equals` on each of the objects of the collection in turn. Instead, clever techniques are used to reduce the number of items of the collection that need to be checked. Some of these techniques require there to be an integer (called a *hashcode*) associated with each of the possible values that can be stored in the collection. And to speed up the execution of methods like `contains`, the objects in the collection are arranged so that the ones that have the same hashcode are kept together.

In order to support this, the class `java.lang.Object` has a method called `hashCode`:

```
public int hashCode();
```

The integer that is returned is one that is unique for that object.

However, this method is inappropriate when a class declares a method called `equals`. What we need to do is to provide our own version of `hashCode` that *override*s the one of `java.lang.Object`.

The WWW page that documents hashCode says: ‘If two objects are equal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce the same integer result. It is not required that if two objects are unequal according to the equals(Object) method, then calling the hashCode method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hashtables.’

The above class declaration gives one possibility for hashCode. However, there are many other possibilities. Instead, you could get hashCode to return the value of iYear*10000 + iMonth*100 + iDay. Although it would still work if you got it to return the value of iYear + iMonth + iDay, or the value iDay, or the value 42, these will lead to poorer performance as the same integer value is being produced for unequal objects.

4.10.5 Stage E5: using the new version of the Date class

The following program uses some of the facilities of the new class declaration for Date:

```
// This program makes tNoelDate, a variable of              // NoelProg.java
// the class type Date, refer to an object of class Date
// representing Christmas Day 1999.
// It then makes tOtherDate refer to another Date object,
// and then uses both == and equals to compare the two variables.
import java.io. BufferedReader;
import java.io. InputStreamReader;
import java.io. IOException;
public class NoelProg
{
   public static void main(final String[] pArgs) throws IOException
   {
      final Date tNoelDate = new Date(1999, 12, 25);
      System.out.println("tNoelDate is: " + tNoelDate);
      final BufferedReader tKeyboard =
                     new BufferedReader(new InputStreamReader(System.in));
      System.out.print("Type in the date, e.g., 1999-12-25: ");
      System.out.flush();
      final String tOtherDateString = tKeyboard.readLine();
      final Date tOtherDate = new Date(tOtherDateString);
      System.out.println("tOtherDate is: " + tOtherDate);
      System.out.println("tUsingOperator: " + (tNoelDate==tOtherDate));
      System.out.println("tNoelDate.equals: " + tNoelDate.equals(tOtherDate));
      System.out.println("tOtherDate.equals: " + tOtherDate.equals(tNoelDate));
      System.out.println("tNoelDate.iIsLeap: " + iIsLeap(tNoelDate));
      System.out.println("tOtherDate.iIsLeap: " + iIsLeap(tOtherDate));
   }
   private static boolean iIsLeap(final Date pDate)
   {
      int tYear = pDate.getYear();
      return (tYear%400==0) || (tYear%4==0 && tYear%100!=0);
   }
}
```

4.11 The role of a class

You should view a class declaration as being split into two sections. The text of the headings of the public methods (together with the names of any public fields) gives information as to *what* services are offered by the class. The text of the bodies of the public members together with the text of the private members give the details of *how* the services are to be provided.

The class declaration for Date hides three ints and provides four constructors to create a date and a rather formal way of accessing the fields of a date by means of the three access methods getYear, getMonth and getDay. This idea of hiding fields behind access methods is often termed *information hiding* or *encapsulation*.

It may seem like a complicated way of providing an object and accessing it. However, you should view the class declaration as documenting a design decision. At the moment we have chosen to represent a date by three ints. At a later stage, we may feel that that is wrong: for example, we might choose three shorts, three bytes, or a single int giving the number of days since a particular date. Since we have retained control over the access to the fields of the class by providing access methods, we can make changes like this with only minimal impact to the code of the program: we know that the only code that needs to be changed is located in the methods of the class.

5 Another example of data abstraction: the class Person

5.1 A class called Person

Here is a class declaration for a class called Person. It is rather basic as it can be used to store only a name, a date of birth and a height:

```
import java.util.StringTokenizer;                                  // Person.java
public class Person {
   public Person() { this("", 0.0F, new Date()); }
   public Person(String pString) {
      StringTokenizer tTokens = new StringTokenizer(pString, "%");
      oName = tTokens.nextToken();
      iHeight = Float.valueOf(tTokens.nextToken()).floatValue();
      oDateOfBirth = new Date(tTokens.nextToken());
   }
   public Person(String pName, float pHeight, Date pDate) {
      oName = pName; iHeight = pHeight; oDateOfBirth = pDate;
   }
   public String getName() { return oName; }
   public boolean equals(Object pObject) {
      return oName.equals(((Person) pObject).oName);
   }
   public String toString() {
      return oName + "%" + iHeight + "%" + oDateOfBirth;
   }
   protected String oName;
   private float iHeight;
   protected Date oDateOfBirth;
}
```

Although this class declaration has `protected` fields, for the time being treat the `protected` fields as if they were `private` fields. The distinction between the two will be discussed later.

5.2 Using the class `Person`

And here is a Java application that tests some aspects of the class `Person`:

```
import java.io.BufferedReader;                             // UsePerson.java
import java.io.InputStreamReader;
import java.io.IOException;
public class UsePerson {
   public static void main(String[ ] args) throws IOException {
      BufferedReader input =
            new BufferedReader(new InputStreamReader(System.in));
      Person tGirlfriend =
            new Person("Smith", 5.5F, new Date(1973, 2, 27));
      System.out.print("Girlfriend> ");  System.out.println(tGirlfriend);
      Person tWife = new Person();
      System.out.print("Wife> ");  System.out.println(tWife);
      Person tBaby;
      tWife = tGirlfriend;
      System.out.print("Wife> ");  System.out.println(tWife);
      tBaby = new Person(input.readLine());
      System.out.print("Baby> ");  System.out.println(tBaby);
      System.out.println(tWife.getName().equals(tBaby.getName()));
   }
}
```

When this program is executed, it produces output like:

```
Girlfriend> Smith%5.5%1973-02-27
Wife> %0.0%0-00-00
Wife> Smith%5.5%1973-02-27
Smith%1.5%1990-4-9
Baby> Smith%1.5%1990-04-09
true
```

6 Grouping classes into packages

6.1 Package declarations

By default, a class/interface declaration belongs to the *default package*. And `.class` files are stored in the current directory.

It is useful to be able to group related classes/interfaces together. And for this, Java has the concept of a *package*.

You can use a *package declaration* to indicate that a class/interface belongs to a particular package. For example, suppose you have a file containing the text of a class called `Date` and that you want it to belong to a package called `dateutils`. You just need to insert a package declaration at the start of the file:

```
package dateutils;
```

Any class/interface declaration that contains this line belongs to this package. The `.class` files associated with these files of source code must appear in a directory called `dateutils`. And any client that wishes to use this class could use an *import declaration*, such as:

```
import dateutils.Date;
```

If instead some class/interface declarations each have a package declaration that takes the form:

```
package utils.dateutils;
```

then the `.class` files should be in a subdirectory called `dateutils` that is itself in a directory called `utils`. Any client that wishes to use the class called `Date` belonging to this package could use the import declaration:

```
import utils.dateutils.Date;
```

6.2 Setting the CLASSPATH

When the Java compiler/interpreter is executed, it looks for any packages in the directories that are mentioned in the *CLASSPATH*. By default, the CLASSPATH is empty, and if this is the case it will instead look for packages in the current directory.

So, if you have put some class/interface declarations into a package called `utils.dateutils`, the `utils` directory must be a subdirectory of the current directory.

Although the use of a subdirectory of the current directory is a useful place to hide the files of a package, this mechanism can be too restrictive. For example, if you build a number of useful classes and store them in one or more packages, it would be useful to put these in a standard place. The Java compiler/interpreter allows you to specify other directories in which it can find packages by setting the CLASSPATH.

The way in which this is done depends on whether you are using a Unix or an MS-DOS command line. When using Unix, then for csh/tcsh, an example is:

```
setenv CLASSPATH .:/users/dcl0bjc/classes:/users/dxy3abc/public_html
```

or, if you are using sh, ksh or bash, this would be:

```
CLASSPATH=.:/users/dcl0bjc/classes:/users/dxy3abc/public_html
export CLASSPATH
```

At an MS-DOS prompt, you could type something like:

```
set CLASSPATH=.;C:\project\classes;D:\myjava
```

Note that for Unix, items in the list of directories are separated by a colon, whereas the semicolon is used at an MS-DOS prompt. If you also want the compiler/interpreter to look in the current directory, then it must be included in the CLASSPATH: a dot can be used in the CLASSPATH in order to refer to this directory.

The Java compiler/interpreter knows how to find the packages that form part of Java's Development Kit, and so there is no need to include anything in the CLASSPATH to help the compiler/interpreter find these packages. Note: this was not the case with earlier versions of the Development Kit.

6.3 It's a small world: how can unique names be generated?

Java specifies a convention for generating globally unique names for classes/interfaces. The convention is that a package name starts with the components of the author's Internet address (in reverse order). Examples are:

organization	domain name	an example of a class/interface name
Sun Microsystems	sun.com	com.sun.xxx.yyyyy
IBM	ibm.com	com.ibm.wwww.vvv.uuuuu
University of Durham	dur.ac.uk	uk.ac.dur.aaaaa.bbb.ccc.dddd

If the University of Durham wanted to establish a convention for the uniqueness of the names of classes/interfaces, it could utilise a person's username. So I might prefer to put the `Date` class mentioned earlier into the package:

```
package uk.ac.dur.dcl0bjc.utils.dateutils;
```

And if I wanted this package to be accessible from the WWW (see below), then it would be sensible to put the files of this package into the directory:

```
/users/dcl0bjc/public_html/uk/ac/dur/dcl0bjc/utils/dateutils
```

Having done this, if you needed to use this package in a Java application, you would need something like:

```
import uk.ac.dur.dcl0bjc.utils.dateutils.Date;
```

and you would need to set the CLASSPATH:

```
setenv CLASSPATH .:/users/dcl0bjc/public_html
```

in order for the Java compiler/interpreter to find the `.class` files.

The reason for putting these `.class` files in a directory that is below a user's `public_html` directory is that they can then be accessed by a Java applet running on a WWW browser elsewhere in the world. You can use the `CODEBASE` attribute of an `APPLET`/`OBJECT`/`EMBED` tag if you want to indicate that an applet's `.class` files are all stored in a particular place, e.g.:

```
CODEBASE=http://www.dur.ac.uk/barry,cornelius/
```

6.4 Compiling from a private directory into one that is visible from the WWW

Although it may be useful to put your `.class` files into a publically accessible place, you may want to hide the source files. So, if the current directory contains some Java source code and the current directory is inaccessible from the WWW, you can easily arrange for the Java compiler to put the `.class` files into a different directory (a directory that is accessible from the WWW) by using the `d` option of the `javac` command, e.g.:

```
setenv CLASSPATH .:/users/dcl0bjc/public_html
javac -d /users/dcl0bjc/public_html Date.java
```

If the file `Date.java` contains the line:

```
package uk.ac.dur.dcl0bjc.utils.dateutils;
```

then the directory `/users/dcl0bjc/public_html/uk/ac/dur/dcl0bjc/utils/dateutils` will be used by the compiler to store the `Date.class` file. If need be, it will automatically create any directories that do not exist.

7 Object-oriented programming

7.1 Introduction

We have seen that classes can be used to describe objects existing in the problems that you wish to solve. Although the programs we have written have used objects, many people view this as just *object-based programming*: you need to use both *inheritance* and *dynamic binding* before you are doing *object-oriented programming*. These two topics form the main thrust of this section.

7.2 Using inheritance to form a subclass

So far, the classes we have produced have been for objects that are distinct from one another: a date is nothing like a person, and vice-versa. However, there will be occasions when a new class is in fact a more specialized form of another class.

For example, if we now have to produce a program that manipulates data about students, we will need a class to represent a student. Such a class will have a lot in common with the class representing a person which we have already produced. Instead of producing a completely new class for a student, we can derive the `Student` class from the `Person` class:

```
public class Student extends Person {
   ...
}
```

This is called *inheritance*: the class `Student` is said to *inherit* from the class `Person`: the class `Student` is the *subclass* and the class `Person` is the *superclass*.

Note: you cannot derive a subclass from a class that has the modifier `final`, for example:

```
public final class String { ... }
```

Note: unlike C++, in Java, you cannot derive a class from more than one class, i.e., Java does not have *multiple inheritance*.

7.3 A class called `Student`

Here is a class declaration for the class `Student`:

```
import java.util.StringTokenizer;                            // Student.java
public class Student extends Person {
   public Student(String pName, float pHeight, Date pDateOfBirth,
                  String pCourseName, int pStudentNumber) {
      super(pName, pHeight, pDateOfBirth);
      iCourseName = pCourseName; iStudentNumber = pStudentNumber;
   }
   public int getStudentNumber() { return iStudentNumber; }
   public boolean equals(Object pObject) {
      return oName.equals(((Student) pObject).oName);
   }
   public String toString() {
      return oName + "=" + oDateOfBirth
                   + "=" + iCourseName + "=" + iStudentNumber;
   }
   private String iCourseName;
   private int iStudentNumber;
}
```

Suppose you declare an object to be of the subclass:

```
Student tStudent = new Student( ... );
```

As well as having the members of the subclass, the object has all the members of the superclass. So, the object `tStudent` has the members:

- `Student`, `getStudentNumber`, `equals`, `toString`,
  `iCourseName`, `iStudentNumber`

from the class `Student` and the following members:

- `Person`, `Person`, `Person`, `getName`, `equals`, `toString`,
  `oName`, `iHeight`, `oDateOfBirth`

from the class `Person`.

So an object of the class `Student` has five fields that are called `oName`, `iHeight`, `oDateOfBirth`, `iCourseName` and `iStudentNumber`. The constructor for the class `Student` has arguments that are used to initialize not only the fields of the class `Student` but also the fields from the class `Person`. In the body of the constructor, a special method called `super` is used to initialize the fields of the superclass (`Person`).

Because each of these five fields is declared to be `private` or `protected`, they are inaccessible to a client of the class `Student`. However, a client can use any public members of the class or any public members of the superclass. Examples are:

```
System.out.println(tStudent.getName());
System.out.println(tStudent.getStudentNumber());
```

7.4 Package members and `protected` members

Previously, we have declared members of classes to be either `public` or `private`. We look now at what it means for a member to have a `protected` modifier or to have no modifier at all.

If a member of a class has no modifier at all, it can be accessed by the code of any class within the same package. Such a member is sometimes called a *package member*.

If a member of a class has a `protected` modifier, it can be accessed by the code of any class within the same package or by the code of any subclass (whether or not it is in the same package).

So, the code of any method of a subclass may access any `public` and `protected` members of a superclass. Consider the class `Person` again. If we want some members of the class `Person` to be accessible in `Person` and in any subclass of `Person` but generally to be inaccessible, then those members can be `protected` members of the class `Person`. However, if we want a member of the class `Person` to be inaccessible in the code of the subclass, then it needs to be a `private` member of the class `Person`.

The class declaration for `Person` has `oName` and `oDateOfBirth` as `protected` fields and `iHeight` as a `private` field. So the code of a method of the class `Student` is able to access the fields `oName` and `oDateOfBirth` but is unable to access `iHeight`.

Some people argue that it is inappropriate for a subclass to be able to access fields of its superclass: they would argue that it is better for these fields to be `private` and for the superclass to provide `public` methods to access them.

7.5 Method overriding

You can give a method of a subclass the same name as a method of the superclass. This often occurs when more appropriate code can be devised for the method of the subclass. This is called method *overriding*. Note: you cannot override a method that has the modifier `final` (or `static` or `private`) in the superclass.

There are two examples of this with the `Person` and `Student` classes: the methods `equals` and `toString` appear in both the superclass `Person` and the subclass `Student`.

Method *overriding* should not be confused with method *overloading* which was introduced earlier.

7.6 Using the class `Student`

Here is a Java application that tests some aspects of the class `Student`:

```
public class UseStudent {                                  // UseStudent.java
   public static void main(String[ ] args) {
      Person tPerson = new Person("Jones", 1.6F, new Date(1969,12,25));
      System.out.println(tPerson);
      Student tStudent =
         new Student("Smith", 1.85F, new Date(1970,6,12), "Computing", 27);
      System.out.println(tStudent);
      tPerson = tStudent;
      System.out.println(tPerson);
   }
}
```

7.7 Dynamic binding

So far, a reference variable of the type `Person` has been given values that causes it to refer to a `Person` object. However, a reference variable can be given a value that causes it to refer to an object of its class or *any subclass of that class*. For example, in the `UseStudent` program, the variable `tPerson` is first made to refer to an object of class `Person`, but, at the end of the program, it is made to refer to an object of class `Student`.

So, suppose you have written a method:

```
public void task(Person pPerson) {
   ...
}
```

The code of the method `task` is written in terms of the variable `pPerson`. We can pass as an argument to `task` an object that is of class `Person` or an object that is of any subclass of `Person`. If the code of `task` calls a method and this method is one that has been overridden in the subclass, then the actual method that is called will depend on what kind of object has been passed to `task`. For example, if `task` calls `equals` then `Person`'s `equals` method will be called if the object passed as an argument is of class `Person`, whereas `Student`'s `equals` method will be called if the object passed as an argument is of class `Student`. So the actual version of the `equals` method that will be called is unknown until runtime: it depends on what kind of object `pPerson` refers to. This is known as *dynamic binding*.

The code of the method `task` will also continue to work if, later, another subclass of `Person` is produced: the code of `task` does not have to be modified every time a new subclass of `Person` is produced.

7.8 Inheritance should be used for is-a relationships

Earlier, we used the class `Date` when constructing the class `Person`, and we have now used the class `Person` when constructing the class `Student`. We used inheritance to produce the class `Student` from the class `Person`, whereas the class `Person` contains a field (called `oDateOfBirth`) of type `Date`. This is called *composition* (or *layering*): the class `Person` is *composed* of a field of type `Date`.

Earlier, we said we used inheritance because the new class 'is a more specialized form of another class'. It is best to use inheritance for is-a relationships and composition for has-a relationships. So, one test for deciding whether to use inheritance or composition is to see whether it makes sense to use the words *is a* or *has a*. For example, 'every person is a date' is nonsense whereas 'every person has a date for his/her date of birth' makes sense.

8 Another example of OO programming: 2D shapes

8.1 The class `Shape`

We can use the following class for objects that are two-dimensional geometrical figures. The class includes a constructor to create an object representing a shape at some position in two-dimensional space. It also includes a method called `translate` that moves a shape to a new position relative to its current position.

```
public class Shape {                                            // Shape.java
   public Shape(int vX, int vY) {
      iX = vX; iY = vY;
   }
   public Shape() { this(0, 0); }
   public int getX() { return iX; }
   public int getY() { return iY; }
   public void translate(int vX, int vY) { iX += vX; iY += vY; }
   public boolean equals(Object rObject) {
      return iX == ((Shape) rObject).iX && iY == ((Shape) rObject).iY;
   }
   public String toString() { return iX + ":" + iY; }
   private int iX, iY;
}
```

8.2 The class `Circle`

Suppose we now want a class `Circle` to represent shapes that are circles. We can create this class by inheritance from the class `Shape` as follows:

```
public class Circle extends Shape {                            // Circle.java
   public Circle(int vRadius, int vX, int vY) {
      super(vX, vY);  iRadius = vRadius;
   }
   public Circle() { this(0, 0, 0); }
   public int getRadius() { return iRadius; }
   public boolean equals(Object rObject) {
      return super.equals(rObject) && iRadius == ((Circle)rObject).iRadius;
   }
   public String toString() { return super.toString() + ":" + iRadius; }
   private int iRadius;
}
```

Objects of this class have three fields `iX`, `iY` and `iRadius`. Once again, the constructor for this class uses the special method called `super`:

```
super(vX, vY);
```

in order to initialize the `iX` and `iY` fields (with the values that are passed through `vX` and `vY`).

The bodies of the `equals` and the `toString` methods show a different use of the `super` keyword. In these methods, it appears as `super.`*methodname*`(...)`. This notation means: ‘apply the method *methodname* as defined in the superclass to the current object’.

In this example, instead of using `super.`*methodname*`(...)`, the `getX` and `getY` methods of the superclass could be used. For example, `toString` could be declared as:

```
public String toString() { return getX() + ":" + getY() + ":" + iRadius; }
```

Note that:

```
public String toString() { return iX + ":" + iY + ":" + iRadius; }
```

would not be possible unless the `iX` and `iY` fields of the class `Shape` were changed from `private` fields to `protected` fields.

8.3 The class `Rectangle`

In a similar way, the class `Rectangle` can also be built from the class `Shape`:

```
public class Rectangle extends Shape {                     // Rectangle.java
   public Rectangle(int vWidth, int vHeight, int vX, int vY) {
      super(vX, vY);  iWidth = vWidth;  iHeight = vHeight;
   }
   public Rectangle() { this(0, 0, 0, 0); }
   public int getWidth() { return iWidth; }
   public int getHeight() { return iHeight; }
   public boolean equals(Object rObject) {
      return super.equals(rObject)
             && iWidth == ((Rectangle) rObject).iWidth
             && iHeight == ((Rectangle) rObject).iHeight;
   }
   public String toString() { return super.toString()
                                     + ":" + iWidth + ":" + iHeight; }
   private int iWidth;
   private int iHeight;
}
```

8.4 Using the class `Shape` and its subclasses

The following program uses the classes `Shape`, `Circle` and `Rectangle`. It reads some data describing some shapes from a file called `data`. The file could contain the values: 4, 2, 100, 200, 30, 50, 1, 150, 200, 30, 2, 200, 200, 50, 80, 1, 250, 200 and 40 (where each value is on a separate line of the file). This data is meant to be interpreted as follows: there are four shapes; the first one is a rectangle with an x-coordinate of 100, a y-coordinate of 200, a width of 30, a height of 50; the second shape is a circle with an x-coordinate of 150, a y-coordinate of 200, a radius of 30; and so on.

```
import java.io.BufferedReader;                          // FileToScreen.java
import java.io.FileReader;
import java.io.IOException;
public class FileToScreen {
   public static void main(String[ ] args) throws IOException {
      BufferedReader input =
            new BufferedReader(new FileReader("data"));
      String line = input.readLine();
      int numShapes = Integer.parseInt(line);
      Shape[] shapes = new Shape[numShapes];
      for (int shapeNumber = 0; shapeNumber<numShapes; shapeNumber++) {
         line = input.readLine();  int shape = Integer.parseInt(line);
         line = input.readLine();  int x = Integer.parseInt(line);
         line = input.readLine();  int y = Integer.parseInt(line);
         switch ( shape ) {
            case 1:
               line = input.readLine();
               int radius = Integer.parseInt(line);
               shapes[shapeNumber] = new Circle(radius, x, y);
               break;
            case 2:
               line = input.readLine();
               int width = Integer.parseInt(line);
               line = input.readLine();
0663:                int height = Integer.parseInt(line);
0664:                shapes[shapeNumber] = new Rectangle(width, height, x, y);
0665:                break;
0666:          }
0667:       }
0668:       for (int shapeNumber = 0; shapeNumber<numShapes; shapeNumber++) {
0669:          Shape tShape = shapes[shapeNumber];
0670:          tShape.translate(1, 2);
0671:          System.out.println(tShape);
0672:       }
0673:    }
0674: }
```

The program stores the details about the shapes in an array called `shapes`:

```
0648:       Shape[] shapes = new Shape[numShapes];
```

and uses the following two statements to put values into the array:

```
0657:                shapes[shapeNumber] = new Circle(radius, x, y);

0664:                shapes[shapeNumber] = new Rectangle(width, height, x, y);
```

So the program does not use the array to store any references to objects of the class `Shape`: instead, each element is either a reference to a `Circle` object or a reference to a `Rectangle` object.

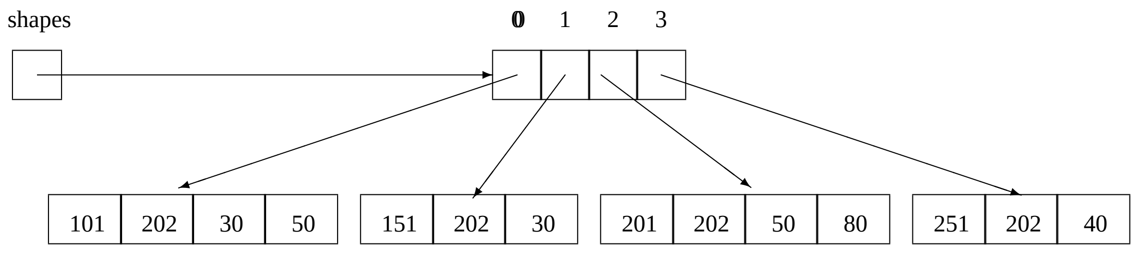
At the end of the program, there is a `for` statement whose aim is to output the details about the shapes that have been stored. It repeatedly executes the following three statements:

```
0669:          Shape tShape = shapes[shapeNumber];
0670:          tShape.translate(1, 2);
0671:          System.out.println(tShape);
```

In the first of these, the variable `tShape` is made to refer to either a `Circle` object or a `Rectangle` object. Then the `translate` method is applied to the object. Because neither `Circle` nor `Rectangle` declare a `translate` method, it will be the `translate` method of the superclass (`Shape`) that will be used. Finally, the `println` statement will use either `Circle`'s or `Rectangle`'s `toString` method in order to print the shape referred to by `tShape`. This is another example of dynamic binding.

When the program is executed with the above data, this `for` statement produces the following output:

```
101:202:30:50
151:202:30
201:202:50:80
251:202:40
```



9 Exception handling

9.1 What is exception handling?

A method often detects situations which it knows it cannot handle. It may be that the arguments for the method were inappropriate; it may be that a series of calculations has led to a situation that should not occur; it may be that its attempt to allocate space using `new` has failed; and so on. What should the programmer of this method do when such untoward events (*exceptions*) arise?

The method could output an error message and then terminate execution. However, the user of the method might be extremely unhappy if this happens: he/she might want to do some 'cleanup' code before the program terminates.

Instead, the programmer of the method could return some value that signifies that an error has occurred. However, returning an error value may be inconvenient to the user of the method as the point of call of the method may not be the best place to handle the error. So his/her code has to be littered with error-handling code.

Some programming languages allow the code of the method to signify that an exception has occurred and this is then handled by some code that occurs elsewhere in the program. In Java, a *try statement* consists of a *try block* together with zero or more *exception handler*s (each introduced by the keyword `catch`) and an optional *finally clause*:

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

A try statement can be used to indicate that a piece of code wishes to handle exceptions. In the code executed by the try block, a *throw statement* is used to signify that an exception has occurred. When a throw statement is executed, control is transferred to the exception handler of the most recently entered try statement containing an appropriate exception handler. It is possible to write an exception handler that handles all exceptions, and to write one that re-throws an exception.

How does a try statement end? If an exception occurs, the last-statement-to-be-executed will be in an exception handler; otherwise, it will be in the try block. The last-statement-to-be-executed may be a statement that causes a transfer of control (such as a `return`, `continue` or a `break` statement) or it may the statement that appears at the end of the exception handler or the try block. If a try statement has a finally clause, the statements of the finally clause will then be executed. If the last-statement-to-be-executed is one that causes a transfer of control, the finally clause will be executed before control is actually transferred to its new destination. So, if a try statement has a finally clause, it will always be executed.

Because a finally clause provides a way of guaranteeing that some code will be executed before a block is left, it is sometimes useful to write try statements that have a finally clause but do not have any exception handlers.

9.2 Altering `Date` to deal with invalid dates

The `Date` class given earlier can be modified to deal with invalid dates in the following way:

```
// A class for representing values that are dates.
// Barry Cornelius, 20th September 1999
import java.util. StringTokenizer;
public class Date
{
   private int iYear;
   private int iMonth;
   private int iDay;
   ...
   public Date(final int pYear, final int pMonth, final int pDay)
          throws InvalidDateException
   {
      iYear = pYear;  iMonth = pMonth;  iDay = pDay;
      iCheckDate();
   }
   ...
   public int getYear()
   {
      return iYear;
   }
   ...
   public void setYear(final int pYear)
          throws InvalidDateException
   {
      iYear = pYear;
      iCheckDate();
   }
   ...
   private void iCheckDate()
          throws InvalidDateException
   {
      if (iYear<1900 || iYear>2100 ||
          iMonth>12 || iDay>31)
      {
         throw new InvalidDateException();
      }
   }
}
```

The `Date` class requires a file containing the following supporting class:

```
//                                              // InvalidDateException.java
public class InvalidDateException extends Exception {
   public InvalidDateException() {
      super();
   }
}
```

The following version of the `NoelProg` program contains some code that catches the exceptions caused by inappropriate uses of the constructors and methods of this new version of the `Date` class:

```
...
public class NoelProg
{
   public static void main(final String[] pArgs)
          throws InvalidDateException,IOException
   {
      final Date tNoelDate = new Date(1999, 12, 25);
      System.out.println("tNoelDate is: " + tNoelDate);
      final BufferedReader tKeyboard =
                    new BufferedReader(new InputStreamReader(System.in));
      Date tOtherDate = new Date();
      while (true)
      {
         System.out.println("Type in the date, e.g., 1999-12-25");
         final String tOtherDateString = tKeyboard.readLine();
         try {
            tOtherDate = new Date(tOtherDateString);
            break;
         }
         catch(InvalidDateException pInvalidDateException) {
            System.out.println("Invalid date");
         }
      }
      System.out.println("tOtherDate is: " + tOtherDate);
      ...
   }
}
```

10 Interfaces

10.1 What is an interface?

Earlier, in *The role of a class*, it was suggested that, when looking at a class declaration, you should distinguish between the text that describes *what* services are offered and the text that describes *how* these services are provided. The *what* describes the interface whereas the *how* describes the implementation. Java allows us to document the *what* by means of a construct called an *interface*.

So, in Java, an interface is a construct that gives a list of related methods (and/or constants). Here is an example that lists a set of methods for manipulating a date:

```
public interface DateIF {
   public int getYear();
   public int getMonth();
   public int getDay();
   public void setYear(int pYear);
   public void setMonth(int pMonth);
   public void setDay(int pDay);
   public boolean equals(Object pObject);
   public int hashCode();
   public String toString();
}
```

With a class, we use `new` and a constructor (i.e., a class instance creation expression) to create an object, an instance of the class. It does not make sense to create an instance of an interface (and for this reason an interface does not have a constructor).

10.2 Producing classes that conform to an interface

Instead, the purpose of Java's `interface` construct is to describe the interface to which a set of classes conform, i.e., each class implements the interface.

For example, there are several ways of providing a class for representing a date each of which stores the details of a date in a different way: we could use three `int`s representing a year, a month and a day; a `short` and two `byte`s representing a year, a month and a day; one `int` that stores the number of days since the *beginning of time*; and so on. So we could provide several classes, each one of which conforms to the `DateIF` interface.

We should document that a class implements an interface. This is done by means of an `implements` clause:

```
import java.util. StringTokenizer;
public class Date implements DateIF
{
   private int iYear;
   ...
   public String toString()
   {
      return iYear + "-" + iMonth/10 + iMonth%10 + "-" + iDay/10 + iDay%10;
   }
}
```

If `Date` says that it `implements DateIF`, then `Date` must at least include declarations for each method that is defined in `DateIF`.

10.3 Using interfaces

Suppose that, besides `Date` we have also created other classes that implement the `DateIF` interface, e.g.:

```
public class DateA implements DateIF { ... }
public class DateB implements DateIF { ... }
```

If the code of some client needs to refer to a class (e.g., a parameter of some method is a date), then it should use the name of the interface (e.g., `DateIF`) rather than the name of a class (e.g., `Date`). For example:

```
private static boolean iIsLeap(final DateIF pDateIF) {
   final int tYear = pDateIF.getYear();
   return (tYear%400==0) || (tYear%4==0 && tYear%100!=0);
}
```

The only time when the code of a client needs to refer to one of the classes is when it wants to create a date. Then it has to choose which implementation to use, e.g., to choose between one of the following statements:

```
DateIF tDateIF = new Date();
DateIF tDateIF = new DateA();
DateIF tDateIF = new DateB();
```

The variable `tDateIF` is a reference variable that is of an interface type. It can refer to any object which is of a class that implements the interface.

10.4 Other points

Here are some other points:

- Only constants (i.e., variables that are `static final`) and methods can be listed in an interface.
- An interface can be derived from another interface or from several interfaces:

  ```
  interface a extends b, c { ... }
  ```

- A class can implement more than one interface:

  ```
  public class x implements y, z { ... }
  ```

- A class can be derived from another class as well as implement one or more interfaces:

  ```
  public class x extends w implements y, z { ... }
  ```

We saw earlier that multiple inheritance is not possible in Java. An interface is Java's way of achieving some of the possibilities of multiple inheritance. It is an idea that is borrowed from the programming language Objective C.

11 Starting another thread

11.1 The class `java.lang.Thread`

Often the user of a program does something that causes the program to do some task that is time-consuming. In this situation, you may prefer the user to have the ability to get on with something else at the same time as the time-consuning task. In Java, you can put the time-consuming task into a separate *thread* of execution.

It is easy to start another thread: you just need to create an object of the class `Thread` (from the package `java.lang`), and execute its `start` method. So, suppose a method, e.g., the `main` method of a program, contains:

```
Thread tThread = new Thread();
...
tThread.start();
```

The the call of `start` does two things:

- it starts the execution of `tThread.run()` in a separate thread;
- it then immediately returns to the statement following the call of `tThread.start()`.

So we now have two threads of activity that are running concurrently: the `main` method and the `tThread.run` method.

This is not so exciting as it sounds because `java.lang.Thread`'s `run` method does nothing because it has a null body: it stops executing straightaway. And so we are just left with the thread of execution that is executing the `main` method.

11.2 Deriving the class `ClockStdout` from `java.lang.Thread`

However, because Java has inheritance, we can derive a class from `java.lang.Thread` and provide a `run` method that does something useful.

In the code below, a class called `ClockStdout` is derived from `java.lang.Thread`, and `ClockStdout`'s declaration overrides `Thread`'s `run` method. The code of `ClockStdout`'s `run` method is an infinite loop inside which we first get the current date and time, then output that to the *standard output*, and then wait for two seconds.

```
import java.util.Date;                                  // ClockStdout.java
public class ClockStdout extends Thread {
   public void run() {
      while ( true ) {
         Date tDate = new Date();
         System.out.println(tDate);
         try { Thread.sleep(2000); }
         catch ( InterruptedException tInterruptedException ) { }
      }
   }
}
```

11.3 Using the class `ClockStdout` in the `UseClockStdout` program

The `main` method of the `UseClockStdout` program (given below) creates an object (`tClockStdout`) of the class `ClockStdout`, and then calls `tClockStdout`'s `start` method. However, the class `ClockStdout` does not itself declare a `start` method, and so it is `java.lang.Thread`'s `start` method that gets called. As explained earlier, this does two things:

- it causes `tClockStdout`'s `run` method (i.e., the infinite loop) to start executing in a separate thread;
- it then immediately returns to execute the rest of the `main` method.

So we now have two threads of activity that are running concurrently: the `main` method and the `tClockStdout.run` method. Having started the `tClockStdout` thread, the `main` method then goes on to output the digits from 0 to 7 stopping for one second after it has output each digit:

```
public class UseClockStdout {                         // UseClockStdout.java
   public static void main(String[ ] args) {
      System.out.println("UseClockStdout program");
      ClockStdout tClockStdout = new ClockStdout();
      tClockStdout.start();
      for ( int count = 0; count < 8 ; count++ ) {
         System.out.println("count is: " + count);
         try { Thread.sleep(1000); }
         catch ( InterruptedException tInterruptedException ) { }
      }
      System.out.println("UseClockStdout program");
   }
}
```

Here is the sort of output that the program produces:

```
UseClockStdout program
count is: 0
Sat Jun 14 15:49:15 GMT+01:00 1997
count is: 1
count is: 2
Sat Jun 14 15:49:17 GMT+01:00 1997
count is: 3
count is: 4
Sat Jun 14 15:49:19 GMT+01:00 1997
count is: 5
count is: 6
Sat Jun 14 15:49:21 GMT+01:00 1997
count is: 7
UseClockStdout program
Sat Jun 14 15:49:23 GMT+01:00 1997
Sat Jun 14 15:49:25 GMT+01:00 1997
Sat Jun 14 15:49:27 GMT+01:00 1997
Sat Jun 14 15:49:29 GMT+01:00 1997
...
```

You can see that the output is from both threads, and that the program will not finish because the `tClockStdout` thread is an infinite loop. So, if you execute this program, you will need to press *Ctrl/C* to stop its execution.

Two ways of getting the program to terminate properly are:

- The `main` method can terminate the execution of `tClockStdout`'s thread by the call:

```
tClockStdout.stop();
```

- The `main` method can terminate the execution of the program by the call:

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

11.4 Using `synchronized` for accessing a variable from different threads

If you wish to access the same variable from more than one thread, you will need to use the `synchronized` keyword to ensure that accesses to the variable are performed correctly. You can control access either by means of a *synchronized statement* or by using *synchronized method*s. An example of the use of synchronized methods is:

```
public class Store {
   public Store(int vStore) { iStore = vStore; }
   public synchronized int get() { ... return iStore; }
   public synchronized void put(int vStore) { iStore = vStore; ... }
   ...
   private int iStore;
}
```