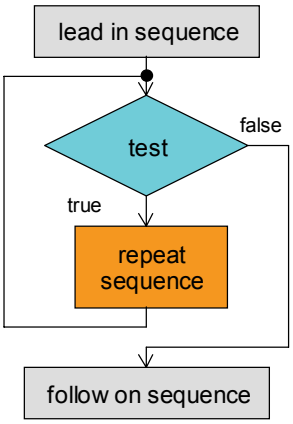


2 Computer Languages: & Java Programming

Introduction

It is assumed that the reader has a working knowledge of some computer language. However, key aspects of programming are summarizes in this chapter using the Java language to support the illustrations developed later in the text that demonstrate how graphic algorithms can be implemented in a practical way.

Libraries accessed through high level programming languages provide most graphics facilities now in common use. The demonstrations and illustrations in this section are all prepared using Java's standard window display libraries. Some of the topics, which will be explored in later chapters, will be presented using specially constructed high-level graphic and geometric-modelling language facilities that are extensions not supported by standard high-level languages. In order to outline how these extensions might be implemented it is necessary to provide an introduction to computer language processing, and it is convenient, to overlap this task with a summary of the Java language facilities used through out this book.

Later chapters set out to present the way in which a high-level computer language like Java is processed in stages to control the hardware of a computer system in the desired way. This is done using a series of simplified system-simulations. These are written in Java, using Java graphics user interface facilities to visualize the operation of the computer system and to illustrate the way that the various language levels translate from one to the other. The lowest level is micro coding and machine language programming, the next level up is an assembly language translation system, and finally at the top of the language system hierarchy is Mini-JC the kernel of a high level Java or C like programming language. Access to graphic facilities can be made at each of these language levels. Although display processors need to be examined briefly at the hardware level to understand their fundamental capabilities, programming them, can be done at all language levels above their machine code.

A. Thomas, *Integrated Graphic and Computer Modelling*,
DOI: 10.1007/978-1-84800-179-4_2, © Springer-Verlag London Limited 2008

Structured Programming Constructs

The core of an imperative-language system is based on four kinds of construction.

- Simple command statements and sequences.
- Conditional statements and sequences.
- Repeat statements and sequences.
- Sequences of statements in a hierarchical block structure.

In Java these occur in the following ways:

Names

In order to issue commands in any language it is necessary to identify the objects to which the commands will be applied. In natural language these references are names or nouns. In computer languages there are two kinds of references to simple objects. The first are called literal references and in a sense they are the objects they represent. Examples are numbers such as *2.304*, which though they are character strings, directly represent the particular numbers they encode. In order to treat this sequence of characters merely as a sequence of characters it is necessary to enclose them within quotation marks. This identifies them as a character string literal: a *String, "2.304"*. Character strings must be represented using double quotes: *"67", "234 items*", or *"Fred",* single characters using single quotes: *'k' or 'G'*. Names are character strings (but not in quotes) that start with an alphabetic character and optionally continue with further alphabetic or numerical characters. They must be treated like algebraic variables in that they are a name that can represent any value or object. A numeric variable has to be given a value before an expression containing it can be evaluated. Variables can be rearranged in algebraic expressions in "valid" ways, independently of the values they represent. Strings and characters can be reordered to implement such text manipulation within larger language statements. Naming allows general commands to be expressed that can be applied to many different particular values or objects that a name could represent.

Simple Command Statements

Many simple commands are in reality sub-program names, for example:

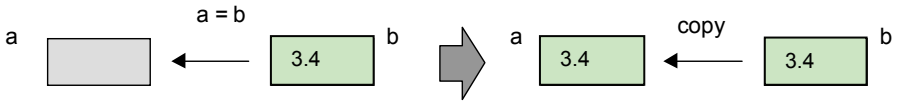
```
IO.writeString("Hello World");
```

The command *IO.writeString( )* is a call to a sub-program elsewhere in the system, which takes the data *"Hello World"* and writes it out to the display screen. Other simple commands duplicate, rename or generate objects using assignment statements.

```
number1 = 3.78;
number2 = IO.readInteger( );
number3 = number2;
number4 = number3*number1;
```

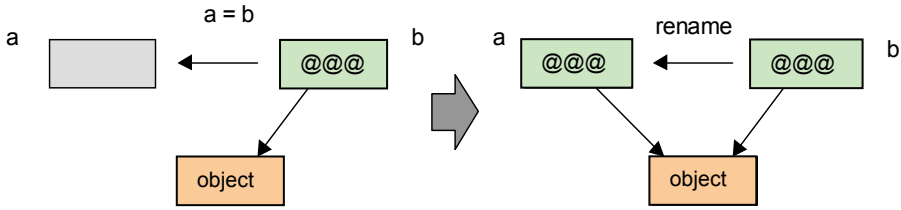
These assignment statements are commands, in the first case to associate a variable name *number1* with the real value, 3.78: In the second to call the sub-program *IO.readInteger( )* to get a number from the computer keyboard and store it

as a variable called *number2*, in the third as a command to transfer the value stored in *number2* to another variable called *number3,* and finally in the fourth the values of two variables are combined in an arithmetic expression creating a new value which is then assigned to the variable *number4*. The assignment statement can be interpreted in two ways in Java. In the first, shown below, it is a copy command. In other cases it can be thought of as a renaming or multiple naming-command.

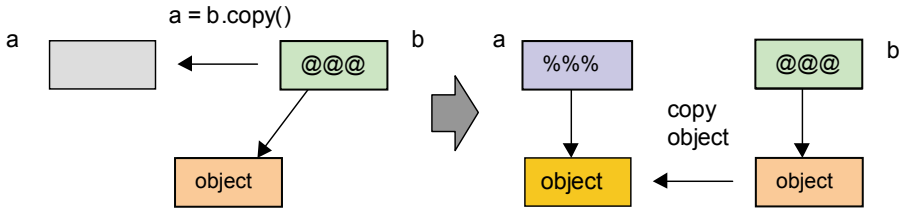


In this example if **a** and **b** are simple variable names referring to data representing numbers or characters, or truth values, then the assignment operation is one of duplication. The simplest way of thinking about the operation is that **a** and **b** represent boxes, and the assignment takes a copy of what is in box **b** and places it in box **a**.

If on the other hand, **a** and **b** are the names of a more complex object, then the assignment *a = b;* means that the object named **b** can also be referred to or called by the name **a**. The simplest way of visualising this case is as follows.



Where the data object referred to by a name is small, the box associated with the name is used to hold the data. Where the data object referred to by a name is too large for this or has a variety of possible sizes, then the name-box holds an indirect reference to the object shown in the diagram above by @@@. This means the assignment can be executed by exactly the same operation as that used in the simple case: copying the reference @@@ from one box to the other, but to get a duplicate or new object the statement has to be written *a = b.copy( );*



Clearly, further operations using the new names in each of these two cases must be handled differently. In the first case there are two objects, which can be acted on independently using their corresponding names, in the second there is one object and either name will select it. If the first name is used to change the object, then, when the second name is used to access the object that it refers to, following the change, it will address the modified object; not the unchanged one it originally referred to.

Declaring Variables and Initialising Objects

In carrying out an assignment unless the operation is policed by the system, any "type" of object could be associated with a name. If the name was intended to represent a number and by accident a truth-value is assigned to it, then clearly unlooked for results would be produced! Most programming languages type-check variables and restrict assignments to permitted associations. In order to do this a necessary part of the program writing is to specify the type of data a variable can hold. This is done in declaration statements.

```
int   variable1,   variable2,   variable3;        //  integer variables
double   variable4 = 3.2,   variable5 = 6.1;  // double length floating point variables
static TextWindow    IO    = null;                //  a null reference to a TextWindow object
```

It is generally a good idea to initialise variables with start up values like the assignments shown in the second example, even if these are going to be changed later. Where a variable refers to a more complex object, then this initialisation may be deferred in the way shown for the *TextWindow* object. Where it is not then an object has to be constructed by the system. This is achieved by issuing the ***new*** command. A useful example of this is initialising a text window to support input and output to the program, which can be done in the following way:

```
static TextWindow    IO    =  new TextWindow  (20, 30, 500, 60 );
```

The name of the object type: *TextWindow* precedes the variable name: *IO*, in the same way used to declaring simple objects. However, the ***new*** command is required to initialise a new *TextWindow*, by a call to the sub-program that sets up an object of this type, before its reference can be assigned to the variable name *IO*. In Java the procedure for building more complex objects is called a "*constructor"*, and it has the same name as the type of object it generates. The qualification of this declaration, by the keyword *static* makes the variable a *class* variable. This allows it to be used directly in simple programs.

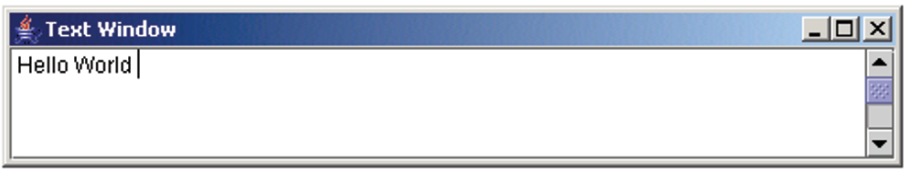
A Simple Java Program

The classic example of a simple program is one that outputs the greeting:

Hello World

```
public class Program1 {
     static TextWindow IO = new TextWindow(20,170,500,200 );
     public static void main(String[] args){
          IO.writeString(" Hello World ");
     }
}
```

Notice that the output command *writeString("hello world")* is coupled to the object name *IO* by a full stop character. This is because the *TextWindow* object: *IO* carries out the operation of displaying the message, and the sub-program that does the work, is part of this more complex object. This program when it is run generates the following display on the screen



The command *writeString()* is a call to a sub-program, elsewhere in the system, which is part of the definition of the text window object, and which takes the data *"Hello World"* and writes it out to the display screen. It is an example of several commands needed to enter and display data. Essential, if useful programs are to be written.

The facilities in Java to handle the input and output of data are very flexible to cover a variety of different modes of interaction between the user and the system, and consequently are fairly complex. In order to simplify most of the examples of programs presented in this book, a reduced set of commands solely for entering and displaying text information in a text window is provided.

```
class TextWindow extends JFrame{

      TextWindow(int col, int row,int width, int height){}     // Constructor

      public String readTextString(){}
      public String readString(){}
      public char readCharacter(){}
      public void readSpaces(){}
      public String readLine(){}
      public byte readByteInteger(){}
      public short readShortInteger(){}
      public int readInteger(){}
      public long readLongInteger(){}
      public float readReal(){}
      public double readLongReal(){}

      public void writeString(String str){}
      public void writeLine(){}
      public void writeCharacter( char ch){}
      public  void writeByteInteger(byte number, int align){}
      public  void writeShortInteger(short number, int align){}
      public  void writeInteger(int number, int align){}
      public  void writeLongInteger(long number, int align){}
      public void writeReal(float number, int align,int frac){}
      public void writeLongReal(double number, int align,int frac){}
      public void newLine(){}

      public void quit()
}
```

This *TextWindow* object *IO*, once it is set up, by calling its constructor in the way illustrated in the "*Hello World*" program, supports the list of commands or methods given above. Since a subprogram with a particular name can only return one type of data object, this list of methods introduces most of the basic types of data that the system user handles as text. The first eleven methods return numbers of different types, which can be assigned to variables of matching types in the following way:

```
char  character   =  IO. readCharacter( );      // single characters
String   string1  =  IO.readTextString( );      // character strings
String   string2  =  IO.readString( );          // character strings
String   string3  =  IO.readLine( );            // character strings
byte number1      =  IO.readByteInteger( );     // 8 bit integers
short    number2  =  IO.readShortInteger( );    // 16 bit integers
int      number3  =  IO.readInteger( );         // 32 bit integers
long number4      =  IO.readLongInteger( );     // 64 bit integers
float number5     =  IO.readReal( );            // 32 bit floating point values
double number6    =  IO.readLongReal( );        // 64 bit floating point values
```

These newly defined variables *character, string1, string2, string3, and number1.. number6,* will receive input from the keyboard. Their contents can in turn be output to the display in the *TextWindow* by the matching commands:

```
IO.writeCharacter( character);
IO.writeString( string1);
IO.writeString( string2);
IO.writeString( string3);
IO.writeByteInteger( number1, 5);
IO.writeShortInteger( number2, 5);
IO.writeInteger( number3, 5);
IO.writeLongInteger( number4, 10){}
IO.writeReal( number5, 10, 5);
IO.writeLongReal( number6, 10, 5);
```

Input to the system is best thought of as a flow of characters, in a single-file stream from the keyboard, in the order in which they are typed into the system. This will include character codes for formatting the text display like the carriage return, which finishes one line and starts the next in the display. The *IO.readLine()* command returns all the remaining text in the current line of input up to and including the carriage return. The *IO.newLine()* command does the same thing but does not return the String of characters making up the rest of the line. Its useful function is to clear input stream characters such as the carriage return that might interfere with subsequent read commands. In a related way the *IO.writeLine()* command places a carriage return character into the output steam of characters being sent to the text window. The *IO.readTextString* command and *IO.readString* command returns the next sequence of characters, upto the next "space" character code. The *IO.readTextString()* command removes any leading space characters before looking for input. The sequence *IO.readSpaces(); IO.readString();* being equivalent to *IO.readTextString()*.

Formulae, Expressions and Equations

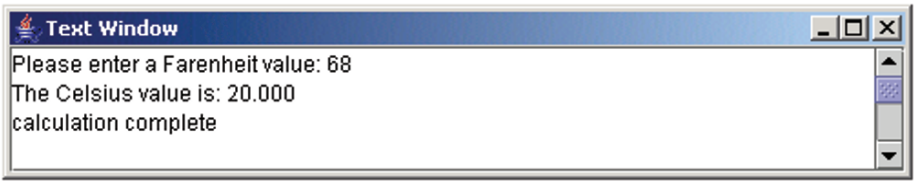
A natural extension to the assignment of simple variables is made, where the assignment transfers the result of evaluating an expression into a new variable box. In this case the assignment can be viewed as a way of generating a new variable value.

$$c = (f - 32.0) / 9.0 * 5.0;$$

This provides the simplest route into writing useful programs. Many scientific relationships and results are recorded mathematically in the form of formulae and equations. These usually translate into assignment statements and expressions in a conveniently directly way.

A simple program to convert a Fahrenheit temperature value into its corresponding Celsius value can be written in the following way.

```
public class Program2{
    static TextWindow IO = new TextWindow(20,170,500,200 );
    public static void main(String[] args){
        IO.writeString("Please enter a Farenheit value: ");
        double f = IO.readLongReal(); IO.newLine();
        double c = (f-32.0)/9.0*5.0;
        IO.writeString("The Celsius value is: ");
        IO.writeLongReal(c,6,3);
        IO.writeLine();
        IO.writeString("calculation complete \n");
    }
}
```



Statement Sequences: Blocks and Subprograms

A small program like this is made up from a list of simple commands. However the flow of control from one statement to the next, often needs to be rearranged to follow more complicated routes through different statement-sequence blocks, and facilities have to be provided to move from block to block in a program in a controlled manner. One of these block-structuring approaches has already been mentioned, and consists of giving a name to a commonly used block of code as a sub-program, and accessing it by issuing its name as a single command. The *readInteger()* command given above refers to such a block of code which carries out the reading operation demanded by the program, obtaining the number from the keyboard as it is typed into the computer system.

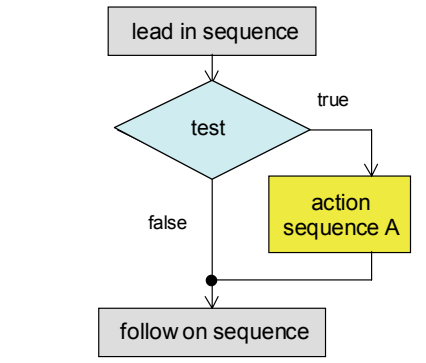
Conditional Statements

The simplest command that requires a statement sequence to be divided into blocks is the conditional command, which is illustrated by the two examples given below. In each of these cases a special pair of symbols < and > is used as brackets to indicate a section of code that has not been expressed using correctly structured commands. These brackets are called meta-symbols because they are not part of the language, but denote either an *approximation name* or an *abstract name* for an operation, which will be replaced in the final completed program by an equivalent sequence of correct commands. They are useful to denote "pseudo" code, in other words, statements, which are a rough approximation of what the final program is intended to do, but need to be distinguished from finished code. This helps to develop a program in organised steps, as it is being set up and designed. The meta symbols < >, in a similar way but more formally, are also used to define the grammatical components of larger language structures. An example would be a sentence defined as:

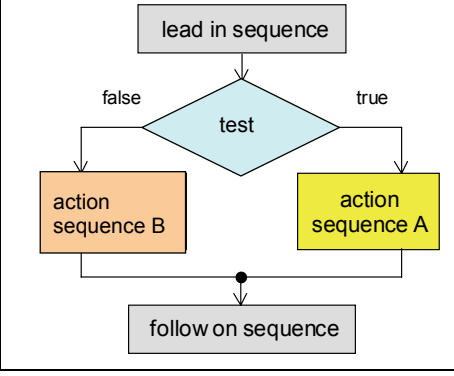
<sentence> := <subject> <verb> <object>

In order to get a correct sentence the elements in < > brackets have to be replaced by the real words that make up, or are deemed appropriate as, the subject, the verb or the object of the final sentence.

<lead in sequence>
if(< test >)**{**< action sequence A>**}**
<follow on sequence>



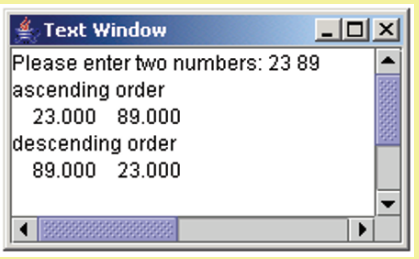
<lead in sequence>
if(< test>)**{**< action sequence A>**}**
else{<action sequence B>**}**
<follow on sequence>



The conditional consists of a block of code, which must be processed only if a certain condition is met. A test statement specifies the condition, and the condition is met when this test is found to be true. Its associated operation can then be executed.

An extension of the same idea is given in the second example. Here there are two alternative sequences of code. The first is executed when the condition is met in other words when the test is true, the second is executed when the test fails and gives a false result. It is convenient to visualise the flow of control defined by these statements in the way shown above in the diagrams on the right.

```
public class Program3{
   static TextWindow IO = new TextWindow(20,170,500,200 );

   public static void main(String [ ] args){
      IO.writeString("Please enter two numbers: ");
      double a = IO.readLongReal();
      double b = IO.readLongReal();

      IO.writeString("ascending order \n");
      if(a<b){
          IO.writeLongReal(a,10,3);
          IO.writeLongReal(b,10,3);
          IO.writeLine();
      }else{
          IO.writeLongReal(b,10,3);
          IO.writeLongReal(a,10,3);
          IO.writeLine();
      }
      IO.writeString("descending order \n");
      if(a>b){
          IO.writeLongReal(a,10,3);
          IO.writeLongReal(b,10,3);
          IO.writeLine();
      }else{
          IO.writeLongReal(b,10,3);
          IO.writeLongReal(a,10,3);
          IO.writeLine();
      }
}
```

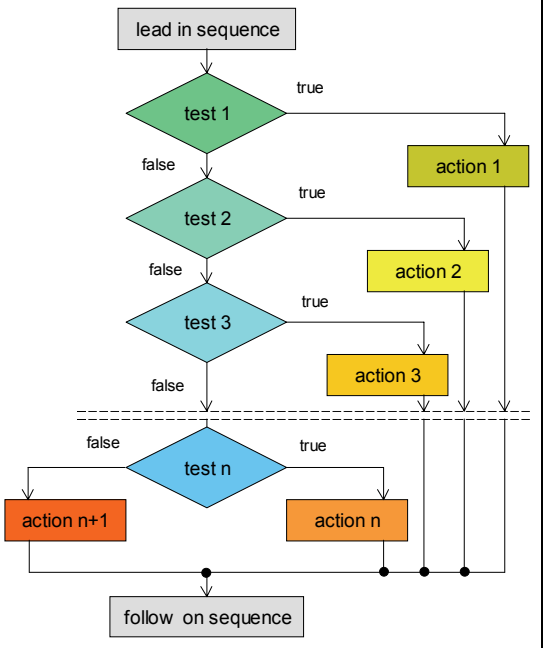


A program to order two numbers for output can be written using a single conditional test in the way shown above. However it is possible to place conditional statements within the statement sequences already controlled by other conditional statements.

This gives a nested structure, which can be used to reorder more than two numbers and output them as an ordered list, in a single step. This is loosely comparable to the analogue use of measured rods to order a collection of values in a single step, described in chapter one as a “one-shot” operation.

```
public static void main(String[] args){
      IO.writeString("Please enter three values: ");
      int a = IO.readInteger(); int b = IO.readInteger(); int c = IO.readInteger();

      if(a<b)
            if(a<c)
                  if (b<c) { IO.writeInteger(a,9);IO.writeInteger (b,9); IO.writeInteger (c,9);}
                  else    { IO.writeInteger(a,9); IO.writeInteger (c,9); IO.writeInteger (b,9);}
            else  if (b<c) { IO.writeString("Impossible case :except  ring order ");}
                  else    { IO.writeInteger(c,9); IO.writeInteger (a,9); IO.writeInteger (b,9);}
      else if(a<c)
                  if (b<c) { IO.writeInteger(b,9); IO.writeInteger (a,9); IO.writeInteger (c,9);}
                  else    { IO.writeString("Impossible case :except  ring order ");}
            else if (b<c) { IO.writeInteger(b,9); IO.writeInteger (c,9); IO.writeInteger (a,9);}
                  else    { IO.writeInteger(c,9); IO.writeInteger (b,9); IO.writeInteger (a,9);}
}
```

```
if       (<test1>){<Action1>}
else if (<test2> ){<Action2>}
else if (<test3>){<Action3>}
else if (<test..>){<Action..>}
…
else if (<test n>){<Action n>}
else             {<Action  n+1>}
```



A special case of this nested arrangement is provided by the example shown above where the tests are applied sequentially in a list. In this example a list of conditional tests is processed and whenever a test is successful the dependent code sequence is executed after which, control leaves the conditional statement and passes to the next statement. To order a sequence of numbers using this construction would require the following arrangement.

```
public static void main(String [ ] args){
      IO.writeString("Please enter three values: ");
      int a = IO.readInteger(); int b = IO.readInteger(); int c = IO.readInteger();

      if      ((a<b)&&(b<c))  { IO.writeInteger(a,9); IO.writeInteger (b,9); IO.writeInteger (c,9);}
      else if((a<c)&&(c<b))  { IO.writeInteger(a,9); IO.writeInteger (c,9); IO.writeInteger (b,9);}
      else if((c<a)&&(a<b))  { IO.writeInteger(c,9); IO.writeInteger (a,9); IO.writeInteger (b,9);}
      else if((b<a)&&(a<c))  { IO.writeInteger(b,9); IO.writeInteger (a,9); IO.writeInteger (c,9);}
      else if((b<c)&&(c<a))  { IO.writeInteger(b,9); IO.writeInteger (c,9); IO.writeInteger (a,9);}
      else if(((c<b)&&(b<a)) { IO.writeInteger(c,9); IO.writeInteger (b,9); IO.writeInteger (a,9);}
      else IO.writeString("Impossible case :except  ring order ");
}
```

In this example the tests are clear but more complex. Each test is an expression, which combines the truth-values of two simple binary relationship tests such as (a<c) to give a final single truth-value, the result of evaluating the overall test expression.

These truth-values are called Boolean values, which can be represented in the program by literal representations in the same way that numerical values can. In this case they are one of two values represented by the words *true* and *false*. Boolean expressions are formed by combining Boolean variables using the operators *"not"*, *"and"* and *"or"*. These are represented in Java by the characters: !, &&, ||, respectively. The following operator, *"truth-tables"* define their actions.

input		output
A	▶	*! A*
true	▶	*false*
false	▶	*true*

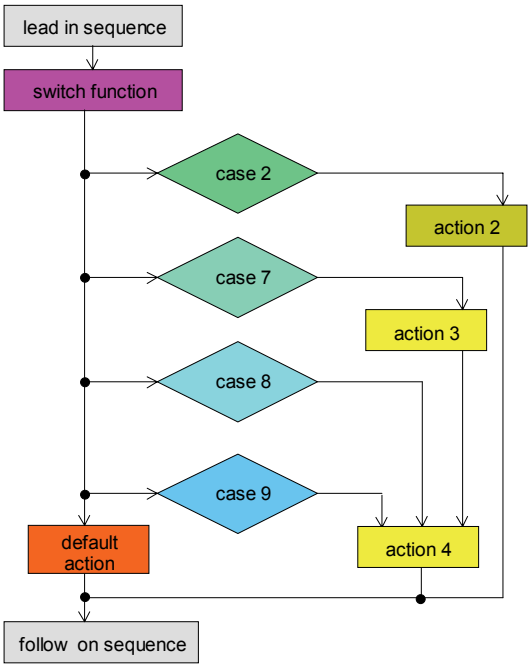
input	input		output		output
A	*B*	▶	*A&&B*	▶	*A||B*
true	*true*	▶	*true*	▶	*true*
true	*false*	▶	*false*	▶	*true*
false	*true*	▶	*false*	▶	*true*
false	*false*	▶	*false*	▶	*false*

Expressions can be constructed using operator precedence rules (*not* > *and* > *or*) and brackets in much the same way that arithmetic algebraic expressions are built up and the resulting values can be assigned to Boolean variables.

```
boolean  itIsRaining = true;
boolean result = !((a < b) && (b < c)&& itIsRaining || (a == b))||(x != y)
```

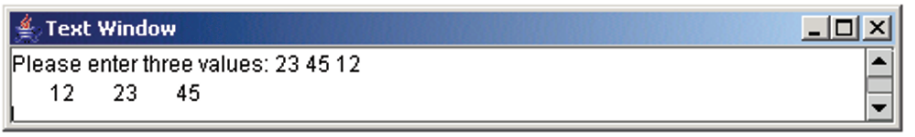
The <*switch statement*> provides an alternative statement to the nested conditional that allows a similar kind of multiple-choice. In this case there is not a sequence of tests for true or false, but a function that generates an integer value. This number is used to select the label case-number that it matches, and then the code associated with the case is executed. Notice in this case that the *break* statement is necessary to pass control on to the next command. Where it is missing control passes to the next case in sequence, below. In the example shown above, if the selection function gives an integer value 7 then actions 3 and actions 4 are executed. If the selection function gives 8 or 9 then only action 4 is executed.

```
switch( <selection function> ){
   case 2: <action 2>; break;
   case 7: <action 3>;
   case 8:
   case 9: <action 4>; break;
   ....
   default:<default action>;
}
```



The application of this statement to sorting three numbers can be carried out in the following way.

```
public static void main(String[] args){
      IO.writeString("Please enter three values: ");
      int a = IO.readInteger(); int b = IO.readInteger(); int c = IO.readInteger();

      int j = 0;
      if(a<b)j=j+1;
      if(a<c)j=j+2;
      if(b<c)j=j+4;

      switch(j){
         case 0: IO.writeInteger(c,9); IO.writeInteger (b,9); IO.writeInteger (a,9);break;
         case 1: IO.writeInteger(c,9); IO.writeInteger (a,9); IO.writeInteger (b,9);break;
         case 3: IO.writeInteger(a,9); IO.writeInteger (c,9); IO.writeInteger (b,9);break;
         case 4: IO.writeInteger(b,9); IO.writeInteger (c,9); IO.writeInteger (a,9);break;
         case 6: IO.writeInteger(b,9); IO.writeInteger (a,9); IO.writeInteger (c,9);break;
         case 7: IO.writeInteger(a,9); IO.writeInteger (b,9); IO.writeInteger (c,9);break;
         default:IO.writeString("Impossible case :except  ring order ");break;
      }
      IO.writeLine();
}
```



Decision Tables

These three examples lead to a useful programming construct, which is helpful in designing programs that handle complex relationships. This is the decision table. When a relationship test is evaluated or when a *boolean* variable is defined, combinations of their truth values can be used to select different actions. When these conditions and actions are complex then it is useful to set out all the possible combination of *boolean* variable values to ensure that each outcome, from all possible inputs, is allocated to an appropriate action. In essence this is giving all input combinations of values an output action rather than a value, which is done in the case of the truth table definition of *boolean* operator functions. Laying out these relationships in a table allows the relationships between the tests and the actions they require to be systematically examined and reduced to their simplest form, it also saves mistakes resulting from unaccounted cases being overlooked.

The ordering of three numbers depends on the primitive operation of comparing pairs of numbers. There are six relationship pairs generated by three variables {a, b, c} which are (a, b), (a, c), (b, c) and (b, a), (c, a), (c, b). If these are all tested using the < test, then this gives six *boolean* values and six possible tests of the form *if(a<b) A1 else A2*, *if (b<a) A3 else A4* etc.. However the results of these tests are not all independent. If *(a<b)* is true then *(b<a)* will be false. These tests are not opposites and cannot therefore be treated by one test *if(a<b) then A1(or A4) else A2 (or A3)*. The problem is the case where *(a==b)*. Where *(a==b)* both *(a<b)* and *(b<a)* will be false. When the actions A1 and A2 are considered for ordering the two values *a* and *b*, it is clear that where *(a == b)* either outputting *(a, b)* or *(b, a)* gives the same result. Analysing the actions in relationship to the tested conditions in this case allows a single test to be used to replace two. A decision table for this problem can be set up as follows:

(a<b)	true				false			
(a<c)	true		false		true		false	
(b<c)	true	false	true	false	true	false	true	false
Action	A7	A6	A5	A4	A3	A2	A1	A0
Output	a, b, c	a, c, b	error	c, a, b	b, a, c	error	b, c, a	c, b, a

In each of these examples the program is designed to select and then output the sorted numbers in one step as a complete ordered list of numbers. There is a limit to the number of elements that can be treated in this "one-shot" way, with three numbers there are only 3! in other words 6 possible output lists. When the number of elements in the unordered list is raised to 4, then the number of output orders goes up to 24. With seven numbers, this number would expand to 7! in other words 5040

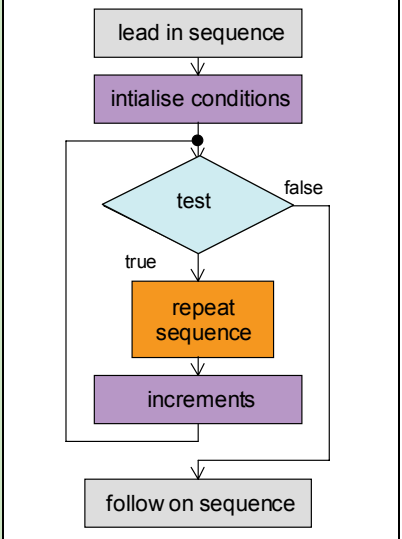
output statements would be needed in the program to carry out the task in the same way. Laying out the decision table would show that 7 numbers would require 7*6/2 binary relationship tests. As a decision table this would set up 2^{21}, in other words over 2 million potential actions. Since only 5040 of these are not error messages, this approach clearly has strict practical limits.

The sorting program can be greatly simplified if the sorted output is built up step by step rather than being generated in a single step as a one-shot operation. If each step removes the current largest element in the input list, only **n** steps will be required to create the output list in the modified operation to order **n** values. This sequential process can be greatly simplified as a program if it can be expressed as an operation, which can be applied repeatedly, to the same set of data.

Repeat Statements

The next commands that modify the way sequences of code are processed are the repeat commands. There are three main forms commonly used. The first is the *for*-loop. This contains three fields in a control section followed by a sequence of dependent statements within { } brackets. The first field sets up initial conditions, usually a counting variable. The second field sets up a finishing condition usually a relationship test applied to the counting variable and finally the third field defines the changes that must be executed at the end of each repetition cycle. The latter is usually the increment or decrement of the counting variable.

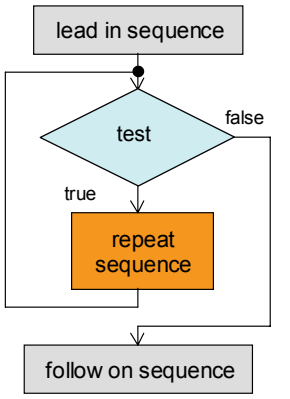
```
for(int i = 0; i< integerArray.length; i++){
      IO.writeInteger(integerArray[i],6);
}
```



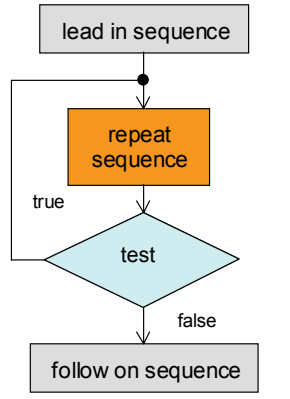
The *for*-loop is often associated with actions on arrays. An array is a collection of objects where each object can be accessed by giving the array name followed by the index of the object in the array, in square brackets. If the counter in a *for*-loop is used to index elements in the array then each object in it can be visited in order. A program to write out all the integers in an array of integer numbers can be written in

the way shown above. The other two repeat commands are more primitive in that they merely control the repetition of a block of code by a terminating test. The two forms of this command apply this test at the beginning and at the end of the repetition cycle respectively. The following diagrams illustrate the flow of control set up by these statements using the equivalent programs to the one above for writing out the contents of an array.

```
int i = 0;
while( i< integerArray.length){
    IO.writeInteger(integerArray[i],6);
     i = i+1;
}
```



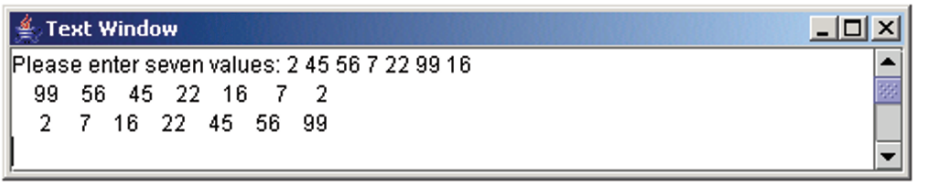

```
int i = 0;
do{
    IO.writeInteger(integerArray[i],6);
     i = i +1;
} while( i< integerArray.length);
```



The difference between these two examples is that the first can cope with an array with no contents, while the second needs the array to have at least one entry if an error is not to be generated by the system attempting to access a non-existent element.

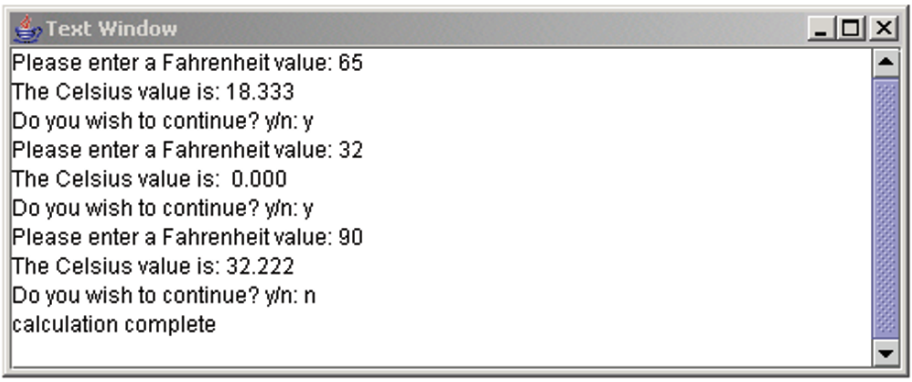
Using the repeat command allows a set of numbers held in an array to be ordered by a simple repetitive swapping operation. If the largest element is selected each cycle through an array and the value is stored at the beginning of the array, then after **n** passes for an array of **n** elements the array will end up being sorted into descending order.

```
public static void main(String[] args){
      int[] integerArray = new int[7];
      IO.writeString("Please enter seven values: ");

      for(int i=0;i<7;i++) integerArray[i] = IO.readInteger();
      for(int j=0;j<7;j++){
            for(int i=j+1;i<7;i++){
                  if(integerArray[j]<integerArray[i]){
                        int temp=integerArray[j];
                        integerArray[j]=integerArray[i];
                        integerArray[i]=temp;
                  }
            }
      }
      for(int i=0;i<7; i++)IO.writeInteger(integerArray[i],6);
      IO.writeLine( );
      for(int i=6;i>=0;i--)IO.writeInteger(integerArray[i],6);
}
```



The application of the second form of repeat command allows the program to be set up to handle variable amounts of input data. A program to convert Fahrenheit temperature measures to Celsius values can be written in the following way, where the system asks at the end of each calculation if another is required.

```
public class Program 4{
      static TextWindow IO = new TextWindow(20,170,500,200 );

      public static void main(String[] args){
            String str = "";
            do{  IO.writeString("Please enter a Fahrenheit value: ");
                  double f = IO.readLongReal();IO.newLine();
                  double c = (f-32.0)/9.0*5.0;
                  IO.writeString("The Celsius value is: ");
                  IO.writeLongReal(c,6,3); IO.writeLine();
                  IO.writeString("Do you wish to continue? y/n: ");
                  str = IO.readString();IO.newLine();
            }while(str.equals("y"));
            IO.writeString("calculation complete \n");
      }
}
```



The sequences of commands in a repeat loop have to be designed carefully, to be applicable again and again as the program cycles. An example of a construction, which can be applied repetitively to a sequence of data, can be seen in the case of the formulae used to define the sum, the average, and the variance of a list of values.

Sum: $$S = \sum_{i=1}^{i=n} x_i$$

Average: $$\bar{x} = \sum_{i=1}^{i=n} \frac{x_i}{n} = \frac{1}{n}.\sum_{i=1}^{i=n} x_i$$

Variance: $$\sigma^2 = \sum_{i=1}^{i=n} \frac{(\bar{x} - x_i)^2}{n}$$

$$\sum_{i=1}^{i=n} \frac{(\bar{x} - x_i)^2}{n} = \sum_{i=1}^{i=n} \frac{\left(\bar{x}^2 - 2.\bar{x}.x_i + x_i^{\,2}\right)}{n}$$

$$= \frac{1}{n}\sum_{i=1}^{i=n} \bar{x}^2 - 2.\bar{x}.\sum_{i=1}^{i=n} \frac{x_i}{n} + \sum_{i=1}^{i=n} \frac{x_i^{\,2}}{n}$$

$$= \bar{x}^2 - 2.\bar{x}.\bar{x} + \sum_{i=1}^{i=n} \frac{x_i^{\,2}}{n}$$

$$= \sum_{i=1}^{i=n} \frac{x_i^{\,2}}{n} - \bar{x}^2$$

The sum is simple to calculate within a "*for*" loop or a "*while*" loop. The average also can be calculated in a single repeat loop, in one of two ways depending on whether the length of the list is known at the beginning of the repeat command or

only when the list has been completely processed. In the first case if the length of the list is known then each element can be divided by the list length and then added to the total. In the second case a count has to be kept of each new element added to the sum of list elements, when the list is complete the answer is the sum divided by the number of elements.

The variance in contrast, appears to require two loops the first to calculate the average, the second to calculate the variance. Rearranging the formula, algebraically in the way shown above allows the variance to be calculated in one loop. By calculating the sum of the squared elements, $(x_i.x_i)$ and the average $\overline{x}$ within the loop, the final result can be obtained by squaring the average and subtracting it from the average of the sum of the squares. This is one example from a variety of different "recurrence relationships" designed to use repeat commands to provide compact and efficient program code.

Another example of this process occurs with the choice of names for variables. A program to generate the Fibonacci series can be set up by defining the n^{th} element as the sum of the $n\text{-}1^{th}$ and the $n\text{-}2^{th}$ elements in the series.

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

```
nextElement = lastElement + lastButOneElement;
```

Once this statement has been executed the *lastElement* and the *lastButOneElement* no longer hold the values appropriate to their names. In order to complete a sequence of statements that can be repeated, it is necessary to redefine these variables in the following way:

```
nextElement = lastElement + lastButOneElement;
output(nextElement);
lastButOneElement = lastElement;
lastElement =  nextElement;
```

Exactly the same redefinition is needed to draw a curve based on plotting a string of line segments.

```
plotLine(leadingPoint, laggingPoint, colour);
laggingPoint = leadingPoint;
leadingPoint =  calculateNextPoint();
```

A more complex repeat pattern can be set up to implement a "finite state machine". This is a program where the current state determines the action, and the next state is determined by the combination of new inputs and the current state. A diagram of this kind of mechanism is a very useful tool for programming a variety of problems. It is often easier to visualise the interactions needed to make a program function correctly if bubbles labelled by the state names are drawn out to represent the states, and state transitions are shown by arrows, from one bubble to another, with each arrow associated with the inputs that cause the transition.

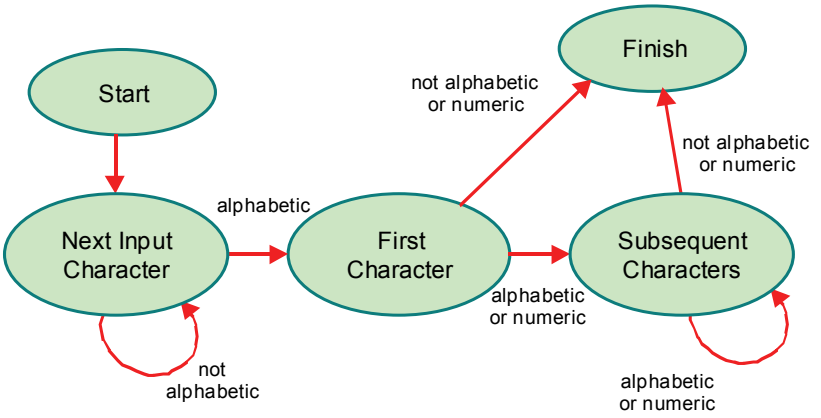
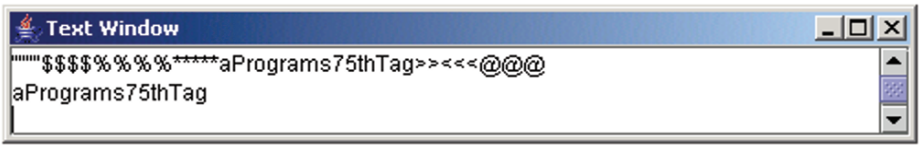


Figure 2.1 State transition diagram for a name recognition program

```
public static void main(String[] args){
      String str = ""; char ch = '\0';
      boolean notFinished = true;
      int start = 0, nextCharacter = 1;
      int firstCharacter= 2, subsequentCharacters = 3, finish=4;
      int state = start;
      while(notFinished){
            switch(state){
            case 0:state = nextCharacter; break;
            case 1:ch = IO.readCharacter();
                  if(((ch>='a')&&(ch<='z'))||((ch>='A')&&(ch<='Z'))) state = firstCharacter;
                  break;
            case 2:str = str+ch;
                  ch = IO.readCharacter();
                  if(((ch<'a')||(ch>'z'))&&((ch<'A')||(ch>'Z')) &&((ch<'0')||(ch>'9')))state = finish;
                  else state = subsequentCharacters;
                  break;
            case 3:str = str + ch;
                  ch = IO.readCharacter();
                  if(((ch<'a')||(ch>'z'))&&((ch<'A')||(ch>'Z'))&&((ch<'0')||(ch>'9'))) state = finish;
                  break;
            case 4:notFinished= false; break;
            }
      }
      IO.writeString(str + "\n");
}
```



A switch statement, acting on a state variable, contained within a repeat loop, can be used to implement this kind of process in the way shown in Figure 2.1 to recognise a name within a sequence of characters. Though this structure is very powerful and will cope with many programming tasks, its limitation is its fixed number of state variables. Many programming tasks require more memory than this in order to build up a varying number of partial results, before the overall task can be completed, the amount of memory depending on the nature of the input received: the classical example of this kind of task is evaluating an expression.

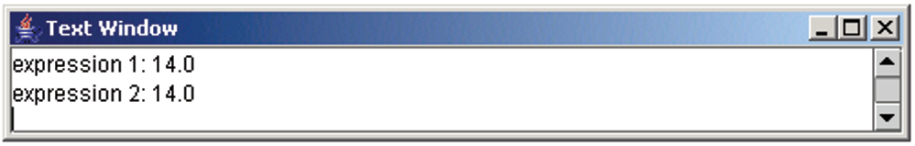
Sub-programs, Procedures, Functions and Methods

One way of getting this extra memory as it is needed is to use recursive procedures – sub-programs, which call themselves. The terms: sub-program, subroutine, procedure, function, and method; are with small variations interchangeable and depend on the computer language being used. In object oriented languages the preferred term referring to a sub program is the term "method". In Pascal and Modula-2 the terms procedure and function-procedure are used to distinguish sub-programs which returned no value, and sub-programs which returned a value, like *value = sin(alpha);*. In Java and C, because statements are considered to have values, (allowing strings of assignments such as a = b = c = d; to be written) any subprogram can be written as a function. However, methods are allowed to return a void value, and therefore can behave as procedures that cannot be used in assignment statements!

```
Public class Program5{
    static TextWindow IO = new TextWindow(20,170,500,200 );
    static double mul (double x,double y)  {return x*y;}
    static double div  (double x,double y)  {return x/y;}
    static double sub (double x,double y)  {return x-y;}
    static double add (double x,double y)  {return x+y;}
    public static void main(String[] args){
        double a=2,b=3,c=4,d=10,e=1,f=4,g=2;
        double expression1 = a*b+c*(d-f)/(e+g);
        double expression2 = add(mul(a,b),mul(c,div(sub(d,f),add(e,g))));
        IO.writeString("exrpession 1: "+ expression1+"\n");
        IO.writeString("exrpession 2: "+ expression2+"\n");
    }
}
```

The declaration of functions in a simple program to evaluate an arithmetic expression using function calls is shown in Program 5. As before it is necessary to qualify the definition of each method by the keyword *static*. In this example the same expression is presented in two different ways. In the first it is written in the conventional form using arithmetic operators and brackets. In the second each operator is replaced by a function call. Each function method executes its corresponding arithmetic operation in a standard way on the two values passed to it and returns the result. This example illustrates the way methods are defined and the way they are called. In the function definition the type of the return value has to precede the function name, and the values passed to the function have to be given

working names in order for the function code to be written, and each has to have its type defined. When the function is "called" the real parameter names matching the dummy, working names in the function definition have to be placed in the method's argument list, in the correct order to match the dummy arguments in the method definition. This matching process allows the function to be applied to any variables that the calling statement specifies.



When a method is called, control is passed to the sub-program code. The parameter values in the calling statement are copied to the dummy variables in the sub-program code. When the method's computation is complete its resulting value is passed back, and treated, as a value associated with the function name in the calling statement, as if the calling name were a simple variable.

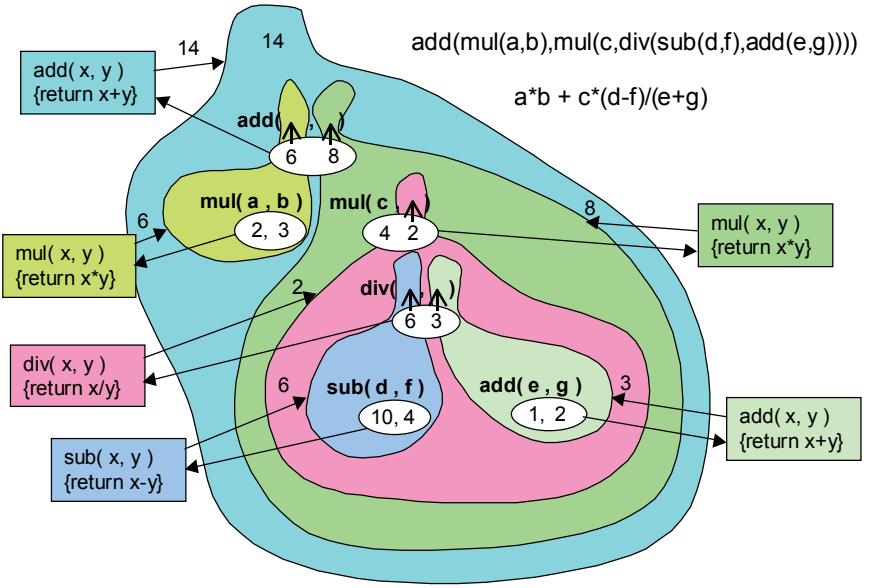


Figure 2.2 Parameter matching for sub routine calls

When a more complex object is passed as an argument to a sub-program, its indirect reference will be copied to a local variable in the method. This means, in this case, the object is not duplicated. For simple objects such as integers the value is passed and therefore it is duplicated. Arrays are more complex objects and they have their references passed to methods rather than the whole array being duplicated. Consequently an array with one element will act as a convenient example of a pass by reference. An example, where this treatment makes a difference is the subprogram to swap the contents of two variables.

```
Public class Program6{
    static TextWindow IO = new TextWindow(20,170,500,200 );
    static void swap1(double x, double y){ double temp = x;  x = y; y = temp; }
    static void swap2(double [] x, double[] y){ double temp = x[0];  x[0] = y[0]; y[0] = temp;}

    public static void main(String[] args){
        IO.writeString("please enter two different numbers: ");
        double a = IO.readLongReal();  double b = IO.readLongReal();
        double[] c = new double[]{a}; double[] d = new double[]{b};
        swap1(a,b);
        IO.writeString("swap1: " + a + " "+ b +"\n");
        swap2(c,d);
        IO.writeString("swap2: " + c[0]+ " "  + d[0] +"\n");
    }
}
```

In program 6, swap1 shows a swapping function which exchanges the contents of the local parameter variables, but which has no effect in the space of the calling program. In contrast swap2, by passing the references to two arrays, by exchanging their contents, provides the result back to the calling program, because it uses the same references to the arrays in the function that are used in the calling program.

When a subroutine is called, storage space for its internal and parameter variables, (local variables) are allocated to the program automatically by the language system. This extra memory space is arranged in a stack data-structure, where the last element added to the stack, is the first element returned from the stack. If a procedure calls itself then it will build up a sequence of memory spaces on the stack, which will be taken off the stack as the procedure returns to its calling statement. This recursive procedure calling supports a different way of implementing a repetitive operation. For example writing out the contents of an array can be done either forwards or backwards, by the following procedures:

```
    static void forwardOrder(int i, double[] array){
        if(i<array.length)IO.writeLongReal(array[i],6,2);else return;
        forwardOrder(i+1, array);
    }
    static void backwardOrder(int i, double[] array){
        if(i<array.length)backwardOrder(i+1, array);else return;
        IO.writeLongReal(array[i],6,2);
    }
```

In these examples each call to the routine will set up a local variable *i,* which will be placed in the stack starting with 0. A new value for *i* will be generated, and incrementally increased, by each routine call, until it is equal to the length of the array, when the procedure will return through all its intermediate calls back to its start, releasing memory space for *i* as it goes. By placing a write statement before the recursive call the local values of *i* will be used as they increase, by placing the write statement after the recursive call the local value of *i* will be used as they decrease during the return path of the calling sequence. It is essential that some way of stopping such a chain of recursive calls be built into recursive procedures, in this case testing to see if the end of the array has been reached stops the sequence of calls.

Two standard data structures can be handled in an elegant way using this approach. The first is the simple list, either as an array in the way shown in the example, or as a dynamic linked list data structure. The second is the tree data structure where elements are hierarchically linked to two or more lower level elements. The expression used in Program 5 can be represented as the operator tree shown diagrammatically in Figure 2.3. This will be explored more fully in a later chapter.

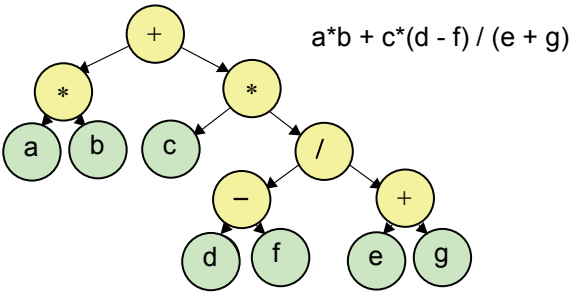


Figure 2.3 An arithmetic expression operator tree

Recursive programs can handle more complex data structures than simple finite state machines, mainly because of the way they can manage the growth and release of data stored on a stack.

Associated with the tree structure is a programming strategy called “divide and conquer”. Where this approach is applicable it usually provides a more efficient algorithm, than alternatives. Consider the sorting problem. To order a list of values that can appear in any order, it is necessary to remove the largest value *n* times from a list of *n* elements, and each selection will take *n* comparisons. Overall this requires n^2 comparison operations. If the original list is divided into two halves, and this is done recursively until there is only one element in each list, then the return operation can be one of merging lists in order. At any level this will consist of systematically merging lists already ordered lower down the recursive chain. Program 7 shows an example of a merge-sorting algorithm of this type. Figure 2.4 shows that the number of comparisons in this approach is reduced to the order of *n.log(n)* for a list of *n* elements.

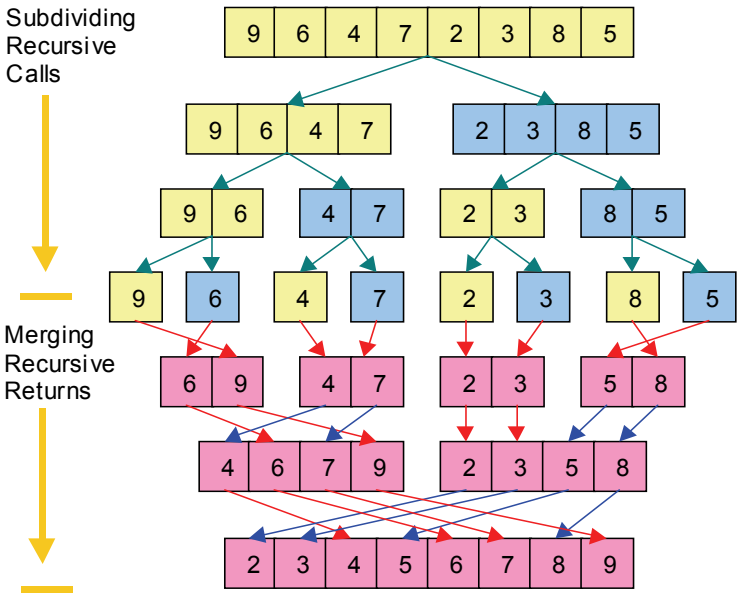
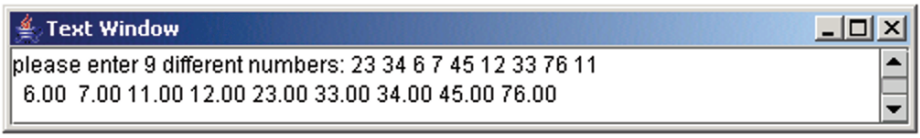


Figure 2.4 Divide and conquer mergeSort procedure

```
public class Program7{
   static TextWindow IO = new TextWindow(20,170,500,200 );  static int num= 9;
   public static void main( String[ ]  args ){
        double[][] a = new double[2][num];  int level=0, left=0, right =num;
        IO.writeString("please enter "+num+" different numbers: ");
        for(int i=0;i<num;i++) a[0][i]= IO.readLongReal();
        mergeSort(level,left,right,a);
        for(int i=0;i<num;i++) IO.writeLongReal(a[0][i],6,2);
   }
   static void mergeSort( int level, int left, int right, double[ ][ ] a){
        int nextLevel = (level+1)%2;
        if((right - left)==1){ a[1][left] = a[0][left];  return;}
        int middle = (left+right)/2;
        mergeSort( nextLevel, left, middle, a);   mergeSort(nextLevel, middle, right, a);
        merge(nextLevel, left, middle, right, a);
   }
   static void merge( int level, int left, int middle, int right, double[ ][ ]a ){
        int r = (level+1)%2, s = level, t, i =left ,j = middle, k = left;
        while(((i<middle)||(j<right))){ t = 0;
             if((i>=middle)|| ((j<right) &&(a[s][j]<=a[s][i]))) {a[r][k++]= a[s][j];t=t+1;}
             if((j>=right) || ((i<middle)&&(a[s][j]>=a[s][i]))) {a[r][k++]= a[s][i];t=t+2;}
             switch( t ) {case 1: j++; break;  case 3: j++; case 2: i++; }
        }
   }
}
```



The merging operation is a linear sequential operation analogous to the action of closing a zip fastener, in the way shown schematically in Figure 2.5!

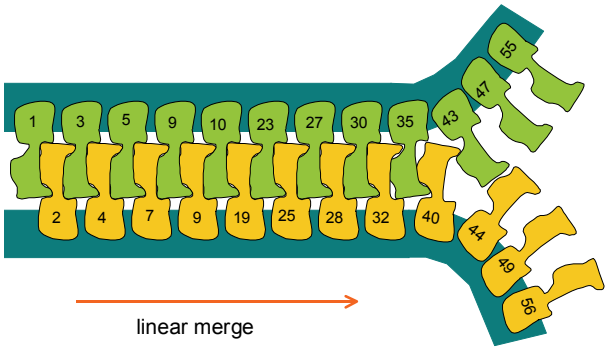


Figure 2.5 Linear merging

Types, Classes and Objects

Java is an object-oriented language. This means that it provides more than the basic structured programming constructs discussed above. The object-oriented approach extends the way data types are handled, including more complex data structures and algorithms within a common, unified conceptual framework.

A data-type appears to be a relatively simple idea when viewed at the text level. The symbols used to represent numbers and words immediately indicate that their interpretation must be handled in a different way. They are different data types. At the machine level in the computer system all the information is in the same form: strings of binary digits. In this setting the data gives no indication of its type. Its type is only determined by the context in which it can be used correctly. A bit string is a number when it is processed as a number, but it could equally well be processed as a character string if it were passed to the input-stream of a printer. A data type is defined by the permitted operations that can be carried out on data of that type. The structure of the data and the valid operations on it are inextricably entwined.

Data Structures and Algorithms

The way data structures and algorithms have to be considered together can be illustrated by the following example. If an eight-bit data value is used to represent whole numbers then it can be used to represent positive integers in the range 0 -- 255, but the information defining the use that can be made of these bit patterns has to be held outside the pattern itself. If the number is negative this will change its type and hence for example, the way it can be added to a positive number. It is quite possible to use one bit from the eight bits to determine which of these two types is present,

though the number range held in the remaining 7 bits of data would be limited to half the original, only giving: 0 to 127: and the program to process the two data formats would still have to be different.

If the bit patterns, in positive binary numeric order are placed round a circle from 0 to 255, then the value range from –128 to +127 can represented by a shift round the circle: shown by the red and blue labels in Figure 2.6. If 0 to 255 is represented by 00000000 to 11111111 (blue) then –128 to 0 to +127 can be represented by the sequence 10000000 to 00000000 to 01111111 (red). This allows the addition of positive and negative numbers to be the same operation merely giving a result offset along the original positive number line. To turn a positive number to a negative number in this representation, or vice versa, merely requires the number's bit pattern to be mapped horizontally across the circle, shown by the green arrows in Figure 2.6

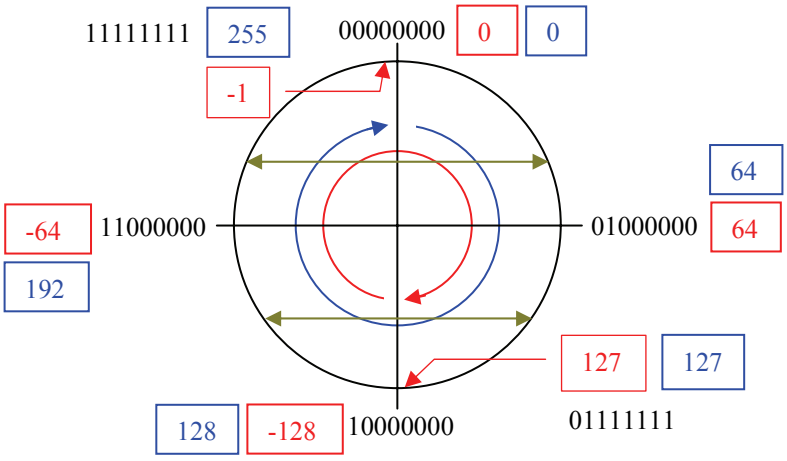


Figure 2.6 Two integer types: mapped onto the same bit patterns

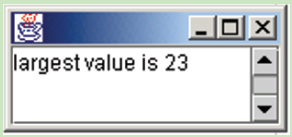
This mapping gives the two's complement representation for negative numbers. 00000001 as a positive 1 becomes 11111111 as negative 1, and 127 with bit pattern 01111111 becomes 10000001 as –127. If the binary values in each bit position are inverted (0⇨1 and 1⇨0), and a binary 1 added to the new overall 8-bit binary number, the result is a conversion from a positive to a negative number, and the reverse process converts values the other way. In this case the number range for positive or negative numbers is still halved but the permitted operations become the same for both.

Element Names, Object Names and Collection Names

Simple variable names are ways of accessing memory locations, which contain changeable bit patterns. The type associated with the name determines the correct operations that can be carried out on these bit patterns. Object names in contrast are also ways of accessing memory locations that hold bit patterns, but these bit patterns are the address of objects, which can consist of many words of data. The first is a direct reference to the value the second is the reference to a reference in other words

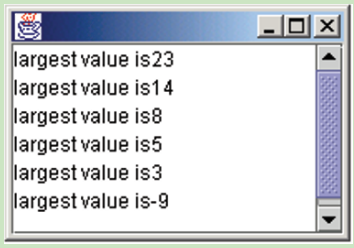
an indirect reference. The two names directly reference different types of data, the first a variable value the second an address or a pointer. Simple variable names are not sufficient to allow useful programs to be written. Names for more complex objects are necessary and this can be demonstrated by attempting to sort a collection of numerical values stored as individually named, simple variables for example *{a, b, c, d, e, f }*. If the *"repeated selection of the largest"* algorithm is used then the only way it can be coded is as a sequence of commands of the form:

```
int a= 5, b=8, c= -9, d=23, e=14, f=3, t=0;
int m= Integer.MIN_VALUE;
if(m<a){ t= a; a=m; m=t;}
if(m<b){ t= b; b=m; m=t;}
if(m<c){ t= c; c=m; m=t;}
if(m<d){ t= d; d=m; m=t;}
if(m<e){ t= e; e=m; m=t;}
if(m<f){ t= f; f=m; m=t;}
Output.writeString("largest value is "+ m +"\n");
```



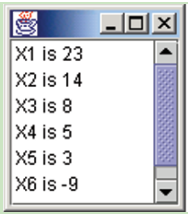
This requires as many statements as there are variables. To order the set of numbers this sequence could be placed in a repeat loop as long as ordered output is all that is wanted: writing out the value of *m* at the end of each iteration.

```
int a= 5,b=8,c=-9,d=23,e=14,f=3,t=0;
for(int i=0;i<6;i++){
    int m= Integer.MIN_VALUE;
    if(m<a){ t= a; a=m; m=t;}
    if(m<b){ t= b; b=m; m=t;}
    if(m<c){ t= c; c=m; m=t;}
    if(m<d){ t= d; d=m; m=t;}
    if(m<e){ t= e; e=m; m=t;}
    if(m<f){ t= f; f=m; m=t;}
    Output.writeString("largest value is"+ m +"\n");
}
```



If a new list of ordered variables, is wanted then this selection sequence will have to be duplicated for each value, or included in a list of procedure calls to this selection code structured as the sub program *selectTheLargest.*

```
int X1= selectTheLargest();  Output.writeString(" X1 is "+ X1 +"\n");
int X2= selectTheLargest();  Output.writeString(" X2 is "+ X2 +"\n");
int X3= selectTheLargest();  Output.writeString(" X3 is "+ X3 +"\n");
int X4= selectTheLargest();  Output.writeString(" X4 is "+ X4 +"\n");
int X5= selectTheLargest();  Output.writeString(" X5 is "+ X5 +"\n");
int X6= selectTheLargest();  Output.writeString(" X6 is "+ X6 +"\n");
```

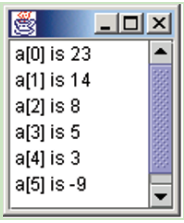


To make this work it is necessary to make the variables *a* to *f* static global variables.

```
static int selectTheLargest(){
    int m= Integer.MIN_VALUE;
    for(int i=0;i < 6;i++){
        if(m<a){ t= a; a=m; m=t;}  if(m<b){ t= b; b=m; m=t;}  if(m<c){ t= c; c=m; m=t;}
        if(m<d){ t= d; d=m; m=t;}  if(m<e){ t= e; e=m; m=t;}  if(m<f ){ t= f;  f=m; m=t; }
    } return m;
}
```

The solution to this potentially massive duplication of the same code pattern is the use of array naming. The collection of numbers is treated as a single object and is given a group name. Within the group the individual elements are identified by a second, variable name holding an index value. This is implemented by storing the collection of numbers in neighbouring locations in memory. The array name is then associated with the base-address of the first location in memory used for the block of data, and the index is used to give the offset from this position for each element in the collection by adding the index value to the base-address of the array. This allows a short single program to handle different sized arrays of numbers with the same code.

```
int[] a = new int[]{5,8,-9,23,14,3};
int t;
for(int i=0;i<a.length;i++)
    for(int j=1;j<a.length;j++)
        if(a[j]>a[j-1]){ t=a[j]; a[j]=a[j-1];  a[j-1]=t;}
for(int i=0;i<a.length;i++)
    jOutput.writeString(" a["+i+"] is "+ a[i] + "\n");
```



If many arrays are defined in a program, and they are placed next door to each other in memory: they cannot be increased in size. This becomes a limitation for example if more data are entered into the program than space has been allocated for them. Another limitation of this particular form of group naming is that an array has to be a collection of elements of the same type. It is often useful to have a name for a bundle of elements that are of different types. In C, Pascal and various, other programming languages this possibility has been catered for by providing "*structure"* names for collections of differently typed variables. In order to access these individually a different naming convention has evolved. The name of the group is followed by a period, followed by the name of the individual variable. The structure is not of great use by itself but duplicated it provides a building block for more flexible linked data structures to handle collections of data that vary in size: dynamic data structures.

In an array, neighbouring elements can be obtained by adding offsets from a current index value. Stepping one by one through an array's index values allows all its elements to be processed. This depends on the data being stored in adjacent memory locations. An alternative approach to this task of storing data collections, allows individual elements to be stored anywhere in memory, so that collections can be incrementally built up or reduced in size as required. To do this reference information must be included with each data-element to allow its neighbours to be located. "*structure*" data types proved particularly suited to this construction. Such a structure can be made up from variables for holding the primary data, along with

variables of the type of the structure itself. These structure variables hold the location addresses of neighbouring structures in a collection. In Java these structures will be objects containing link variables of their own type, as references to neighbours.

Arrays allow lists of elements to be stored. An equivalent dynamic data structure can be built up from *structures* of the type *ListElement* defined in the following way:

```
class ListElement{
    public ListElement left=null, right=null;
    public Object object = null;
    public boolean comparable = true;
    ListElement(){}
    ListElement(boolean comparable){ this.comparable=comparable; }
}
```

This class definition is a template, which defines the data framework that each *ListElement* must have. It also provides two procedures, which create new structures of this type called constructors. A list is a sequence of these objects linked by the references *left* and *right*.

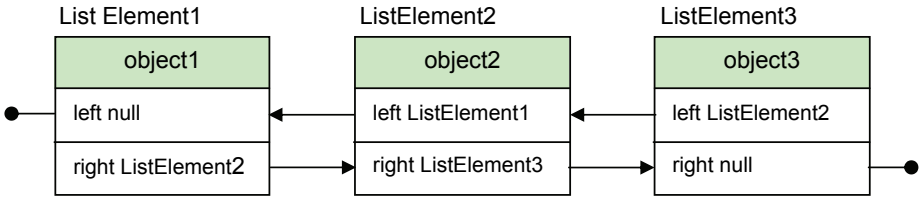


Figure 2.7 Double linked list

Although this list does not have a name for the data collection as a whole, the linked list structure allows all objects in its length to be processed using code of the form:

```
reference = firstElement;
while(reference != null){
    output(reference.object);
    reference = reference.right;
}
```

The list is accessed through a variable of type *ListElement*. Multiple lists can be set up and accessed by holding their leftmost element in a *ListElement* variable. This is the approach found in C, Pascal, Modula2 and similar languages. The only problem is that it is possible to build a variety of linked list structures using the *ListElement* as the building block. Since different accessing functions need to be used with these data structures, more information needs to be stored with each construction than a simple *ListElement* reference. In other words these list structures are of different types, and to handle them in a consistent way they should be given individual variable names with a type that reflects the kind of list-object that they reference.

Object Oriented languages such as Java not only allow these more complex data collections to be constructed but also to be given names as single objects. As objects of the same type, it will be possible to operated on them in common ways, and these operations implicitly define the type of object they are. The class definition of a *List* provides a reference to the first *ListElement* in the list and the last *ListElement*. More than this it includes the methods for operating on these structures in the following way:

```
class List{
    public int length = 0;
    public boolean comparable = true;
    public ListElement start = null,  finish = null;

    List( ){ };
    List(boolean c){this.comparable = c;};

    public List makeNewList(List lst){
        List lst0 = new List(true);  ListElement ref = lst.start;
        while(ref!=null){  lst0.append(ref.object);  ref=ref.right; }
        return lst0;
    }

    public void setComparable(boolean c){comparable = c;}

    public ListElement push(Object n){
        if (n==null) return null;
        ListElement m = new ListElement(comparable);
        m.object = n;   m.right = this.start;  m.left = null;
        if (this.start == null){this.finish = m;} else{ this.start.left = m;}
        this.start = m;  this.length = this.length + 1;
        return m;
    }
    public Object pop(){
        if (this.start == null)return null;
        Object m = this.start.object;  this.start = this.start.right;
        if(this.start != null) this.start.left = null;  else this.finish = null;
        this.length = this.length - 1;
        return m;
    }
    public ListElement append(Object n){
        if (n==null) return null;
        ListElement m = new ListElement(comparable);   m.object = n;
        if(this.finish == null){ this.finish = this.start = m;}
        else{  this.finish.right = m;  m.left = this.finish;  finish = m; }
        this.length = this.length + 1;
        return m;
    }
```

```
    public Object remove(){
        if(this.finish == null)return null;
        Object m = this.finish.object;
        if(finish.left == null){ this.start= this.finish = null;}
        else{  this.finish.left.right = null;  this.finish = this.finish.left; }
        this.length = this.length - 1;
        return m;
    }
    public Object delete(ListElement m){
        if(m == null) return null;
        else if (m.left == null) return this.pop();
        else if (m.right == null) return this.remove();
        else{ m.left.right = m.right;  m.right.left = m.left;
              this.length = this.length - 1;
        } return m.object;
    }
    public ListElement insertBefore(ListElement after,Object n){
        ListElement m=null;
        if (n==null) return null;
        if(after == null) m=this.append(n);
        else if(after.left == null) return this.push(n);
        else{
            m = new ListElement(comparable);  m.object = n;
            ListElement before = after.left;  before.right = m;  m.left = before;
            m.right = after;   after.left = m;  this.length = this.length + 1;
        }  return m;
    }
    public ListElement insertAfter(ListElement before,Object n){
        ListElement m=null;
        if (n==null) return null;
        if(before == null) m=this.push(n);
        else if ( before.right == null)return this.append(n);
        else{
            m = new ListElement(comparable);  m.object = n;
            ListElement after = before.right;  after.left = m;  before.right = m;
            m.left = before;  m.right = after;  this.length = this.length + 1;
        }  return m;
    }
    public List joinTo(List b){
        if(this.start==null)return b;
        if(b.start==null)return this;
        List a = new List();
        a.start = this.start;  a.finish = b.finish; this.finish.right= b.start;
        b.start.left = this.finish;
        return a;
    }
}
```

Lists and Trees

Compared with an array a list has a major draw back. Finding an element in a list involves following the links from one end of the list to the other searching for the required element. The same is true for an array where elements are stored in any order. However, if values are stored in order in an array, elements can be found by dividing the array into two halves selecting the half containing the target and then recursively subdividing the new reduced sub-array in the same way until the target is "*found*" or determined to be "*not present*". Instead of taking '*n'* steps for a list of '*n'* elements long this process requires '*log(n)'* steps. However adding values to an ordered array of values will involve moving entries along to make room for a new member, which on average still adds a serious overhead to the work.

In contrast the tree data structure allows a fast "*find*" operation to be applied while at the same time providing a fast insertion method. Tree data structures can be constructed from *ListElement* objects in the way shown in Figure 2.8 merely by employing a different linking strategy. Like bit patterns, the data element, building blocks, are the same but the overall type is determined by the permitted operations on the data. These will be provided by the Tree class methods. The only draw back is that tree building operations based on inserting new elements can distort the balanced shape of the tree, which is the property that makes fast '*find'* operations work.

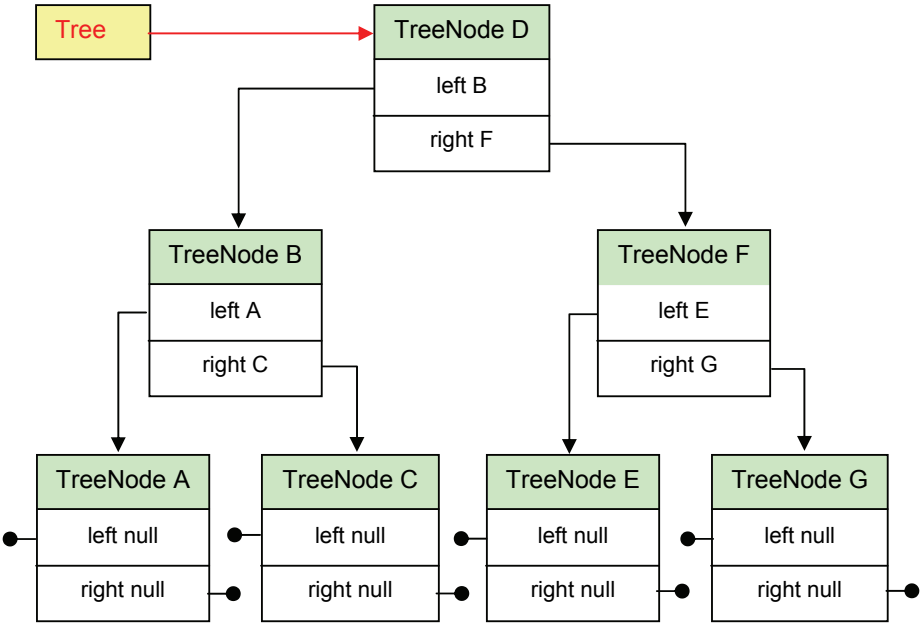


Figure 2.8 ListElements linked as a tree data structure

The tree like the list can be accessed and used through recursive procedures. Accessing elements in trees can be done in various ways depending on the application, however there are three simple tree traversal algorithms, which are used again and again. These can be expressed using the same three code statements but arranged in three different orders. Where the tree is used to store an ordered list of values shown in Figure 2.9, the output order for the prefix traversal will be D, B, A, C, F, E, G. The output order will be A, B, C, D, E, F, G for the infix traversal, and A, C, B, E, G, F, D for the postfix traversal.

```
abstract class Tree{
    static TextWindow IO = null;
    public boolean comparable = true;   public ListElement root = null;
    public Tree(){ };
    public Tree(TextWindow IO){Tree.IO = IO;};
    public abstract ListElement insert(ListElement ls, ListElement nw);
    public abstract void output(Object o);
    public void prefixTraversal(ListElement tree){
        if(tree==null)return;
        output(tree.object);
        prefixTraversal(tree.left);
        prefixTraversal(tree.right);
    }
    public void infixTraversal(ListElement tree){
        if(tree==null)return;
        infixTraversal(tree.left);
        output(tree.object);
        infixTraversal(tree.right);
    }
    public void postfixTraversal(ListElement tree){
        if(tree==null)return;
        postfixTraversal(tree.left);
        postfixTraversal(tree.right);
        output(tree.object);
    }
}
class StringTree extends Tree{
    public StringTree(TextWindow IO){Tree.IO = IO;};
    public ListElement insert(ListElement ls,ListElement ln){
        if(ls==null)ls= ln;
        else{String o1 = (String) ls.object, o2 = (String)ln.object;
            int test = o1.compareTo(o2);
            if (test > 0)    {ls.left  = insert(ls.left, ln);}
            else if(test < 0){ls.right = insert(ls.right,ln);}  // new string matches existing string
        }return ls;
    }
    public void output(Object o){IO.writeString((String)o+" ");}
}
```

```
public static void main( String[ ]  args ){
    ListElement root = null;String str=" ";
    StringTree stree = new StringTree(IO);
    IO.writeString("please enter 7 strings: ");
    for(int i=0; i<7;i++){
        str= IO.readTextString( );
        ListElement treeNode = new ListElement();
        treeNode.object=str;
        root= stree.insert(root, treeNode);
    }
    stree.infixTraversal(root);   IO.writeLine();
    stree.prefixTraversal(root); IO.writeLine();
    stree.postfixTraversal(root);IO.writeLine();
    }
}
```

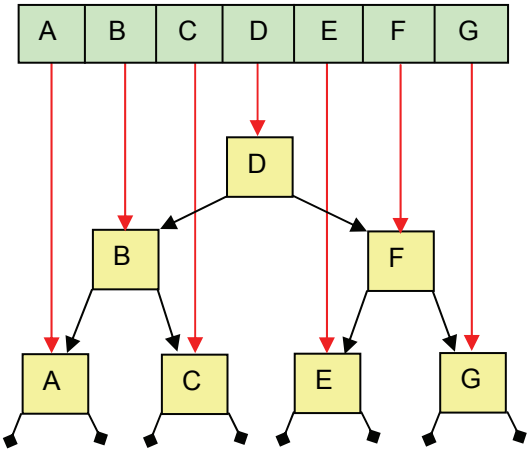


Figure 2.9 Rebalancing tree structures

If an already ordered sequence of input values is used to build a tree, using these procedures will create a single list, not a tree. There are several more sophisticated tree-building algorithms designed to keep the resulting tree reasonably well balanced as it is constructed. These can be found presented in detail in books on data structures and algorithms. However, an alternative, simpler approach to this problem is to record the number of levels in each tree-object, and when it becomes too unbalanced restructure the tree.

In practice it is often useful to switch the data structures used to implement data collections when processing different steps in a task. An example of the way this can be done is provided by tree data structures. An array can be used to rebalance a linked list tree structure built from an ordered list of values. If the linked list is traversed in infix order and the output is placed in an array of the appropriate size. The entries in the array will, by design, be in value order. If these are then accessed using a recursive binary subdivision of the array, selecting the middle value, and entering it back into a new linked list tree, the resulting tree will be balanced in the way shown in Figure 2.9.

A tree structure can be implemented either using an array or a double linked list. The tree given in Figure 2.8 could be stored in an array in the way shown in Figure 2.10, and output in infix-order A, B, C, D, E, F, G produced using the following code:

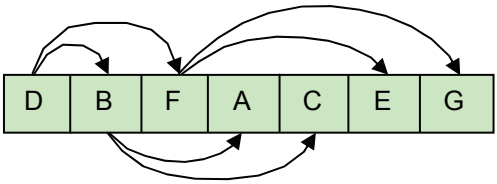


Figure 2.10 A tree in an array

```
public void infixTraversal(char[ ] tree, int index){
    if( index >= tree.length)return;
    infixTraversal(tree, index*2+1);
    output(tree[index]);
    infixTraversal(tree, index*2+2);
}
```

This arrangement is used to implement an alternative set of tree data types called *heap* structures: access is still based on the order of values but not in the same way.

Stacks, Queues and Deques

The same data structures are used again and again in different contexts, so it is natural that they should be considered as objects from the same class. However there are a series of common, dynamic, list based data structures that are essentially different types, that can be built up from the same *ListElement* units. The most general structure is the *List*, with the accessing methods given in the class listed above. However, a list that can only be accessed at one end is called a "*stack*" or a LIFO (last in first out list), similarly a list which has inputs at one end and outputs at the other is called a "*queue*" or a FIFO (first in first out list), and a queue that can go forwards or backwards is called a "*deque*" or double ended queue.

To identify these specialised lists as distinct types in Java can be achieved by grouping the methods that define these types in a similar construct to a "*class"* called an "*interface"*. The methods can all be implemented in the same class, however access to them can be limited by declaring variables of the types defined by the

interface names. The implementation class is then set up to "*implement*" these interfaces.

```
interface Stack{
    public ListElement push(Object n);
    public Object pop();
}
interface Queue{
    public ListElement append(Object n);
    public Object pop();
}
class List implements Stack, Queue{
    //  as above
}
```

This allows Stack and Queue objects to be generated by statements of the form:

```
Stack stack1  =  new List( );
Queue queue =  new List();
```

In these two cases *stack1* and *queue* can only use the methods in their interface definitions, not all the methods available in the List class.

Although Java provides a library of such structures in the *Collections* package, it is often necessary to build purpose-built linked list structures for graphics applications. An important example is producing an ordered threaded list of a polygon boundary. A list of coordinates representing a polygon boundary cannot be changed without losing the structure of the boundary however it is often necessary to arrange the coordinates in sequential order.

Although it is possible to create two coordinate lists each with its own order, they are of little use unless they can be cross-linked. This requires the *ListElement* structure to be extended to allow list elements to be linked together where they refer to the same coordinate object. The extended *ListElement* structure used in later examples is defined in the following way. The reference names link1 and link2 are provided to support this cross-reference between different list structures.

```
class ListElement{
    public ListElement left=null, right=null, link1=null, link2=null;
    public Object object = null;
    public int tag = 0;
    public String name = "";
    public boolean comparable = true;

    ListElement(){ }
    ListElement(boolean comparable){ this.comparable=comparable; }
}
```

Sets, Abstract Data Types and Encapsulation

What emerges is a collection of types, which can be implemented as linked lists, indexed arrays, or as tree structures without changing their external overall behaviour. If a "black box" approach is taken to defining a data type, then only the operations needed to define the correct behaviour of objects of the type need to be made visible to the user. All that needs to be known is the correct output that can be expected from an operation, generate from a given set of inputs, without the need to specify any mechanism for turning one into the other. This introduces *abstraction* and *hierarchy*.

Set objects provide a good example of this kind of data type where there are a variety of ways that they can be implemented. One way in which the set operations of union, intersection difference and symmetric difference can be implemented is based on ordered lists and a merge operation outlined in the merge-sort algorithm illustrated in Figure 2.5. The set must be represented by an ordered list of objects where no element is duplicated. The union of two sets can then be implemented by processing the two lists sequentially the smallest values first. The two lists are merged by comparing the two "next" elements from each list, and outputting the smaller if they are different, but outputting only one, and discarding the other, if they are the same. This gives a new list that conforms to the structure of the set, having no duplicate elements and holding an ordered series of values. The intersection of the sets can be implemented by only outputting one copy of any elements that match, discarding the rest. Clearly the implementation can use linked lists or arrays, and the user does not need to know which.

If the programmer wishes to work with the abstract properties of a polygon without having to consider its implementation at a "lower" level, the true representation of a polygon can be hidden in a polygon class. When the class is implemented in a program, a particular data structure to represent the polygon can be chosen, for example a list of vertex co-ordinates. A method called area can then be written to calculate the area of a polygon modelled in this way. However, the user may provide information in various ways through different class constructors to define the polygon. The system will then have to generate the internal representation of the polygon as a list of co-ordinates from the information given to the constructors. If a particular polygon has been given the variable name *polygon1* then the area of the polygon can be returned by the "area" method using a statement of the form *polygon1.area( )*, and the real data structure need never be referred to.

Java allows objects like polygons to be treated in the same way that numbers are treated in simpler languages. Their type is defined by a class definition, which includes the operations that are permitted on the objects from the class of that type. The class definition also contains the data structures used by the sub-programs (called methods), to implement the operations on objects of the class.

If the polygon class is implemented well, then a polygon object can be worked with in much the same way that an integer object can be worked with, in expressions, relationship-tests and the like. The rules governing these operations will be different and often more complex for "higher level" objects, but a more unified programming environment results.

The object-oriented approach also allows a more natural use of names to be adopted so that program code reflects the objects, which are being worked on, like polygons, in a more direct way, than was possible in previous programming languages. In Java the main program building blocks become the class and interface definitions. These define the data structures necessary to implement objects of the class, and to manage the set of objects generated by the class. Each class has the potential mechanism to generate data structures to represent or "instantiate" multiple objects of the type that the class defines. These objects can be worked with using variables of the type the class defines, which can be used in programs, very much like variables holding numbers, by assigning object to them.

Hierarchy Inheritance and Abstraction.

This process of hiding implementation details is called encapsulation and makes many programming tasks much clearer. An important aspect of the hierarchy supported by the class structure is that one class can be defined as a refinement or modification of another class. The new class "inherits" much of its structure from its parent class, but has properties and methods of its own. This makes it possible to implement programs in a way that minimises duplication, which is very important in maintaining programs, so that changes can be carried out in as few locations as possible. It also allows template classes to be defined where the full implementation is left unfinished to be implemented by inherited classes. These are called Abstract classes illustrated by the *Tree* class given above. For example a *StringTree* class can inherit from the *Tree* class its general tree traversal methods but must provide the *output* and the *insert* procedures in the specialised form required by String objects for their *write* and *compareTo* methods.

In Java direct inheritance is permitted from only one "super" class. In real life, objects can be thought of belonging to many classificational sets. A car is a "vehicle" and also a "manufactured object". The flexibility this demands in a programming language, however, can lead to ambiguities and complex errors. Consequently, Java provides the different construction called an Interface illustrated in the case of List objects. This extends the methods and names, which can be applied to objects in one class as though they were objects from a different class of a different type

The next step is to provide equivalent input output facilities for graphic objects to that provided for text. One of the operations necessary for spatial objects such as polygons is presenting them in graphic displays. In the next chapter a simple display window class is introduced which will allow basic display operation to be executed.

http://www.springer.com/978-1-84800-178-7

Integrated Graphic and Computer Modelling
Thomas, A.
2008, XXI, 685 p., Hardcover
ISBN: 978-1-84800-178-7