# ELEC1004
# Object-Oriented Programming

University College London
Richard James
r.james@ee.ucl.ac.uk

## 1 Introduction

The aim of this course is to provide a grounding in the fundamental elements of object-oriented programming in particular and programming in general, assuming no initial knowledge of the subject. The Java language is used to teach these concepts. The syllabus is as follows:

- High level languages
  - Need for high level languages
  - Brief history of the development of high level languages
  - Compiled versus interpreted languages Procedural versus object oriented languages
- The Linux operating system
  - Terminal use and File management
  - Text editors
  - Developing and running Java programs
- Introduction to Programming
  - Algorithms and programs
  - How to structure a program
  - Flow diagrams
- The Java Programming Language
  - Introduction to object oriented programming
  - Applications
  - Classes
  - Data types
  - Operators
  - Terminal input/output
  - File input/output
  - Control flow
  - Arrays
  - Methods
  - Inheritance
  - Applets

Recommended texts:

[A] P. J. Deitel, H. M. Deitel, "**Java: How to Program**", Prentice Hall, 2004

[B] R. Morelli, "**JAVA, JAVA, JAVA! Object Oriented Problem Solving**", Prentice Hall, 2003

[C] R. Winder and G. Roberts, "**Developing JAVA software**", J Wiley, 2000 [COMPUTER SCIENCE D 20 JAV:WIN]

## 1.1 Programming Languages

Programming languages can be categorized into three types; machine languages, assembly languages and high-level languages.

At the lowest level a computer executes its own machine language. The instruction set that makes up this language is machine dependent and is defined by the hardware design. Programming in machine-languages is time-consuming. This code is difficult to read, which increases this risk of programmatical error. Finally, it is not portable.

To ease development Assembly language was invented. Key words are used to represent the instruction set and data may be represented by decimal numbers. In order for a computer to execute an assembly-language program it must be converted into machine code. Assemblers were developed for this purpose. Assembly language made programming simpler, but the code is still not portable and many instructions are required to perform even simple tasks.

High-level languages overcome these two difficulties. The instructions resemble mathematical notation, and a single line can accomplish much compared to a line of assembly language. Programming is a faster process and the code is, for the most part, machine independent. There are two ways in which high-levels codes may be executed. Firstly, they may be converted into machine code, for a specific target processor, using a program known as a compiler. Alternatively, interpreter programs have been developed which execute the code directly, but more slowly. Since compilation can be time consuming, interpretation is attractive during the development stages of a program.

Listing 1 shows a simple high-level program that adds two numbers (this is shown for the sake of illustration, the syntax will be described in more detail later) and Listings 2 and 3 show the equivalent assembly code for two different CPUs, the 68k and the x86.

Listing 1: A high-level (Java) code that adds two numbers

```
1 public static void addition() {
2     int x = 3;
3     int y = 4;
4     int z = x + y;
5 }
```

Listing 2: 68k assembler code that adds two numbers

```
1 movem.l d5-d7,-(sp)
2 moveq   #3,d7
3 moveq   #4,d6
4 move.w  d6,d5
5 add.w   d7,d5
6 movem.l (sp)+,d5-d7
7 rts
```

Listing 3: x86 assembler code that adds two numbers

```
1  push    ebp
2  mov     ebp,esp
3  sub     esp,0
4  mov     dword ptr [x],3
5  mov     dword ptr [y],4
6  mov     eax,dword ptr [x]
7  add     eax,dword ptr [y]
8  mov     dword ptr [z],eax
9  mov     esp,ebp
10 pop     ebp
11 ret
```

## 1.2 A Brief History of High-Level Language Development

The early high-level languages, C, FORTRAN and BASIC are procedural. The unit of programming is a function. Groups of actions that collectively perform some task are collected together in a function and functions are grouped together to form a program. Typically, in engineering, the actions are a mathematical calculation, e.g. modelling a transistor.

| | |
|---|---|
| Early 1950's | FORTRAN (FOrmula TRANslation) |
| | COBOL (COmmon Business Oriented Language) |
| Mid 1950's | ALGOL (ALGOrithmic Language) |
| 1960's | Chaotic period with many languages |
| | BASIC (Beginners All-purpose Symbolic Language) |
| 1970's | Structured languages encouraging good programming: PASCAL, C |
| 1980's | Object Oriented Languages: C++ |
| 1990's | Java |

Java, C++ are object-oriented languages. The unit of programming is the class from which objects are created (instantiated). An example of an object is a window on a computer screen. The class will contain all the information needed to define a window, height, width, colours, menu bars, etc. A class is rather like the blue print of a house. You can create many instances (objects) from a class, e.g. you can open several windows on your screen. Java and C++ classes contain methods similar to procedural language functions, allowing objected-oriented design to be combined with traditional engineering requirements.

## 1.3 History of Java

The Java programming language was originally called Oak. It was developed at Sun Microsystems around 1990 by James Gosling with inspiration from Bill Joy. Intended as control software for embedded microprocessors in consumer items (cable set-top boxes,VCR's, toasters) and also personal data assistants (PDA). Consequently it needed to be be platform independent, since the processors would come from multiple manufacturers as well as reliable and compact.

The interactive TV and PDA markets did not take off at that time, but in 1993 the internet and the Web began to explode. Sun changed the name to Java and shifted to internet applications.

## 1.4 Why Learn Java?

Java was chosen for this course for a number of reasons. It is most widely used object-orientated language and knowledge of Object-Oriented Design is requested by many employers and by the IET. It is a well-structured language that is relatively easy to learn. Once you have mastered Java other object-oriented languages, such as C++, will be much easier to learn if you have to do so. As an added bonus the procedural aspects of Java are identical to those of C. A knowledge of Java is a pre-requisite for those wishing to take third and forth year options given by the Computer Science Department.

## 1.5 Getting and Using Java

In order to create and execute a Java program you must first download the the Java Development Kit 6 (JDK) from `http://java.sun.com/javase/downloads/index.jsp`. An introductory tutorial can be found here: `http://java.sun.com/docs/books/tutorial/`

## 2 A Simple Java Application

Applications are Java programs that are executed from the computer's command window, for example the from the UNIX terminal window or the MS Windows XP Command Prompt window. Applets are Java programs that run in Web browser or the appletviewer (provided in the JDK). In the first part of the course we will concentrate on the use of applications.

All Java programs are classes. The program begins with a couple of comment lines. The text between the two delimiters /* and */ is ignored by the compiler and generates no bytecode. Such lines are added to help the reader understand the program. At the start of a program it is desirable to add a few comment lines detailing the author of the class, the creation date of the code and a brief description. Another possible type of comment is a comment line; any text that follows the delimiter // will be ignored by the compiler.

```
1  /* HelloWorld.java implements an application that
2  ** prints "Hello World!" to standard output */
3  public class HelloWorld
4  {
5      // main method, entry point for Java application
6      public static void main(String[] args)
7      {
8          System.out.println("Hello World!");
9      } // end main method
10 } // end class HelloWorld
```

Line 3 begins the class declaration for the class HelloWorld.

```
public class HelloWorld
```

Note that the filename must be the same as the class name, and should have a .java extension. It is a convention to begin class names with an upper case letter. The class name is known as an identifier. Class names can comprise a series of letters, digits, underscores and dollar signs. However, the identifier name must not begin with a number. For class names formed from multiple words, begin each word with an upper case letter but omit the spaces, e.g. HelloWorld. The keyword public indicates a general access to this class and will be explained in detail later.

The parentheses, { } on lines 4 and 10 mark the start and end of the class declaration. A class usually contains one or more methods. In an application one of these must be the main method; the starting point of every Java application.

```
public static void main(String[] args)
```

The meaning of the static keyword will be explained later. For the time being take it for granted that the main method should be defined in this manner. Methods are callable sections of code that are designed to perform some task. On completion of this task they are able to return some information to the caller. However, in this case, the void keyword indicates that no information is to be returned. Following the method name are parentheses ( ). The text within is a list of arguments sent from the caller to the method, as illustrated by String[] args in the example. The parentheses, { } on lines 4 and 9 mark the start and end of the body of the method declaration.

Line 8 is the command which displays the desired output. System.out is the standard output object and println is a method that outputs to the command window.

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

The characters between quotation marks form a string literal. Each executable statement should terminated with semicolon.

## 2.1 Compiling and Executing the Program

The compilation and execution of this code can be de divided into a number of steps. First of all the program must be entered into an editing program, for example Notepad or emacs. The next step is the compilation of the Java program. For the example shown above this is achieved using the following command:

```
javac HelloWorld.java
```

A file called HelloWorld.class is created which contains bytecodes. Bytecodes represent the tasks to be executed in the program and are executed by the Java Virtual Machine (JVM), which is part of the JDK. A virtual machine is a program that simulates a computer. The virtual machine has knowledge of the underlying operating system and hardware; however the bytecode is insulated from this. The bytecode is not dependent on the computer hardware and so it is portable. In order to execute the bytecode the following command is entered into the command window:

```
java HelloWorld
```

This starts the JVM. The next step is known as loading. Any .class files used by the program are transferred into primary memory. Once the classes are loaded a bytecode verifier run, which ensures that they are valid and that no security restrictions are violated.

Finally, the program is executed. Java uses a clever mix of interpretation and just-in-time (JIT) compilation to improve performance. Since compilation is time consuming, code that only runs once will tend to be interpreted whereas heavily used code, such as loops, may be compiled.

## 2.2 Programming Style

The following list describes a number of important things to consider when writing a program, in order to make it easier to read and understand.

- Program heading: All programs must start with a set of comment lines that give the title of the program;
  - a brief, e.g. one line, description of the;program;
  - the author of the program;
  - the date(s) on which the program was created;
  - the date(s) of any revision(s) of the program.
  - Comments: Helpful notes to the reader and to yourself can be included in English in the source code explaining what parts of the program do. Such comments must be proceeded by a // or placed between a /* and a */.
- Indentation: Programs have levels of operation which can be indicated by indenting sections of code from the left-hand side. This makes the program much more readable.
- Top-down: Top-down design of a program means that a program to perform complex tasks is broken down into smaller pieces. Each piece may be further and divided and redivided until each piece resembles a simple task. Each piece may be written separately, and then the whole is assembled. Java is well-suited to this design methodology since it includes the idea of a methods; a separate method could perhaps be written to perform each separate task.

- Variable names: Always use names that help identify the meaning of the variable, so you might choose the name temp to store the values of temperature in a set of data, or grade to hold the letter grade which a candidate got in an examination. As a matter of good programming, do not use single letters, e.g. volt and not v for a variable holding a voltage.

# 3 An Introduction to Classes

This section begins by describing a simple class that holds the name of a person. The class is defined as follows:

```
1  // Person.java
2  // stores the name of a person
3  public class Person
4  {
5      // private field that stores the name
6      private String name;
7
8      public void setName(String str)
9      {
10         name = str;
11     }
12
13     public String getName()
14     {
15         return name;
16     }
17 }
```

Each instance of the class will have its own copy of the variable called name. This type of variable which is declared within the class declaration, but outside any method declarations is known as an instance variable or a field.

The private keyword indicates that the name variable is only accessible from methods defined within the Person class. This is known as data hiding. Variables or methods that are public may be accessed externally. It is good practice to make all instance variables private.

In order to change and to read the value of the instance variables it is common to provide get and set methods. Here the set method is declared as:

```
    public void setName(String str)
```

This function does not return anything to the caller and so its return type is void. When the method is called it is passed a string which is then assigned to the instance variable, name. An advantage of using a set method is that input argument can be checked in order to ensure that the instance variables do not take on unexpected values.

The get function is declared as:

```
    public String getName()
```

This declaration states that the method does not require any arguments when it is called, since the parameter list is empty, and that it returns a String to the caller. The return statement ends the execution of a method and returns program execution to the caller. It is also the means by which arguments are returned to the caller. In this case it returns the String name.

In order to use this class it is necessary to first create an instance of it. The following code creates two instances of the Person class, calls the set method of each, and then finally displays the names.

```
1  // PersonTest.java
2  // Creates two instances of the Person class
3  public class PersonTest {
```

```
4       public static void main(String[] args)
5       {
6           Person per1 = new Person();
7           Person per2 = new Person();
8
9           per1.setName("John");
10          per2.setName("David");
11
12          System.out.printf("First  person is %s\n", per1.getName());
13          System.out.printf("Second person is %s\n", per2.getName());
14      } // end main method
15  } // end class PersonTest
```

Line 6 initializes the local variable per1 with the result of new Person(). Note that the parenthesis are required. This expression creates a new instance of the class Person and returns a reference to this object. A reference stores the location of an object in the computer's memory, and will be described in more detail later. On Line 9 the . operator means call setName method belonging to the per1 object. This sets the name field of this particular object to "John". Finally line 12 displays the name stored in the object.

In order to run this program is necessary to compile both Person.java and PersonTest.java, using the following command:

```
javac Person.java PersonTest.java
```

PersonTest can the be executed using:

```
java PersonTest
```

This produces the following output:

```
First person is John
Second person is David
```

## 3.1 Constructors

Previously when the Person object was created the name variable was initialized to null. It is possible to provide a value for name when the object is created, by declaring a constructor. The new keyword calls the class's constructor. The call to the constructor is indicated by the class name followed by parenthesis. If a class declaration does not explicitly include a constructor then the compiler provides one by default. When defining a constructor it is possible to define the number and type of arguments used for the initialization. More than one constructor may be present in a class allowing creation of instances to be initialized by different sets of input parameters.

The code that follows is a modified version of the Person class introduced above in which initialization is performed via a constructor.

```
1   // Person2.java
2   // stores the name of a person
3   public class Person2 {
4       // private field that stores the name
5       private String name;
6
7       public Person2(String str)
8       {
9           setName(str);
10      }
```

```
11
12      public void setName(String str)
13      {
14          name = str;
15      }
16
17      public String getName()
18      {
19          return name;
20      }
21  }
```

Note that with a constructor defined the default constructor is no longer generated and so a call to new Person2() gives the compilation error: cannot find symbol : constructor Person2()

```
1  // PersonTest2.java
2  // Test the creations of the Person2 class
3  public class PersonTest2 {
4      public static void main(String[] args)
5      {
6          Person2 per1 = new Person2("John");
7          Person2 per2 = new Person2("David");
8
9          System.out.printf("First  person is %s\n", per1.getName());
10         System.out.printf("Second person is %s\n", per2.getName());
11     } // end main method
12 } // end class PersonTest2
```

The diagram below shows the Unified Modeling Language (UML) class diagram for the Person2 class. The top row gives the class name. The second section details the class fields, including instance and static variables. For this class only the instance variable name is present. The final section details the constructor and the class methods. The Person2 class contains a constructor and a get and set method. The + and − indicate the access permissions. + represents public access and − private access.

| **Person2** |
| --- |
| − name : String |
| <<constructor>> Person2(name : String)<br>+ setName( name : String )<br>+ getName( ) : String |

## 3.2 Static Members

We were introduced earlier to a static class method called main. A static member is used to store classwide information. For instance if a class contains a static int, only one copy of this int will exist for all the objects of the same class. Furthermore, this int is available if no objects of the class have been instantiated. Static class member should be accessed by using the . operator on the class name.

For example consider a class that stores the date. We might want to introduce a default date member, which is used to initialize the class when no argument constructor is called. The default date would be best set as a static member of the class.

# 4 Language Fundamentals

In this section the primitive data types are introduced. You will learn the types available and how to declare and initialize them. Additionally the operations available for these types, such as addition and multiplication, are detailed.

## 4.1 Variables

The primitive Java data types are:

| | | |
|---|---|---|
| byte | Holds an integer within the range: -128 to 127 | [8 bits] |
| short | Holds an integer within the range: -32768 to 32767 | [16 bits] |
| int | Holds an integer within the range: -2147483648 to 2147483647 ($-2^{31}$ to $2^{31}-1$) | [32 bits] |
| long | Holds an integer within the range: -9223372036854775808 to 9223327036854775807 ($-2^{63}$ to $2^{63}-1$) | [64 bits] |
| float | Holds a floating point number within the range: $\pm 1.40129846432481707 \cdot 10^{-45}$ to $\pm 3.40282346638528860 \cdot 10^{38}$ | [32 bits] |
| double | Holds a floating point number within the range: $\pm 4.94065645841246544 \cdot 10^{-324}$ to $\pm 1.79769313486231570 \cdot 10^{308}$ | [64 bits] |
| char | Holds a single 16-bit Unicode character. It has a minimum value of '\u0000' and a maximum value of '\uffff" | [16 bits] |
| boolean | Holds either true or false | [-] |

All integers are signed two's complement integers. For integral variables the default choice is usually int; however, for large data sets it may be preferable to use short or even byte types to save memory. A long should be used when you need a range of values wider than those provided by int.

Floating point types are defined according to the IEEE 754 floating point standard. The default choice for decimal values is the double. In order to save memory it may be preferable to choose the float type. These two data types should never be used for precise values, such as currency. For that, the java.math.BigDecimal class should be used instead.

The data types described above were all primitive types. Java also provides the String class, which is used to hold a string of characters, e.g. "This is a string." Strings are immutable objects, which means they can not change in value after they are created. The behaviour of immutable objects is similar to the primitive types. This difference between primitive data types and objects will be discussed in more detail later.

Declarations are made as in the following examples:

```
int     iValue = 2;
long    lValue = 23567894532L;
float   fValue = 3.1f;        // f or F suffix for float values
double  dValue1 = -3.65d;     // d or D suffix for double values
double  dValue2 = 1.23;       // assumed double
double  dValue3 = 4.562e-5;   // 'e' = exponent
char    cValue = 'g';
boolean bValue = false;
String  sValue = "another string";
```

In the example above the integer type is initialized by a base 10 number. It is also possible to initialize such types by hexadecimal or octal integers. Hexadecimal initializers should be prefixed by the character 0x and octal initializers should be prefixed by a 0.

### 4.1.1 Default Values

It is not strictly necessary to assign a value when a variable is declared in a class (also known as a field). Such variables that are declared but not initialized will be set to a reasonable default by the compiler. Generally speaking, this default will be zero or null, depending on the data type. However, relying on such default values is generally considered bad programming style. The default value for each primitive type is given below.

| Data Type | Default Value (for fields) |
|---|---|
| byte | 0 |
| short | 0 |
| int | 0 |
| long | 0L |
| float | 0.0f |
| double | 0.0d |
| char | '\u0000' |
| String (or any object) | null |
| boolean | false |

Local variables are different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error.

## 4.2 Names of Variables and Methods

Any data which is used in a program has to be stored and retrieved from the memory of the computer. The user is free to invent his or her own names for the storage locations in memory where the data is held. We are talking here of quantities which, in general, will change as the program runs (that is, variables) rather than those which don't change (constants). Also methods which the user writes in the program (as opposed to those like System.out.println() which already exist) can be given a name of the user's choice. However, each variable or method in a program must be given a legal name, or identifier.

To be legal, the name must be a sequence of letters, digits, and underscores (_). Additionally, the dollar sign and pound sign characters may be used, but by convention, they are always avoided. Any other symbols, such as %, & etc. are forbidden. By convention variable and method names should begin with a lower case letter, (an underscore is also permitted but avoid this! It is used in a specialised manner in more advanced applications). Starting a name with a number is not allowed. If the name you choose consists of only one word, spell that word in all lowercase letters. If it consists of more than one word, capitalize the first letter of each subsequent word, for example studentName.

Always use names that help identify the meaning of the variable, as in the above example, so you might choose the name temp to store the values of temperature in a set of data, or grade to hold the letter grade which a candidate got in an examination. As a matter of good programming avoid single letter names, e.g. use volt and not v for a variable holding a voltage. Using all upper case letters in a name is, by convention, reserved for Constants which are discussed below. Note that the compiler distinguishes between upper and lower case letters so that, for example, the method System.out.println() cannot be written System.out.PrintLn(). In addition, a variable name cannot be one of the reserved words. These are words, such as main, which already have a meaning within the Java language. A list of reserved words is follows:

| | | | | | |
|---|---|---|---|---|---|
| abstract | assert | boolean | break | byte | case |
| catch | char | class | continue | default | do |

| | | | | | |
|---|---|---|---|---|---|
| double | else | extends | false | final | finally |
| float | for | if | implements | import | instanceof |
| int | interface | long | native | new | null |
| package | private | protected | public | return | short |
| static | strictfp | super | switch | synchronized | this |
| throw | throws | transient | true | try | void |
| volatile | while | | | | |

Additionally the following three words are reserved though are not used by Java:

| | | |
|---|---|---|
| const | goto | var |

So, for example, the following are legal identifiers:

| | | | | |
|---|---|---|---|---|
| rate | radius | batch45 | topRate | top_rate |

whereas the following are not:

| | |
|---|---|
| 45b | (it begins with a number) |
| top rate | (space not allowed) |
| rate_% | (contains the %) |
| long | (this is a reserved word) |
| top-rate | (don't confuse the dash - with the underscore _) |

As an exercise, state which of the following are legal identifiers, and for those which aren't, say why:

| | | | |
|---|---|---|---|
| case | long_int | 5thday | noonDay |

## 4.3 Constants

Constants may be created by using a final declaration, e.g.

```
public final double K_BOLTZMANN = 1.380650324e-23;
```

By convention the name of such a constant is written entirely in upper case letters. The value of a constant declared in this manner cannot be altered after it has been initialised. The mathematical constants, $\pi$ and $e$ (the base of natural logarithms) are provided by the Java Math class, as Math.PI and Math.E respectively.

It is good practice to declare arguments passed to a method as final if they are not altered by the method.

## 4.4 Operators

The following operators are called arithmetic operators:

| | |
|---|---|
| + | addition |
| − | subtraction |
| * | multiplication |
| / | division |
| % | the remainder, or modulus, operator |

These are all binary operators, that is they need a quantity on each side of them in order to make sense. These quantities are called the operands.

We have already encountered one of the assignment operators, =, earlier. There are a further four assignment operators, as detailed below:

| Operator | Type | Example | Equivalence |
|---|---|---|---|
| = | assignment | x = y | |
| += | addition assignment | x += y | x = x + y |
| −= | subtraction assignment | x −= y | x = x − y |
| *= | multiplication assignment | x *= y | x = x * y |
| /= | division assignment | x /= y | x = x / y |

These operators allow certain statements to be written more concisely. For example if we wanted to add 5 to the variable x we could either write x=x+5; or x+=5;. A discussion of the assignment operator, =, as applied to instances of objects will follow later.

Relational operators are binary operators used to perform tests on their two operands:

| | | | |
|---|---|---|---|
| == | relational is equal to | != | relational is not equal to |
| < | less than | > | greater than |
| <= | less than or equal to | >= | greater than or equal to |

A discussion of the equality operator, ==, as applied to instances of objects will follow later. Using relational operators in combination with an if statement makes it possible to control the flow of a program.

Logical operators allow more complex conditions to be formed by combing simpler conditions together:

| | |
|---|---|
| && | logical AND |
| \|\| | logical OR |

The logical complement operator, !, is a unary operator that performs a logical NOT on the boolean that it prefixes.

Incrementing and decrementing operators are shown below. These are the unary operators ++ and −− which, respectively, add one to or subtract one from a variable, i.e.

| Operator | Type | Equivalence |
|---|---|---|
| ++expr | unary preincrement | expr = expr + 1 |
| expr++ | unary postincrement | expr = expr + 1 |
| −−expr | unary predecrement | expr = expr − 1 |
| expr−− | unary postdecrement | expr = expr − 1 |

These unary operators can come before or after the variable (but not both). The effect may be different in the two cases: If the operator comes before its operand, then the increment or decrement is carried out before the value of the variable is used in the expression in which it occurs. If the operator comes after its operand, then the increment or decrement is carried out after the value of the variable is used in the expression in which it occurs.

The following program illustrates the difference between preincrementing and postincrementing:

```
1  // Demonstrate preincrementing and postincrementing
2  public class IncrementingExample
3  {
4      public static void main(String[] args)
5      {
6          int i=1;
7          System.out.printf("pre-incrementing: %d\n", ++i);
8          System.out.printf("no  incrementing: %d\n", i);
9          i=1;
10         System.out.printf("post-incrementing: %d\n", i++);
11         System.out.printf("no   incrementing: %d\n", i);
12     }
13 }
```

```
pre-incrementing: 2
no  incrementing: 2
post-incrementing: 1
no   incrementing: 2
```

There are another set of operators, known as bitwise operators that can be applied to integer types, which perform logical operations on the individual bits that make up a number. These operators can be applied to int and long values. If an operand is shorter than an int, it is promoted to an int prior to the operation. These operators are less commonly used and will not be examined.

| | |
|---|---|
| << | shifts a bit pattern to the left |
| >> | shifts a bit pattern to the right |
| & | performs a bitwise AND operation |
| \| | performs a bitwise inclusive OR operation |
| ^ | performs a bitwise exclusive OR operation |
| ~ | performs a bitwise complement |

## 4.5 Operator Precedence

The order in which calculation is performed is determined by the precedence of the operators involved. For example if calculating $x \cdot y + z$ the multiplication of $x$ and $y$ is performed before the addition of $z$. The precedence is given in the table below. The closer to the top of the table an operator appears, the higher its precedence. Operators with higher precedence are evaluated before operators with a lower precedence.

| | | | | | |
|---|---|---|---|---|---|
| parenthesis | () | [] | | | |
| postfix | expr++ | expr-- | | | |
| unary | ++expr | --expr | +expr | -expr | ! |
| multiplicative | * | / | % | | |
| additive | + | - | | | |
| relational | < | <= | >= | > | |
| equality | == | != | | | |
| logical AND | && | | | | |
| logical OR | \|\| | | | | |
| assignment | = | *= | /= | += | -= |

Operators on the same line have equal precedence. When operators of equal precedence appear in the same expression, a rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated from left to right; assignment operators are evaluated right to left. The order of evaluation for a simple expression is demonstrated below.

| a | * | b | / | c | + | d | * | e |
|---|---|---|---|---|---|---|---|---|
| | 1 | | 2 | | 4 | | 3 | |

If you are unsure about the order of evaluation of an expression then use parenthesis to ensure the correct order. Using redundant parenthesis can sometimes improve the readability of code.

# 5 Standard Mathematical Functions

The standard Java math library (Math class) contains several standard mathematical functions. The more commonly used ones are listed below.

| | |
|---|---|
| Math.sin($x$) | Returns the trigonometric sine of $x$ |
| Math.cos($x$) | Returns the trigonometric cosine of $x$ |
| Math.tan($x$) | Returns the trigonometric tangent of $x$ |
| Math.acos($x$) | Returns the arc cosine of $x$; $0 \leq x \leq \pi$ |
| Math.asin($x$) | Returns the arc sine of $x$; $-\pi/2 \leq x \leq \pi/2$ |
| Math.atan($x$) | Returns the arc tangent of $x$; $-\pi/2 \leq x \leq \pi/2$ |
| Math.atan2($y, x$) | Returns $\theta$ from the conversion of $(x, y)$ to polar coordinates $(r, \theta)$ |
| Math.exp($x$) | Returns e (the base of natural logarithms) raised to the power of $x$ |
| Math.log($x$) | Returns the natural logarithm (base e) of $x$ |
| Math.log10($x$) | Returns the base 10 logarithm of $x$ |
| Math.abs($x$) | absolute value of $x$ |
| Math.pow($x, y$) | Returns the $x^y$ |
| Math.sqrt($x$) | Returns positive square root of $x$ |
| Math.ceil($x$) | Returns the smallest integer that is greater than or equal to $x$ |
| Math.floor($x$) | Returns the largest integer that is less than or equal to $x$ |
| Math.max($x, y$) | Returns the greater of $x$ and $y$ |
| Math.min($x, y$) | Returns the smaller of $x$ and $y$ |
| Math.toDegrees($\theta$) | Converts an angle measured in radians to degrees |
| Math.toRadians($\theta$) | Converts an angle measured in degrees to radians |

It also includes the hyperbolic trigonometric functions, as well as the constants Math.PI, $\pi$, and Math.E, $e$ the base of natural logarithms. A complete list can be found here:
http://java.sun.com/javase/6/docs/api/java/lang/Math.html

# 6 Errors

Errors can be divided into two groups, compilation errors and run-time errors. Compilation errors are detected when javac is run. They arise due to syntax errors in the entered code, for example typos. Bytecode will not be generated if compilation errors are detected.

Run-time errors can be encountered when executing code that has been successfully compiled by javac. An example of such an error would be attempting of open a file that does not exist.

## 6.1 Compilation Errors

The example below shows the code and error report generated for a piece of code that contains a typo.

```
8  totalCharge = cap*potDiff
9  chargeDensity = totalCharge/plateArea;
```

```
Example1.java:8: ';' expected
                        totalCharge = cap*potDiff
                                                 ^
1 error
```

The error message gives first the file name that generated the error, Example1.java in this case. Following this is the line number where the error occurred. In this case the statement on line 8 is not correctly terminated by a semicolon. The compiler reaches the next statement on line 9 without finding a semicolon and then complains. An ^ is output, pinpointing the error.

There are many other types of compilation errors, for example if the type of the arguments passed to a method do not match its declaration. Another example is trying to assign a double to an int type:

```
3      double dValue = 3.3;
4      int iValue = dValue;
```

```
Example2.java:4: possible loss of precision
found   : double
required: int
                int iValue = dValue;
                             ^
1 error
```

This error may be resolved by casting (converting) the double type to an int by changing line 4 to:

```
    int iValue = (int)dValue;
```

Note that the following line compiles, as an int type is promoted automatically to a double type with no loss of precision:

```
    dValue = iValue;
```

## 6.2 Run-time errors (Execution errors)

Run-time errors can either be fatal or non-fatal. If the error is non-fatal the program will execute to completion; however, the result will be incorrect. Fatal errors occur when the system cannot recover from an error that is encountered. Exceptions are the customary way in Java to indicate to a calling method that an abnormal condition has occurred. When such an error occurs an exception object is thrown, and unless it is handled in some way, the program will quit prematurely and report the type of error that occurred. How to throw and catch exceptions will be dealt with in the second year course.

### 6.2.1 Fatal run-time errors

The following code attempts to divide the integer value 1 by the integer value 0.

```
8       int xplus = 1/0;
9       System.out.printf("xplus = %d\n", xplus);
```

Line 8 causes an java.lang.ArithmeticException to be thrown and the program exits.

```
Exception in thread "main" java.lang.ArithmeticException: / by
zero
        at Example3.main(Example3.java:8)
```

### 6.2.2 Non-fatal run-time errors

When a non-fatal run-time error occurs the program compiles; however, the answer is wrong. The code does not correctly represent the calculation you are attempting to perform. An example of this would be attempting to evaluate directly $\sin(x)/x|_{x=0}$. The cause of such an error can usually be found by checking the program output against the desired answer calculated by hand for a number of trial input values. The following example shows the code and output when attempting a (floating point) division by zero.

```
8       double xplus = 1.0/0.0;
9       System.out.printf("xplus = %f\n", xplus);
10      double xminus = -1.0/0.0;
11      System.out.printf("xminus = %f\n", xminus);
12      double xundef = 0.0/0.0;
13      System.out.printf("xundef = %f\n", xundef);
```

```
xplus = Infinity
xminus = -Infinity
xundef = NaN
```

# 7 Input and Output

The section describes how to read and write from and to the command window. The procedure for reading and writing files is closely related and is also demonstrated.

## 7.1 Outputting to the Command Window

The following three types of print statements all perform the same function, they output the value of x:

```
1  int x = 3;
2  System.out.println("The value of x is " + x);
3  System.out.print  ("The value of x is " + x + "\n");
4  System.out.printf ("The value of x is %d\n", x);
```

The function println automatically moves onto a new line after displaying a String whereas print and printf do not. For print and printf it is necessary to include a newline character \n to do so.

In lines 2 and 3 the effect of the + operation acting on the String creates a new String which is the concatenation of `"The value of x is "` with the value of x converted to a String. This conversion is automatic. If an object is used in place of an int, the toString() method of the object will be called instead and the result will be concatenated.

Within a String is possible to use the following special characters:

| | |
|---|---|
| \n | new line |
| \r | line feed (jump to start of line) |
| \t | tab character |

Additionally it is possible include a quotation mark within the string using \", a backslash using \\ and a percent sign using %%.

The printf (print formatted) function accepts a string parameter, called the format string, which specifies a method for rendering a number of other parameters. It allows more precise control over the output than println or print do. For example the number of significant figures displayed for a floating point type can be controlled. The values you want to be displayed should be passed as arguments to printf and the format of these should be defined in the format string using the following specifiers:

| | |
|---|---|
| %d | Outputs an integer value |
| %f | Outputs a floating point value |
| %b | Outputs a boolean |
| %c | Outputs a character |
| %s | Outputs a string |

There are many other specifiers, but these the important ones to remember. The following example illustrates the use of a printf statement to output an int and a double. The .3 limits the displayed precision of the double to 3 decimal places.

```
int    iValue = 2;
double dValue = 114.32534;
System.out.printf("iValue = %d, dValue = %.3f\n", iValue, dValue);
```

## 7.2 Getting Input from the Command Window

Getting input from the command window is made simple using the Scanner class which is part of the java.util package. The following listing shows how to use this package to read an int, a double and a String.

```
1  import java.util.Scanner;  // Make Scanner class visible.
2
3  public class InputTest
4  {
5      public static void main(String[] args)
6      {
7          Scanner sc = new Scanner(System.in);
8
9          System.out.print("Enter an integer: ");
10         int iValue = sc.nextInt();
11         System.out.printf("Entered: %d\n", iValue);
12
13         System.out.print("Enter an decimal: ");
14         double dValue = sc.nextDouble();
15         System.out.printf("Entered: %f\n", dValue);
16
17         System.out.print("Enter a string: ");
18         String str = sc.next();
19         System.out.printf("Entered: %s\n", str);
20     } // end main method
21 } // end class InputTest
```

The import statement makes the Scanner class visible to the code. This allows us to refer to the Scanner class without having to enter java.util.Scanner each time. For example if the import statement were not present, line 7 would have to be written as

```
java.util.Scanner sc = new java.util.Scanner(System.in);
```

If the entered variable does not match the expected type, for example if a string is entered when a integer is expected a java.util.InputMismatchException is thrown.

## 7.3 Input and Output with Dialog Boxes

The javax.swing package contains many classes to help create Graphical User Interfaces (GUIs). Here, the dialogs provided by the package are used to input an integer value and then display it.

```
1  import javax.swing.JOptionPane;
2
3  public class DialogInputTest
4  {
5      public static void main(String[] args)
6      {
7          String strIn, strOut;
8          strIn = JOptionPane.showInputDialog("Enter an integer");
9
10         int iValue = Integer.parseInt(strIn);
11
12         strOut = String.format("Entered: %d", iValue);
13
14         JOptionPane.showMessageDialog(null, strOut);
15     } // end main method
16 } // end class DialogInputTest
```



The showInputDialog method brings up a dialog that contains an edit box and that displays the string prompt that is passed to it, in this case "Enter an integer." If the user dismisses the dialog with Cancel or by clicking the cross, showInputDialog returns null. If the user clicks OK, the contents of the edit box are returned as a string type. Normally you would convert this string to a certain type and then use it in a calculation. In the example the string is converted to an integer using the Integer.parseInt method. If you want to input a decimal number the string may be converted to a double type using:

```
double dValue = Double.parseDouble(str);
```

Next a string, strOut, is prepared that contains the text to output. Strings may be formatted using the static method String.format, which behaves like the printf method introduced earlier. The final method call showMessageDialog displays strOut. The first argument passed to this method sets the screen position of the dialog box. In this case the null argument positions the dialog at the centre of the screen.

## 7.4 Reading from a File

The Scanner class can be used to read from a file as well as from the command window. To read from the command window a Scanner object was initialized with System.in. In order to read from a file it should be initialized with a File object. The File object is an abstract representation of file and directory pathnames. It should be created and initialized with the filename you wish to read from.

```
1  import java.util.Scanner;
2  import java.io.File;
3
4  public class FileReadTest
5  {
6      public static void main(String[] args)
7          throws java.io.FileNotFoundException
8      {
9          Scanner in = new Scanner(new File("filename.txt"));
10         String str = in.next();      // read string from file
11         System.out.println(str);     // output string to command window
12         in.close();
13     } // end main method
14 } // end class FileReadTest
```

It is possible to read in int and double data types as was demonstrated earlier, using the nextInt() and nextDouble() methods respectively. It is important to release hold of a file once you have finished reading from it using the close() method. An exception is thrown when attempting of open a file that does not exist. Valid Java programming language code must honor the Catch or Specify Requirement. This means that code that might throw certain exceptions must be enclosed by a try statement and the exceptions must be appropriately handled. Alternatively, a throws clause can be included, as is demonstrated on line 7 in the example above. Take it for granted that this line must be included. Exceptions will not be tested, and will be dealt with in more detail in the second year course.

## 7.5 Writing to a File

The java.util package includes the Formatter class to facilitate writing to a file. A formatter object should be initialized with the file name you wish to write to. In order to write to the file the format method is provided, which behaves like the printf method introduced earlier. The following example writes two lines to a file called filename.txt.

```
1  import java.util.Formatter;
2
3  public class FileWriteTest
4  {
5      public static void main(String[] args)
6          throws SecurityException, java.io.FileNotFoundException
7      {
8          Formatter out = new Formatter("filename.txt");
9          out.format("FileWriteTest output\n");
10         out.format("Display integer: %d\n",3);
11         out.close();
12     } // end main method
13 } // end class FileWriteTest
```

The contents of filename.txt are as follows:

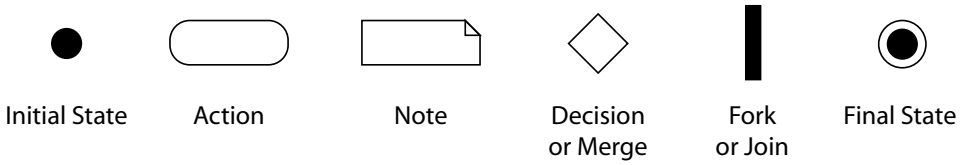
```
FileWriteTest output
Display an integer: 3
```

# 8 Control Structures

This section introduces the building blocks for structured programming: the if, else, for, do..while and switch statements.

## 8.1 Flow Diagrams

Unified Modeling Language activity diagrams are used to describe the workflow of a section of software.




Arrows represent transitions and indicate the order in which actions occur. Notes are similar to Java comments and should be connected to the element they describe with a dotted line. These diagrams are used to demonstrate some of the building blocks introduced in this section.

## 8.2 The if statement

The general form of an if statement is as follows

```
if(test){
    ............;
    ............;
}
next statement;
```

What is enclosed by the curly brackets { and } can be any number of individual executable statements and is called a compound statement. The test is carried out; if it is true then the compound statement is executed. If it is false then the compound statement is skipped and next statement is executed. As an example consider, the following bit of code where it is assumed that j and k have been assigned values somewhere above:

```
if(j<k){
    min = j;
    System.out.println("j is smaller than k");
}
```

The curly brackets can be omitted if there is only one statement in a compound statement.

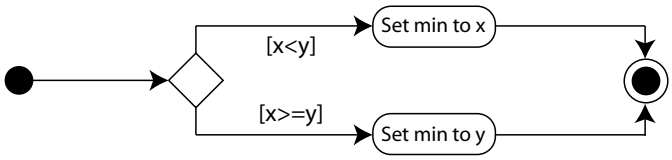
The general form of an if else statement is

```
if(test){
    ............;    // compound statement 1
    ............;
}
else{
    ............;    // compound statement 2
    ............;
}
```

If the test is true, compound statement 1 is executed; if it is false then compound statement 2 is executed. (Again curly brackets can be omitted if there is only one statement in a compound statement.) As an example suppose that x and y are two variables (and to keep it simple suppose that we know they are never equal) which have been declared and assigned values somewhere above and that min has been declared also:

Listing 4: find the minimum of x and y

```
if(x<y){
    min = x;
}
else{
    min = y;
}
```




The conditional operator ?: can sometimes be used in place of an if-else statement or very simple switch. The general form of a conditional construct is

```
expression1?expression2:expression3
```

This whole construct has a value determined as follows: Expression1 is evaluated first (and is often a test). If it is true then expression2 is evaluated and this value is the value of the whole construct. If expression1 is false then expression3 is evaluated and this value is the value of the whole construct. The previous example, listing 4, can alternatively be written as

```
min = (x<y)?x:y;
```

If x is less than y, then the whole construct to the right of the assignment operator = has the value of x, otherwise it has the value of y. The conditional operator is often avoided as it can be difficult to read.

### 8.3 The for Statement

The general form of a for statement is

```
for(expression1; test ; expression2){
    ............;
    ............;
}
next statement;
```

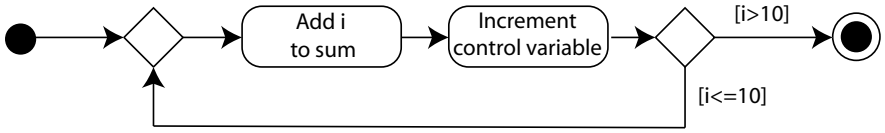
First expression1 executed (typically this initialises some variable used as a loop counter). Then the test is carried out. If it is true then the compound statement in curly brackets is executed, followed by the operation of expression2. Then the test is carried out again, and so on (compound statement, expression2, test) until the test is false when the next statement is executed.

The following example demonstrates a for loop that calculates the sum of the integers from 1 to 10:

```
int i=0, sum=0;
for(i=1; i<=10; i++){
    sum = sum + i;
}
```

When there is only one statement between the curly brackets, the brackets may be omitted. Reducing things further still it is possible to include the calculation of sum in the increment portion of the for loop:

```
int sum=0;
for(int i=1; i<=10; sum += i, i++);
```




This is not recommended though as it makes the code more difficult to read. It is common to declare the counter within the for statement. The scope of the variable i is restricted to the for loop, i.e. it will not be recognised as a declared integer variable after the for statement line.

## 8.4 The while Statement

The general form of the while statement is

```
while(test){
    ............;
    ............;
}
next statement;
```

The way the loop works is as follows: the test is carried out, and if it is true all the statements in the compound statement are executed in turn. One of these statements will normally change some variable which is controlling the test. Then the test is carried out again. If it is still true then the once again all the statements in the compound statement are executed. This carries on until the test is false, when the next statement is executed. If the compound statement consists of only one statement then the curly brackets can be omitted.

The previous example may be rewritten using a while loop as:

```
// loop to calculate the sum of the first ten integers
int i = 1, sum = 0;
while(i<=10){
    sum = sum + i;
    ++i;
}
System.out.printf("The sum is %d\n", sum);
```

The loop here is used to calculate the sum of the first ten integers and it operates as follows: i is set to one and sum to zero initially. Then the test i<=10 is carried out. The first time this is true (since i is 1) so the compound statement is executed. The current value of i is added to sum (so that sum is now 1) and i is incremented to 2. The test is carried out again and is still true, so 2 is added to sum and i becomes 3. This continues until finally i is 10. This is added to sum (which now contains the value $1+2+3+.....+10=55$) and i is incremented to 11. Now the test is false so the next statement after the while loop is executed, which output the result. We could rewrite the compound statement using a post-increment operator and omitting the curly brackets:

```
int i = 1, sum = 0;
while(i<=10)
    sum = sum + i++;
```

Note that for a while loop, the test is carried out before the compound statement is executed. There may be cases where the values of variables used in the test are not known beforehand and it may be that the test is always false. In that case the statements in the compound statement would never be executed.

## 8.5 The do ... while Statement

The general form of a do ... while statement is

```
do{
    ............;
    ............;
}while(test);
next statement;
```

This is similar to a while statement with the exception that the statements within the body of the do ... while loop are always executed at least once, i.e. when the do ... while statement is encountered the statements within the loop are executed and then the test is performed. If the test returns true the statements within the loop are again executed, if it returns false the next statement after the do...while loop is executed. The example can written as:

```
// loop to calculate the sum of the first ten integers
int i = 1, sum = 0;
do{
    sum = sum + i;
    ++i;
}while(i<=10);
```

## 8.6 The switch Statement

The general form of the switch statement is

```
switch(expression){
    case constant1:
        ............;
        ............;
        break;
    case constant2:
        ............;
        ............;
        break;
    default:
        ............;
        ............;
}
```

The expression is evaluated (it will often be just a variable with an already-assigned value). This value is compared with the constants, which must be distinct and of integer type (this can be of type int, or type char). If the expression has the same value as one of the constants, execution continues with the first statement after this constant. ALL the remaining statements in the curly brackets are then executed in turn, unless the statement break; is encountered, in which case the switch statement terminates. If the expression does not match one of the constants, then the statements after the default label are executed in turn. The default can be absent, in which case no statements are executed if the expression does not match one of the constants. As an example, consider the following program which counts the number of occurrences of the letters 'a' and 'b', and the number of other characters in a String:

```
1  // Count the occurrences of chars a and b in a string
2  public class SwitchExample1
3  {
4      public static void main(String[] args)
5      {
6          String str = "This is a brief line of text";
7          int aCount = 0, bCount = 0, otherCount = 0;
8
9          for(int k=0; k<str.length(); k++)
10         {
```

```
11              // get char at kth position in str
12              char ch = str.charAt(k);
13
14              switch(ch){
15                  case 'a':
16                      ++aCount;
17                      break; // exit switch
18                  case 'b':
19                      ++bCount;
20                      break; // exit switch
21                  default:
22                      ++otherCount;
23                      break; // optional; will exit switch anyway
24              } // end switch
25          } // end for
26          System.out.printf("aCount=%d, bCount=%d, ",aCount, bCount);
27          System.out.printf("otherCount=%d\n", otherCount);
28      } // end main
29  } // end SwitchExample1
```

The variables aCount, bCount and otherCount are used to store the numbers of characters. You will see that the body of the for loop consists of a single switch statement. A character is taken from the keyboard by getchar() and assigned to variable ch. The variable ch is compared to the two cases `'a'` and `'b'`. If it is `'a'`, aCount is incremented by one and the break statement takes us out of the switch statement (and back to the for test where another character is taken in); if it is `'b'`, bCount is incremented by one; if it is neither of these cases then the variable otherCount is incremented by one. If we were not interested in the characters other than `'a'` and `'b'`, then the line

```
        default: ++otherCount;
```

could be omitted (as could the declaration of the variable otherCount). In the above example, a break statement was used in each of the non-default cases. We now give a slightly modified version of the method main of this program in which aCount and otherCount are used in the same way but instead of counting the number of occurrences of `'b'`, we count the total number of occurrences of both `'a'` and `'b'` using the variable abCount.

```
1   // Count the occurrences of chars a and (a+b) in a string
2   public class SwitchExample2
3   {
4       public static void main(String[] args)
5       {
6           String str = "This is a brief line of text";
7           int aCount = 0, abCount = 0, otherCount = 0;
8
9           for(int k=0; k<str.length(); k++)
10          {
11              // get char at kth position in str
12              char ch = str.charAt(k);
13
14              switch(ch){
15                  case 'a':
16                      ++aCount;
17                  case 'b':
18                      ++abCount;
19                      break; // exit switch
20                  default:
21                      ++otherCount;
22                      break; // optional; will exit switch anyway
```

```
23                 } // end switch
24             } // end for
25             System.out.printf("aCount=%d, abCount=%d, ",aCount, abCount);
26             System.out.printf("otherCount=%d\n", otherCount);
27         } // end main
28 } // end SwitchExample2
```

The break statement in case 'a' has been removed. Consequently if ch matches this case, aCount will be incremented by one and then, since there is no break statement between this and the next statement

```
        ++abCount;
```

of the switch, abCount will also be incremented by one before a break is encountered. This process of executing all statements not separated by a break statement is sometimes called "fall-through" or "drop-through". After a case label, it is not necessary to have any statements. This occurs when we require the same actions to be taken in two different cases. As a last example, we simplify the preceding program so that only the total number of occurrences of both 'a' and 'b' are counted. The body of the switch becomes:

```
    case 'a': case 'b':
        ++abCount;
        break;
```

The increment of abCount now occurs if ch matches either 'a' or 'b'.

The expression in brackets after the word switch in a switch statement must be int or char. Hence the following would produce an error message:

```
        float f;
        switch(f){
            ..........
```

## 8.7 The break and continue Statements

It is possible jump out of a for, do or do{..}while loop using the break statement. Execution then continues with the next statement after the control statement.

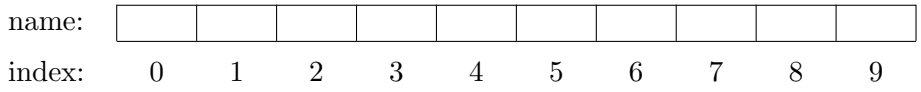
The continue command when executed in a loop will jump to the next iteration of the loop, skipping the remaining commands in the loop for the current iteration.

# 9 Arrays

Arrays are containers used to store a number of elements. Each of these elements must be of the same type. An array may be declared and created using a statement such as:

```
int[] arr = new int[10]; // an array of 10 integers
```

The structure of the array is illustrated below. The location of an element in the array may be specified by a label known as the index. The index is zero-based in Java.




Square brackets are used to denote indexing; for instance to obtain the forth entry in the above array one would use arr[3]. This entry can be assigned in the same way as a normal int, for example arr[3]=2. The index is usually an int value. An index of byte, short or char type is promoted to int; however, a long index causes a compile error. Since arrays are objects they can be created using the new operator, which automatically initialises the contents with the default value of their type.

Arrays can be created and initialised by an array initialiser in curly brackets, for example

```
float math_con[] = {3.142F, 1.414F, 2.718F, 0.866F, 0.577F};
```

This initialises math_con[0] to 3.142, math_con[1] to 1.414, and so on. The array is given the size of the number of initialisers.

Once an array has been created it is not possible to change its size. An implementation of a resizable array is provided by the ArrayList class.

If an array is created with an object type rather than a primitive type, then what is stored at each index is a reference to an object. An instance can then be created for each entry in the array using the new expression. The reference returned by new can then be stored in the corresponding index in the array. This process is shown below for a class called MyClass.

```
final int SIZE = 4;
MyClass[] aObjects = new MyClass[SIZE];
for(int k=0; k<aObjects.length; k++)
    aObjects[k] = new MyClass();
```

## 9.1 ArrayIndexOutOfBoundsException

An array access (read or write) with an index less than 0 or greater than the length of the array generates an ArrayIndexOutOfBoundsException exception.

```
1 public class Example2{
2     public static void main(String[] args){
3         double[] xx = new double[5];
4         for(int i=0; i<7; i++){
5             // attempt to overfill array by 2 elements
6             xx[i] = i;
7         }
8     }
9 }
```

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 5
        at Example2.main(Example2.java:6)
```

## 9.2 Passing Arrays to Methods

An array of primitive data types is not itself a primitive data type but an object. Its name is the reference to that instance and to pass an array to a method one simply passes the name of the array. As the name is a reference this is another example of passing by reference. An alteration to one or more of the elements of the array will be reflected by an identical change in the array calling method, as one is altering the same entity, i.e. through a reference or a pointer to that instance. The following program illustrates this. The array is an array of int. A reference to the array object is passed to a method and all the elements are incremented. You can see that the elements of iArray in the calling method main are incremented by this procedure. Next a single array entry is passed to a method and increment. However, the single entry is passed by value and so the change in the method is not reflected back to main.

```
1  // PassArray.java
2  public class PassArray
3  {
4      public static void main(String[] args)
5      {
6          int[] iArray = {1, 2, 3, 4};
7          printArray(iArray);
8          incrementArray(iArray);
9          printArray(iArray);
10         incrementValue(iArray[0]);
11         printArray(iArray);
12     } // end main method
13
14     private static void incrementArray(int[] array)
15     {
16         for(int k=0; k<array.length; k++)
17             array[k]++;
18     }
19     private static void incrementValue(int iValue)
20     {
21         iValue++;
22     }
23     private static void printArray(int[] array)
24     {
25         for(int k=0; k<array.length; k++)
26             System.out.printf(" %d", array[k]);
27         System.out.println();
28     }
29 } // end class PassArray
```

```
1 2 3 4
2 3 4 5
2 3 4 5
```

You may insulate your array in the main method from changes in the method to which the array was passed by using a copy method, but in the case of arrays, you do not need to write your own copy method, you may use the Java clone method. To do this line 8 should be replaced by

```
incrementArray((int[])iArray.clone());
```

# 10 Multidimensional Arrays

Many programming languages store multidimensional arrays in a single block of memory large enough to hold all elements of the array. Java does not do this. Instead multi-dimensional arrays are built from many one-dimensional arrays, in other words it builds up "arrays of arrays".

As a consequence rows may be different sizes. Also, each row is an object, an array, that can be used independently. The following example demonstrates how to declare and create a two-dimensional array:

```
final int NROWS = 10;
final int NCOLS = 5;
double[][] volts = new double[NROWS][NCOLS];
```

A multidimensional array may be initialized too:

```
double[][] eye3 = { { 1.0, 0.0, 0.0 }, // 3 x 3 identity array
                    { 0.0, 1.0, 0.0 },
                    { 0.0, 0.0, 1.0 } };
```

Initialization can also be performed for non-rectangular matrices, for example

```
int[][] varRowLen = { { 1, 2 },
                      { 2, 3, 4, 5, 6 },
                      { 7 } };
```

A similarly sized array can be created using the new expression:

```
int[][] varRowLen = new int[3][];
varRowLen[0] = new int[2];
varRowLen[1] = new int[5];
varRowLen[2] = new int[1];
```

Note were the first line written `int [][] varRowLen = new int[][3];` this would be a syntax error.

Every array object contains an instance variable called length, which allows the size a multidimensional array can be easily found. The following example demonstrates this by displaying a 2D integer array.

```
private static void displayArray2D(int[][] array) {
    for( int i=0; i<array.length; i++ )
    {
        for( int j=0 ; j<array[i].length; j++ )
            System.out.printf(" %d", array[i][j]);
        System.out.println();
    }
}
```

# 11 Classes

In this section classes are dealt with in more detail. To illustrate concepts such as overloading and overriding a class is introduced, which is a representation of a complex number. The class is a blueprint that describes the features common to all complex numbers, i.e. they all have real and imaginary parts. The Class Diagram for the Complex class is shown below.

| **Complex** |
|---|
| − real : double<br>− imag : double |
| <<constructor>> Complex(real : double, imag : double)<br>+ getReal( ) : double<br>+ getImag( ) : double |

The class contains the instance variables called real and imag which hold the real and imaginary part of the complex number respectively. It is possible to instantiate many instances of the Complex number class. Each of these objects can represent complex numbers with different real and imaginary parts.

Since real and imag are private, they are not accessible to the outside world. The get methods, getReal and getImag, are provided so that the user of the class may obtain the real and imaginary values of a given object.

In this case set methods are not provided. Instead values are assigned to real and imag when an object is created, via the constructor. The constructor takes as arguments two variables, which are the desired real and imaginary parts of the object. It is called when the class is instantiated using the new keyword.

## 11.1 Access Modifiers

The names of all variables, be they static or instance, and methods can be preceded by an access modifier. This modifier may be one of the following three keywords; private, protected or public. The protected modifier will be described in more detail in Section 12.1.

| | |
|---|---|
| **private** | Variables and methods prefixed by the private keyword may only be accessed by methods within the same class. They cannot be accessed externally. |
| **public** | Variables and methods prefixed by the public keyword can be accessed by external code in addition to methods of the same class. |

It is desirable to make all instance and static variables private as this prevents them being altered from outside the class. This prevents instance variables from being given illegal values by external code. With private instance variables, if a class is behaving unexpectedly then a method of the class must contain an error, and pin-pointing the error is then usually straightforward.

With the variables declared as private, how does the user interact with the class? When writing a class you should provide a well defined interface, a set of public methods, which are able to modify the instance variables. The user is only aware of the interface and is insulated from the specific implementation, i.e. the instance variables you have chosen. If a method is only used internally within a class, the user does not need to see the method and it can be made private.

## 11.2 Implementation

An implementation of the Complex class follows.

```java
public class Complex
{
    private double real;
    private double imag;

    public Complex(double real, double imag)
    {
        this.real = real;
        this.imag = imag;
    }

    public double getReal() {
        return real;
    }

    public double getImag() {
        return imag;
    }
}
```

The argument list of the constructor contains two variables named real and imag. This causes some ambiguity; the instance variables named real and imag and the variables on the argument list have the the same name. In order to resolve this ambiguity a this. prefix may be attached to a variable or method name to specify that it belongs to the class. Generally it is best to avoid such conflicts by choosing the variable names on the argument list more carefully:

```java
    public Complex(double realIn, double imagIn)
    {
        real = realIn;
        imag = imagIn;
    }
```

In order to use this class it is necessary to create some Complex objects. The following code is a simple Java application that creates two such objects and outputs their values.

```java
// ComplexTest.java
public class ComplexTest
{
    public static void main(String[] args)
    {
        Complex a = new Complex(1.0, 2.0);
        Complex b = new Complex(2.0, 0.0);

        System.out.printf("%f+%fi\n", a.getReal(), a.getImag());
        System.out.printf("%f+%fi\n", b.getReal(), b.getImag());
    }
}
```

```
1.000000+2.000000i
2.000000+0.000000i
```

## 11.3 Overloading

Two methods can be defined with the same name, provided their argument lists differ. This allows for polymorphism; values of different data types can be handled using a uniform interface. For example two constructors may be provided for the Complex class in order to provide an alternative means to create an object. This second constructor provides a simpler means to create a Complex object when a number is purely real. To the Class Diagram the following is added:

| <<constructor>> Complex(real : double) |
|---|

The implementation becomes:

```
1  public class Complex
2  {
3      private double real;
4      private double imag;
5
6      public Complex(double real, double imag) {
7          this.real = real;
8          this.imag = imag;
9      }
10
11     public Complex(double real) {
12         this.real = real;
13         this.imag = 0.0;
14     }
15     ...
16 }
```

A Complex object with no imaginary part may then be created using the following code.

```
Complex b = new Complex(2.0);
```

## 11.4 Overriding

Every class is implicitly a subclass of Object (in the absence of any other explicit superclass). Object contains the method toString. A call to System.out.print(obj) will automatically invoke the toString method of the object named obj. You can override toString in your class to provide your own String representation. In the Class Diagram this method is defined as:

| + toString( ) : String |
|---|

The most straightforward implementation of this method follows.

```
1  public class Complex {
2      ...
3      // return a String representation of the invoking object
4      public String toString()
5      {
6          if (imag < 0)
7              return real + " - " + (-imag) + "i";
8          return real + " + " + imag + "i";
9      }
10 }
```

This method provides a consistent String representation for the class. From the point of the view of the user, it is much simpler to output the value of the object to the Command Window.

```
1  // ComplexTest.java
2  public class ComplexTest {
3      public static void main(String[] args)
4      {
5          Complex a = new Complex(1.0, 2.0);
6          Complex b = new Complex(2.0, 0.0);
7
8          System.out.println(a);
9          System.out.println(b);
10     }
11 }
```

```
1.0 + 2.0i
2.0 + 0.0i
```

## 11.5 Operations

The current implementation of the Complex class is able to store and display the value of a complex number, but it provides no means for carrying out calculations. In order to do this, we must add methods to the class for this purpose. For example we could include a method to add together two Complex objects:

| + plus( other : Complex ) : Complex |
|---|

A naive first implementation might look something like this.

```
1  public class Complex {
2      ...
3      // do not implement this kind of thing!
4      public Complex badplus(Complex other)
5      {
6          real += other.real;
7          imag += other.imag;
8          return this;
9      }
10 }
```

The problem with this implementation is apparent in the following Java application which tests this method.

```
1  // ComplexTest.java
2  public class ComplexTest {
3      public static void main(String[] args)
4      {
5          Complex a = new Complex(1.0, 2.0);
6          Complex b = new Complex(2.0, 0.0);
7
8          System.out.println(a.badplus(b));
9          System.out.println(a);
10     }
11 }
```

Note that java does not provide operator overloading like C++ does, and so it is not possible to write code such as a+b. Instead, the plus method must be called in the standard way. This application outputs the following:

```
3.0 + 2.0i
3.0 + 2.0i
```

The badplus method has actually modified the instance variables of the object named a. From the point of view of the user, this behaviour is unexpected. To overcome this problem the plus method should be modified so that it does not change the value of the instance variables of the object to which it belongs. In other words the plus operator should create a new Complex object.

```
1  public class Complex {
2      ...
3      public Complex plus(Complex other)
4      {
5          return new Complex(real+other.real, imag+other.imag);
```

```
6       }
7
8       public static Complex plus(Complex a, Complex b)
9       {
10          return new Complex(a.real+b.real, a.imag+b.imag);
11      }
12  }
```

The code above shows two possible implementations of the plus method, both of which may be included in the class. Both perform the same task, but are called slightly differently, as is demonstrated in the following application.

```
1   // ComplexTest.java
2   public class ComplexTest {
3       public static void main(String[] args)
4       {
5           Complex a = new Complex(1.0, 2.0);
6           Complex b = new Complex(2.0, 0.0);
7
8           System.out.println(a.plus(b));
9           System.out.println(Complex.plus(a,b));
10          System.out.println(a);
11      }
12  }
```

```
3.0 + 2.0i
3.0 + 2.0i
1.0 + 2.0i
```

Both methods produce identical results and have no effect on the instance variables of the object a, as desired.

## 11.6 Immutable Objects

For objects such as Complex objects there is no real need to allow changes to the instance variables after the object is created. In order to more strongly enforce this requirement, we can make the instance variables final. This means that they can only be assigned within the constructor. If any other method within the class then tries to modify these variables a compiler error results. Such objects are known as immutable. A common example of such a type is the String class.

```
1   public class Complex
2   {
3       private final double real; // Immutable; no way to change instance
4       private final double imag; // variables after construction
5
6       public Complex(double real, double imag) {
7           this.real = real;
8           this.imag = imag;
9       }
10      ...
11  }
```

## 11.7 Testing Equality

How can we compare the values of two complex numbers? The following application creates two Complex objects with the same real and imaginary parts and then attempts a naive comparison.

```java
// ComplexTest.java
public class ComplexTest {
    public static void main(String[] args)
    {
        Complex c = new Complex(1.0, 2.0);
        Complex d = new Complex(1.0, 2.0);

        System.out.println(c == d);
    }
}
```

The variables c and d are references, which locate the Complex objects in memory. This code creates two objects in distinct memory locations and as a result the two references, c and d, differ. The output of the code is as follows:

```
false
```

The compiler does not know how to test if two Complex objects represent the same complex number. We have to provide a means (a method) for it to do so. The usual way to do this is to add a method to the class called equals.

| + equals( other : Complex ) : boolean |
|---|

This method may be implemented as follows:

```java
public class Complex {
    ...
    public boolean equals(Complex other)
    {
        if(real==other.real && imag==other.imag)
            return true;
        return false;
    }
}
```

When we then test this method we can see that the comparison works as a user would expect.

```java
// ComplexTest.java
public class ComplexTest {
    public static void main(String[] args)
    {
        Complex c = new Complex(1.0, 2.0);
        Complex d = new Complex(1.0, 2.0);

        System.out.println(c.equals(d));
    }
}
```

```
true
```

## 11.8 Other Methods

This complex number is lacking several important methods. Create your own complex number class based on this template. Add methods called minus, multiply, reciprical and divide. The divide method can make use of the reciprical method. The reciprical method can be based on the following expression:

$$z^{-1} = z^* \cdot |z|^{-2}.$$

## 11.9 Copying and Cloning

Because the Complex number class is immutable we do not have to worry about unexpected changes to objects after they have been created. However, for mutable classes such as the Person2 class introduced in Section 3.1 we have to be more careful. In this section the concept of copying and cloning objects will be illustrated using the Person2 class.

### 11.9.1 Copying

The following Java application attempts to make a copy of a Person2 object.

```
1  // PersonTest2.java
2  public class PersonTest2
3  {
4      public static void main(String[] args)
5      {
6          Person2 per1 = new Person2("John");
7          Person2 per2 = per1;
8          per2.setName("David");
9
10         System.out.printf("First  person is %s\n", per1.getName());
11         System.out.printf("Second person is %s\n", per2.getName());
12     } // end main method
13 } // end class PersonTest2
```

However, Line 7 merely assigns the reference stored in per1 to the variable per2. This means that both per1 and per2 are pointing to the same object in memory. Therefore after calling setName, both per1.getName() and per2.getName() yield the same result.

```
First  person is David
Second person is David
```

In order to overcome this problem it is possible to provide a method which creates a copy of an object. This copy is an independent entity. A possible implementation of such a method follows.

```
1  // Person2.java
2  public class Person2
3  {
4      ...
5      public Person2 copy()
6      {
7          return new Person2(name);
8      }
9  }
```

A value can be then assigned to per2 by making a call to this method, so that line 7 of the application becomes:

```
7          Person2 per2 = per1.copy();
```

The subsequent call to setName only affects the newly created object and not the original, so that the program correctly outputs:

```
First  person is John
Second person is David
```

### 11.9.2 Cloning

The Java Object class has a method called clone which may be used to make copies. This may be overridden to provide a general means to replicate instances of your class.

```java
// Person2.java
public class Person2 {
    ...
    public Object clone()
    {
        return (Object) new Person2(name);
    }
}
```

Within the application the clone method may be called in the following manner:

```java
        Person2 per2 = (Person2)per1.clone();
```

## 11.10 Passing by Reference

As was demonstrated in Section 9.2, the primitive data types are passed to a method by value. If the method makes a change to the primitive data type it is actually modifying a copy of the original variable and changes are not reflected back to the calling method. We saw that arrays are in fact objects, and when passing an array to a method, it is in fact a reference that is passed. This makes sense from a performance point of view, as copying large arrays is time consuming. Similarly objects are passed to methods by reference.

The following Java application demonstrates a method that is passed a Person2 object. The method changes the object using a call to setName and this change is seen in the calling method.

```java
// PersonTest2.java
public class PersonTest2
{
    public static void main(String[] args)
    {
        Person2 per = new Person2("John");

        changeName(per);

        System.out.printf("Person is %s\n", per.getName());
    } // end main method

    private static void changeName(Person2 per)
    {
        per.setName("Fred");
    } // end method changeName
} // end class PersonTest2
```

```
Person is Fred
```

## 11.11 Static Methods and Variables

In the Person2 class the variable called name is an instance (non-static) variable. Every object has its own name variable. Any method that makes use of an instance variable should identically be non-static.

Static variables are shared by all objects of the same class. In other words they belong to the class. A get or set method that acts on a static variable should itself be static.

In summary, static variables and methods belong to the class and the non-static variables and methods belong to the object.

In the example that follows a static variable is added to the Person2 class that counts the number of objects that have been created. This might be useful if you wanted to write a program that modelled the behaviour of people. They would act differently according to the number of people present.

```java
// Person2.java
public class Person2
{
    private String name;
    // static field, number of objects created
    private static int count = 0;

    public Person2(String str) {
        setName(str);
        count++;
    }
    public void setName(String str) {
        name = str;
    }
    public String getName() {
        return name;
    }
    public static int getCount() {
        return count;
    }
} // end class Person2
```

```java
// PersonTest2.java
public class PersonTest2
{
    public static void main(String[] args)
    {
        System.out.printf("Count = %d\n", Person2.getCount());

        Person2 per1 = new Person2("Fred");
        Person2 per2 = new Person2("Sarah");

        System.out.printf("Count = %d\n", Person2.getCount());
    } // end main method
} // end class PersonTest2
```

```
0
2
```

# 12 Inheritance

Inheritance is a means by which the behaviour of a class may be absorbed into a new class and expanded upon. In Java the existing class is known as the superclass. The new class that extends the capabilities of the superclass is known as the subclass.

Different classes will have certain amounts in common with each other. Inheritance allows the relationships between classes to be described. For instance a superclass Vehicle could be defined, with properties such as speed, and both the Bicycle and Car could be subclasses that derive from the Vehicle class.

## 12.1 Access Modifiers

The three accessor modifiers, private, protected and public are described below.

| | |
|---|---|
| **private** | Variables and methods prefixed by the private keyword may only be accessed by methods within the same class. They cannot be accessed by subclasses. |
| **protected** | Variables and methods prefixed by the protected keyword may only be accessed by methods within the same class or by subclasses. |
| **public** | Variables and methods prefixed by the public keyword can be accessed by external code in addition to subclasses and methods of the same class. |

## 12.2 Example

In the following example a superclass called Component is defined, which represents a general electrical component that posses a complex impedance. It also includes methods for adding Components in parallel and in series. Three subclasses, the Resistor, Inductor and Capacitor derive from the Component class. The hierarchy UML class diagram is shown below.

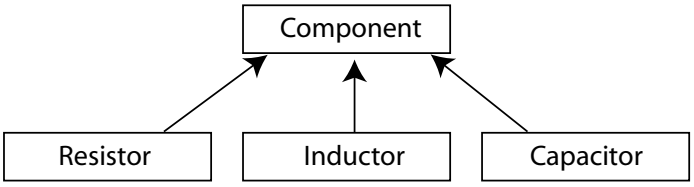



Figure 1: Component hierarchy UML class diagram

| **Component** |
|---|
| − impedance : Complex |
| <<constructor>> Component(impedance : Complex)<br>+ getImpedance( ) : Complex<br>+ inSeriesWith( com : Component ) : Component<br>+ inParallelWith( com : Component ) : Component |

| **Resistor** |
|---|
| − resistance : double |
| <<constructor>> Resistor(R : double) |

| **Inductor** |
|---|
| − inductance : double |
| <<constructor>> Inductor(L : double, freq : double) |

| Capacitor |
| --- |
| – capacitance : double |
| <<constructor>> Capacitor(C : double, freq : double) |

```
// Component.java
public class Component
{
    private Complex impedance;  // Component impedance
    private String name;        // Component name (debug purposes)

    public Component(String name, Complex impedance)
    {
        this.name = name;
        this.impedance = impedance;
    }

    public Complex getImpedance()
    {
        return impedance;
    }

    public String getName()
    {
        return name;
    }

    // Combine two components in series
    public Component inSeriesWith(Component com)
    {
        String newName = "("+name+"+"+com.name+")";
        Complex temp = impedance.plus(com.impedance);
        return new Component(newName, temp);
    }

    // Combine two components in parallel
    public Component inParallelWith(Component com)
    {
        String newName = "("+name+"||"+com.name+")";
        Complex temp =  impedance.multiply(com.impedance).divide(
                        impedance.plus(com.impedance));
        return new Component(newName, temp);
    }
}
```

In order to declare a subclass in Java the extend keyword should be used in the class declaration. It should come after the subclass name and should be followed by the superclass name.

```
// Resistor.java
public class Resistor extends Component
{
    private double resistance;

    public Resistor(double R)
    {
        super("Resistor", new Complex(R,0.0));
        resistance = R;
    }
}
```

Since the impedance instance variable of the superclass is private it may not be directly accessed by the subclass. It is possible to include a setImpedance method in the Component class to allow modification of this variable. In this example the superclass constructor is called from the Resistor constructor in order to initialize the impedance, using the keyword super. Note that if the super keyword is used, it must be the first statement in the subclass constructor.

```
1  // Inductor.java
2  public class Inductor extends Component
3  {
4      private double inductance;
5
6      public Inductor(double L, double freq)
7      {
8          super("Inductor", new Complex(0.0,2.0*Math.PI*freq*L));
9          inductance = L;
10     }
11 }
```

Since the call to the superclass constructor must appear as the first statement in the subclass constructor, the calculation of the impedance must be embedded within the super call. For more complex calculations of the impedance it may make more sense to change the impedance member variable of the superclass to protected. It may then be directly modified by the subclass after the calculation of the impedance has been carried out.

```
1  // Capacitor.java
2  public class Capacitor extends Component
3  {
4      private double capacitance;
5
6      public Capacitor(double C, double freq)
7      {
8          super("Capacitor", new Complex(0.0,1.0/(2.0*Math.PI*freq*C)));
9          capacitance = C;
10     }
11 }
```

A Java application follows, which makes use of the classes introduced above to simulate the circuit shown in Fig. 2.

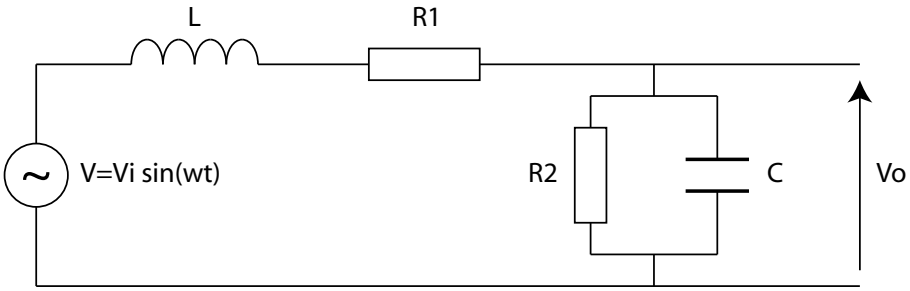



Figure 2: Circuit diagram for ComponentTest

Firstly, the program inputs the component values as well as the input voltage $V_i$ and frequency, making use of the showInputDialog method of the JOptionPane class. Component objects are then created for each resistor, capacitor and inductor that appear in the circuit diagram. Finally, the program calculates and outputs the magnitude and phase of $V_o$.

```
1  import javax.swing.JOptionPane;
2
```

```
3  public class ComponentTest
4  {
5      public static void main(String[] args)
6      {
7          // Read in circuit values
8          double resist1 = inputDouble("Input Resistance, R1 (Ohms)");
9          double resist2 = inputDouble("Input Resistance, R2 (Ohms)");
10         double capac = inputDouble("Input Capacitance, C (Farads)");
11         double induct = inputDouble("Input Inductance, L (Henrys)");
12         double volts = inputDouble("Input voltage amplitude (Volts)");
13         double freq = inputDouble("Input voltage frequency (Hz)");
14
15         // Create circuit objects
16         Resistor  res1 = new Resistor(resist1);
17         Resistor  res2 = new Resistor(resist2);
18         Capacitor cap  = new Capacitor(capac, freq);
19         Inductor  ind  = new Inductor(induct, freq);
20
21         // create L+R1 combination
22         Component indRes1 = ind.inSeriesWith(res1);
23
24         // create C||R2 combination
25         Component capRes2 = cap.inParallelWith(res2);
26
27         // get C||R2 impedance
28         Complex impCR2 = capRes2.getImpedance();
29
30         // create L+R1+C||R2 combination
31         Component total = indRes1.inSeriesWith(capRes2);
32
33         // get total impedance
34         Complex impTotal = total.getImpedance();
35
36         // Complex Output Voltage, Vo
37         Complex voltOut = (impCR2.multiply(volts)).divide(impTotal);
38
39         // Output amplitude and phases
40         double mag = voltOut.abs();
41         double phase = voltOut.phase();
42
43         // Display results
44         String str =
45         String.format("Magnitude = %f V\nPhase = %f", mag, phase);
46         JOptionPane.showMessageDialog(null, str);
47     }
48
49     private static double inputDouble(String prompt)
50     {
51         String str = JOptionPane.showInputDialog(prompt);
52         return Double.parseDouble(str);
53     }
54 }
```

# 13 Recursion

Recursion is when a method calls itself within the body of its definition. The method is said to be recursive. All methods can be used recursively. A standard example involves a method which takes a positive integer value n and returns the sum of the first n integers:

```
public int sum(int n)
{
    if (n==1)
        return 1;
    else
        return n + sum(n-1);
}
```

Obviously if $n$ is 1, then the value 1 must be returned. Otherwise, the value of $n$ is added to sum(n-1) which will be the sum of the previous $n-1$ integers. It is here that the method calls itself. Let's look in some detail how this works for, say, $n = 3$. Assume that the method is first called with a statement such as

```
s = sum(3);
```

The formal parameter n is put equal to 3. This is not unity, so the method should return $3 + \text{sum}(2)$. To do this it needs to calculate sum(2). Hence it calls itself with parameter now equal to 2 (this should be thought of as occurring "inside" the "first call" of the method). To calculate sum(2), it needs sum(1) in order to return $2 + \text{sum}(1)$, so this "inner call" has another "innermost call" to itself. Now sum(1) returns the value 1 to the inner call, which can then return $2 + 1 = 3$ to the first call of itself. This returns $3 + 3 = 6$ to the method that first called the method sum. You can think of these recursive calls as occurring "Russian doll-like", one inside the other. When we finally reach a definite numerical value without involving another method call, the whole set of nested calls works back to the original first call of the method, substituting the values returned by each recursive call. The call sum($n$) will create n nested copies of the method, with different actual parameters each time, before regressing level-by-level to the first call. A common error in writing recursive methods is to omit to include the definite numerical value which does not involve a method call. In the above method this was the line

```
if (n==1) return (1);
```

This type of return in a recursive method is often called the "base case" which, ultimately, all method calls arrive at, before regressing to the first call. Omitting the base case can lead to an endless nesting of method calls.

# 14 Applets

This section describes how to write a simple a Java applet. This applet outputs "Hello world!" to the screen.

```
1  // The HelloWorld class implements an applet that
2  // simply displays "Hello World!"
3  import java.applet.*;
4  import java.awt.*;
5
6  public class HelloWorld extends Applet
7  {
8      public void paint(Graphics g)
9      {
10         g.drawString("Hello world!", 50, 25);
11     }
12 }
```

The import statements

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

allow the program, HelloWorld, to access classes in the Java library, which contain the methods needed to display an applet, as well as the Abstract Windowing Toolkit (awt) classes, which contain methods needed when opening a window.

```
public class HelloWorld extends Applet
```

The keyword extends allows your class, HelloWorld, to use all the methods in a Java Library class called Applet. HelloWorld is now referred to as a subclass of Applet which is known as a superclass.

The obligatory method for an applet is paint. An applet may contain additional methods; however, this simple example does not. Applets do not have a method called main.

```
public void paint(Graphics g)
```

The method paint allows graphics or text to be written, i.e. painted, to the applet window. The argument, Graphics g, is essential. It transfers a Graphics object from the library, which can be used to draw to the screen.

```
g.drawString("Hello World!", 50, 25);
```

The statement above will print the characters in quotes, i.e. Hello world!, in the Applet Window starting 50 pixels to the right and 25 pixels down relative to the top left hand corner of the Applet Window.

## 14.1 Compiling and Running a Java Applet

A Java applet can be compiled using the following statement:

```
javac HelloWorld.java
```

If the compilation is successful a byte code file, e.g. HelloWorld.class will be created. Do not attempt to execute an applet with the java command. First of all it is necessary to create an HTML file:

```
1 <html>
2     <head> <title> A Java Applet called HelloWorld </title> </head>
3     <body>
4         <applet code="HelloWorld.class" width=150 height=50> </applet>
5     </body>
6 </html>
```

This should be saved as an appropriately named text file with an .html extension, for example HelloWorld.html. The program can be run using web browser or the appletviewer, provided as part of the JDK. To execute the HelloWorld applet using the appletviewer the following command can be used:

```
appletviewer HelloWorld.html
```

Always check your program first with appletviewer before using a web browser in order to correct all compilation and runtime errors.