Programming

in

ava

CONTENTS

OVERVIEW OF JAVA

Learning Module Objectives

When you have completed this learning module you will have an overview of:

- The Java language
- Java portability
- Java Libraries

THE JAVA LANGUAGE

Development of Java

The Java language was originally developed by Sun Systems who required a platform-independent programming language which would produce run time code small enough to run on microprocessor empowered devices such as cable TV switchboxes.

With the growing popularity of the World Wide Web, the Java development team produced a browser which enabled Java code to be incorporated in web pages. With this development, the popularity of Java increased astronomically, and most modern internet browsers are Java enabled.

The increased capabilities of newer releases of Java have made it a popular language for large-scale programming applications. The addition of ODBC support has established Java as a powerful tool for enterprise-wide development.

Object-oriented design

From the beginning, Java was developed as an Object Oriented language, making it simple to design a system which makes full use of re-usable components.

C-like language structure

The syntax of the actual Java language was primarily based on C++, and in fact the program instructions are so similar that C programmers will have very little difficulty in learning Java.

However, many of the features which tend to make C++ programs bug-prone have been removed or replaced in Java. It is impossible in Java to write code which infringes on the memory of other applications, or other sections of the same application. Deallocation of unused memory, which is another major source of bugs in C++, is no longer the responsibility of the programmer.

These problems, which are the major cause of the General Protection Faults so often seen in Windows applications, cannot occur in systems developed in Java.

Client/Server applications

Java is especially suited to developing client/server applications. Full support for multithreaded applications simplifies the task of writing server programs, and client programs are easily developed using the GUI (Graphical User Interface) libraries.

Java probably provides the best tools available in any programming language for communication over a network.

Java and the Internet

Java is perhaps best known for its popularity in the creation of programs which may be accessed via the internet. Java Applets are programs which are designed to be run through a web browser. Originally popular for animated graphics on web pages, they have quickly become a powerful intranet tool . Program code can be stored centrally in one place, and downloaded to the client's machine at run time. This eliminates the problem of ensuring that all users have the same version of the software, while taking the load off the server machine, since applets are run on the client's computer.

The name Applet suggests that these are small application programs, but in fact they can be any size, and many web pages make far more productive use of this technique than simply providing animated graphics.

Security

The Java development team has all along been fully conscious of the security implications of running applets downloaded from the internet, and of incorporating reusable components in program development.

The security measures built into the Java design work well, and are being continually reviewed.

JAVA PORTABILITY

Java Byte Code

Since every different model of processor has its own set of machine code instructions, it is impossible to produce machine code that will be run by more than one type of processor. This is a severe restriction on the portability of software, since a program written for one type of computer will not run on another.

To get around this problem, Java code is compiled to a form known as Byte Code. While not quite as low level as machine code, it is a form which is compact in size and easily converted to machine code.

The Java Virtual Machine

The Java run-time environment is known as the Java Virtual Machine (JVM). It includes a byte code runner which will convert the byte code to machine code and run it.

Provided that a JVM is available for a particular processor, Java programs written on any other machine can be run on that processor without modification or recompiling.

JVM's are available for most of the commonly used computers, including Windows machines, Unix boxes, IBM machines and the Apple Mac. Browsers such as Netscape and Explorer include a version of the JVM which may be used to run applets. Many modern devices which incorporate microprocessors have a built-in JVM, so that Java may soon become the major development tool for embedded systems.

This allows software packages written in Java to have a user base over many different platforms, without any need for redevelopment.

Impact on performance

Since byte code must be converted to machine code at run time, there will obviously be a negative impact on performance.

Many techniques are being developed to overcome this. One of these is what is known as a Just In Time compiler, which compiles sections of code which are reused many times within a program to machine code, which is cached. This results in a huge performance improvement, and has made it possible for Java to become competitive even in applications where performance is a major issue.

Unicode

Until recently, the coding system commonly used for printable characters was ASCII, which features a 7 bit code incorporating numbers, symbols and letters of the Western European alphabet.

However, there has been an increasing demand for support for other writing symbols, such as Arabic and Japanese, to be incorporated in computer systems. This has resulted in the specification of Unicode, a 16-bit code featuring international alphabets and symbols, in addition to the letters, numbers and symbols defined in the ASCII code.

Java uses Unicode for all of its internal representation of data in character format. This simplifies the development of software for an international client base.

JAVA LIBRARIES

Classes

Since Java is an Object Oriented language, programs are built from a collection of objects. In order to simplify the creation of objects, classes are provided which act as a 'blueprint' for objects of similar functions. The Java libraries consist of a wide range of classes useful for many different types of programming application. It is these comprehensive libraries, rather than the language itself, which provide the power of Java.

Some examples of the programming tasks for which classes are provided are:

- GUI (Graphic User Interface) objects
- File handling
- Printing
- Mathematical functions
- Networking
- Security features
- Database manipulation

The Core Libraries

These are libraries which must be provided with any Java Development Kit (JDK).

Standard Extensions

These are additional functions which may be provided with the JDK. If they are provided, they must be the standard Java libraries for these functions.

Third party libraries

Innumerable additional classes are available from various sources. Some of these may be purchased, while others can be downloaded for free from the internet. Software development may be speeded up considerably by making use of these pre-written classes. However, two points should be borne in mind before using third party libraries:

- Care must be taken to ensure that they come from a trusted source, and do not contain destructive code
- Some vendors provide classes which will only work on a particular platform, so that programs which include them will not be fully portable. This defeats the object of Java, since its major advantage is its portability. In particular, many libraries provided by Microsoft are Windows-specific, and will not work on other platforms.

Java Beans

Java Beans are re-usable components which may be included in any program. These may be developed by the organisation to provide standard code for various aspects of the business, or they may be acquired from third parties. Beans are available for a wide range of tasks, for example built in spreadsheets or word processors that can be included in user programs.

Review Questions

1.	What is an applet?
2.	What is Byte Code, and what is its impact on program performance?
3.	What is Unicode?
4.	List three types of programming task which can be simplified by the use of the classes provided in the Java libraries.
5.	What points should developers bear in mind before using third party libraries?

OBJECT ORIENTED PROGRAMMING

Learning Module Objectives

When you have completed this learning module you will have :

- An overview of the purpose of object oriented programming
- An understanding of the terms used
- An understanding of the four main principles of object oriented program, which are abstraction, encapsulation, inheritance, and polymorphism

OVERVIEW

Purpose

Manufacturers of modern equipment rarely manufacture an entire product from scratch. Products are generally built from components, which may be manufactured by a different company altogether, or by a different department within the same company.

If a particular component is unavailable, it is often perfectly possible to replace it with a component obtained from a different supplier. Although the replacement component may be completely different in its inner design, it can still be used provided that it has the same connections, and does the same job.

Products can often be adapted to different purposes or environments by simply changing one of the components. For example, a computer manufacturer can supply machines destined for either Europe or the United states by simply including either a 220 volt power supply or a 110 volt power supply.

Object Oriented design attempts to apply this same principle to the development of computer systems. The user's requirements are broken down into a set of components, referred to as objects. Each object is defined as to how it will connect to other objects, and as to the output that it will produce for a given input. The designer of a stereo set does not need to know the inner workings of each component, but only its required input and the expected output. In the same way, objects written by one programmer may be used in a program written by another, provided that the interface and the output are known.

Reliability

Object oriented design does not allow one object to directly change data which belongs to another object; it can only do so by sending a request to that object to make the required change. This tends to produce a much more robust and easily maintained system. It is easy to include checks to ensure that the changes requested are valid. It is also simple to track down problems, since a set of data is changed in one place only.

Ease of modification

Since users of an object are unaware of the inner workings of that object, it is easy to modify it in such a way that it will not affect the rest of the system. Provided that the interface and the type of output remain the same, any changes will not affect other components of the system. An object could be entirely rewritten without other parts of the system being aware of it.

Design

OOP design consists of identifying objects and the relationships between them. Loosely speaking, nouns mentioned in the functional specification of the system will often become objects, while verbs will become 'methods', or activities, of an object.

Objects may be related in three ways:

Uses …	In this type of relationship, one object uses another, usually to obtain information. For example, an order object can use a stock object to verify whether there is sufficient stock to fulfill the order.
Has a …	Orders, for example, can be said to have items, and students have examination results. In this type of relationship, one object will generally create and/or manipulate one or more instances of another object.
Is a …	In this type of relationship, one object is a subset of another type of object. For example, temporary employees and permanent employees are both subsets of employees. They will have data and methods common to employees, and may contain additional data and methods of their own.

Object Oriented design will therefore consist of identifying and classifying objects, deciding what data and methods should belong to each object, and specifying the relationships between them.

TERMS USED

Introduction

Object Oriented programming has its own terminology. In many cases, the terms used are complex-sounding words used to describe simple concepts familiar to most programmers. Some of the terms are described below, while the terms used in stating the four major principle of object oriented programming are described in the next section.

Class

A class is a template or blueprint from which objects are created. A program which deals with employees would need to define an employee class, describing the data which would need to be held for an employee, and the actions which would be required in dealing with an employee.

This class may be reused within different parts of a program. For example, a program may need information about members of staff within a department, and also about the department managers. An object can be created for staff members, and another for managers, from the same employee class.

Object

As stated earlier, objects can be viewed as the components from which a program is built. A class defines the data and activities associated with an object of a specific type. The actual object contains the current values associated with the data, and can be called upon to perform an activity. For example, a Printer object can contain data, such as the location at which printing should take place, and the paper type currently loaded. Activities which could be requested from the printer object could include initialising the printer, printing, changing the print location etc.

Class instance

This is another way of referring to an object.

State

The state of an object refers to the current settings of its data items.

Method

A method is an activity or task carried out by an object. Methods in fact correspond very closely with subroutines or functions in procedural languages.

Fields

Data items held within an object

Instance variables

Another name for fields.

Accessor methods

Methods which allow data to be retrieved from an object

Mutator methods

Methods which allow data to be modified within an object

Messages

Requests for an object to perform an activity i.e. method calls.

THE FOUR MAIN PRINCIPLES

Introduction

There are four concepts which need to be understood when attempting to implement an object oriented design. These are detailed below.

Abstraction

An abstraction is the definition of the important characteristics, or details, of an object, with respect to the task which the program is designed to perform.

There are many different details that could be recorded about a person, for example, but the details that would be abstracted for particular applications would be different. The information stored by the income tax department would be a different set to that stored by his bank, and different again to that stored by the estate agents from whom he rents a property.

This concept is common to all types of system design.

Encapsulation

Manufacturers of integrated circuits encapsulate the circuit in a plastic covering, leaving only the connections visible to the user of the component. The inner workings of the IC are not visible, and cannot be changed by the person building a circuit with it.

This same principle is used in OOP. Objects are 'encapsulated' so that other parts of the system cannot see how they work inside, or make any changes to their data or methods. To use an object within a program, it is only necessary to know what input it expects, and what output it produces.

Data contained within an object can only be changed by invoking a method of that object.

Inheritance

In a previous section, the 'is a …' type of relationship was mentioned. This is a very common type of relationship, and it is often possible to define hierarchical relationships between objects. For example:

	An accounts clerk
is a	Permanent employee
is an	Employee
is a	Person
is an	Object

In Java, all hierarchies begin with a common factor, which is known as object, and a class for this is included in the Java libraries.

In the example above, a class would be defined for Person, based on the Object class, and inheriting data and methods from Object. It would then extend the class, adding data and methods of its own. The data could include the person's name and address.

The Employee class would then be based on the Person class, inheriting its data and methods, and adding a few of its own. This continues down the hierarchy.

Other classes, such as Customer, could also be based on the Person class, and extend it to include data and methods specific to customers.

This concept makes it possible to write code which can be re-used many times within the system. However, care must be taken at the design stage to correctly identify hierarchies in order to make efficient use of this facility.

Polymorphism

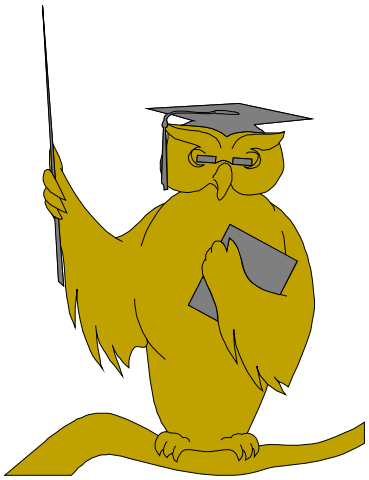
Polymorphism simply refers to the ability to use the same name to refer to different things. This often happens within OOP. The following three examples illustrate three different uses of polymorphism.

An Employee object may have a Print method which prints the employees details. An Order object may also have a Print method, which prints the order and its associated items. Both have the same name, but do different things.

A class may inherit data and methods from a class above it in the hierarchy. However, it is possible to override any of these methods or data items by defining methods or data items with the same name within the new class.

Within a class, it is possible to have two methods of the same name, but each requiring a different data type as its input. At run time, the JVM will call the appropriate method depending on the data type sent to it. For example, an employee may have two methods named setSalary, one of which accepts a string argument and converts it to a numeric data type before using it, and one of which accepts a numeric argument. This makes for flexibility, as it appears to the rest of the program that the Employee object can accept either a string or numeric argument, and can be invoked with whichever is most convenient.

Review Questions – How Would You…



1.	Give three advantages of using Object Oriented Programming
2.	State three types of relationships that can exist between objects
3.	Explain the difference between a class and an object
4.	What is meant by the term ‘method’?
5.	What is the advantage of encapsulation?
6.	Explain the principle of inheritance, and state its advantage.

LANGUAGE BASICS

Learning Module Objectives

When you have completed this learning module you will have seen how to:

- Create, compile and run Java programs
- Use the language components to build a program
- Code class and method headers
- Code data declarations
- Code assignments, making use of the Java operators, casting and method calls
- Use program control features

CREATING, COMPILING AND RUNNING A JAVA PROGRAM

Source files

Java source code is composed of classes. Each source file must have one public class, which will be its entry point. The name of the public class and the name of the source file must match exactly, except that the source file must have the extension of .java. For example, a source file containing a public class named MyProg must be named MyProg.java. Names are case sensitive.

Compiled files

Compiled files in the form of byte code will be named classname.class, so that a class named MyProg will reside in MyProg.class.

Packages

Classes may be grouped into packages. Each package will have its own directory, containing the relevant files.

Compiling

To compile a Java source, the command is
javac filename.java
which may be typed in from the command prompt.

The path must previously have been set to include the bin directory underneath the directory in which the Java development kit was installed.

There are several switches which may be used with the Java compiler. Typing javac with no arguments will list the switches available. Some examples include:
-classpath <path> Additional user-defined classes may be found in <path>
-sourcepath <path> Specify where to find input source files
-d <directory> Specify where to place the compiled files

Any compilation errors will be displayed on screen.

Running a program

To run a program which has been compiled, the command is
java classname
where classname is the name of the public class. This will have been compiled into a file named classname.class

f

JAVA LANGUAGE COMPONENTS

Introduction

Like any language, Java has components, or building blocks, from which a program can be put together. Before learning the actual syntax of Java, it is necessary to have some idea of what these components are.

Classes

All program code must form part of a class in Java. The programmer can code his own classes, as well as making use of classes which are provided in the Java libraries, and classes which have been written by other programmers.

Statements

Classes contain statements, which include data definitions, assignments and method calls. Statements are terminated by semi-colons (;).

Blocks

Statements may be grouped together into blocks as required by the program logic. Braces {} indicate the beginning and end of a block.

Data definitions

All variables must be defined before they can be used.

Methods

Similar to functions in C , these contain all the statements necessary to carry out a particular task. If a class is to be used as a stand-alone program, the public class must have a method named **main** which will be the starting point of the program.

Modifiers

Class, data and method definitions can be prefixed with modifiers which set properties such as whether the item is public (i.e. can be used from other packages).

Assignments

These can be used to place a value into a variable, or to make one object refer to another.

Conditions

These are used in conjunction with program flow control statements

Operators

Operators can be used with assignments or with conditions. They include arithmetic, relational and bit pattern operators.

Method calls

Methods can be called from the current class, from the Java core library classes and from other classes included in the program, provided that access to them is allowed. Some methods can be invoked even when no object has been created from the class (these are known as static methods); others are attached to an object.

Many common tasks such as displaying messages on the screen or manipulating strings are done via method calls to the Java library classes.

Since the methods included with Java are numbered in thousands, a major task in becoming proficient in Java is to become familiar with these classes. A few of them are introduced in this course; the Java documentation contains details of all classes and their methods.

Program control

Java has several statements which control program flow, including conditionals, loops and switches.

Class instantiation

This creates an object from the template provided by a class.

Comments

Comments in Java programs can take three forms:

//	Indicates that the rest of the line should be treated as a comment.
/*	Indicates the beginning of a comment block which may span more than one line.
*/	Indicates the end of a comment block begun with either /* or /**
/**	Indicates the beginning of a comment block which can be used to form part of the program documentation. The utility **javadoc** can be used to build program documentation from these comment lines.

CLASS AND METHOD HEADERS

Class headers

Java programs consist of one or more classes, each of which must be introduced by a class header, followed by the body of the class enclosed in braces {}. The body of the class will contain the definition of its data and methods.

The header takes the form:
[*modifier*] **class *name*** [**extends** *name*] [**implements** *name ...*]

extends *name* and **implements** *name ...* are optional, and are covered in Chapter 5.

Modifier — This is optional, and can include one or more of the following:

- **public** — This indicates that the class can be accessed from other packages.
- **final** — This indicates that the class cannot be extended. Extension of classes is covered in Chapter 5
- **abstract** — This indicates that the class must be extended to be of use. Abstract classes are covered in Chapter 5.

class — The reserved word **class** must be used here to introduce the class. As with all Java code, it is case sensitive.

name — This is the name chosen by the programmer for the class. Class names are case sensitive, must begin with a letter, and can contain letters (including letters from other alphabets, such as the Greek alphabet), numbers and the underscore character. They are not restricted in length. Reserved words cannot be used as class name. A list of reserved words is given in Appendix A.

It is an accepted Java convention that class names should be nouns, with the first letter of each word capitalised. Examples could be: **MyClass, Employee, LineItem**

The definition of a class could therefore look like this:

public final class Payslip { …….. }

where the section between the braces contained the data and methods making up a payslip.

Method headers

As with classes, a method definition would consist of a method header followed by the body of the method enclosed in braces. Each class can contain as many methods as needed.

Method headers take the form:
modifier ReturnDataType name (*argument ...*)

modifier — Several modifiers are available with methods. The common ones include:

- **public** — This method may be invoked from anywhere, provided that the class which contains it is also public.

	static	The method forms part of the class, not of an object created from the class, and can therefore be used even when no object has been created.
	final	The method cannot be overridden by a method of the same name when the class forms part of an inheritance hierarchy.
ReturnData Type	As with functions in other languages such as C, Pascal and Basic, methods can return a value. If no value is to be returned by this method, then the return data type should be the reserved word **void**. If a value is to be returned, this should then be the data type of the value returned. Valid data types are listed in the next section.	
name	This is the name given to the method by the programmer. Method names follow the same rules as class names. It is a common convention to begin the names of accessor methods with **get** and mutator methods with **set**, for example **getBalance** and **setSalary**.	
argument	Data can be passed to the method when it is called in the form of an argument. The list of arguments enclosed in brackets should include a data type and a name for each argument to be passed to the method. This serves as a data definition for each argument. If no arguments are to be passed, then the brackets should still be included, with nothing between them. An example of an argument list is: (int Salary, char ActionCode)	

The following examples illustrate method headers:

public static void printPayslip (string EmpNum)

int CalcOvertime (int hours, int rate)

static void refreshScreen ()

As stated earlier, program execution is begun at the method named **main** of the public class. The arguments passed to this method will be the arguments placed on the command line when the program is executed. Unlike C programs, these arguments will not include the program name. The header for the **main** method is usually coded like this:

public static void main (String[] args)

Java programs do not return values, and the command line arguments are defined as an array of strings.

Skeleton program

The following program will compile and run, but will do nothing. It can be used as a skeleton program, inserting data definitions and methods as needed.

```
public class TestProg
{
  public static void main(String[] args) {}
}
```

Please complete Exercise 2 from the exercise pack

DATA DECLARATIONS

Data Types

Java contains eight primitive data types, which form part of the language. Additionally, classes are provided for complex data types such as strings, numbers containing decimal fractions, and dates. The primitive data types are covered in this section. Strings, numbers with decimal fractions and dates are covered in Chapter 4.

The primitive data types

There are four integer, two floating point, one character and one Boolean data type included as part of the Java language. These are detailed below.

Integer data types	**long**	Long integers occupy 8 bytes, and are capable of storing signed numbers 18-19 digits in length.
	int	These occupy 4 bytes, and store signed numbers 8-9 digits in length.
	short	Short integers occupy 2 bytes, and store numbers in the range –32768 to +32767.
	byte	Byte variables occupy 1 byte and store numbers in the range –128 to +127
floating point data types	**float**	Float variables occupy 4 bytes. They can be used to store numbers with fractional parts. However, since the fractional part is stored as a binary fraction, and binary fractions do not always convert exactly to decimal fractions, there may be some rounding errors, and they are not suitable for storing currency values for accounting functions. Float variables store 6 to 7 significant digits with a range of approximately ±3.4E+38
	double	This is a double precision floating point variable occupying 8 bytes. As with Float, it is stored as a binary fractions. Doubles have 15 significant digits and can store numbers in the range of approximately ±1.8E+308
Character data	**char**	This stores a single Unicode character occupying two bytes. Unlike C, character variables in Java cannot be used for arithmetic. They can, however, be converted to and from the integer data types to obtain a numeric Unicode value.
Boolean data types	**boolean**	This is used to store a boolean value of **true** or **false**, and can be used in conditions. It occupies 4 bytes. Unlike C, the boolean data type is not compatible with the numeric data types, and cannot be converted to or from them.

Variable declarations

Variables must be declared and given a data type before being used. Variable declarations may appear anywhere within a Java program.

The scope, or area of visibility, of a variable is the block in which it was declared. Sections of the program outside this block will be unable to access the variable.

Variables which need to be accessed throughout a class should be declared at the front of the class, before any methods are defined. Variables which need to be used throughout a method must be declared in the outer block of the method.

Variable names

Variable names follow the same rules as class names. Variable names must be unique within the block in which they are declared.

Declaring a variable

Variable declarations take the following form:

[*modifier*] *type name* [, *name* ....] [= *expression*]

modifier The modifier is optional, and in most cases is omitted. Valid modifiers include:

- **final** This indicates that the value of the variable cannot be changed i.e. it is a constant. In this case, a value or expression must be allocated to it within the declaration.
- **public** If the class is public, this indicates a global variable. Global variables conflict with the principles of object oriented programming, and should not be used. It is often useful, however, to declare a global constant, so that **public** is almost always used in conjunction with **final.**
- **static** This indicates that the variable will be held once per class, as opposed to once for each object created from the class.

type This can be any of the valid primitive types, or **String**. Strings are covered in the next chapter. The valid primitive types are: **long, int, short, byte, float, double, char** and **boolean**.

name This should be a valid variable name, unique within the block. If necessary, a list of variable names can be supplied, each of which will be allocated with the same data type.

It is an accepted convention in Java to name constants in capital letters.

expression Optionally, the declaration can initialise the variable to a given value. The expression can contain literals, operators and references to existing variables, and follows the rules of assignments which are covered in the next section.

Variables declared within methods should be initialised by the programmer, as they will not be automatically initialised to any particular value at run time.

The following are examples of variable declarations.

```
int x, y;
final double DISCOUNTRATE = 0.15
double monthlySalary = annualSalary/12
```

Defining arrays

The creation of an array requires two stages: defining the array, and creating the array, although it is possible to use a shortcut to achieve both steps in one statement.

The definition of the array takes the form
datatype [] *name*;
for example
double [] result;

Note that no dimensions are given for the array at this point. This step simply creates a pointer which can be used to address an array.

Before using the array, it must be created and allocated a size as follows:
name = **new** *datatype* [*n*]
where *name* and *datatype* match those used in the declaration, and *n* is a literal specifying the number of elements in the array, or the name of a data item containing the number of elements in the array.

The statement used to create the array declared in the previous example with 20 elements would be:
result = new double [20]

These two stages can be combined into one, in which the syntax would be as follows:
datatype [] *name* = **new** *datatype*[*n*]
for example
double[] result = new double[20]

It is also possible to specify a set of values to be stored in the array, as follows:
datatype [] *name* = { *literal1, literal 2 ...*};
for example
double [] result = {85, 60, 72, 48, 57, 42, 78};

It should be borne in mind when it comes to assignments that arrays are treated as objects rather than primitive data types in Java.

To refer to a particular element in an array, a subscript, or pointer, is used, so that the fifth element in the **result** array would be referred to as **result[4]** (Note that the positions in an array are numbered beginning at zero). The subscript can also be a dataname containing the required value.

Using literals in assigning a value

Literals in Java take different forms, depending on the type of data with which they are associated.

Below is a breakdown of the format of literals for different data types.

Byte, short, int	A number, not enclosed in quotes, optionally prefixed by a sign or A number in Hexadecimal format prefixed by 0x Examples: 142; -120; 0xA14E
Long	As for byte, short and int, but must be suffixed by a capital L. Examples: 4000000000L; -138742L

double	A number, not enclosed in quotes, using a full stop for the decimal point. Alternatively, the number could be stated in scientific format. Examples: 472.75; -0.048; 8.85E-12
float	As for doubles, but must be suffixed by a capital F. Examples: -4.87F; 3.0E+8F
char	Character literals are enclosed in single quotes, which may contain the actual character, an escape sequence, or the numeric Unicode equivalent of the character prefixed with \u. Examples: 'H' '\b' '\u0008'
boolean	Values can either be **true** or **false**. Quotes are not used.
String	String literals are enclosed in double quotes. Examples: String Message = "Received with thanks" String Prompt = "Please enter your account number"

Please complete Exercise 3 from the exercise pack

ASSIGNMENTS AND OPERATORS

Assignments

An assignment is used to manipulate the values of variables, or to make one object point to another. As with C, assignments may become fairly long and complex, since all arithmetic and logic operations are carried out with assignments.

Assignments generally have the syntax
variablename = expression

Expression can be a simple literal, the name of a variable, a call to a method which returns a value, or a complex expression built from variable names, literals, method calls and operators. The operators are discussed later on in this section.

Using assignments with objects

Assignments work differently for objects than for primitive data types. The statement
variablename1 = variablename2
will place a copy of the contents of variable name 2 into variable name 1.

However, this is not the case with objects, including strings and arrays.
The statement
objectname1 = objectname2
will cause object name 1 to point to object name 2, so that they both refer to the same area of memory. Any subsequent changes made to object name 1 would then obviously affect object name 2.

If a copy of an object is required, then an assignment must not be used. Many objects contain a method named **Clone** which can be used to copy their contents to another object.

Operators

There are many different operators available in Java. These are classified below.

Arithmetic

+	Addition. Note: the + sign can also be used for string concatenation.
-	Subtraction
*	Multiplication
/	Division. If both of the operands are integer, integer division with truncation is carried out; otherwise, the answer will contain fractional parts.
%	Modulus. This will give the remainder portion of an integer division carried out on the two operands e.g. 15 % 6 would give the answer 3.

Note that there is no exponential (to the power of) operator. A method of the Math class can be used to achieve this, as detailed in Chapter 4.

Increment and Decrement

var++ Where var indicates a variable name, var++ will increment the contents of the variable by 1. This will be done after the variable has been used in the expression, so that after the following statements:

```
int n = 5;
int x;
x=3*n++
```

n would contain 6, since it has been incremented, and x would contain 15, since it is calculated using the value of n before incrementation.

	var--	This works in the same way as var++, except that the contents of **var** would be decremented by 1.
	++var	This increments the contents of **var** before var is evaluated for the expression, so that after the following statements: int n = 5; int x; x=3*++n n would have the value of 6, and x would have the value of 18, since the incremented value is used in evaluating the expression.
	--var	This works in the same way as ++var, except that **var** is decremented by one.
Relational and Boolean	==	This tests for equality. The double equals sign avoids confusion with the single equals sign used for assignments.
	!=	Not equal to
	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
	!	Not
	&&	And
	||	Or
Bit manipulation	**&**	Logical And
	|	Logical Or
	^	Exclusive Or
	~	Not
	>>	Bit shift right, extending sign bit
	<<	Bit shift left
	>>>	Bit shift right, extending with zeros
Condition	**?**	This is used within an assignment to choose between two possible values. Its usage is as illustrated in the following example: **SalesTaxPercent=(TaxDue=='Y')? 20:0;** The condition TaxDue=='Y' is used to decide whether to place the value 20 or the value 0 into SalesTaxPercent.

Shortcut

Programs often need to use the current value of a variable in calculating its new value, for example

x=x+10

This can be shortened to

x+=10

in Java. This produces the same result as the first example.

This shortcut can be used with most of the arithmetic and bit manipulation operators, so that the following are valid:

+=; -=; *=; /=; %=; &=; |=; ^=; <<=; >>=; >>>=

Casting

It is often necessary to assign values of one data type to a variable of another data type. For this type of assignment, the Java compiler checks to see whether this could result in loss of data. For example, assigning a short integer value to a long integer value can never result in loss of data, but assigning a long integer value to a short integer could do so. For this reason, the compiler will reject those assignments which may result in data being lost.

However, it is possible to override this by a technique known as **casting.** Casting informs the compiler that the programmer is aware that loss of data may occur, but wishes the statement to be executed regardless.

Casting can also be used with objects, but this is covered in a later chapter.

Casting simply involves prefixing the expression in an assignment with the data type, in brackets, to which the expression is to be converted.

Examples:
Salary=(int) AnnualSalary / 12
Result = (char) 65

The following conversions do not need casting, as they cannot result in loss of data:

byte	to	short, int, long, float, double
short	to	int, long, float, double
int	to	long, float, double
long	to	float, double
float	to	double
char	to	int

Method calls

Many methods are designed to return a value. Those that do so can be included in an expression in the same way as variables. The data type returned by the method is defined in the method header, as seen in the previous section.

Calls to methods which do not return a value can be coded as stand-alone statements.

To invoke a method defined in the same class, code the method name followed by brackets containing any parameters which it requires, for example:
TaxDue=CalcTax(Salary, TaxCode);
would invoke the method CalcTax using variables Salary and TaxCode as parameters, and would assign the value returned by CalcTax to the variable TaxDue.

To invoke a static method from another class, prefix the method name with the classname and a period (.), for example
MthAvg=Math.round(Annual/12)
invokes the **round** method from the Java built-in Math class to round off the result of the calculation to the nearest whole number.

To invoke a method which is not static, prefix the method name with the name of an object which has been created from that class, for example:
LengthOfString=MyString.length()
uses the length method of an object MyString which has been created as class String.

The Java built-in libraries contain many useful methods which can carry out a wide variety of functions. Some of these are covered in later chapters; however, the Java language documentation contains a full list of classes and methods, including the data types returned and the parameters expected for each method.

Order of evaluation

Java assignments can become extremely long and complicated, often using several operators in a single statement. It is therefore important to understand the order of evaluation.

The table below lists the different levels in the order of evaluation to which each operator belongs. Those at a higher level will always be carried out before those at a lower level.

Method calls
Brackets, !, ~, ++, --, Unary plus and minus (ie sign associated with a number), cast
*, /, %
+,-
<<,>>,>>>
<, <=, >, >=
==,!=
&
^
|
&&
||
?
=,+= and other equivalent shortcuts

Sample program: notes

Brackets may always be used to control the order of evaluation.
The following sample program takes the amount of a loan and an interest rate as arguments from the command line. Note: these are whole numbers.

It then calculates simple interest for one year, and converts this to a monthly interest value to the nearest whole dollar. This is then displayed on the screen.

The program makes use of the following methods from the Java libraries:
System.out.println(String)
displays the value of String on the system output.
Integer.parseInt(String)
converts a string value to an integer.
Math.round(double)
rounds a double variable to the nearest whole number, returning a long integer.

Sample program

```
public class interest
{
 public static void main(String[] args)
 {
  int Loan,Rate;
  long MthlyInt;
  Loan=Integer.parseInt(args[0]);          // First argument is the loan
  Rate=Integer.parseInt(args[1]);          // Second argument is the rate
  MthlyInt=
  Math.round(Loan*(double)Rate/1200);      // Casting to double is needed
                                           //  to prevent integer division
  System.out.println(MthlyInt);
 }
}
```

Please complete Exercise 4 from the exercise book

PROGRAM CONTROL FEATURES

Introduction

As with most programming languages, the flow of Java programs can be controlled with selections, loops or switches. There is no GOTO in the Java language.

Conditional statements

Conditional statements are coded according to one of the following:

if (*condition*) *statement;*
or
if (*condition*) *statement;*
else ***statement;***
or
if (*condition*) {*block}*
or
if (*condition*) {*block}*
else **{*block*}**

condition is any valid boolean expression, which may include variable names, literals, method calls and operators, as well as variables with data type boolean.

The condition can be used to select either a single statement or a block of statements.

Examples:

```
if {CustType==”Dealer”) AmtDue -= (AmtDue * 0.1);

if (FinalMark < 50)  Result  = “Fail”;
else                 Result = “Pass”;

if (SaleQty <= StockQty && SaleValue <= CreditAvail)
  {
   StockQty -=SaleQty;
  CreditAvail -= SaleValue;
  Invoice.Write();
  }
else Invoice.Reject();
```

For … loop

The for .. loop is generally used when a statement or block of statements needs to be executed a fixed number of times.

The syntax is:

for (*InitiationStatement;* *Condition; IncrementationStatement*)
statement;
or
for (*InitiationStatement;* *Condition; IncrementationStatement*)
{*block*}

The initiation statement should assign an initial value to a counter variable, and optionally declare it as well.
Examples could be:
Counter=1; //Initialises a predefined variable Counter to 1
short i=10; //Declares a variable named i and initialises it to 1

If the counter variable is declared here, it can only be used within the loop.

The condition is checked before each cycle of the loop; when the condition becomes false, the loop terminates.

The incrementation statement is used to increment the counter after each cycle of the loop. A simple increment (eg i++) can be used to increment the counter by 1, or any assignment expression can be used to increment it by some other value.

The following example prints the square of all the numbers between 1 and 10.

```
public class Squares
{
 public static void main(String[] args)
 {
  for (int i=1; i<11;i++)
     System.out.println(+ i + " squared is " + i*i);
 }
}
```

/* Note: The + operator is used in the println method to concatenate the items to be printed. Since a string is expected here, and the first item to be printed is an integer, the concatenation operator precedes it to convert it to a string. */

Break

The break statement can be used with any of the loops to break out of the loop prematurely. It will cause execution to continue with the next statement after the loop.

A variation of this is **break** *label* which can be used to break out of a hierarchy of nested loops. It would be used when an exception in an inner loop should terminate not only the inner but the outer loop.

The label is placed at the beginning of the outer loop; however, when the break is executed, the program will continue with the next statement after the outer loop
e.g.:

```
int i;
mainloop:
for (i=1;i<10;i++)
{
  for (j=1;j<10;j++)
  { …….
    if (errorcondition)  break mainloop;
  }
}
statement 1
```

When **errorcondition** became true, program execution would continue with **statement1**, regardless of the current values of **i** and **j.**

While … loop

This loop can be used to repeat a statement or block of statements while a

condition remains true. The condition is tested before any of the statements are executed, so that if the condition is false at the outset, the loop will not be executed at all.

Example:
while (MoreEntries)
Entry.Process;

Do …. while loop

This is similar to the previous type of loop, except that the condition is checked after processing the statement or block of statements, so that it will always be executed at least once, even if the condition is false at the outset.

Sample Program: Check for Prime Numbers

```
/* ***************************************
 * This program will input a number
 * from the command line, and check
 * to see whether it is a prime number
 * ***************************************/

public class prime
{
 public static void main(String[] args)
 {
    int numIn, counter;
   double numRoot;
   numIn = Integer.parseInt(args[0]);

/* ***************************************
 * Factors between 2 and the square
 * root of the number entered will be
 * searched for
 * ***************************************/
    numRoot=Math.sqrt(numIn);
    counter=2;
     boolean factorised=false;
    String result;

    do
    {
      if (counter > numRoot)      // Factors cannot be greater
                break;            // than the square root
     if(numIn % counter == 0)    // If dividing by the counter gives
            factorised=true;      // remainder 0, a factor was found
     counter ++;
   }
   while (! factorised);

   if (factorised)
     System.out.println(+ numIn + " is not a prime number");
   else
     System.out.println(+ numIn + " is a prime number");
  }
}
```

Switch

The switch is used to select alternate program paths depending on the value of a variable.

The general syntax is:

```
switch(variablename)
{
    case value:
          statement …..;
```

```
            break;
    case value:
            statement …..;
            break;
etc
    default:
            statement .....;
            break;
}
```

The contents of *variablename* are tested against each of the **case** values listed. If a match is found, the statements between the matching **case** and the next **break** are executed.

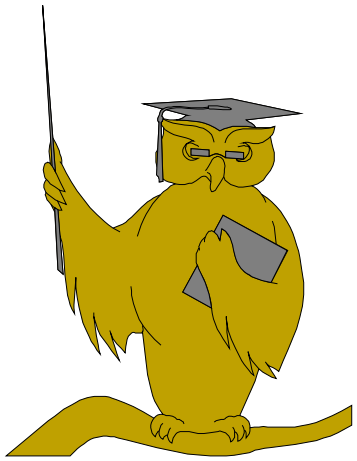
A **case** with its relevant statements would be coded for each value which is of interest to the program. Optionally, a **default** can be included. The statements listed under the default will be executed if there are no matches.

Sample program using **switch**:

```
// ********************************************************
// * This program will carry out a calculation
// * entered on the command line. The calculation
// * should be entered as a number, followed by a
// * space, followed by an operand, followed by a
// * space and a number. Valid operands are +,-,x
// * and /.
// ********************************************************
public class calc
{
 public static void main(String[] args)
 {
  double Num1, Num2, Ans=0;
  char Op;
  Num1=Integer.parseInt(args[0]); //First argument is a number
  Op=args[1].charAt(0);            //First character of second
                                   //argument is the operand
  Num2=Integer.parseInt(args[2]); //Third argument is a number

// ********************************************************
// Decide on the correct calculation depending on
// the operand entered
// ********************************************************

   switch(Op)
   {  case '+':
         Ans=Num1+Num2;
         break;
      case '-':
         Ans=Num1-Num2;
         break;
      case 'x':
         Ans=Num1*Num2;
         break;
      case '/':
         Ans=Num1/Num2;
         break;
      default:
         System.out.println
         ("Can't do that - I'm only a cheap calculator");
         break;
   }
   System.out.println("Answer is " + Ans);
 }
}
```

Note: The **charAt** function is used here to obtain the first character of a string, and place it in a character variable.

Review Questions – How Would You…



1.	Compile and run a Java program?
2.	Group together statements in a block?
3.	Code the header for a public class? How must this relate to the name of the program source file?
4.	Define a variable which will be able to store a whole number 15 digits in length?
5.	Define an integer constant containing the number 300 which can be accessed from anywhere?
6.	Calculate the sum of variables named a, b and c, storing the answer in a variable named d?
7.	Code a Java statement which will display your name on the screen?
8.	Given two numeric variables named a and b, code the statements which will subtract the larger number from the smaller number placing the result in a variable named c.

Please complete Exercise 5 from the exercise pack

INTRODUCTION TO LIBRARY CLASSES

Learning Module Objectives

When you have completed this learning module you will have seen how to use methods provided in some of the classes defined in the core library java.lang, including:

- Character
- Integer
- Double
- Math
- String

OVERVIEW

Libraries

The Java libraries provide an enormous range of features which enhance the language capability. This chapter looks at some of the methods provided in classes defined in the library java.lang.

This chapter is not meant to be a full cover of the classes contained in this library. A few useful methods have been selected from some of the classes, to give the student an idea of what is available. Many of the classes have predefined constants which can also be useful to the programmer, and a few of these are introduced here.

Full documentation on java.lang and other libraries is provided in the Java language specifications.

Method calls

Methods will be listed in subsequent sections in the format:
ReturnDataType MethodName(ParameterDataType ...)

ReturnDataType specifies the data type which will be returned by this method, or **void** if no data type is returned.

MethodName should be prefixed by the class name when calling the method. Each of the sections below deal with one class, so the class name will be the same for each method within the section.

ParameterDataType specifies the data types of parameters required by the method, if any. If more than one parameter is required, they should be separated by commas.

For example, the following method is listed in the section dealing with the Character class:

char toUpperCase(char)

This method could be invoked as follows:

Var1 = Character.toUpperCase(Var2)

where Var1 and Var2 are variables defined as type **char.**

Summary of methods introduced in this chapter

The library java.lang includes what are known as wrapper classes for several of the primitive data types. These include Character, Integer, Long, Float and Double. These provide several methods useful in manipulating each data type.

java.lang.Math contains mathematical functions, while String and StringBuffer deal with manipulation of character strings.

THE CHARACTER CLASS

Introduction

The class **Character** provides useful constants and methods for use with variables of data type **char.**

Constants

MIN_VALUE	This contains the lowest value which can be stored in a character field, equivalent to defining a literal '\u0000'
MAX_VALUE	The largest possible character value; equivalent to '\uffff'.

Methods

boolean isDefined(char)	Returns true if the supplied value has been defined in the Unicode table.
boolean isLowerCase(char)	Returns true if the character is defined as lower case in the Unicode table.
boolean isUpperCase(char)	Returns true if defined as upper case.
boolean isDigit(char)	Returns true if defined as a digit in the Unicode table.
boolean isLetter(char)	Returns true if defined as a letter
boolean isLetterOrDigit(char)	Returns true if defined as either a letter or a digit.
boolean isSpace(char)	Returns true if the supplied parameter contains a space, tab, line feed or form feed.
char toLowerCase(char)	Returns the lower case equivalent of the supplied parameter, if any; if there is none, the original parameter is returned.
char toUpperCase(char)	Returns the upper case equivalent of the supplied parameter, if any; if there is none, the original parameter is returned.

Note: Since the Unicode character set includes the alphabets of many different countries, methods such as **isLetter** etc. will return true if the supplied character is a letter in **any** of the defined alphabets.

Please complete Exercise 6 from the exercise pack

THE INTEGER CLASS

Introduction The class **Integer** provides useful constants and methods for use with variables of data type **int** and compatible data types**.**

Constants

MIN_VALUE	The lowest integer value, i.e. -2147483648
MAX_VALUE	The lowest integer value, i.e. 2147483647

Methods

String toString(int)	The argument is converted to string format.
int parseInt(String)	The contents of the string are converted to integer format. The string may only contain digits, optionally preceded by a minus sign.

Note: The class **Long** has a similar definition of constants and methods which may be used with long integers.

THE DOUBLE CLASS

Introduction

The class **Double** provides useful constants and methods for use with variables of data type **double** and compatible data types**.**

Constants

MIN_VALUE	The lowest value which can be stored in this data type.
MAX_VALUE	The highest value which can be stored in this data type
NEGATIVE_INFINITY	A value which is taken to represent negative infinity
POSITIVE_INFINITY	A value which is taken to represent positive infinity
NaN	Not a Number: a value which indicates that the result of a calculation or assignment was not a valid number (for example the result of attempting to divide by zero)

Methods

string toString(double)	This will return a string containing either the number, or, if the double contains a value which is not a number, or represents positive or negative infinity, the string value "NaN", :infinity" or "-infinity"
double parseDouble(string)	This will attempt to interpret a string of characters as a number of type double. The string can be either in normal or scientific format. This method is new in Java 1.2; programs written with earlier versions of Java had to use the DoubleValue method, which is much clumsier, since an object had first to be created from the class.
boolean isNaN(double)	This will return true if the argument contains a value representing NaN (not a number); this generally happens if an illegal arithmetic operation is carried out on a double variable.
boolean isInfinite(double)	Returns true if the argument contains a value representing positive or negative infinity.

Note:

The class **Float** has a similar definition of constants and methods which may be used with variables of data type **Float.**

THE MATH CLASS

Introduction

The **Math** class contains constants and methods commonly used in mathematics. For a full list of the Math functions, consult the Java language documentation.

Constants

E	The exponential constant: the base of natural logarithms
PI	The ratio of the circumference of a circle to its diameter. (π)

Methods

double sin(double)	The sine of the angle in radians given as an argument.
double cos(double)	The cosine of the angle in radians given as an argument.
double tan(double)	The tangent of the angle in radians given as an argument.
double asin(double)	The arc sine in radians of the argument.
double acos(double)	The arc cosine in radians of the argument
double atan(double)	The arc tangent in radians of the argument
double max(double,double)	Returns the greater of the two arguments *
double min(double, double)	Returns the smaller of the two arguments *
double pow(double,double)	This returns the first argument raised to the power of the second.
double random()	Returns a random number greater than 0 and less than 1.
long round(double) or **int round(float)**	Returns the value of argument rounded to the nearest whole number.
double sqrt(double)	Returns the square root of the argument.

Note: items marked * can also use **float, int or long** as data types for arguments and return values.

Please complete Exercise 7 from the exercise pack

THE STRING CLASS

Introduction

String is a special class, in that whenever a variable is defined as being of type String, an object is created of the class String.

Most of the methods in String are not static, in other words they belong to the object rather than to the class. This means that unlike the methods examined so far, the method name should be prefixed with the name of the object rather than the class; the object being the variable that was defined as String. The method will then operate on the data that is stored within the object. For example, to place the uppercase equivalent of a string variable Surname into a string named NameIndex, the method toUpperCase would be invoked as follows:
NameIndex=Surname.toUpperCase()

Once created, strings cannot be amended. The methods provided with the String class examine, compare, extract substrings, and make copies from String objects. If a string is to be modified, the StringBuffer class should be used. StringBuffer is not covered in this chapter, since further concepts in creating and dealing with objects need to be covered first.

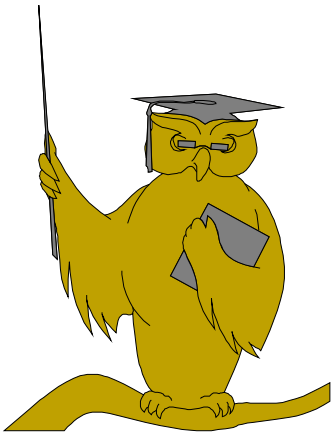
There are a large number of methods defined in the String class. A few of the more commonly-used ones are listed below.

Methods

void String(char[]) **void String(byte[])**	Creates the string from the array .
boolean equals(String)	Returns true if the argument is equal to the string.
int length()	Returns the number of characters contained in the string.
char charAt(int)	Returns the character at the position pointed to by the argument.
char[] toCharArray()	Creates a new character array containing characters contained in the string.
boolean equalsIgnoreCase(String)	Returns true if the two strings are equal if case is ignored.
int compareTo(String)	returns a negative number if this string is alphabetically lower than the argument string, zero if they are equal, and a positive value when this string is higher alphabetically than the argument.
boolean RegionMatches (int,String,int,int)	The first argument is an offset within the current string, the second is a string to be compared with it, the third is an offset within the argument string, and the fourth is a length. Substrings at the given offset and of the given length are compared between the two strings. A True value is returned if they are equal.
boolean startsWith(String)	Returns true if the string starts with the given sequence of characters.
boolean endsWith	Returns true if the string ends with the given sequence.

int indexOf(String)	If the argument exists as a substring within the current string, the starting position of the substring is returned.
String substring(int,int)	Returns the substring beginning at the position indicated by the first argument, and ending at the second.
String concat(String)	Returns a concatenation of the current string and the argument string.
String replace(char,char)	Returns a copy of the string with all characters matching the first argument replaced by the second.
String toLowerCase()	Returns a copy of the string converted to lower case.
String toUpperCase	Returns a copy of the string converted to upper case.
String trim()	Returns a copy of the string with leading and trailing spaces removed.

Review Questions – How Would You…



1.	Convert a String value to a Double, assuming that the string contained a valid value?
2.	Check whether a character contained a digit?
3.	Raise a number stored in a variable **Num** to the power of three?
4.	Obtain the uppercase equivalent of a string?
5.	Check whether a variable containing a name begins with the sequence “Mr”

Please complete Exercise 8 from the exercise pack

CLASSES, INTERFACES AND OBJECTS

Learning Module Objectives

When you have completed this learning module you will have seen how to:

- Create and make use of an object
- Write classes from which objects can be created
- Create a subclass by extending a superclass
- Create and use abstract classes and methods
- Create and use interfaces
- Create and use inner classes

CREATING AN OBJECT FROM A CLASS

Purpose

Up to this point, we have looked at simple programs containing only one class with static methods and local variables. In fact, we have created programs which have a single component only – the public class with which program execution begins.

Just as a complex electrical product needs many components, complex programs are built of many objects.

An object is a collection of data and methods, and is built from a class, which is the prototype specifying what these data and methods will consist of. Many objects may be built from the same class. Each one will have a unique **state**, since each stores a separate copy of the data contained by the object.

Both arrays and strings, which we have looked at in earlier chapters, are objects. However, they are not created in the same way as standard objects. This section deals with creating and using standard objects.

Creating an object

There are two steps in creating an object; however, there is a shortcut which allows both steps to be carried out in one statement.

Firstly, a reference variable for the object must be created. This is a pointer which will be used to access the object, and it is declared in much the same way as a data declaration. The pointer should not be confused with the object itself. Creating a pointer simply sets aside a small amount of memory which can hold the memory address of an object. It does not reserve any memory for the actual object..

To create a reference variable, the syntax is:
classname objectname

classname is the name of a class which may either have been defined within the program, or may be contained in a library.

objectname is a name allocated by the programmer, and will be used to reference the object. Object names have the same naming rules as classes.

Example:
StockItem ItemPurchased;

Before the object can be used, it must be created, and, in many cases, values assigned to it. The number and type of values which must be assigned when an object is created are defined in the class.

The syntax for creating an object is as follows:

objectname = **new *classname***(*valuelist*);

objectname is the name given to the object when the reference was created.

classsname is the name of the class from which the object will be created.

valuelist is a list, separated by commas, of the values which will be assigned to the object. The values may be expressed as literals, datanames, or expressions.

Example:
ItemPurchased = new StockItem(StockCode,1,"Sale")

Using a method belonging to an object

The class from which the object was created will define the methods which are available, and the parameters which they require.

Once an object has been created, the methods are invoked by the name of the method preceded by the object name, as opposed to the classname.

Example:
ItemPurchased.SubtractFromStock(SaleQuantity)

Imported classes

Classes are grouped into packages. All classes from a given package reside in the same directory. When creating an object, the compiler will expect to find the class either within the current package, or in the java.lang package. If classes are to be used from other packages, an **import** statement must be included at the front of the source file.

Example:
import java.util.*;

Using an Object – Examples

The following examples using classes defined in the Java libraries will illustrate the creation and use of objects.

Using the Gregorian Calendar class

The GregorianCalendar class defines an object which can store a date and time, and carry out various functions on it. In practice, many organisations define their own calendar functions, as the GregorianCalendar class has many drawbacks:

- Months are numbered from zero, so that the 10th of December 1999 would be entered as 99 11 10. This can lead to confusion and program errors, particularly since days of the month are numbered from 1.
- It allows the entry of two digit years, but will not handle them correctly at the change of the century. The year 99 is assumed to be 0099, and increments to the year 0100.
- Although it allows a number of days to be added or subtracted to or from a date, it does not give the difference of two dates in days, which is a function often required in commercial applications.
- It does not do proper validation on dates entered, but simply interprets incorrect dates as best it can.

However, it serves as a good example in creating and manipulating objects, and it is a fairly simple matter to write a class which extends GregorianCalendar to provide a useful commercial date class.

This sample program does the following:

- Since GregorianCalendar is in the java.util library, this must be imported
- Uses command line arguments for a day, month and year, and a number of days to be added to this date.
- Defines an object Date1 as a GregorianCalendar, and creates an instance of it
- Stores the date entered on the command line in Date1
- Uses the **add** method of Date1 to add the given number of days to this date
- Uses the **get** method of Date1 to retrieve the day, month and year after the calculation in order to display the result on the screen. **DAY_OF_MONTH, MONTH** and **YEAR** are constants defined in GregorianCalendar to indicate which fields are to be accessed.

```
import java.util.*;
public class test1
{
 public static void main(String[] args)
 {
  int Day,Mth,Year,DayToAdd;
  String StrDate;

  Day=Integer.parseInt(args[0]);
  Mth=Integer.parseInt(args[1]);
  Year=Integer.parseInt(args[2]);
  DayToAdd=Integer.parseInt(args[3]);
  Mth=Mth-1         // Convert from user form of months 1 to 12
                    // to Gregorian form of months 0 to 11
  GregorianCalendar Date1;
  Date1=new GregorianCalendar(Year,Mth,Day);
  Date1.add(Date1.DAY_OF_MONTH,DayToAdd);
  StrDate=Date1.get(Date1.DAY_OF_MONTH)
  + ":" + (Date1.get(Date1.MONTH)+1)  // Convert back to user form
  + ":" + Date1.get(Date1.YEAR);
  System.out.println(StrDate);
 }
}
```

Using the StringBuffer class

The StringBuffer class is similar to String, except that it allows the contents of the string to be modified.

The following program calculates interest using a balance and a rate entered on the command line.

It creates an instance of the StringBuffer class containing text to be printed, and uses the **insert** method of the StringBuffer to insert the calculated interest into this string before printing it.

```
public class int1
{
 public static void main(String[] args)
 {
 double Balance, Interest, Rate;
 Balance=Double.parseDouble(args[0]);
 Rate=Double.parseDouble(args[1]);
 Interest=Balance * Rate / 100;
```

```
  StringBuffer OutString = new StringBuffer("Interest of  will be charged");
  OutString.insert(12,Interest);      // 12 is the position at which
                                      // Interest is inserted
  System.out.println(OutString);
  }
}
```

Using the BigDecimal Class

The BigDecimal class defined in java.math (not to be confused with java/lang/Math!) allows numbers to be stored and manipulated as decimal rather than binary fractions, eliminating the rounding errors which occur when converting decimal fractions to binary and vice versa. Methods are provided for such operation as arithmetic, rounding, comparison and conversion.

The following example is a modified version of the 'Cheap Calculator' program which was originally used to illustrate the **switch** statement.

It has been changed to create a BigDecimal object for each of the two operands, and for the answer. Instead of using the Java language operators to perform arithmetic, the **add, subtract, multiply** and **divide** methods are used. Note that the **divide** method needs two further arguments: the number of decimal places to store in the answer, and the rounding method. The rounding methods available are predefined constants in the BigDecimal class.

```
// *******************************************************
// * This program will carry out a calculation
// * entered on the command line. The calculation
// * should be entered as a number, followed by a
// * space, followed by an operand, followed by a
// * space and a number. Valid operands are +,-,x
// * and /.
// *******************************************************
//
// * Modified version - uses BigDecimal class to
// * store the numbers and perform the arithmetic
// *******************************************************
import java.math.*;
public class calcdec
{
 public static void main(String[] args)
 {
  BigDecimal Num1, Num2, Ans;
  char Op;
  Num1=new BigDecimal(args[0]);   //First argument is a number
  Op=args[1].charAt(0);           //First character of second
                                  //argument is the operand
Num2=new BigDecimal(args[2]);     //Third argument is a number

  Ans=new BigDecimal(0);

// *******************************************************
// Decide on the correct calculation depending on
// the operand entered
// *******************************************************

   switch(Op)
   { case '+':
```

```
            Ans=
            Num1.add(Num2);
            break;
        case '-':
            Ans=
            Num1.subtract(Num2);
            break;
        case 'x':
            Ans=
            Num1.multiply(Num2);
            break;
case '/':
            Ans=
            Num1.divide(Num2,2,Num1.ROUND_HALF_UP);
            break;
        default:
            System.out.println
            ("Can't do that - I'm only a cheap calculator");
            break;
    }
    System.out.println("Answer is " + Ans);
  }
}
```

Please complete Exercise 9 from the exercise pack

CLASSES FROM WHICH OBJECTS CAN BE CREATED

Overview

Objects consist of data and methods. The class from which the object is created defines what data will be stored, and what methods will be available for use with the object.

Instance Variables

The Instance Variables define the data to be stored by the object. Instance Variables are not the same as local variables used by an object's methods. They are defined outside of any method, usually at the front of the class. Whereas the local variables of methods do not retain their value outside of the method, the instance variables are retained throughout the lifetime of the object, and can only be modified by the methods of the object.

They can be of any valid data type, and can also be objects. They should always be given the modifier **private** to enforce the encapsulation rule of OOP.

The following is an example of the definition of object variables for a class named **Loan:**

```
class Loan
{
// *  Instance Variables
// **********************

private String          Customer;
private double          CurrentBalance;
private double          Rate;
private GregorianCalendar DateLastTran;

…… methods etc. would be defined here
}
```

Constructors

If an object is to be created from a class, it needs one or more special methods known as **constructors**.

The purpose of a constructor is to place initial values in the instance variables. The constructor method will usually require parameters, in order to be able to get the initial values from the process which created the object.

The constructor method must have the same name as the class.

Many classes provide more than one constructor, in order to allow flexibility in the number and type of parameters supplied. For example, if one of the parameters is a date, two constructors can be provided, one of which requires the date as a parameter, and one of which defaults to today's date.

Again, if one of the instance variables is an integer, two constructors may be provided. The first will expect this parameter in the form of an integer. The second will expect the parameter in the form of a string, and will convert it to an integer.

Since all constructors have the same name, being that of the class, the run time environment decides which of the constructors to invoke, depending on the number and type of arguments supplied.

The following example shows the constructors for the Loan class described in the previous example. The first constructor expects the amount of the loan and the rate as doubles; the second expects them as strings, and converts them to doubles.

```
public Loan(String Cust, double Amt, double Rt)
{
Customer = Cust;
CurrentBalance = Amt;
Rate = Rt / 100;
DateLastTran=new GregorianCalendar();
}

public Loan(String Cust, String Amt, String Rt)
{
Customer = Cust;
CurrentBalance = Double.parseDouble(Amt);
Rate = Double.parseDouble(Rt) / 100;
DateLastTran=new GregorianCalendar();
}
```

Methods

Public methods in a class can be used by the process which created the object in order to initiate an action. Typically, these will include accessor methods, which return data stored in the object, and mutator methods, which change the values of data stored within the class.

The **return** *expression* statement is used to return a value to the process which invoked the method, eg
return Balance;
This should be the last logical statement within the method.

Additionally, a class may have its own private methods which may be called from within other methods of the class, but may not be called from outside the class.

It is also possible to include a method named **finalize()** which will be executed once the object is no longer required. This may be used to release system resources, such as files, used by the object. However, it is better practice to release resources as soon as they are no longer needed by the object.

Example

The full coding for the sample **Loan** class, together with a public class **loancalc** which tests it, is shown below.

As well as the two constructor methods, **Loan** contains the methods:
MthlyInterest : Calculates monthly interest on the loan, adds it to the balance, and returns the amount of interest calculated
Payment: Retrieves a payment amount as a parameter, and subtracts this from the balance
getBalance: Returns the balance

```
import java.util.*;
//  ***********************************
//  public class to test the Loan class
//  ***********************************

public class loancalc
{
  public static void main(String[] args)
  {
  double Interest, Balance;
  Loan CarLoan = new Loan("Fred", "120000", "25");
  Interest=CarLoan.MthlyInterest();
  CarLoan.Payment(3000);
  Balance = CarLoan.getBalance();
  System.out.println(+ Interest + " " + Balance);

  }
}

//  *****************************************
//   Definitiion of the Loan class
//  *****************************************
class Loan
{
// *  Instance Variables
// **********************

private String          Customer;
private double          CurrentBalance;
private double          Rate;
private GregorianCalendar DateLastTran;

//  Constructors
//  ************

public Loan(String Cust, double Amt, double Rt)
{
Customer = Cust;
CurrentBalance = Amt;
Rate = Rt / 100;
DateLastTran=new GregorianCalendar();
}

public Loan(String Cust, String Amt, String Rt)
{
Customer = Cust;
CurrentBalance = Double.parseDouble(Amt);
Rate = Double.parseDouble(Rt) / 100;
DateLastTran=new GregorianCalendar();
}

//  Other public methods
//  ********************
public double MthlyInterest()
{
double Interest;
```

```
Interest = CurrentBalance * Rate / 12;
CurrentBalance = CurrentBalance + Interest;
return Interest;
}

public void Payment(double Pay)
{
CurrentBalance = CurrentBalance - Pay;
DateLastTran=new GregorianCalendar();
}

public double getBalance()
{
return CurrentBalance;
}

}
```

Static methods & fields

Some performance gain is achieved by defining variables (such as constants) and methods which do not require the presence of an object with the modifier **static**. This means that there is one instance per class, rather than one instance per object, of these items.

Initialisation blocks

Any initialisation which does not require input from the process creating the object can be done in an initialisation block, if the programmer wishes. This is just a block at the front of the class enclosed in braces and containing the initialisation statements:

class ***classname {***
{ initialisation statements}
instance variables
methods
}

Please complete Exercise 10 from the exercise pack

SUBCLASSES

Purpose

In the second chapter, we looked at **inheritance** as being one of the principles of OOP. Inheritance is said to implement an is-a… relationship between objects. For example, in a building society an investor is a customer, and so is a borrower. A class **Customer** could be used as a base class, or superclass, to deal with both types of customer.

Each of them will, however, have additional data and methods unique to the customer type. **Subclasses** could therefore be built from **Customer** to cater for investors and borrowers. Since they will both need to use the original data and methods from **Customer**, they will inherit these. Additional data and methods will be defined in the subclasses for the specific needs of each type. They are therefore said to **extend** the superclass.

The class header

The class header for a subclass **HouseLoan** which is to extend the **Loan** class in the previous example would be coded:

class HouseLoan extends Loan {……}

Instance Variables

Since additional data will be stored, for example the serial number of the title deeds, these must be defined at the front of the subclass. The instance variables defined in the original class will be inherited, and need not be defined.

The constructors

Constructors must be provided which will initialise these new instance variables. These constructors will usually call the constructors from the superclass once they have initialised their own variables; this is done by the statement **super(***parameterlist***).** For example, the **HouseLoan** class may begin like this:

```
class HouseLoan extends Loan
{
private String TitleDeeds;
public HouseLoan(String Cust, double Amt, double Rt, String Deeds)
{ super(Cust,  Amt,  Rt);
TitleDeeds=Deeds;

}
public HouseLoan(String Cust, String Amt, String Rt, String Deeds)
{ super(Cust,  Amt,  Rt);
TitleDeeds=Deeds;

}
…..}
```

Methods

Methods will be inherited from the superclass, so that it is not necessary to redefine them in the subclass. However, in some circumstances subclasses may need these methods to work in a different way. In that case, they can be written in the subclass using the same name as the method in the superclass; the new method in the subclass will override or hide the method in the superclass.

Additional methods may be needed in the subclass. They can be coded as normal.

Access to fields

The methods defined in the subclass will not have access to the instance variables stored in the superclass. They can only be accessed by invoking the accessor and mutator methods of the superclass.

Multiple inheritance

Some OOP languages allow multiple inheritance, where a subclass can inherit from several superclasses. This is not allowed in Java, as it has been found to cause problems due to ambiguity.

Levels of inheritance

Inheritance can occur over as many levels as required, for example class **b** can be written as a subclass of class **a,** and inherit from it. Class **c** can be written as a subclass of **b**, and inherit all data and methods from **b**, including those which it inherited from **a**. Class **d** could then be written as a subclass of **c** and so on.

The Object superclass

In Java, all objects implicitly extend the superclass **Object**, and inherit its data and methods.

Assigning a subclass object to a superclass

There are often cases where a particular programming tasks does not need to see the differences between objects representing a subclass and those representing a superclass. For example, we may have a subclass **manager** which extends the class **employee**. The program which prints payslips will not need to work with the additional information stored in **manager**, and can process managers in the same way as other employees.

If an object **staff** had been created from **employee**, and an object **boss** had been created from **manager**, it would be acceptable for the payslip program to include a statement **staff=boss;** and continue to process the boss as an ordinary staff member.

However, it would not be acceptable to do this in reverse, i.e. **boss=staff**, since **boss** contains additional information.

Casting and instanceof

If it was necessary to assign **staff** to **boss**, this can be done by casting, as follows: **boss=(manager) staff**. This will allow the assignment to take place. However, if the current **staff** object was not originally a **manager**, a run-time error can occur.

It is therefore a good idea, before assigning **staff** to **boss**, to check that the current **staff** object is actually a manager, as follow:

```
if (staff instanceof manager)
  boss=(manager)staff;
```

The **instanceof** operator checks the internal contents of the object to ascertain its original type.

Final classes and methods

Some classes should not be extended. If this is the case, the modifier **final** can be included in the class header, and the compiler will not allow it to be extended.

It is also possible that the writer of a class may not want some of the methods to be overridden by methods of the same name in a subclass. The modifier **final** can then be included in the method header.

There is a positive impact on performance when classes and/or methods are coded as **final**.

Please complete Exercise 11 from the exercise pack

ABSTRACT CLASSES

Purpose

When there are multiple levels of inheritance, the original superclass can sometimes become little more than a framework around which other classes are written. For example, the superclass **Object** simply defines what sort of methods an object should have, but at that level, it is impossible to define exactly what the methods will do.

It is possible to write a class as being **abstract**, which means that it cannot be implemented as an object, but can only be extended as the basis for other classes. An abstract class may have some methods fully defined, whereas others will just exist to show that any subclasses extending this class will need to implement these methods. In that case, the method is said to be abstract as well as the class.

If a class contains any abstract methods, the class must also be abstract.

The class header

The keyword **abstract** included in the class header indicates that the class is abstract, for example:

abstract class Customer

Abstract methods

An abstract method contains a method header only, as follows:

public abstract void UpdateBalance();

This acts as a placeholder for the method, and forces all subclasses of **Customer** to implement their own versions of **UpdateBalance()**.

Non-abstract methods

An abstract class can contain many methods which are not abstract, as well as abstract methods.

Example

The **Number** class in the java.lang library is a perfect example of an abstract class. It is the framework around which classes such as Integer, Double, Float, BigDecimal etc. are built. It is worth having a look at Number.java in the directory src\java\lang directory to see a working example of an abstract class. If the library source code has not been loaded down in your installation, the next chapter contains a description of how to do so.

INTERFACES

Background

In the real world, there are many examples of interfaces which allow different components to fit together. For example, an electric drill made by one manufacturer may be able to make use of attachments supplied by another manufacturer, because the interface between the drill and the attachments is the same in both cases. Interfaces in Java therefore define what must be present for one component to successfully work with another.

Purpose

Java does not allow multiple inheritance, so that a class cannot be built from more than one superclass. However, it may be necessary to ensure that a given group of classes conform to certain standards. For example, Java provides a class that can be used to sort an array of objects. This needs to be able to work with many different types of objects. It is not possible to have them all descend from the same superclass, since they may need to extend widely different classes.

An interface can be used to ensure that all classes which are to be dealt with in a similar fashion (eg array sorting) have the basic framework in place.

Coding an interface

An interface is coded as follows:
public interface *Interfacename*
{*method header ...}*

Implementing an interface

An interface is therefore simply a named collection defining the names and data types of methods which must be defined in any class implementing the interface. A class may implement one or more interfaces, in addition to optionally extending another class.

The class header would be coded as follows:
class *classname* [**extends** *classname*] **implements** ***Interfacename***
[,*Interfacename ...*]

The class must define actual methods for the placeholder methods defined in the interface.

Using instanceof with interfaces

It is possible to use the **instanceof** operator on an object to check whether it implements a particular interface, i.e.
if (*objectname* **instanceof** *interfacename***) ……**

Comparable interface

The Comparable interface is defined in java.lang, and is used to check whether array sorting can be carried out on an object.

The Comparable interface contains a method **compareTo** which requires an object as a parameter, and returns an integer.

Any class implementing **Comparable** must include an actual method which conforms to these standards.

This can then be used by the array sort to compare one object to another.

INNER CLASSES

Overview

Inner classes are classes which are defined inside other classes, and have the advantage that they can access the private variables of the outer class.

They are particularly useful with event-driven objects such as GUI (Graphic User Interfaces). GUI's are dealt with in a later chapter, and the usefulness of inner classes will become apparent then. For the meantime, we will just look at how to code an inner class.

Coding an inner class

This is simply a case of coding one class inside the other, as follows:

```
class x
{
  public x {…..statements}
  public void a { ….statements}

  class y
  {
   public y { …statements}
   public void b { …statements}
   }
}
```

y then becomes an inner class of x.

Review Questions – How Would You…

1.	Create an object from a class named **Employee**, which requires the employee number and name as parameters?
2.	Code the instance variables and constructor of the class **Employee** referred to in the last question?
3.	Code the class header of a class **manager** which is to be a subclass of **Employee?**
4.	Override a method from a superclass when coding a subclass?
5.	Ensure that a method cannot be overridden in any subclass of the subclass which you are writing?

THE JAVA DEVELOPMENT KIT

Learning Module Objectives

When you have completed this learning module you will have seen how to:

- Find and organise files in the Java development environment
- Use switches with the Java compiler javac
- Use switches with the Java runtime environment java
- Use the archive facility jar
- Use the documentation utility javadoc

FILE ORGANISATION

Introduction

The Java installation files will form a predefined directory structure underneath the directory in which Java was installed. It is up to individual organisations to implement a directory structure suitable for storing the source and object files of their own applications programs.

The java installation files

After installation, the Java directory will contain several subdirectries. These include:

bin	This contains the executable files, and the path must be set to this directory in order to run the Java run-time, compiler and utilities.
lib	This contains the library classes which are needed for program development.
src	This contains the source code of the Java libraries in a series of subdirectories. If this is empty, or does not exist, the source code can be loaded down from the archive file src.jar using the Java archive utility jar.
doc	This contains the Java documentation. If no documentation exists, documentation can be built from the Java library source code using the javadoc utility.

User files

Program source files are named with an extension of .java, and compiled files have the extension .class. The compiler will create a separate .class file for each class defined within a source file, so that a single source can produce several objects. All the class files required for an application should reside in the same directory; this is referred to as a package. It should be noted that class names must be unique within a package, otherwise the compiled versions will overwrite each other.

JAVA COMPILER AND RUNTIME ENVIRONMENT

javac

javac is the java compiler. Unless otherwise specified, it will expect to find source files in the current directory, and will place compiled files in the same directory. Classes will be searched for in the current directory and in the java library directory and any directories listed in the classpath environment variable. The behaviour of the compiler can be altered by the uses of switches on the command line. Some of the more useful ones include:

-g	Generate debugging information
-O	Optimise for performance
-classpath	Followed by a path name to a directory, this gives an alternative directory to search for user-defined class files
-d	Followed by the pathname to a directory, this specifies the directory into which compiled files will be placed.

A full list of the compiler switches may be obtained by typing **javac** at the command prompt with no arguments.

Example

javac –O –classpath \jdk1.2\dev\class loancalc.java

java

The program java is the byte code runner which executes compiled program code. By default, all classes referred to are expected to be found in the current directory as well as any directories listed in the classpath environment variable. Classes may also be stored in zip or jar files for convenience and space saving. A few switches are available, including:

-version	This displays the version of Java which is running
-cp	This switch is followed by a list of pathnames to directories and/or zip and/or jar files, separated by semi-colons, which contain the class files to be run. This instructs the run-time environment to look for classes in these areas. If this is set, it should include the current directory if any classes are located there.
-D	Sets a system property, which may be tested in programs. It is entered as –D*propertyname=value*

JAVA ARCHIVE FACILITY

Advantages of archive files

Java archive files are a convenient way to keep all the files which make up a package together. Primarily, their purpose is to contain class files, but they can contain other files that are needed by the package, such as sound and graphics files. The runtime environment can run programs directly from the archive files, so there is no need to download them.

The advantages of this method include:

- It is a convenient medium for transporting files, since only one file need be copied or downloaded through the network
- It provides better control, since individual files cannot get lost
- It simplifies the directory structure
- Since archive files can be compressed, it can save space
- The utility **javakey** provides a method of adding a signature to the archive file, so that changes by unauthorised parties can be detected.

The archive utility jar

jar stands for Java Archive, and is the name of the Java archiving utility program. It is also the file extension which is placed on the archive files, so that they are known as jar files. Jar can be used both to archive and to download from an archive.

The syntax for running jar from the command prompt is:
jar *options* [jar-file] [-C dir] files ...

Some of the options available are listed below. On the command line, they should be listed together with no spaces between them.

The optional –C dir is used to indicate a change to another directory.

Files can be listed separately or using wildcards.

Options include:

c	create a new archive file
t	list table of contents for an archive file
x	extract files from an archive
u	update existing archive
f	specify archive file name
0	Do not compress

Examples

```
jar cf my.jar *.class
jar xf my.jar *.*
jar t my.jar
```

JAVADOC

Purpose

The Javadoc utility is used to create documentation in HTML format from a Java source file. It is useful both for documenting applications which have been developed, and obtaining documentation from the source files of library classes.

Action

Javadoc will build documentation based on:

- public classes
- methods, other than private methods
- interfaces
- variables, other than private variables
- program comments which use the /** … */ syntax.

This documentation will be in HTML format, and can be accessed by a browser. It will be fully indexed.

Program comments

These should include a description of the class at the front of each class, and documentation relating to each public method and variable, placed before the definition of the method or variable. HTML modifiers can be included in the text.

Running Javadoc

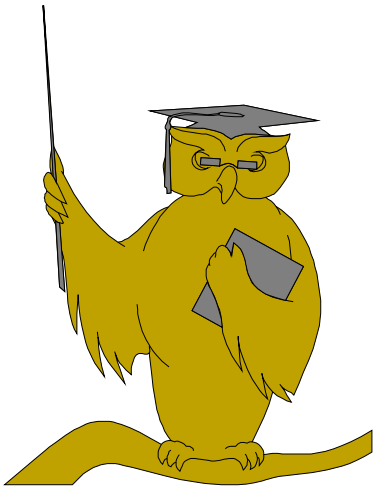
From the command line, enter:
javadoc –d *directoryname sourcefilelist*

directoryname is the directory in which the documentation will be created
sourcefilelist is a list of source files which are to be documented.

Accessing the documentation

A browser such as Explorer can be used to view the documentation. Opening the file index.html in the directory in which the documentation was created will give access to the documentation.

Review Questions – How Would You…



1.	Include classes from another directory when compiling a Java program?
2.	Find out which version of the Java run-time environment is installed?
3.	Archive all the class files in the current directory into a jar file?
4.	Extract all the source files archived in src.jar?
5.	Include comments in a source program suitable for producing documentation with Javadoc?

Please complete Exercise 12 from the exercise pack

EXCEPTION HANDLING

Learning Module Objectives

When you have completed this learning module you will have seen how to:

- Check for exceptions
- Write an exception handler
- Propagate an exception to the next level

OVERVIEW

Run-time errors

There are many reasons why programs can have run time errors.

In many cases, these could have been avoided if the programmer built proper checks into the program in the first place.

For example, bad input typed in by the user has often been known to cause such errors as Index Out of Bounds. If the program checked the user's input to ensure that it is within an acceptable range before attempting to use it, this need not happen.

In other cases, the error can be caused by circumstances which cannot be avoided by the programmer. A file needed by the program may have been accidentally deleted by a user, or the system may have run out of memory.

In all cases of run-time errors, it is important to either deal with the error if possible, or shut the program down with a meaningful error message and minimum loss of data if the error cannot be dealt with.

Exception handling in Java

If a method cannot complete normally, it "throws" an exception. The exception is an object containing details of the problem which occurred. The exception can then be "caught" by some other part of the program, known as an exception handler. This exception handler can do one of several things, for example:

- Fix the problem, if possible
- Release resources, for example close files etc, and propagate the error to the process which called the method in which the exception handler is located, or propagate it back to the Java runtime environment so that the program is closed
- Display a meaningful error message, and allow program execution to continue.

The programmer must decide what type of action is appropriate for each type of exception.

If an exception is not "caught" within the method which threw it, it is propagated back to the method which called it. If it is not caught there, it is propagated back up the hierarchy until an exception handler is found, or, if none is provided, back to the Java runtime environment.

Exception classes

Java provides a hierarchy of exception classes, all of which originate from the superclass **Throwable**. Some of these can be thrown by the built-in Java classes. A programmer can also throw any of these exceptions from his own methods, in which case he must decide which exception class is appropriate. Programmers can also define their own exception classes by extending any of the existing classes.

Some examples of the predefined exception classes are listed below. A full list is available in the Java documentation.

ClassNotFoundException
IllegalAccessException
IOException
EOFException
FileNotFoundException
MalformedURLException
ProtocolException
SocketException
UnknownHostException
UnknownServiceException

These exception classes include data, such as an error message which can be set by the programmer, and several useful methods such as:
printStackTrace(), which prints a trace of where the error occurred, and the methods through which the program arrived at that point
getMessage(), which retrieves the error message contained in the exception object.

INCLUDING EXCEPTION HANDLING

Throwing an exception

If an error occurs which cannot conveniently be handled at that point in the program, an exception can be thrown as follows:

if(*condition*)
  throw new *exceptiontype()*

exceptiontype is the name of an exception class. It could be one defined by Java, including those listed in the previous section, or it could be the name of an exception class defined by the programmer.

This will terminate the current method, and search for an exception handler which deals with that exception class, or a superclass of that exception class.

Optionally, an error message may be stored within the exception object, as follows:
if(*condition*)
  throw new *exceptiontype("Error message Text")*

Propagating an exception further up the hierarchy.

Many of the Java library classes can throw exceptions in this way.
If it is known that a method can throw an exception which will not be handled within the method, the compiler must be informed by adding the clause **throws** *exceptiontype* to the end of the method header, where *exceptiontype* is the class of the exception object which is thrown.

For example, a method which processes a file is always prone to exceptions, and the method header may be coded:

public void GetCustInfo(String Custno) throws IOException

Many of the methods in library classes do this, especially those that deal with file handling.

Catching an exception

The method which called this method must either catch the error or propagate it to the next level up. If it is to be propagated, then the calling method must also specify **throws IOException** in its header.
This is done by enclosing the statements which may cause an error in a **try … catch** block, as follows:

```
try
{
statements which may cause an error ….
}
catch (exceptionclass objectname)
{
statements which deal with the error
}
```

The exception class in the catch block must be either the one thrown by the statements in the try block, or a superclass of it.

The statements in the catch block can use the methods associated with the object caught in order to retrieve information from it, for example:

```
catch(IOException badFile)
{System.out.println(badFile.getMessage());
badFile.printStackTrace;
etc
}
```

Example

This example shows the Calculator program from a previous example modified to include exception handling, so that if an invalid number is entered on the command line, a user-friendly error message is displayed.

```
// ********************************************************
// * This program will carry out a calculation
// * entered on the command line. The calculation
// * should be entered as a number, followed by a
// * space, followed by an operand, followed by a
// * space and a number. Valid operands are +,-,x
// * and /.
// ********************************************************
public class calc
{
 public static void main(String[] args)
 {
  double Num1=0, Num2=0, Ans=0;
  char Op;
// ********************************************************
// Number format checking has now been included here
// ********************************************************
  try
   {
   Num1=Integer.parseInt(args[0]); //First argument is a number
    }
    catch (NumberFormatException ex)
    {
    System.out.println
    ("First No. was not entered in the correct format – please try again");
    System.exit(0);
    }

   Op=args[1].charAt(0);              //First character of second
                                      //argument is the operand
   try
   {
   Num2=Integer.parseInt(args[2]); //Third argument is a number
    }
    catch (NumberFormatException ex)
    {
    System.out.println
    ("Second No. was not entered in the correct format – please try again");
    System.exit(0);
    }
```

```
// ********************************************************
// Decide on the correct calculation depending on
// the operand entered
// ********************************************************

    switch(Op)
    {  case '+':
           Ans=Num1+Num2;
           break;
       case '-':
           Ans=Num1-Num2;
           break;
       case 'x':
           Ans=Num1*Num2;
           break;
       case '/':
           Ans=Num1/Num2;
           break;
       default:
           System.out.println
           ("Can't do that - I'm only a cheap calculator");
           break;
    }
    System.out.println("Answer is " + Ans);
  }
}
```

Including final statements with a try…catch block

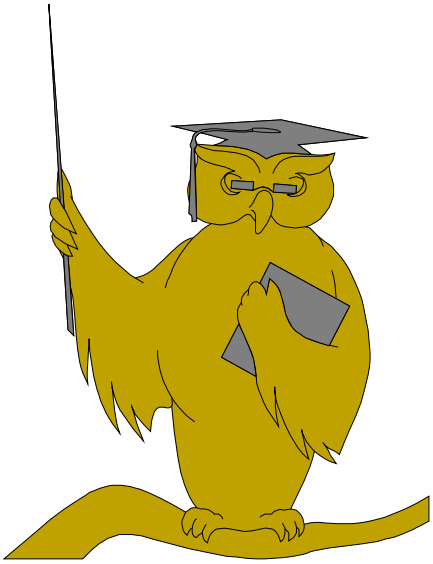
When an error is detected in the **try** block, control is passed directly to the **catch** block, and all later statements in the **try** block are ignored.

However, there are often statements (such as closing files etc) which should be carried out whether or not an error occurred. These can be included in a **finally** block, which is executed at the end, either after the **try** block completed successfully, or after the final statement in the **catch** block.

This would be laid out as follows:

```
try
{
statements which may cause an error ….
}
catch (exceptionclass objectname)
{
statements which deal with the error
}
finally
{
statements which should be carried out whether or not an error occurred
}
```

Review Questions



1.	Give two examples of runtime errors which could be avoided by the programmer
2.	Give two examples of runtime errors which cannot be avoided by the programmer
3.	How would you indicate to the compiler that an error may be thrown within a method but will not be handled by that method?
4.	How would you throw an error?
5.	In an exception handler, how can the error message be retrieved?

Please complete Exercise 13 from the exercise pack

THE GRAPHIC USER INTERFACE CLASSES

Learning Module Objectives

When you have completed this learning module you will have :

- An overview of the process of defining a Graphic User Interface (GUI) using the Swing package
- An understanding of the classes used in creating a GUI, and their hierarchical structure
- A n understanding of event handling
- Seen how to use a selection of the Swing components to build a GUI
- An understanding of Layout Managers

OVERVIEW

Graphic User Interfaces

Users have come to expect an interface that allows them to work with familiar screen objects, such as menus, text boxes, command buttons and scrollbars, to communicate with any applications program. The Java GUI components make it possible to create this kind of interface.

Releases prior to Java 1.2 relied primarily on the AWT (Abstract Window Toolkit) classes, which used the GUI objects of the underlying operating system (e.g. Windows, Unix or Mac). This had the advantage that users would be presented with GUI objects which looked and behaved in a familiar manner. The disadvantage in practice was that applications ported to a different platform were prone to bugs caused by differences in the underlying components.

To get around this problem, release 1.2 introduced the Swing library. The Swing components use an in-built GUI, and are no longer platform-dependent. They may be customised to retain the look of a particular platform, or may use the default "look and feel", which is known as Metal. In order to allow applications developed with Swing to run on older runtime environments, a plug-in is downloadable from the internet.

Several new components, such as tables and toolbars, were introduced with Java 1.2.

Understanding GUI's

Since the AWT and Swing libraries contain several hundred classes, the creation of a GUI can appear confusing to beginners. In order to put things in perspective, programmers need to have a good understanding of the topics listed below.

The hierarchy of the GUI components

GUI components derive from a common class Component which is a subclass of object, and form a hierarchy of classes for individual purposes. Since each class inherits data and methods from the class above it, it is necessary to understand this hierarchy in order to know what classes and methods will be available with any particular component.

Event handling

GUI applications are event driven, rather than following sequential procedures. The user initiates an event by pressing a key on the keyboard, or moving or clicking the mouse. This is detected by the operating system, and placed in an event queue.

The Java run-time environment processes the event queue, and searches for an appropriate method of an object in the applications program which has been registered to deal with that event. Control is then passed to that method, and returned to the run-time environment on completion of the method.

Programmers need to be familiar with the type of events which may occur, and with registering and writing event handlers.

Individual components

To work with a component, it is necessary to know many things about it, for example, what events it can trigger, what data and methods it has available, and how it will look and behave on the screen.

Layout managers

It is not recommended that programmers specify fixed positions on the screen for the objects displayed on it, since users will often resize a window, with peculiar results. Again, not all users work on identical equipment, and a program which looks good at one screen resolution may not do so at another.

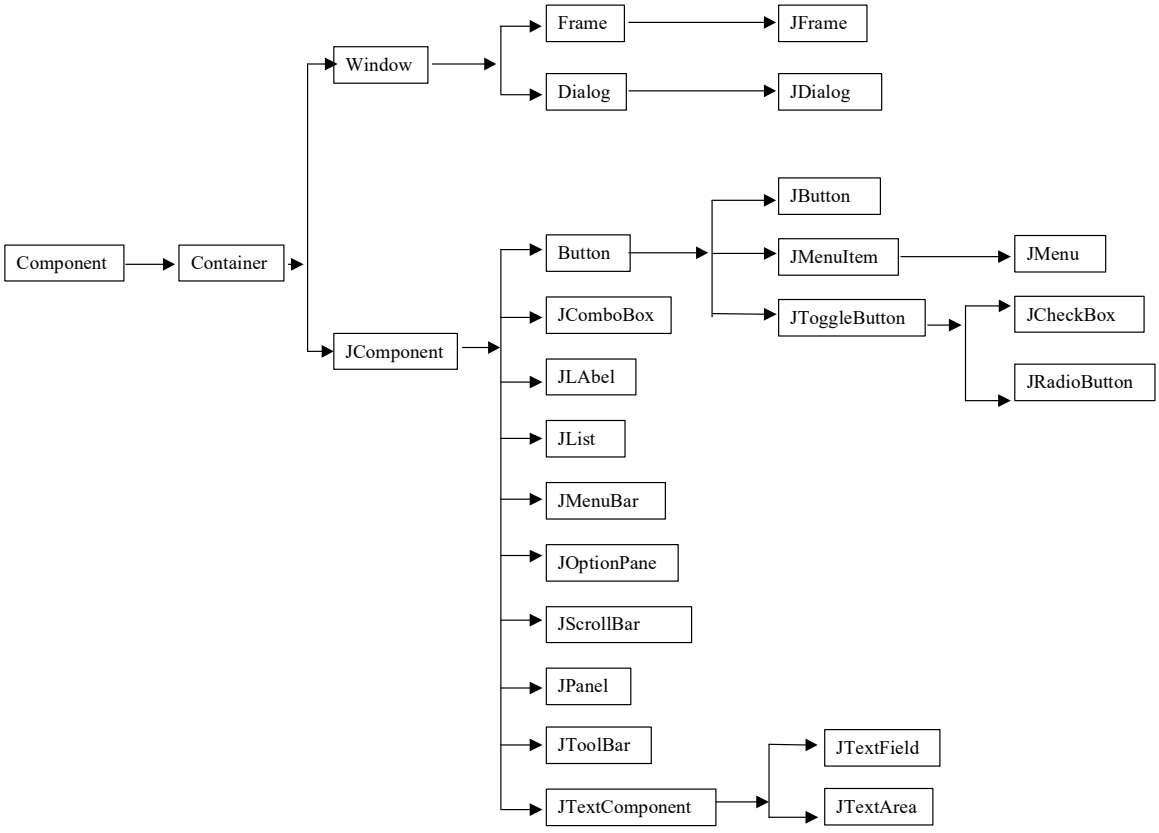
It is recommended that programmers use one of the layout managers provided, which will adjust the position and size of the components to fit the available screen space.

Programmers need to be familiar with the layout managers, and be able to select a manager which will produce the desired effect.

Getting more information

It is not possible to cover all the components available on a course of this duration. A few components will be covered in order to give a good understanding of GUI's and how to use them. Programmers can obtain more information from the Java documentation.

GUI CLASS HIERARCHY



The hierarchy

As can be seen from the above diagram, all GUI classes are subclasses of Component, which is defined in the AWT library, and is itself a subclass of class Object. Its methods and data are inherited by class Container, which defines further methods and data, which are inherited by Window and JComponent.

The Window branch contains subclasses that define two different types of screen window: Frame, which is a general purpose window, and Dialog which is a pop-up dialog box. Frame and Dialog are AWT components; further functionality is given to them in the Swing classes JFrame and JDialog.

JComponent, the other branch of the hierarchy, is a Swing class, and holds general functionality for objects which can be displayed in Frames or Dialog boxes. Under this are defined the individual GUI components, such as buttons, scrollbars and text boxes.

Before looking in detail at frames, dialog boxes and GUI components, it is worth taking a brief look at the classes which form their common origins, since the data and methods held in these classes is available in all of their subclasses.

Some of the methods included in these classes are listed to give an idea of how these fit into the general hierarchy. The actual usage of these methods will become clearer when the individual component classes are documented in a later section, and programming examples are provided.

Component

Data Stored: Screen co-ordinates of the component
Methods: There are a very large number of methods; a few sample ones are

void add*eventtype*Listener(*object*)
There are several methods of this type, for example **addFocusListener**. Each of them registers a 'listener object' for different types of events which may affect this component.

boolean isShowing()
Checks whether the component is currently visible

repaint()
Redraws the component's image on the screen

repaint(x,y,width,height)
Repaints a section of the component defined by the co-ordinates. Note: all screen co-ordinates in Java are given in pixels.

requestFocus()
Requests that focus be given to this component. When an object is focused, it is able to receive input from the keyboard, and is usually highlighted in some way.

setBackground(*colour*)
Sets the background colour of the component.

setBounds(*x,y,width,height*)
Moves and resizes a component

setFont(*font*)
Sets the font, or printing style, of the object.
setForeground(*colour*)
Sets the foreground colour of the object

show()
Causes the component to appear on the screen.

Although this is by no means a complete list, it should give an idea of the functionality of the Component class. All of these methods will be inherited, and possibly redefined, in the subclasses below Component.

Container

Containers maintain a list of GUI components which have been added to them. For example, a frame may contain a menu, some text boxes, a few command buttons etc. Methods include:

void add(*GUIcomponent*)
or
void add(*GUIcomponent, position*)
Adds a GUI component into a container, for example, adding a menu to a frame

remove(*GUI component*)
Removes a GUI component from a container

removeAll()
Removes all components from the container.

Window

This is the class from which two types of window, Frame and Dialog, are derived. Methods include:

toBack()
Moves this window behind any other windows on the screen.

toFront()
Moves this window to the front.

addWindowListener(*object*)
Registers an object as a listener for events affecting this window.

JComponent

This is the subclass from which all individual GUI components, other than windows, are derived. Methods include:

boolean hasFocus()
Returns true if the component currently has the focus.

reshape(x, y, width,height)
Moves and resizes the component

setAlignmentX(x)
Sets the alignment on the X axis

setAlignmentY(y)
Sets the alignment on the Y axis.

setEnabled(boolean)
Sets whether this object is enabled i.e. can receive input from the user

setMaximumSize(object of class Dimension)
Sets the maximum size to which this component can grow on resizing of the window

setMinimumSize(object of class Dimension)
Sets the minimum size for this component

setNextFocusableComponent(component)
Names a component which will receive focus when the user tabs from this component

setOpaque(boolean)
If the argument evaluates to **true**, this causes the background of this component to be filled with the colour of the container.

setPreferredSize(object of class Dimension)
Sets the optimum size for this component

setToolTipText(String)
Sets a text string which will be displayed if the user pauses the mouse over this component. This is usually a short description of the component.

setVisible(boolean)
If the argument evaluates to **true**, this will make the component visible; otherwise, it will hide the component.

The lower levels of the hierarchy

These represent individual components, many of which will be covered in detail in later sections.

Additional GUI classes

There are many GUI classes which do not fit into this hierarchy, but which are used in conjunction with screen components. These are introduced below. Again, this is an overview only, designed to give an idea of what is available. Actual examples of using these classes are given in a later section.

Event objects

When the JVM recognises the occurrence of an event which has a registered listener, it will create an Event object containing the details of the event, and pass this as a parameter to the listener method. This is discussed in more detail in the next section.

Clipboard

This allows data to be transferred in order to implement cut/copy/paste operations.

Useful methods include:
getContents(object)
Retrieves a transferrable object from the clipboard.

setContents(object,owner)
Places a transferrable object onto the clipboard, and sets the owner of the clipboard.

Color

This defines colour constants for the standard colours. The constants are: black, blue, cyan, darkGray, gray, green, lightGrey, magenta, orange, pink, red, white and yellow.

Additionally, a Color object may be constructed using the colour mixture as a parameter as follows:
Color(red,green,blue)
where **red, green** and **blue** may be either float variables with values between 0 and 1, or integer variables with values in the range 0 to 255.

Cursor

This can be used to create an object which holds a bitmap image for a cursor.

It contains the following constants for defining standard cursors: CROSSHAIR_CURSOR, CUSTOM_CURSOR, DEFAULT_CURSOR, HAND_CURSOR, MOVE_CURSOR, TEXT_CURSOR and WAIT_CURSOR. there are also constants for various types of resizing cursors.

Customised cursors can be constructed using the Toolkit class.

Dimension

This is used to store a width and height which can be applied to other objects. It is constructed as follows:
Dimension(width, height)

Font

Font objects are used to contain details of a font which can be applied to objects displaying text. A font object is created using the following parameters:
Font(name,size,style)

name can indicate either:
- one of the logical font names known to Java i.e. Dialog, DialogInput, Monospaced, Serif, SansSerif, or Symbol
  At run time, this will be mapped to an actual font installed on the current machine.
- An actual font name. The drawback with this is that not all machines will have all fonts loaded. However, the GraphicsEnvironment class supplies two methods getAllFonts and getAvailableFontFamilyNames which can be used to find out what fonts are available on the run-time machine.

style This should be one of the predefined constants PLAIN, BOLD OR ITALIC. If both bold and italic are required, use a logical or to give a bitwise union i.e. BOLD|ITALIC.

size Indicates the point size required

FontMetrics

The FontMetrics class is used to calculate actual measurements of a given font on the current screen. To create a FontMetrics object, the constructor is:
FontMetrics(font)

Perhaps the most useful method in this class is stringWidth(String), which calculates the exact width in pixels of the string supplied in this font.

Graphics

Graphics objects are used to store details of a graphics context, such as the current font, colours etc. The Graphics class includes methods for drawing lines and shapes, images and text strings.

Since Graphics is an abstract class, it is not possible to create an object of this type. However, it is the superclass from which various other components define graphic contexts, and can be used as the data type when passing parameters to and from methods which require a graphics context. The **getGraphics** method included in component classes will get the current graphics context for that component.

Its methods include:

setColor(Color)
Sets the colour for subsequent graphics operations

setFont(Font)
Sets the font for subsequent drawing of text strings

drawLine(x1,y1,x2,y2)
Draws a line from the point specified by the co-ordinates (x1,y1) to the point specified by (x2,y2)

fillRect(x, y, width, height)
This will fill a rectangle of the given width and height with the top left hand corner positioned at co-ordinates (x,y) with the current colour.

drawRect(x,y,width,height)
Draws the outline of a rectangle

drawRoundRect(x,y, width, height, arcWidth, arcHeight)
This draws a rectangle with rounded corners. The dimensions of the arcs at the four corners are specified by arcWidth and arcHeight

fillRoundRect(x, y, width, height, arcWidth, arcHeight)
Fills the area bounded by the specified rounded rectangle with the current colour.

draw3DRect(x, y, width, height, raised)
This draws a rectangle with 3D effect. The boolean raised parameter specifies whether the rectangle should appear raised or sunk.

fill3DRect(x, y, width, height, raised)
Fills the 3D rectangle with the current colour

drawOval(x, y, width, height)
Draws an oval with position and dimensions defined by the parameters.

fillOval(x, y, width, height)
Fills the specified oval.

drawArc(x, y, width, height, startAngle, arcAngle)
x and y give the starting position for the drawing. width and height specify the width and height of a rectangle of a rectange which would contain the circle or ellipse of which the arc forms a part. startAngle specifies the angle from the (x,y) position at which the first part of the arc will be drawn, and arcAngle specifies the angle enclosed by the arc. Angle (0) is the horizontal position; positive angles are taken in a anticlockwise direction from there, and negative angles in a clockwise direction. Angles are measured in radians. To convert an angle in degrees to radians, divide by 360 and multiply by 2π.

fillArc(x, y, width, height, startAngle, arcAngle)
Fills the specified arc.

drawPolyline([] xPoints, [] yPoints, nPoints)
Draws a line passing through a series of co-ordinates. The corners of the figure are specified by arrays of x and y co-ordinates. nPoints specifies the number of x and y co-ordinates supplied. If the first pair of (x,y) co-ordinates is the same as the last, the figure will be closed.

drawPolygon([] xPoints, [] yPoints, nPoints)
Similar to drawPolyLine, but automatically closes the figure to form a polygon.

fillPolygon([] xPoints, [] yPoints, nPoints)
Fills the specified polygon.

drawString(String, x, y)
Draws the supplied text string at the position (x,y) using the current font and colour.

drawImage(image, x, y, width,height,ImageObserver)
Draws a graphic image defined by the image parameter, which must be an object of class Image, at the position specified by x and y, scaled to fit into a rectangle defined by width and height. ImageObserver indicates an object which must implement the ImageObserver interface and which will receive notification at different stages of the production of the image. This is often set to null.

dispose()
This releases the graphics context and any resources associated with it.

Graphics Environment

This can be used to obtain descriptions of the graphics devices and fonts available on the current platform.

GraphicsEnvironment is an abstract class, and objects cannot be created directly from it. To create an object containing details of the current environment, a static method getLocalGraphicsEnvironment can be used. as follows:
GraphicsEnvironment *objectname* =
GraphicsEnvironment.getLocalGraphicsEnvironment()
This object can then be used to obtain information about the environment.

Useful methods include:

Font [] getAllFonts()
Returns an array of class Font containing all the fonts available on the platform.

String[] getAvailableFontFamilyNames()
Returns an array of strings containing the names of all font families loaded on the current platform.

GraphicsDevice getDefaultScreenDevice()
This returns an object of type GraphicsDevice containing details of the default screen on the current platform

Image

Image is an abstract class, and cannot be directly created as an object. It is usually used as a container to store details of an image which can be passed as a parameter to other objects. The Toolkit class contains a method for obtaining an image from disk.

Toolkit

The toolkit is an abstract class which can be used to interface with the current environment. Some methods are now deprecated, since they relate to the older AWT GUI interface, and are no longer relevant with the new Swing components. However, many of the functions of the toolkit are still relevant in the new GUI environment.

To create an instance of Toolkit for the current platform, use the following code:
Toolkit *objectname* **= Toolkit.getDefaultToolkit()**
This object can then be used to access other methods of the toolkit.

Useful methods include:

beep()
This produces the beep sound.

Image getImage(Filename)
Returns an object of type image loaded from the specified file name, e.g.
Toolkit tools = Toolkit.getDefaultToolkit();
Image MyImage = tools.getImage("pictures\\MyPict.gif");
Notes:
Images in the form .gif or .jpeg only can be accessed
The double backslash (\\) must be used as a directory separator in Dos or Windows applications, since a single backslash indicates an escape sequence in string literals.

Image getImage(Url)
An URL can be used in place of a filename. The URL must be a previously created instance of class Url.

int getScreenResolution()
Returns the screen resolution

Dimension getScreenSize()
Returns the dimensions of the current screen

Clipboard getSystemClipboard()
Returns an object of type Clipboard containing the contents of the system clipboard. This allows data to be retrieved from the clipboard maintained by the native operating system.

EVENT HANDLING

Introduction

As discussed earlier, when working with GUI components, it is not possible to write procedures which will be followed in a predefined sequence. Control is handed over to the user, who can use the mouse and keyboard to move from one component to another and initiate actions.

Any action taken by the user is known as an event. Events are detected by the operating system and placed in an event queue. The JVM processes the event queue, calling appropriate methods in the applications program to deal with them.

The Event model

The Event Model is the way in which events are monitored and dealt with within the Java language. The Java 1.0 model is significantly different to the Java 1.2 model, although programs written using the old model will still work in Java 1.2. Since the new model is more efficient and flexible, developers are advised to use the new model when writing new applications. This course covers the 1.2 event model.

The event model has three types of object: Event sources, Event Listeners and Event Objects.

Event Source

An event source is a GUI object which is affected by the event. For example, if a user types text into a text box, the text box would be the event source; if the user clicked on a menu item, the menu item would be the event source.

Events will often affect more than one component. For example, if the user moves the mouse from one component to another, both components would register the event, and so would the container of the components.

The applications programmer must decide which events are of interest to the program for each potential event source, and register a listener for each relevant event. Events which do not require any action on the part of the program need not be monitored.

Event object

An event object is a means of communication between the event source and the event listener. When an event occurs, the JVM will check to see whether a listener has been registered for this event. If so, an event object is created containing any necessary information, and methods for retrieving it. The event object is passed as a parameter to the event listener.

Event Listeners

An event listener is the object designated to respond to an event. For example, an object registered as the menu listener for a menu item labeled “Save” would be notified when this item was selected. It should contain the program code which would actually save the user’s work to disk.

Listener objects receive an event object, from which they can obtain more information about the event which occurred.

There are several different types of listeners corresponding to different types of event. Interfaces have been defined in the Java libraries for each type of listener, documenting the methods which are necessary for that type of listener. Each type of listener also has its corresponding event class.

Any object can be designated as the listener for any object, provided that it implements the correct interface. It is up to the programmer to decide which object is the most suitable and can most conveniently carry out the tasks involved. In many cases, the object which contains the event source is suitable. However, it may be useful in some cases to define a separate listener object. This would typically be where many different events required the same action, for example where a user can use either a menu, a toolbar or a key sequence to select the same option.

It is possible to define more than one listener for the same event. In word processing, for example, multiple windows can be created in order to edit more than one document at a time. If the user selects to exit the word processor, the child windows must be aware of this and close themselves down. They could therefore each be registered as event listeners for this event.

Event types

Events can be broadly categorised into two types: semantic events and low-level events.

Semantic events are events as they would be defined by the user, such as clicking on a command button, or changing text in a text field. Each semantic event may be made up of several low-level events, such as key depressed, and key released. Most programs will be mainly concerned with semantic events, but may occasionally need to deal with low-level events.

The AWT defines the following events:

Semantic	ActionEvent	Clicking a button, selecting an item from a list, making a menu selection, pressing enter on a text field etc
	AdjustmentEvent	Adjusting a scrollbar
	ItemEvent	Selecting an item from a check box or list
	TextEvent	Changing the contents of a text component
Low-level	**ComponentEvent**	A component was resized, moved, hidden etc
	KeyEvent	A key was pressed or released.
	MouseEvent	A mouse button was pressed or released
	MouseMotion Event	The mouse was moved or dragged
	FocusEvent	A component lost or gained focus. The component which is currently selected and able to receive input is said to have the focus; when the user tabs away from it or selects another component it is said to lose focus.
	WindowEvent	A window was activated, closed, changed to an icon etc.
	ContainerEvent	A component was added to or removed from a container

Adaptors

Some of the event listener interfaces require several methods to be defined for different actions on the part of the user. It is often the case that the programmer is only interested in one or two of these. To save additional coding for the methods

which are of no interest, those events which require several methods are provided with adaptor classes. An adaptor class defines empty methods for each one required to satisfy the interface. By extending an adaptor class when coding a listener class, the programmer need only define those methods which require program action.

Summary of the AWT event types

The table below shows details of the requirements of the different types of events defined in the AWT The event type is the same as the interface which must be implemented in each case.

Event Type/ Interface	Event object class	Adaptor class (if any)	Methods required within the listener object	Event sources
ActionListener	ActionEvent		actionPerformed	Button List MenuItem TextField
Adjustment Listener	Adjustment Event		adjustmentValue Changed	Scrollbar
Component Listener	Component Event	Component Adapter	componentMoved componentHidden componentResized componentShown	Component
Container Listener	Container Event	ContainerAdapter	componentAdded componentRemoved	Container
FocusListener	FocusEvent	FocusAdapter	focusGained focusLost	Component
ItemListener	ItemEvent		itemStateChanged	Checkbox CheckboxMenu Item Choice List
KeyListener	KeyEvent	KeyAdapter	keyPressed keyReleased keyTyped	Component
MouseListener	MouseEvent	MouseAdapter	mousePressed mouseReleased mouseEntered mouseExited mouseClicked	Component
MouseMotion Listener	MouseEvent	MouseMotion Adapter	mouseDragged mouseMoved	Component
TextListener	TextEvent		textValueChanged	TextField TextArea
Window Listener	WindowEvent	WindowAdapter	windowClosing windowOpened windowIconified windowDeiconified windowClosed windowActivated windowDeactivated	Window

Swing events Several new events have been defined in the Swing libraries. These include:

Event Type/ Interface	Event object class	Adaptor class (if any)	Methods required within the listener object	Event sources
ChangeListener	ChangeEvent		stateChanged	JComponent
Document Listener	Document Event		changedUpdate insertUpdate removeUpdate	JEditorPane JTextField JTextArea
ListSelection Listener	ListSelection Event		valueChanged	JList JTable
MenuListener	MenuEvent		menuCanceled menuDeselected menuSelected	JMenu JPopupMenu

Event objects: The Listener object will often require more information about the event which

occurred. It will receive an event object, and can make use of its methods to retrieve the necessary information.

Useful methods

Some of the methods defined in these classes are listed below. Examples of how they may be used are included in the next section which gives details of how to use the GUI components. All event classes are extended from the superclass **EventObject**, and inherit its methods.

EventObject

Object **getSource**()
Returns a pointer to the object which initiated the event.

ActionEvent

String **getActionCommand**()
Each component can have an action command associated with it. This allows a listener which deals with more than one event source to identify the source.

Adjustment Event

int **getValue**()
Returns the current value held in the adjustment object
int **getAdjustmentType**()
Obtains the type of adjustment(e.g. increment, block increment etc). Constants are supplied for the different types of adjustment.

ContainerEvent

Container **getContainer**()
Returns a pointer to the affected container
Component **getChild**()
Returns a pointer to the component within the container which was affected.

FocusEvent

boolean **isTemporary**()
Returns true if the loss of focus was only temporary, for example when the user switches to another window.

ItemEvent

Object **getItemSelectable**()
returns a pointer to the originating object
Object **getItem**()
returns a pointer to the item affected by the event
int **getStateChange**()
Returns the type of change: Selected or deselected

KeyEvent

int **getKeyCode**()
Returns the integer code associated with this key
char **getKeyChar**()
Returns the character associated with this key

MouseEvent

int **getY**()
Returns the horizontal position of the mouse pointer
int **getX**()
Returns the vertical position of the mouse pointer
int getClickCount()
Returns the number of times the mouse was clicked
boolean **isPopupTrigger**()
Returns true if this mouse event matches the pop-up menu trigger on the current platform

Event handling summary

An event source allows event listeners to be registered, and will invoke them whenever events occur, passing an event object as a parameter.

Event listeners may be any object, but must implement the correct interface, and therefore define all of the expected methods associated with the event type. Event adaptors provide a set of dummy methods for the interface, and can be used as a superclass when defining event handler classes.

USING SWING COMPONENTS

GUI Programs

A GUI program will generally consist of one or more windows containing other components such as text boxes and command buttons. Windows may be either of the JFrame class, which is a standard window, or of the JDialog class, which is a pop-up dialog box.

Rather than placing components directly onto the window, it is normal to place components onto other containers, such as panels, and place the panel on the window. This makes it easier to achieve the required layout.

In simple cases, it is possible to create objects directly from the Swing library classes, but in order to add further functionality, it is usually necessary to write classes which extend the library classes.

The constructor method of the window component is usually used to set the size and title of the window, and to add components to it.

The **main** method of the public class will generally create an instance of the window object, and use its **show** method to display it on the screen. From this point onwards, program control is no longer handled by **main**, but becomes event driven. In order for the program to be able to be shut down, a WindowListener must have been defined which will exit the program when the window is closed.

More windows can be created at any time as needed.

Importing the correct libraries

All libraries other that **java.lang** which are used in the program must be introduced with an **import** statement at the front of the program.

The following libraries may be required by GUI programs:

javax.swing	The Swing components
java.awt	The AWT components
java.awt.event	Event and event listener classes
javax.swing.event	Additional Swing event and listener classes

Other libraries under Swing and AWT are available for advanced features.

Using the Swing components

The following sections introduce a selection of the GUI components, and give examples of their use.

USING SWING COMPONENTS: JFRAME

Introduction

The JFrame component is a class which may be extended to produce standard screen windows. By default, the window will have a title bar containing an icon, title text, minimise and maximise buttons and a close button.

It will also have a **content pane**, on which any components added to it will be displayed. The layout of the content pane will be decided by a layout manager. Layout managers are discussed in a later section; for the next few sections, programs will simply place components according to the default layout manager.

The size of the window must be defined by the programmer; by default, the window will be 0 by 0 pixels in size.

A WindowListener must be registered for the JFrame. It may check for various types of window event, but should at least handle the WindowClosing event in order to shut down the program.

Sample program: a simple window

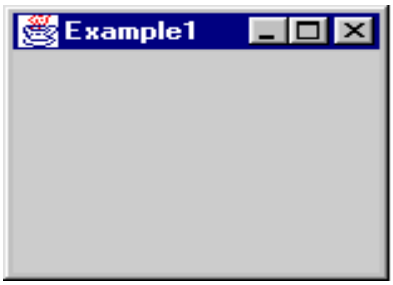
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example1
{
 public static void main(String[] args)
 {
 JFrame Example1 = new EgWindow();  //Create a window
 Example1.show();                   //Display it
 }
}

//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{
//
//* The constructor for EgWindow
//* ***************************

public EgWindow()
 {
 setTitle("Example1");
 setSize(400,300);
```

```
//
//* An inner class is defined as the window Listener
//* It is only interested in the windowClosing event,
//* which will shut down the program
//* ****************************************

  addWindowListener(new WindowAdapter()
      {public void windowClosing(WindowEvent e)
        {System.exit(0);        // Terminate the program
        }
      });
  }
}
```

Sample program notes

- Import statements are included for the AWT, AWT Events and Swing libraries
- The public class creates an instance of the window, and shows it
- The EgWindow class extends JFrame
- The title and size of the window must be set
- An inner class is defined as the windowListener object. This is a useful shortcut when defining windows.
- When the window is closed, the program will terminate using the **exit** method of the Java library class System.
- This program produces a standard window as shown below. The user can move, resize, maximise, minimise and close the window in the standard way.

Sample program output



The content pane

JFrames consist of several sections, or Panes. The two which are of chief interest to the programmer are the Menu Bar and the Contents Pane. Components should never be added to the frame itself, but to the contents pane. In order to access the contents pane, use the following statement:
Container *objectname* = **getContentpane();**
for example:
Container MyFrameContent = getContentpane();
Components can then be added to the window using the **add** method of **MyFrameContent.**

Adding components

Components are added to the contents pane according to the layout manager selected. The default layout manager is Border, which divides the contents pane into five areas: North, South, East, West and Centre. Only one component can be added to each area. The area in which the component is to be placed defaults to Centre. A component could be added to the North segment as follows:
MyFrameContent.add(Label1, "North");

A different layout manager could be selected as follows:
MyFrameContent.setLayout(new FlowLayout());
The Flow layout adds components one after the other, moving to the next line when one line is full

Sample program notes

This program extends the previous example. In addition to the previous code, this program changes the default icon for the frame, and adds a label containing text. Note that only small images are suitable for displaying as icons.

- A pointer to the default toolkit is obtained
- The toolkit is used to obtain the image from disk
- The image is set as the frame's icon
- A pointer to the contents pane is obtained
- JLabel is a Swing component used for displaying text. An instance of Jlabel is created
- The label is added to the contents pane

The Layout Manager will decide where to put it.

Sample program

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example2
{
 public static void main(String[] args)
 {
 JFrame Example2 = new EgWindow();  //Create a window
 Example2.show();                   //Display it
 }
}

//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{

//
//* The constructor for EgWindow
//* ***************************

public EgWindow()
 {
 setTitle("Example 2");
 setSize(150,120);

// * Get a copy of the default toolkit; use it to load
// * one of the demonstration images from the JDK;
// * set the frame's icon image to the one loaded
// ****************************************
```

```
  Toolkit Tools = Toolkit.getDefaultToolkit();
  Image IconImg=Tools.getImage
      ("\\jdk1.2\\demo\\jfc\\swingset\\images\\smalltiger.gif");
  setIconImage(IconImg);

 //
 //* Get a pointer to the content pane, create a label
 //* and add it to the content pane
 //* ***************************************

  Container MyContentFrame = getContentPane();
  JLabel NiceDay=new JLabel("    Have a nice day!");
  MyContentFrame.add(NiceDay);

 //
 //* An inner class is defined as the window listener
 //* It is only interested in the windowClosing event,
 //* which will shut down the program
 //* ***************************************

  addWindowListener(new WindowAdapter()
     {public void windowClosing(WindowEvent e)
       {System.exit(0);
       }
     });
  }
}
```

Sample program output

Please complete Exercise 14

USING SWING COMPONENTS: JPANEL

Introduction

JPanel acts as a container for other components, and can also be used to display text and graphics.

By placing components on a panel rather than directly onto a frame, it is possible to achieve better control of the screen layout, and to group components logically. It is also possible to remove and add panels freely, so that a different set of components can be displayed for different circumstance.

Components can be added to a panel either in its constructor, or externally after an object has been created from it.

By default, panels are created with the Flow Layout, although this can be changed with the **setLayout** method.

Sample program notes

The following program:

- Sets the contents pane to the flow layout
- Creates four labels containing text
- Create a JPanel named Firstpanel, sets its background colour to yellow, and adds two of the labels to it.
- Creates a second JPanel with background set to Red, and adds the other two labels to it
- Adds both panels to the contents pane.

Sample program

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example3
{
  public static void main(String[] args)
  {
  JFrame Example3 = new EgWindow();  //Create a window
  Example3.show();                   //Display it
  }
}

//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{
```

```
//
//* The constructor for EgWindow
//* ***************************

public EgWindow()
  {
  setTitle("Example 3");
  setSize(400,100);


//* The contents pane is given the Flow layout
//* ***********************************

  Container Contents=getContentPane();
  Contents.setLayout(new FlowLayout());

//*
//* Labels containing text are created
//* ****************************

  JLabel Label1 = new JLabel("Mary had a little lamb, ");
  JLabel Label2 = new JLabel("It's fleece was black as soot");
  JLabel Label3 = new JLabel("And into Mary's bread and jam, ");
  JLabel Label4 = new JLabel("His sooty foot 'e put");

//*
//* Create the first panel, set its colour and add two labels to it
//* ***************************************************

  JPanel FirstPanel = new JPanel();
  FirstPanel.setBackground(Color.yellow);
  FirstPanel.add(Label1);
  FirstPanel.add(Label2);

//*
//* Create the second panel, set its colour and add two labels to it
//* ***************************************************

  JPanel SecondPanel = new JPanel();
  SecondPanel.setBackground(Color.red);
  SecondPanel.add(Label3);
  SecondPanel.add(Label4);

//*
//* Add both panels to the contents pane
//* ********************************

  Contents.add(FirstPanel);
  Contents.add(SecondPanel);

//
//* An inner class is defined as the window listener
//* It is only interested in the windowClosing event,
//* which will shut down the program
//* ***************************************
```

```
  addWindowListener(new WindowAdapter()
     {public void windowClosing(WindowEvent e)
       {System.exit(0);
       }
     });
  }
}
```

Sample program output

Using panels to display graphics

Panels are often used to display graphics. In order to do this, it is necessary to write a new class which extends JPanel, and overwrite its **paintComponent** method. This method is called automatically by the system whenever the component is drawn on the screen, either initially or after the window has been minimised or hidden by the user.

The new **paintComponent** method receives the graphics context of the component as a parameter as an object of class Graphics. This Graphics object can then be used to draw the graphics. The first step in the method should be to call **paintComponent** from the superclass in order to draw the rest of the object.

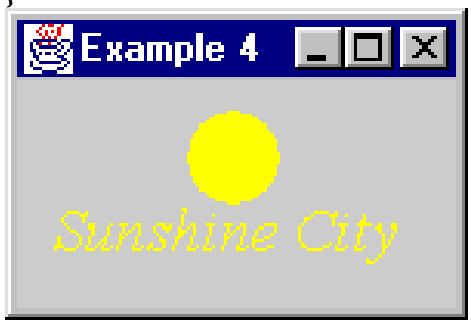
Sample Program

This is demonstrated in the following sample program.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example4
{
 public static void main(String[] args)
 {
 JFrame Example4 = new EgWindow();  //Create a window
 Example4.show();                   //Display it
 }
}

//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{
```

```
//
//* The constructor for EgWindow
//* ***************************

public EgWindow()
  {
  setTitle("Example 4");
  setSize(150,100);

  Container Contents=getContentPane();

  JPanel Panel = new PicturePanel();
  Contents.add(Panel);


  addWindowListener(new WindowAdapter()
      {public void windowClosing(WindowEvent e)
{System.exit(0);
        }
      });
  }
}

//*
//* This class extends JPanel and includes some drawing
//* objects in the paintComponent method
// ***********************************************

class PicturePanel extends JPanel
{

  public void paintComponent(Graphics Gr)
    {
    super.paintComponent(Gr);

    Gr.setColor(Color.yellow);
    Gr.fillOval(55,10,30,30);

    Gr.setFont(new Font("Serif",Font.ITALIC,20));
    Gr.drawString("Sunshine City",12,55);
    }
}
```

Sample Program output



Please complete Exercise 15

USING SWING COMPONENTS: JBUTTON

Introduction

The JButton class defines a button which can be clicked by the user. In order to be useful, an ActionListener must be defined for it, which carries out some action.

Using the JButton

In its simplest form, a button is simply created from the library class, with a text string as a parameter. The text string will appear on the face of the button.

Once the button has been created, its appearance can be altered, if required, using the common Component methods such as **setBackground.** The **addActionListener** method should then be invoked to register a listener for it, and it must be added to its container.

If more than one component shares the same ActionListener, the components can be identified by the **getActionCommand** method of the ActionEvent object. By default, the ActionCommand of a button is the string which appears it. However, it can be changed by the method
setActionCommand("String");

Sample program notes

The following program is an extension of the last sample program, which displayed a text panel and a graphics panel.

This program adds command buttons labelled red, green, blue and yellow. If the user presses a button, the colour of the text and graphics will change accordingly.

Sample program

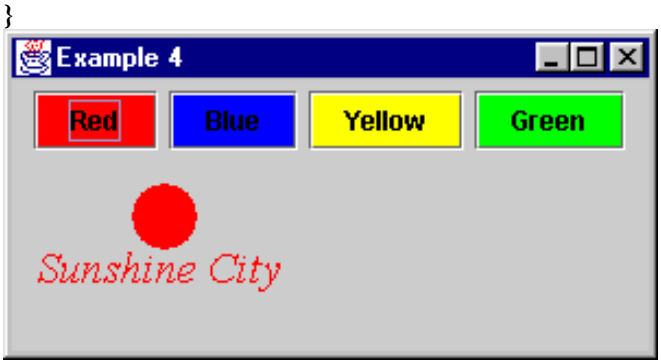
```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example5
{
 public static void main(String[] args)
 {
 JFrame Example5 = new EgWindow();  //Create a window
 Example5.show();                   //Display it
 }
}

//
//* This is the definition of the window, which extends JFrame
//*
*****************************************************************
*

class EgWindow extends JFrame
{

//
//* The constructor for EgWindow
//* ***********************************
```

```
public EgWindow()
  {
  setTitle("Example 4");
  setSize(300,150);

  Container Contents=getContentPane();

//*
//* Create an  instance of the picture panel
//* ****************************************

  PicturePanel Panel = new PicturePanel();
  Contents.add(Panel);

//*
//* Create a new panel to hold the command buttons
//* passing it a pointer to Panel, which will be the
//* action listener for the command buttons
//* ******************************************************


  JPanel BPanel = new ButtonPanel(Panel);
  Contents.add(BPanel,"North");


//*
//* Window listener
//* ******************


  addWindowListener(new WindowAdapter()
      {public void windowClosing(WindowEvent e)
        {System.exit(0);
        }
      });
  }
}

//*
//* This class extends JPanel and includes some drawing
//* objects in the paintComponent method.
//* Since this class contains the objects which will be
//* affected by the command buttons, it also acts as
//* their ActionListener
// ***************************************************************

class PicturePanel extends JPanel implements ActionListener
{

//*
//* Constructor – sets a default colour of Yellow
//* ************************************************

  public PicturePanel()
  { PicColor=Color.black; }
```

```
//*
//* Painting the component: The color is set to
//* the private variable PicColor, which is set
//* according to the last command button clicked
//* ************************************************

  public void paintComponent(Graphics Gr)
    {
    super.paintComponent(Gr);
    Gr.setColor(PicColor);

    Gr.fillOval(55,10,30,30);


    Gr.setFont(new Font("Serif",Font.ITALIC,20));
    Gr.drawString("Sunshine City",12,55);
    }

//*
//* This is the action listener method for the command buttons
//* ************************************************

public void actionPerformed(ActionEvent Evt)
{

String Cmd = Evt.getActionCommand();

if (Cmd=="Yellow")  PicColor=Color.yellow;
if (Cmd=="Green")  PicColor=Color.green;
if (Cmd=="Blue")  PicColor=Color.blue;
if (Cmd=="Red")  PicColor=Color.red;
repaint();

}
private Color PicColor;
}

//*
//* This class extends JPanel to hold the command buttons
//* ************************************************

class ButtonPanel extends JPanel
{


//*
//* Define the buttons with the appropriate text
//* and colour, and add them to the panel
//* ************************************

public ButtonPanel(PicturePanel Listener)
{
super();

JButton RedButton=new JButton("Red");
```

```
RedButton.setBackground(Color.red);
RedButton.addActionListener(Listener);
add(RedButton);

JButton BlueButton=new JButton("Blue");
BlueButton.setBackground(Color.blue);
BlueButton.addActionListener(Listener);
add(BlueButton);

JButton YellowButton=new JButton("Yellow");
YellowButton.setBackground(Color.yellow);
YellowButton.addActionListener(Listener);
add(YellowButton);

JButton GreenButton=new JButton("Green");
GreenButton.setBackground(Color.green);
GreenButton.addActionListener(Listener);
add(GreenButton);
}

}
```

Sample program output



Please complete exercise 16

USING SWING COMPONENTS: JOPTIONPANE

Introduction

JOptionPane is a pop-up standard dialogue box . Unlike other JComponents, this class does not usually have an object created from it, and is not displayed on a frame or panel. It can, in fact, be used in any program, even if the program does not use any other GUI components, but the program must terminate with System.exit(0) if a dialog box is invoked.

It has four static methods which can be used to display different types of dialogue box:

Confirmation Box	Allows the user to select one of two command buttons: Ok or Cancel
Input Box	Allows the user to enter text
Option Box	Allows the user to choose from a set of options
Message box	Displays a message

showMessage Dialog

This method is used to display a dialogue box containing a title, a standard icon, and a message. It does not return any value.

The icon displayed will depend on the message type, which can be any of the following:

ERROR_MESSAGE
INFORMATION_MESSAGE
WARNING_MESSAGE
QUESTION_MESSAGE
PLAIN_MESSAGE

To display a message box, the method is called as follows:
JOptionPane.showMessageDialog(parent, Message, Title, Type)

parent	This is the parent component, for example the current instance of JFrame. if there is no parent, this should be given the value **null**
Message	A string containing the message which is to be displayed. It is also possible to pass an object or an icon for this parameter.
Title	A string containing the title which will appear in the title bar
Type	One of the predefined type constants listed above

Example:

```
JOptionPane (null,"Error: Invalid Date","Date of Birth",
             JOptionPane.ERROR_MESSAGE)
```

Confirmation Box

The confirmation box displays a message, a title and an icon, and allows the user to click one of two buttons labelled Ok and Cancel. It returns an integer representing which of the two buttons was selected. Constants are defined in JOptionPane which can be used to compare to the returned integer:
OK_OPTION
CANCEL_OPTION
YES_OPTION
NO_OPTION
CLOSED_OPTION

CLOSED_OPTION indicates that the box was closed without making a selection.

The method call is as follows:
JOptionPane.showConfirmDialog
(parent, Message, Title, OptionType, Message Type)
Option Type can be set to one of the following constants:
DEFAULT_OPTION
YES_NO_OPTION
YES_NO_CANCEL_OPTION
OK_CANCEL_OPTION

DEFAULT_OPTION displays the OK button only.
The other parameters are the same as for a message box.

Input Box

In its simplest form, an input box will display a message, and provide a text box into which the user can enter a text string. It will also display two command buttons: OK and Cancel.

If OK was selected, the string typed by the user will be returned, otherwise the input box will return null.

The simplest method call is as follows:
Stringvariable=JOptionPane.showInputDialog(Message)
for example,
Surname=JOptionPane.showInputDialog(“Please enter your surname”);

Option Box

An Option Box allows the user to choose between a set of options. The options, usually text strings, are placed in an array, which is passed as a parameter to the method. An integer is returned indicating the element number in the array of the item which the user selected.

The method call is as follows:
JOptionPane.showOptionDialog
(parent, Message, Title, OptionType, Message Type,
icon,options,default)
Icon is an icon which will be displayed, and can be null. Options is the name of the array containing the options, and default is the option which will be selected by default.

The following program illustrates an option box.

```
import java.awt.*;
import javax.swing.*;
public class test1
{
 public static void main(String[] args)
 {
 String[] StrArray = new String [] {"Me","You","Us","Them"};
```

```
int Choice = JOptionPane.showOptionDialog
(null,"Please select one of the following:","Test1",
JOptionPane.YES_NO_CANCEL_OPTION,
JOptionPane.PLAIN_MESSAGE,null,StrArray,StrArray[1]);
System.out.println(StrArray[Choice]);
System.exit(0);
}

}
```

Please complete Exercise 17

USING SWING COMPONENTS: TEXT COMPONENTS

Introduction

These components are covered together, since they are very similar. They include the following:

- JTextField which allows the user to enter text into a single-line text box
- JTextArea allows entry of multiline data; the user presses the enter key at the end of each line.
- JPasswordField which is similar to JTextField, but does not echo the characters typed by the user.

These components are perhaps the most useful method of getting data via the keyboard in Java.

TextAreas give the user the use of editing features such as highlighting a block of text, and cutting, copying and pasting using the control keys Ctrl X., CtrlC and CtrlV.

The data entered into these objects is simply stored as a text string, and is not validated in any way. Validation must be done by the applications program. This could be handled in various ways, such as:

- Add a document listener to the object. This will be activated if the user makes any changes to the text. This is only necessary if validation must take place immediately after every minor change to the text.
- Add a focus listener to the object. When the user tabs or uses the mouse to leave this field, it will be activated, and validation can take place.
- Include a command button that the user can click to indicate that he/she has finished entering text, and place the validation in the listener for the command button.

Methods

There are several useful methods available in these classes. They include:

void setText(String)	Sets the contents of the text component
String getText()	Retrieves the contents
void setEditable(boolean)	Sets whether or not the user can edit the contents
void setColumns(int)	Sets the width of a text field
void setechoChar(char)	Sets the echo character for password fields
char [] getPassword()	Returns the contents of a password field as an array of characters
void setWrapStyleWord(boolean)	Sets whether lines should wrap on word boundaries if line wrapping is enabled.
void setLineWrap(boolean)	Sets or clears line wrap for a text area

Constructors

The constructors for creating new objects from these classes are as follows:

JTextField	**JTextField(String, width)** or **JTextField(width)** The first constructor creates a text field containing the given string, with the column width supplied as an integer. The second requires the column width only, and creates an empty text box.
JTextArea	**JTextArea(String,rows,columns)** or **JTextArea(rows,columns)**
JPasswordField	**JPasswordField(width)**

Scroll panes

Since the actual text typed into a text area by the user may be larger than the area allocated to it on the screen, it is useful to provide scrollbars so that the user can move around the text at will.

This can be done by placing the text area inside a JScrollPane container, which will then allow scrolling through the contents. This is done as follows:

```
JTextArea userData = new JTextArea(20,10);
JScrollPane scrollArea = new JScrollPane(userData);
```

ScrollArea is then added to the contents pane.

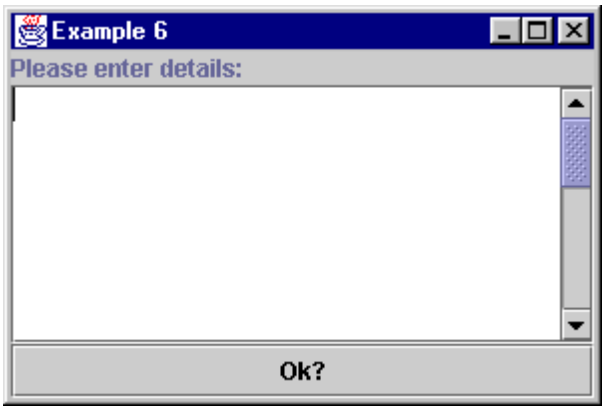
This is illustrated in the following example:

Sample Program

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example6
{
 public static void main(String[] args)
 {
 JFrame Example6 = new EgWindow();  //Create a window
 Example6.show();                   //Display it
 }
}

//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{

//
//* The constructor for EgWindow
//* ***************************
```

```
public EgWindow()
  {
  setTitle("Example 6");
  setSize(300,200);

//* A scroll pane containing a text area is created & added
//* ************************************************

  Container Contents=getContentPane();
  JTextArea UserData = new JTextArea(20,10);
  UserData.setLineWrap(true);
  UserData.setWrapStyleWord(true);
  JScrollPane DataPane = new JScrollPane(UserData);
  Contents.add(DataPane);

  JButton Ok = new JButton("Ok?");
  Contents.add(Ok,"South");

  JLabel Header = new JLabel("Please enter details:");
  Contents.add(Header,"North");

//
//* An inner class is defined as the window listener
//* It is only interested in the windowClosing event,
//* which will shut down the program
//* ********************************************

  addWindowListener(new WindowAdapter()
      {public void windowClosing(WindowEvent e)
        {System.exit(0);
        }
      });
  }
}
```

Sample Program Output



Please complete Exercise 18

USING SWING COMPONENTS: OPTIONS

Introduction

There are three components which allow the user to make a selection from a list of options.

Checkboxes allow the user to select or deselect items by clicking inside the checkbox.

Radio buttons allow the user to select only one of a group of options.

Lists allow the user to browse through a list of items, making selections as needed.

Checkboxes

To create a check box, the constructor is:
JCheckBox *objectname* = new JCheckBox(*String*)

The string supplied will create a label within the checkbox.

The checkbox will generate an action event when the user selects or deselects it.

Useful methods defined in the JCheckBox class include:

boolean isSelected()
Returns true if the item is selected.
void setSelected(boolean)
allows the program to select or deselect the checkbox.

Sample Program

The following program code illustrates displaying checkboxes. However, in practice, it would be necessary to add an ActionListener to each box.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example7
{
 public static void main(String[] args)
 {
 JFrame Example7 = new EgWindow();  //Create a window
 Example7.show();                   //Display it
 }
}
```

```
//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{

//
//* The constructor for EgWindow
//* ***************************

public EgWindow()
  {
  setTitle("Library Membership");
  setSize(300,200);

//* Components are added to the frame
//* ********************************

  Container Contents=getContentPane();
  JPanel Panel = new JPanel();
  JCheckBox Box1 = new JCheckBox("Travel");
  JCheckBox Box2 = new JCheckBox("Education");
  JCheckBox Box3 = new JCheckBox("Literature");
  JCheckBox Box4 = new JCheckBox("Science");
  JCheckBox Box5 = new JCheckBox("Other");
  Panel.add(Box1);
  Panel.add(Box2);
  Panel.add(Box3);
  Panel.add(Box4);
  Panel.add(Box5);

  Contents.add(Panel);

  JLabel Header = new JLabel("Please indicate interests:");
  Contents.add(Header,"North");


//
//* An inner class is defined as the window listener
//* It is only interested in the windowClosing event,
//* which will shut down the program
//* ***************************************

  addWindowListener(new WindowAdapter()
      {public void windowClosing(WindowEvent e)
        {System.exit(0);
        }
      });
  }
}
```

Sample program output

Radio buttons

Checkboxes allow any number of boxes to be checked at one time. However, in many cases, only one of a group of options should be selected at any one time. An example of this would be the options Male and Female. Only one of these could be true for any one individual.

Radio buttons will allow only one of a group of buttons to be selected at one time. To achieve this, an object of class ButtonGroup is created. Individual buttons are then created and added to the group. The buttons must also be added to a container in order to become visible.

To create a radio button:
JRadioButton *objectname* = new JRadioButton(*label*,*initialstate*)
label is a string which will appear next to the button
initialstate is a boolean value indicating whether the button's initial state will be selected or deselected.

To create a button group:
ButtonGroup *objectname* = new ButtonGroup()

To add a button to a group:
groupname.**add(*buttonname*)**

Sample Program

The following program illustrates this. Note that again, in practice, an ActionListener should be added for each button.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example8
{
  public static void main(String[] args)
  {
  JFrame Example8 = new EgWindow();  //Create a window
  Example8.show();                   //Display it
  }
}

//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{
```

```
//
//* The constructor for EgWindow
//* ***************************

public EgWindow()
  {
  setTitle("Employee Details");
  setSize(300,150);

//* Components are added to the frame
//* *****************************

  Container Contents=getContentPane();
  JPanel Panel = new JPanel();

  JRadioButton B1 = new JRadioButton("Single",true);
  JRadioButton B2 = new JRadioButton("Married",false);
  JRadioButton B3 = new JRadioButton("Widowed",false);
  JRadioButton B4 = new JRadioButton("Divorced",false);

  Panel.add(B1);
  Panel.add(B2);
  Panel.add(B3);
  Panel.add(B4);

  ButtonGroup BGroup = new ButtonGroup();

  BGroup.add(B1);
  BGroup.add(B2);
  BGroup.add(B3);
  BGroup.add(B4);


  Contents.add(Panel);

  JLabel Header = new JLabel("Marital Status:");
  Contents.add(Header,"North");


//
//* An inner class is defined as the window listener
//* It is only interested in the windowClosing event,
//* which will shut down the program
//* ****************************************

  addWindowListener(new WindowAdapter()
      {public void windowClosing(WindowEvent e)
        {System.exit(0);
        }
      });
  }
}
```

Sample program output

Lists

A list displays an array of objects inside a list box. The user can select one or more items from the list, although it is possible to restrict his selection to only one item.

To create a new list from an existing array of objects, the syntax is as follows:
JList *listname* = **new JList(*arrayname*)**

Since it often happens that a list contains more items than can be viewed on screen, it is usual to place the list inside a JScrollPane.

The following methods are included in the class JList:
setVisibleRowCount(*integer*)
This sets the number of rows which will be visible at one time. The default is 8.
setSelectionMode(*modeconstant*)
The default selection mode is Multiple, allowing the user multiple selections. To restrict the user to a single selection use the constant SINGLE_SELECTION.
object [] getSelectedValues();
This returns an array of objects selected by the user.
object getSelectedValue();
When only one selection is allowed, this will return the single value selected. It can also be used to return the first selected value where multiple selections are allowed.

Lists need to register a ListSelectionListener which will receive a ListSelectionEvent. However, as the user makes a selection, several events may be generated. The ListSelectionEvent method **getValueIsAdjusting** will return true if this is not the final adjustment.

The following sample program implements a simple, single-selection list inside a scroll pane, and displays a message box containing the selection made. The **getSource** method of the event is used to obtain a pointer to the list which activated the event in order to be able to access its contents.

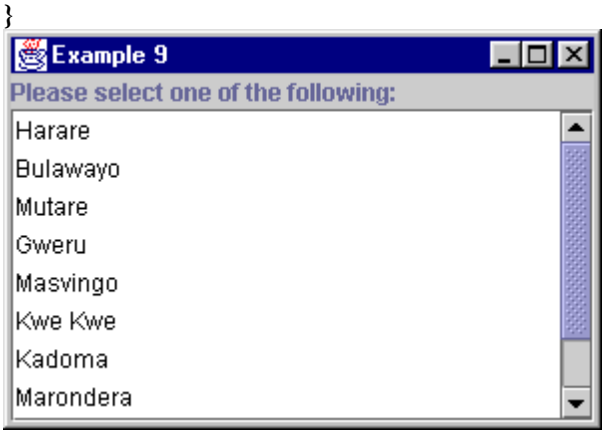
Sample Program source

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example9
{
 public static void main(String[] args)
 {
 JFrame Example9 = new EgWindow();  //Create a window
 Example9.show();                   //Display it
 }
```

```
}

//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{

//
//* The constructor for EgWindow
//* ***************************

 public EgWindow()
 {
 setTitle("Example 9");
 setSize(300,200);

//* A scroll pane containing a list is created & added
//* ****************************************

 Container Contents=getContentPane();
 String[] Towns = new String []
 {"Harare", "Bulawayo","Mutare", "Gweru", "Masvingo", "Kwe Kwe",
  "Kadoma", "Marondera", "Kariba", "Hwange"};

 JList TownList = new JList(Towns);

//*
//* Add a listener for TownList
//* ************************

 TownList.addListSelectionListener(new Listener());


 JScrollPane ListPane = new JScrollPane(TownList);
 Contents.add(ListPane);

 JLabel Header = new JLabel("Please select one of the following:");
 Contents.add(Header,"North");

//
//* An inner class is defined as the window listener
//* It is only interested in the windowClosing event,
//* which will shut down the program
//* ****************************************

 addWindowListener(new WindowAdapter()
     {public void windowClosing(WindowEvent e)
       {System.exit(0);
       }
     });
 }
}

//*
```

```
//* An Listener for ListSelectionEvent
//* ********************************

class Listener implements ListSelectionListener
{
   public Listener()
   {}
   public void valueChanged(ListSelectionEvent ev)
    {
    if (!ev.getValueIsAdjusting())
      {
      JList source = (JList)ev.getSource();
      String Choice = (String)source.getSelectedValue();
      JOptionPane.showMessageDialog(null,Choice);
      }
    }


}
```

Sample Program output



Please complete Exercise 19

USING SWING COMPONENTS: MENUS

Introduction

Menu bars normally are placed at the top of a frame. The menu bar contains menus, which are made up of items and submenus.

The user can call up a menu by clicking its title on the menu bar. He can then make a selection from the menu.

Menu items should have action listeners which carry out the actual task required.

The classes JMenuBar, JMenu and JMenuItem implement this hierarchy.

Incorporating a menu

The first step is to create a menu bar:

JMenubar *menubarname* = new JMenuBar()

This is then added to the frame. Unlike other components, it is not added to the content pane.

***JFramename*.setJMenuBar(*menubarname*)**

The next step is to create menus. For each menu which is to appear:

JMenu *menuname* **= new JMenu(*String*)**

Menu items are created:

JMenuItem *itemname* = new JMenuItem(*String*)

Menu items, separators and submenus are added to the menu in the order in which they are to appear:

*menuname.***add(*itemname*)**

*menuname.***add(*menuname*)**

*menuname.***addSeparator()**

Menus are then added to the menu bar:

menubarname.**add(*menuname*)**

Listeners

Action listeners must be added for each menu item.

Sample program notes

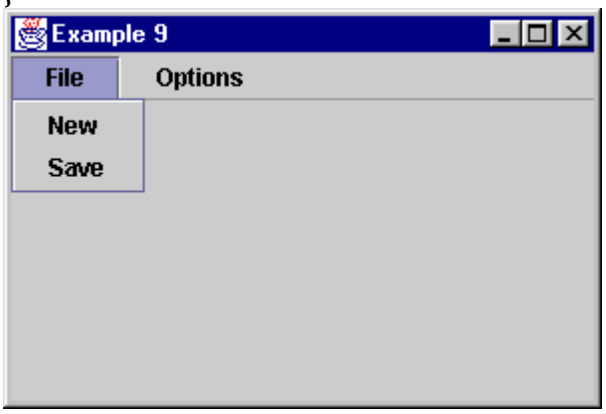
The sample program contains a dummy File and Options menu. Action listeners are included for each item, but they simply display messages to the effect that the method is not yet available.

Sample Program

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
//
//* This is the public class, where program execution begins
//* ************************************************

public class example10
{
 public static void main(String[] args)
 {
 JFrame Example10 = new EgWindow();  //Create a window
 Example10.show();                   //Display it
 }
}

//
//* This is the definition of the window, which extends JFrame
//* ************************************************

class EgWindow extends JFrame
{

//
//* The constructor for EgWindow
//* ***************************

public EgWindow()
 {
 setTitle("Example 9");
 setSize(300,200);
 JMenuBar MainMenu = new JMenuBar();
 this.setJMenuBar(MainMenu);

 JMenu FileMenu = new JMenu("File");
 JMenuItem FileNew = new JMenuItem("New");
 JMenuItem FileSave = new JMenuItem("Save");
 FileMenu.add(FileNew);
 FileMenu.add(FileSave);

 JMenu OptionsMenu = new JMenu("Options");
 JMenuItem Opt1 = new JMenuItem("Option 1");
 OptionsMenu.add(Opt1);

 MainMenu.add(FileMenu);
 MainMenu.add(OptionsMenu);

 FileNew.addActionListener(new Listener());
 FileSave.addActionListener(new Listener());
 Opt1.addActionListener(new Listener());


//
//* An inner class is defined as the window listener
//* It is only interested in the windowClosing event,
```

```
//* which will shut down the program
//* ************************************

  addWindowListener(new WindowAdapter()
      {public void windowClosing(WindowEvent e)
        {System.exit(0);
        }
      });
  }
}

//*
//* An Listener for ActionEvent
//* *************************

class Listener implements ActionListener
{
   public void actionPerformed(ActionEvent ev)
     {
     String Choice = ev.getActionCommand();
     String OutMessage=null;
     if (Choice=="New")  OutMessage="New file choice not yet available";
     if (Choice=="Save") OutMessage="Save file choice not yet available";
     if (Choice=="Option 1")  OutMessage="Option 1 not yet available";
     JOptionPane.showMessageDialog(null,OutMessage);
     }


}
```

Sample Program Output



LAYOUT MANAGERS

Introduction

It is not recommended that components be placed in set positions within their containers, as they may not appear correctly if a screen with a different resolution is used, or if the user resizes the window.

Instead, Java provides a selection of layout managers which decide exactly where and how to place components.

The layout manager is set using the **setLayout(Layout object)** method of the container.

This section takes a brief look at the layout managers which are available.

Border layout

Components are laid out in five areas of the container, referenced as "North", "South", "East", "West" and "Center".

The area needs to be specified when adding a component to a container with Border layout. If no area is specified, "Center" is assumed.

Flow layout

Components are added sequentially to the container, and flow from top to bottom, left to right.

Grid Layout

The container is divided into columns and rows of equal size. When setting this layout, the number of columns and rows is needed as a parameter:
setLayout(new GridLayout(columns,rows))

Box layout

This allows components to be arranged in a series of either horizontal or vertical boxes.

Grid bag layout

This is the most flexible, and correspondingly the most complex, of the layout managers.

It is similar to the grid layout, except that cells are not necessarily the same size, and it is possible to specify the alignment of a component within a cell.

It is best to start with a sketch of the desired layout on paper. This can then be divided up into columns and rows.

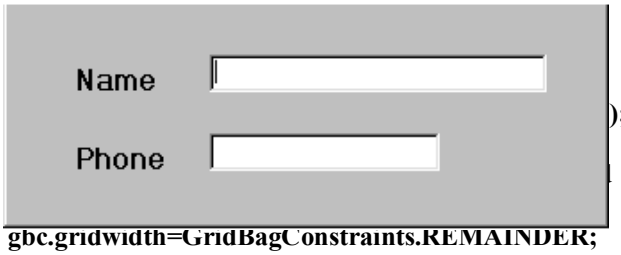
Within the program, a GridBagLayout object is defined and set as the layout manager for the container. A GridBagConstraints object, which holds information about the positioning of components, is also created.

Before adding an item to the container, the GridBagConstraints must be set to contain the layout information for the component. The component is then added to the container as follows:
Container.add(*objectname, constraintsobjectname*)

The type of information stored in the constraints object is:

gridx,gridy The column and row numbers

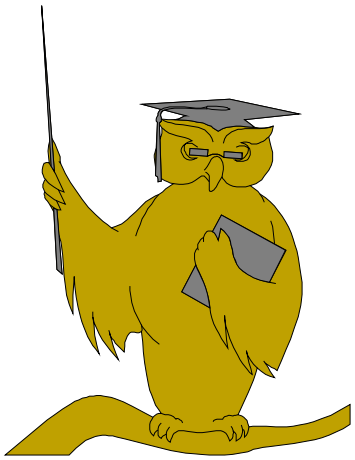
gridheight, gridwidth	The number of columns and rows which the component occupies. If **gridwidth** is set to the constant **GridbagConstraints.REMAINDER**, this will force this component to take up the rest of the line, so that the next component will be placed on the next line down.
weightx,weighty	If set to zero, the component does not change size. Values between 1 and 100 are used to give weighting when the layout manager decides whether to resize the component. For most applications, either 0 or 100 are suitable values depending on whether or not the component should be resized.
fill	Used to specify whether the component will be resized to fill its allotted area, if the area is larger than the component. It may be set to one of the following constants defined in GridBagConstraints: **NONE, HORIZONTAL, VERTICAL** or **BOTH.**
anchor	If the component will not fill its area, this sets the justification within the cell. It may be set to one of the following constants defined in GridBagConstraints: **CENTER, NORTH, NORTHEAST, EAST, SOUTHEAST, SOUTH, SOUTHWEST, WEST,** or **NORTHWEST**. The default value is **CENTER**.
insets	This is used to control the amount by which the component will be inset within its cell. It requires a parameter of class Insets, which defines four values corresponding to the inset from the top, left, bottom and right respectively.
Using the GridBagLayout	A panel named Panel1 is required to be laid out as per the example below, containing components NameLabel, NameText, PhoneLabel and PhoneText.



);

gbc.gridwidth=GridBagConstraints.REMAINDER;
// Next component is the last on the line
Panel1.add(NameText,gbc);
gbc.gridwidth=1;
Panel1.add(PhoneLabel,gbc);
gbc.gridwidth=GridBagConstraints.REMAINDER;
Panel1.add(PhoneText,gbc);

Review Questions



1.	Why is it necessary to understand the hierarchy of GUI components?
2.	How does an event source communicate with an event listener?
3.	What is the purpose of the Graphics object?
4.	What is the difference between check boxes and radio buttons?
5.	How does the Border layout manager position its components?

THE INPUT/OUTPUT CLASSES

Learning Module Objectives

When you have completed this learning module you will have seen how to:

- Use input-output streams and filters
- Program keyboard entry
- Use File, Buffering, Data and Object streams
- Read and write text files, and use number formatting
- Use the File class to manage files and directories

STREAMS AND FILTERS

Streams

All input-output in Java is dealt with as byte streams. The same streaming classes can be used whether data is coming from or going to a file, a printer, the keyboard, a network or large-scale memory storage.

Java provides over 60 classes of stream handlers, which can be confusing for beginners. These classes fall into a hierarchy, as do the GUI components, with subclasses inheriting methods from higher levels.

All stream handlers originate from one of four abstract classes:

InputStream – An incoming stream of bytes
OutputStream – An outgoing stream of bytes
Reader – A n incoming stream of characters
Writer - An outgoing stream of characters

This variety of stream handlers provides many types of functionality in managing data streams. We will make use of the following in this course:

FileInputStream	Reads a stream of bytes from a file
FileOutputStream	Writes a stream of bytes to a file
DataInputStream	Provides methods to read each of the primitive data types
DataOutputStream	Provides methods to write each of the primitive data types
BufferedInputStream	Provides a buffered read
BufferedOutputStream	Provides a buffered write
InputStreamReader	Reads characters in platform-specific format (e.g. ASCII) and converts to Unicode
InputStreamWriter	Converts Unicode to platform-specific character format
FileWriter	Writes text files in platform-specific format
FileReader	Reads text files in platform specific format
BufferedReader	Provides a buffered read in text format
PrintWriter	Writes data in printable format

Note that all I/O streams can throw IOExceptions, which must be either dealt with or thrown back to the JVM. All streams have the method **close()** and should be closed when no longer in use.

Filters

Most input-output operations will require the functionality of two or more of these classes. For example, much greater efficiency when reading files can be obtained by adding buffering.

This can be achieved by "filtering" the data from one stream to another. A program can arrange for data to be retrieved from the BufferedInputStream to achieve buffering, and for the BufferedInputStream to retrieve the data from the FileInputStream.

This would be coded as follows:

```
FileInputStream Myfile = new FileInputStream("MyFile.dat");
```

This attaches a FileInputStream object named Myfile to a disk file named MyFile.dat.

```
BufferedInputStream Mydata = new BufferedInputStream(Myfile);
```

This attaches a BufferedInputStream named Mydata to the FileInputStream named MyFile.

The methods of BufferedInputStream can then be used to read data stored in Myfile.dat.

The declaration of a filter as shown above can be shortened as follows:

```
BufferedInputStream Mydata = newBufferedInputStream( new
                                    FileInputStream("Myfile.dat"));
```

Combinations of streams

There can obviously be many combinations of streams suitable for different circumstances. It is quite common to filter data through three or more input streams to achieve a particular task. The rest of this chapter will look at some common I/O tasks, and the combination of stream handlers suitable for each.

KEYBOARD ENTRY

Introduction

Keyboard entry is taken from the pseudofile System.in. It needs to be passed through the InputStreamReader in order to convert the ASCII characters received from the keyboard to the Unicode characters required by Java.

Since it would be a nuisance to deal with keyboard entry character by character, it is usually passed through BufferedReader. This has a readLine method which returns a character string terminated by the enter key.

Keyboard entry is therefore normally declared as follows:

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

To read a line from the keyboard, the syntax is

```
InputData = Kbd.readLine();
```

where InputData is a string variable.

This is illustrated by the following program, which keeps a total of numbers typed in at the keyboard, and displays a total when the user enters **e** to terminate.

Sample Program

```
import java.util.*;
import java.io.*;
public class keyboard
{
 public static void main(String[] args) throws IOException
 {
 BufferedReader Kbd = new BufferedReader(new
                        InputStreamReader(System.in));
 String InputData;
 char FirstChar;
 double Total=0;
 double Num;
 boolean Finished = false;

 System.out.println("");
 System.out.println("Your friendly adding machine!");
 System.out.println("============================");
 System.out.println("");
 System.out.println("Enter numbers; type e to end");
```

```
do
   {
   System.out.print("Next Number? ");
   InputData = Kbd.readLine();
   FirstChar=InputData.charAt(0);
   if (FirstChar=='e'|FirstChar=='E') Finished=true;
   else
      {Num=Double.parseDouble(InputData);
       Total+=Num;
      }
   }
  while(!Finished);
  System.out.println("Total is " + Total);
  }
}
```

Please complete Exercise 20

DATA FILES

Introduction

Data files store information in a form in which it will be used by the program, as opposed to text files which are stored in human readable form. Text files are covered in the next section. File I/O is based on the FileInputStream and FileOutputStream classes, but since these only deal with byte level I/O, it is usually convenient to filter these through other I/O streams.

To declare FileInputStream and FileOutputStream objects, the syntax is as follows:
FileInputStream *objectname* = new FileInputStream("*pathname*")
and
FileOutputStream *objectname* = new FileOutputStream("*pathname*")

Note: when working with DOS files where the path separator is the backslash character, a double backslash character should be used since a single backslash is interpreted as an escape sequence in string literals.

Data Streams

The DataInputStream and DataOutputStream are useful in that they can read and write all of the Java primitive data types: Float, Double, Byte, Short, Integer, Long Integer, Char and Boolean. By adding a data stream as a filter to or from a file stream, these methods become available.

The file object would be declared as follows:

DataInputStream *objectname* = new DataInputStream
(new FileInputStream("*pathname*"))
or
DataOutputStream *objectname* = new DataOutputStream
(new FileOutputStream("*pathname*"))

When reading or writing the data, the methods of the data stream would be used.

For DataOutputStream, these include:
writeFloat(), writeDouble(), writeByte(), writeShort(), writeInt(), writeLong(), writeChar() and writeBoolean(), all of which return void.

For DataInputStream, there are corresponding read methods, such as readFloat(). These return the data read. The read methods should always be enclosed in a try … catch block which checks for EOFException, which occurs if there is no more data to read.

The following sample program reads numbers from the keyboard, writes them to a disk file, closes it, reads them back again and displays them on the screen.

Sample Program

```
import java.util.*;
import java.io.*;
public class test1
{
 public static void main(String[] args) throws IOException
 {
 BufferedReader Kbd = new BufferedReader(new
InputStreamReader(System.in));
 DataOutputStream Outfile = new DataOutputStream
                  (new FileOutputStream("Test1.dat"));
 String InputData;
 char FirstChar;
 double Num;
 boolean Finished = false;

 System.out.println("");
 System.out.println("Enter numbers; type e to end");

 do
  {
  System.out.print("Next Number? ");
  InputData = Kbd.readLine();

  FirstChar=InputData.charAt(0);
  if (FirstChar=='e'|FirstChar=='E') Finished=true;
  else
     {Num=Double.parseDouble(InputData);
      Outfile.writeDouble(Num);
     }
  }
 while(!Finished);

 Outfile.close();

 DataInputStream InFile = new DataInputStream
                  (new FileInputStream("Test1.dat"));

 Finished=false;

 while(!Finished)
 {try
   {
   Num=InFile.readDouble();
   System.out.println("Number is " + Num);
   }
  catch(EOFException EndofFile)
   {Finished=true;}
 }
}
}
```

Buffered streams

In order to speed up file access, it is worth filtering file access through the BufferedInputStream or BufferedOutputStream. This is often done in conjunction with the data streams, so that the declarations would be as follows:

```
DataOutputStream Outfile = new DataOutputStream
                  (new BufferedOutputStream
                  (new FileOutputStream("Test1.dat")));
```

or

```
DataInputStream Infile = new DataInputStream
                (new BufferedInputStream
                (new FileInputStream("Test1.dat")));
```

Object streams

The ObjectInputStream and ObjectOutputStream are used to write and read entire objects. This will only work if the object implements the Serializable interface. They would filter from the file streams, possibly filtering through the buffered streams. The declarations would therefore be:

```
ObjectOutputStream Outfile = new ObjectOutputStream
                (new BufferedOutputStream
                (new FileOutputStream("Test1.dat")));
```

or

```
ObjectInputStream Infile = new ObjectInputStream
                (new BufferedInputStream
                (new FileInputStream("Test1.dat")));
```

The methods of the object streams can then be used for file access. These include readObject(), which returns an object, and writeObject(Object).

TEXT FILES

Introduction

Text files contain strings built of characters; they must therefore be accessed by streams descended from the Reader and Writer classes which deal with character I/O. The InputStreamReader and OutputStreamWriter will convert characters between Unicode as required by Java and the coding system used on the native platform, usually ASCII. If data is coming from a disk file, these would have to be filtered from the FileInputStream and FileOutputStream, since InputStreamReader and OutputStreamWriter do not have file accessing capabilities.

FileReader and FileWriter

The FileReader and FileWriter classes carry out the same tasks as filtering an InputStreamReader from a FileInputStream, and an OutputStreamWriter to a FileOutputStream. However, since they deal with character level input only, they are often used in conjunction with the BufferedReader and PrintWriter classes which read and write a line of text.

The declarations would be:

BufferedReader *objectname* = new BufferedReader
(new FileReader("*filename*"));

or

PrintWriter *objectname* = new PrintWriter
(new FileWriter("*filename*"));

PrintWriter can equally well be directed to a printer.

The following program displays the contents of a text file one screen at a time, allowing the user to type in the name of the file from the keyboard.

Sample program

```
import java.util.*;
import java.io.*;
public class fileread
{
 public static void main(String[] args) throws IOException
 {
 BufferedReader Kbd = new BufferedReader(new
InputStreamReader(System.in));
 String FileName, InputLine, KbdEntry;
 boolean Finished = false;
 int Counter = 0;
 System.out.println("");
 System.out.println("Enter the name of the file to be read:");
 FileName = Kbd.readLine();

 BufferedReader InFile = new BufferedReader
                (new FileReader(FileName));

 Finished=false;
```

```
while(!Finished)
 {
   InputLine=InFile.readLine();
   if (InputLine==null) Finished=true;
   else              System.out.println(InputLine);

   Counter++;
   if (Counter==20)
   {
   System.out.print("Press enter to continue ");
KbdEntry=Kbd.readLine();
   Counter=0;
   }
 }

}
}
```

Formatting

When producing printed output, it will often be necessary to format numeric data to certain standards, for example using a currency prefix, or specifying the number of digits before or after the decimal point.

The NumberFormat class in the java.text library is used for this purpose. Since the type of formatting often depends on the locale, the relevant formatting details are first retrieved from the system using one of the following:
objectname= **NumberFormat.getNumberInstance()**
objectname= **NumberFormat.getCurrencyInstance()**
objectname= **NumberFormat.getPercentInstance()**

NumberInstance contains the standard number format, Currency Instance the currency symbol and formatting, and PercentInstance is used for expressing a decimal fraction as a percentage.

Once these objects have been created, they can be used to format numeric data, for example:

```
Formatter=NumberFormat.getNumberInstance()
StringName=Formatter.format(Num);
```

There are also methods for setting the number of decimal places:
setMaximumFractionDigits(integer);
setMinimumFractionDigits(integer);

Any digits beyond the maximum will be rounded off.

FILE MANAGEMENT

The File class

The File class provides a facility for retrieving information from the file system or for managing files. For the purposes of the File class, directories are treated as files.

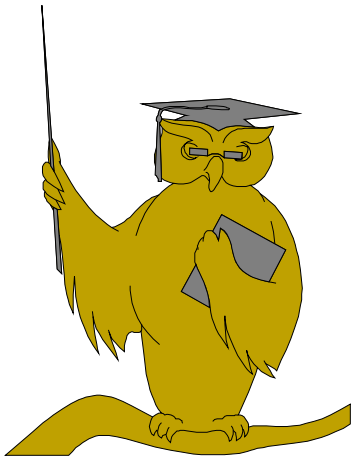
An instance of the File object is set up to point to a file or directory, and the methods of the object can then be used. To set up a File object:

File *objectname* = new File("*pathname*");

The methods of *objectname* can then be used. Some of the more useful methods are:

createNewFile()	creates a file
exists()	Returns true if the file exists
isDirectory()	Returns true if the file is a directory
isFile()	Returns true if the file is not a directory
getAbsolutePath()	Returns the full path to the file
mkDir()	Makes a new directory
length()	Returns the length of the file
list()	If the file is a directory, returns an array of strings containing all the files held in the directory
renameTo(dest)	Renames the file to the name specified by dest.

Review Questions



1.	What is the purpose of filtering?
2.	What is the advantage of the Data streams?
3.	What are the problems encountered when reading or writing character data? What I/O streams handle these problems?
4.	What is the advantage of buffered input and output?
5.	What two streams are usually used for keyboard input?

INTRODUCTION TO DATABASES

Learning Module Objectives

When you have completed this learning module you will have seen how to:

- Connect to a database
- Execute a query and extract data from the result set
- Insert rows into the database

DATABASE CONNECTIONS

Introduction

Java provides support for industry-standard databases which respond to SQL statements via JDBC (Java Database Connectivity).The JDBC-ODBC bridge allows Java to interface with any ODBC database. However, this connection is somewhat inefficient, since two drivers are required to process database access. Many database suppliers such as Informix, Oracle and Sybase provide JDBC interfaces which communicate directly with the database server. However, these are not part of the Java language and must be obtained from the vendor.

From a programming point of view, all databases are treated alike. This course uses a connection to Microsoft Access to illustrate database concepts, but, other than the setting up of the URL which implements the database connection, all databases will behave in the same way.

Preparing to use an Access database with Java

These steps are specific to Microsoft Access. For other databases, contact the Systems Administrator.

For Access and other ODBC databases running through Windows:

- From the control panel, select ODBC 32 bit
- This will bring up the ODBC Source Administrator
- From the User DSN tab, choose Add
- This will bring up the Create new Data Source window.
- Choose Microsoft Access and click Finish
- This will bring up the Microsoft Access setup window.
- Enter a data source name (DSN) and description. The name will be used to refer to the database when setting up an URL for the connection.
- In the Database panel, choose Select. This will bring up a file browser from which the database can be selected. Select the database and click OK
- From the Access setup screen, click OK
- From the Data Source Administrator screen, click OK

Checking for exceptions

Any database statement in Java can throw one or more SQLExceptions. Other exceptions such as ClassNotFound can also occur. Anything to do with the database should be set up in a try…catch block which checks firstly for SQLExceptions, and secondly for general exceptions. Since there may be many SQLExceptions, the code could look like this:

```
try {statements …..}
catch(SQLException ex)
{
while (ex!=null)
  {System.out.println("SQL Exception: " + ex.getMessage());
   ex=ex.getNextException();
  }
 }
catch (java.lang.Exception ex)
  {ex.printStackTrace();
  }
```

Setting up the connection within a Java program

Any programs using databases must import the library java.sql.

The first step is to load the correct driver. When using Access, the statement would be:
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");

The next step is to set up an Url to specify the connection. This will be slightly different for other databases, or for accessing a database over a network. Assume that a database on the current machine had been given the data source name of MyData by following the steps listed in the previous paragraph. The Url to access this database would be set up in Java as follows:

String url = "jdbc:odbc:Mydata";

The next step is to establish a connection to the database. This is done using a Connection object which can be obtained from a static method of the DriverManager class using the Url as follows:

Connection DbConnect = DriverManager.getConnection(url, "","");

This will search through all available drivers on the current platform to find one which can process this Url. The connection should now have been established, and the database can be used.

When no more database activity is required, the connection should be closed as follows:

DbConnect.close();

QUERIES

Introduction

SQL statements are passed to the database server, which processes them and returns one or more rows. These rows are placed in a result set, from which information can be retrieved by the program.

Building an SQL statement

SQL statements are processed as strings. For simple SQL statements such as Select * from Customer; a String variable can be defined containing this text i.e.
String Query = "Select * from Customer";

Often, the query will need to incorporate the contents of a variable. For example, a program may allow the user to type in an employee number from the keyboard, and then search for a matching employee in the database. String concatenation can be used to build up this type of query, remembering to include all commas, quotes and brackets required by the SQL syntax.

If the employee number typed in by the user had been stored in a string named EmpNo, the query could be built up as follows:

String Query = "Select * from Employee where EmployeeNo = '";
String FullQuery = Query + EmpNo + "';";

This will incorporate the current contents of EmpNo into the query string.

Executing an SQL statement

The Statement class can be used to create an object capable of passing SQL statements to the database connection. This is created as follows:

Statement SQLCommand = DBConnect.createStatement();

DBConnect was the name allocated when the connection was established.

The Statement object SQLCommand is now ready to use for executing SQL statements.

To activate the query stored in the string FullQuery, placing the results in a result set named QueryResults, the statement would be:

ResultSet QueryResult=SQLCommand.executeQuery(FullQuery);

The ResultSet class is used to create an object which will contain the results of a query.

Retrieving information from the result set

The result set may contain one or many rows, or even no rows, depending on the type of query.

Result sets contain a method **next()** which sets a pointer to the next row in sequence, returning a boolean value which will be false if there are no more rows.

Within each row of the result set, the columns are stored in an array of strings. Individual columns may be retrieved using the getString method of the result set. getString takes one parameter, which is an integer containing the column number required, and returns the column as a string.

If a query which had just been executed returned two columns, an employee number and a name, and may have returned many rows, the information could be accessed as follows:

```
while (QueryResult.next())
  {
  String Empno=QueryResult.getString(1);
  String EmpName=QueryResult.getString(2);
   }
```

Sample Program Notes

This program retrieves the account number, surname and first names for all rows in a Customer table in a database pointed to by a DSN of JavaCourse

```
import java.sql.*;
public class test1
{
  public static void main(String[] args)
  {

    String url = "jdbc:odbc:JavaCourse";
    String query =
    "select Account, Surname, Firstname from Customer";

    try
    {
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
     Connection con =DriverManager.getConnection(url, "","");
     Statement stmt = con.createStatement();
     ResultSet rs = stmt.executeQuery(query);
     while (rs.next())
     {
     System.out.println("Account No: " + rs.getString(1));
     System.out.println("Surname:    " + rs.getString(2));
     System.out.println("First Name: " + rs.getString(3));
     }
}
catch(SQLException ex)
    {
     while (ex!=null)
       {
       System.out.println ("SQL Exception: " + ex.getMessage ());
       ex = ex.getNextException();
       }
    }
    catch(java.lang.Exception ex)
    {
    ex.printStackTrace();
    }
 }
}
```

Sample program notes

This second program allows the user to type in an account number, and retrieves the account number, surname and firstname from the database for that account only.

```
import java.io.*;
import java.sql.*;
public class data3
{
```

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

   BufferedReader Kbd = new BufferedReader(new
                                InputStreamReader(System.in));
   String url = "jdbc:odbc:JavaCourse";
   String query =
   "select Account, Surname, Firstname from Customer where Account =
'";

   System.out.print("Please enter Account No: ");
   String Acno=Kbd.readLine();
   String fullquery = query +Acno+"';";
   try
   {
    Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
    Connection con =DriverManager.getConnection(url, "","");
    Statement stmt = con.createStatement();
    ResultSet rs = stmt.executeQuery(fullquery);
       if (rs.next())
       {
        System.out.println("Account No: " + rs.getString(1));
        System.out.println("Surname:    " + rs.getString(2));
        System.out.println("First Name: " + rs.getString(3));
        }
        else System.out.println("Account does not exist");
   }
   catch(SQLException ex)
   {
    while (ex!=null)
      {
      System.out.println ("SQL Exception: " + ex.getMessage ());
      ex = ex.getNextException();
      }
   }
   catch(java.lang.Exception ex)
   {
   ex.printStackTrace();
   }
 }
}
```

Please complete Exercise 22

INSERTS

Changing the database contents

SQL statements which change the database are the Insert, Update and Delete commands. This section illustrates the use of inserts. Updates and deletes are handled in the same way. Having learnt to do inserts, anyone with a knowledge of SQL will also be able to code updates and deletes.

Like queries, the SQL updating statements are passed to the database server by an object of the Statement class. However, a different method is used to execute the statement. This is the **executeUpdate** method. It requires a string containing an SQL statement as a parameter, and returns an integer holding the number of rows updated. An example would be:

```
int NumRows = QueryCommand.executeUpdate(Query);
```

Building the SQL statement

Since inserts almost always involve placing the contents of program variables into the database, it will be necessary to build the query string by concatenating text and variables as was seen in the previous section.

Sample program notes

The following program accepts an account number and a surname from the keyboard, and uses them to insert a new row in the database.

Sample program

```
import java.sql.*;
import java.io.*;
public class data2
{
 public static void main(String[] args)
          throws ClassNotFoundException, IOException
 {
  BufferedReader Kbd = new BufferedReader(new
   InputStreamReader(System.in));
  System.out.println("Enter Account");
  String Acno = Kbd.readLine();
  System.out.println("Enter Surname");
  String Surname = Kbd.readLine();


  String url = "jdbc:odbc:JavaCourse";
  String query = "Insert into Customer(Account,Surname) values ('";
  String fullquery;
  try
  {
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   Connection con =DriverManager.getConnection(url, "","");
   Statement stmt = con.createStatement();
    fullquery = query +Acno + "'"+"','"+"'" + Surname + "'" + ");";
   System.out.println("");
   int rows = stmt.executeUpdate(fullquery);
   con.close();
  }
```

```
catch(SQLException ex)
   {
    while (ex!=null)
      {
      System.out.println ("SQL Exception: " + ex.getMessage ());
      ex = ex.getNextException();
      }
   }
   catch(java.lang.Exception ex)
   {
   ex.printStackTrace();
   }
 }
}
```

Please complete Exercise 23

Review Questions

1.	What is a JDBC-ODBC bridge?
2.	What is the purpose of the Connection class?
3.	How would you retrieve multiple rows from a result set?
4.	Which class of object sends SQL statements to the database server?
5.	What method would be used to carry out a statement which modifies the contents of the database?

Appendix A: Reserved Words

The following are Java keywords, having special meaning to the compiler, and cannot be used as class or variable names by the programmer.

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	final	finally	float
for	goto	if	implements	import	instanceof
int	interface	long	native	new	package
private	protected	public	return	short	static
strictfp	super	switch	synchronized	this	throw
throws	transient	try	void	volatile	while