UNIT 1 OBJECT ORIENTED METHODOLOGY-1

Structure **Page Nos.**

1.0 INTRODUCTION

Since the invention of the computer, many approaches of program development have evolved. These include modular programming, top-down programming, bottom-up programming and structured programming. The primary motivation in each case has been the concern to handle the increasing complexity of programs to be reliable and maintainable. These techniques became popular among programmers in 1970s and 1980s.

Due to the popularity of C language, structured programming became very popular and was the main technique of the 1980s. Later this technique also failed to show the desired performance in terms of *maintainability, reusability and reliability*.

As a result of this realisation, a new methodology known as Object oriented programming emerges. This approach to program organization and development attempts to eliminate some of the pitfalls of conventional programming by incorporating the best of the structured programming features with several powerful new concepts. This approach speeds the development of new programs, and, if properly used, improves the maintenance, reusability, and modifiability of software.

So, the major concern for all of us is to know what it is. What are the main features of this approach? How is it better than other approaches? What are the languages which support its various features?

In this unit, we will start with a brief discussion of the manner in which different languages have been developed so as to understand where an Object Oriented programming language fits in. Subsequently, we compare the Object Oriented approach with the procedure-oriented approach. We will also introduce the basic concepts and terminology associated with the Object Oriented (OO) approach. Finally we will talk about common OO languages and applications of OOP in various problem domains.

1.1 OBJECTIVES

After going through this unit, you should be able to:

- find the importance of OO approach;
- define the basic concepts of OO approach;
- differentiate between object and procedure-oriented approaches;
- know about various OO languages;

- describe the applications of OOP, and
- understand the benefits of OO approach.

1.2 PARADIGMS OF PROGRAMMING LANGUAGES

The term *paradigm* describes a set of techniques, methods, theories and standards that together represent a way of thinking for problem solving. According to [Wegner, 1988], paradigms are "**patterns of thought for problem solving".**

Language paradigms were associated with classes of languages. First the paradigms are defined. Thereafter, programming languages according to the different paradigms are classified. The language paradigms are divided into two parts, **imperative** and **declarative** paradigms as shown in the *Figure 1*. Imperative languages can be further classified into **procedural** and **object oriented** approach. Declarative languages can be classified into **functional languages** and **logical languages.** In *Figure1* the examples of languages in each category are also given.

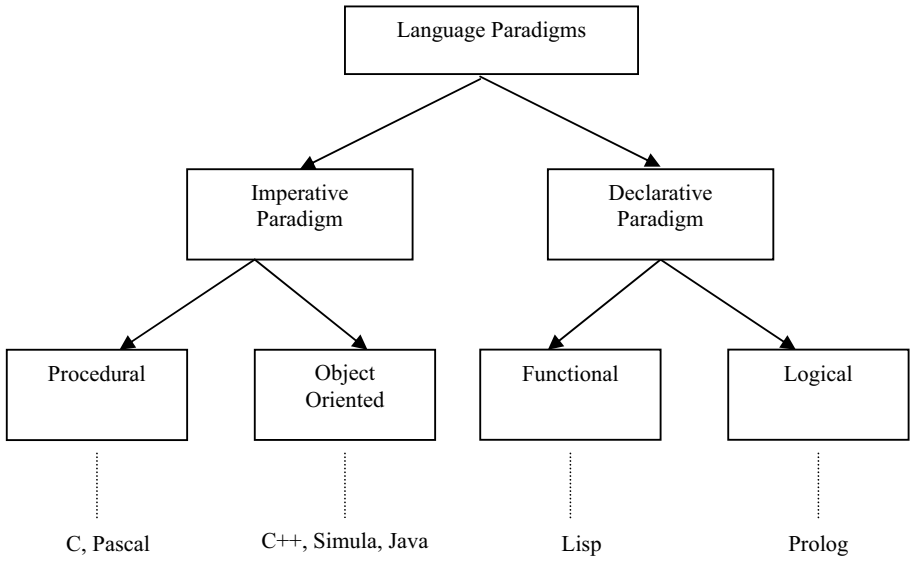


Figure 1: Language Paradigms

Imperative paradigm: The meaning of imperative is **"expressing a command or order"**, so the programming languages in this category specify the step-by-step explanation of command. Imperative programming languages describe the details of **how** the results are to be obtained, in terms of the *underlying machine model*. The programs specify step by step the entire set of transitions that the program goes through. The program starts from an initial state, goes through the transitions and reaches a final state. Within this paradigm we have the procedural approach and Object Oriented approach.

Procedural paradigm: Procedural languages are **statement oriented** with the variables holding values. In this language the execution of a program is modeled as a series of *states of variable* locations. We have two kinds of statements. **Non-executable** statements allocate memory, bind symbolic names to absolute memory locations, and initialize memory. **Executable** statements *like* computation, control flow, and input/output statements. The popular programming languages in this category are *Ada, Fortran, Basic, Algol, Pascal, Cobol, Modula, C, etc.*

Object Oriented paradigm: The Object Oriented paradigm is centered on the concept of the object. Everything is focused on objects. Can you think *what is an*

object? We will discuss the concept of object in detail in further sections. In this language, program consists of two things: first, a set of objects and second the way they interact with each other. Computation in this paradigm is viewed as the simulation of real world entities. The popular programming languages in this paradigm are C++, Simula, Smalltalk and Java.

Declarative paradigm: In this paradigm programs declare or specify what is to be computed without specifying how it is to be achieved. Declarative programming is also known as *Value-oriented programming*. Declarative languages describe the relationships between variables in terms of functions and inference rules. The language executor applies a fixed method to these relations to produce a desired result. It is mainly it is used in solving artificial intelligence and constraint-satisfaction problems. Declarative paradigm is further divided into two categories, **functional** and **logical** paradigms.

Functional paradigm: In this paradigm, a program consists of a **collection of functions**. A function just computes and returns a value. A program consists of calling a function with appropriate arguments, but any function can make use of other functions also. The main programming languages in this category are **Lisp, ML, Scheme**, and **Haskell.**

Logic paradigm: In this paradigm programs only explain what is to be computed not how to compute it. Here program is represented by a set of relationships, between objects or property of objects known as predicate which are held to be true, and a set of logic/clauses **(**i.e. if A is true, then B is true**)**. Basically logic paradigm integrates data and control structures. The **Prolog** language is perhaps the most common example. **Mercury** language is a more modern attempt at creating a logic programming language.

1.3 EVOLUTION OF OO METHODOLOGY

The earliest computers were programmed in **machine language** using **0** and **1**. The mechanical switches were used to load programs. Then, to provide convenience to the programmer, **assembly language** was introduced where programmers use **pneumonic** for various instructions to write programs. But it was a tedious job to remember so many pneumonic codes for various instructions. Other major problem with the assembly languages is that they are machine architecture dependent.

To overcome the difficulties of Assembly language, **high-level languages** came into existence. Programmers could write a series of English-like instructions that a compiler or interpreter could translate into the binary language of computers directly.

These languages are simple in design and easy to use because programs at that time were relatively simple tasks like any arithmetic calculations. As a result, programs were pretty short, limited to about a few hundred line of source code. As the capacity and capability of computers increased, so did the scope to develop more complex computer programs. However, these languages suffered the limitations of reusability, flow control (only goto statements), difficulty due to global variables, understanding and maintainability of long programs.

Structured Programming

When the program becomes larger, a single list of instructions becomes unwieldy. It is difficult for a programmer to comprehend a large program unless it is broken down into smaller units. For this reason languages used the concept of functions (or subroutines, procedures, subprogram) to make programs more comprehensible.

A program is divided into **functions** or **subroutines** where each function has a clearly defined purpose and a defined interface to the other functions in the program. Further, a number of functions are grouped together into larger entity called a **module**, but the principle remains the same, i.e. a grouping of components that carry

out specific tasks. Dividing a program into functions and modules is one of the major characteristics of structured programming.

By dividing the whole program using functions, a structured program minimizes the chance that one function will affect another. Structured programming helps the programmer to write an **error free code** and **maintain control** over each function. This makes the development and maintenance of the code **faster** and **efficient**.

Structured programming remained the leading approach for almost two decades. With the emergence of new applications of computers the demand for software arose with many new features such as **GUI (Graphical user interface)**. The complexity of such programs increased multi-fold and this approach started showing new problems.

The problems arose due to the fundamental principle of this paradigm. The whole emphasis is on doing things. Functions do some activity, maybe a complex one, but the emphasis is still on doing. *Data are given a lower status*. For example in banking application, more emphasis is given to the function which collects the correct data in a desired format or the function which processes it by doing some summation, manipulation etc. or a function which displays it in the desired format or creates a report. But you will also agree that the important part is the data itself.

The major drawback with structured programming are its primary components, i.e., *functions and data structures.* But unfortunately *functions and data structures* do not model the real world very well. Basically to model a real world situation data should be given more importance. Therefore, a new approach emerges with which we can express solutions in terms of real world entities and give due importance to data.

Object Oriented programming

The world and its applications are not organized as functions and values separate from one another. The problem solvers do not think about the world in this manner. They always deal with their problems by concentrating on the **objects**, their **characteristics** and **behavior**.

The *world is Object Oriented,* and Object Oriented programming expresses programs in the ways that model how people perceive the world. *Figure 2* shows different real world objects around us which we often use for performing different functions. This shows that problem solving using the objects oriented approach is very close to our real life problem solving techniques.

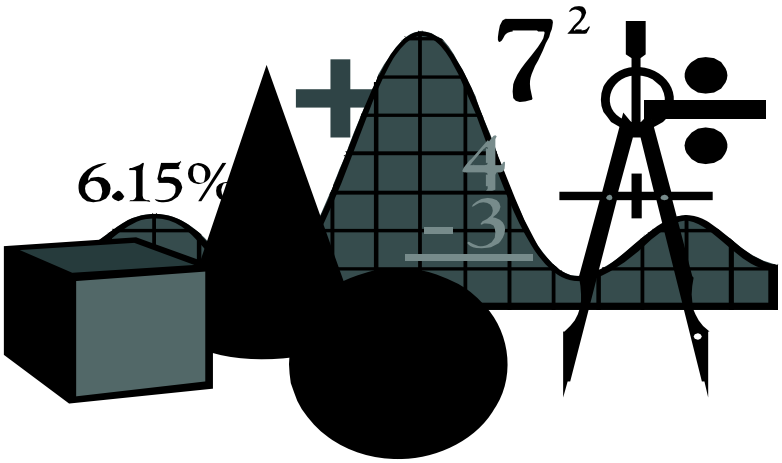


Figure 2: Real world objects

The basic difference in Object Oriented programming **(OOP)** is that the program is organized around the data being operated upon rather than the operations performed. The basic idea behind OOP is to *combine both, data and its functions* that operate on the data into a single unit called **object**. Now in our next section, we will learn about the basic concepts used extensively in the Object Oriented approach.

1.4 BASIC CONCEPTS OF OO APPROACH

Object Oriented methods are favored because many experts agree that Object Oriented techniques are more disciplined than conventional structured techniques. (Martin and Odell 1992)

The main components of Object Oriented technology are '**objects and classes**', '**data abstraction and encapsulation**', '**inheritance**' and '**polymorphism**'. It is very important for you to understand these concepts. Further, in this unit you can find the details of these concepts.

Objects

Fig. 3: Dog object and its behavior

Let's start with **"Object".** The first thing that we should do in the Object Oriented approach is to start thinking in terms of Objects. The problem to be solved is divided into objects. Start analyzing the problem in terms of objects and the nature of communication between them. Program object should be chosen such that they match closely with real-world objects. Let's start creating objects using real-life things, for example, the **dog.** You can create an object representing a dog, It would have data like **How hungry is it**? **How happy is it**? **Where is it**? Now think what are the different functions you can perform on a dog, like **eat, bark, run** and **dig.** Similarly, the following can be treated as objects in different programming problems:

- Employees in a payroll system
- Customers and accounts in a banking system
- Salesman, products, customers in a sales tracking system
- Data structures like linked lists, stacks, etc.
- Hardware devices like magnetic tape drive, keyboard, printer etc.
- GUI elements like windows, menus, events, etc. in any window-based application.

Each object contains ***data and the functions*** that operate on the data. Objects can interact without having to know details of each other's data or functions. It is sufficient to know the type of message accepted and the type of response returned by the object. *For example,* in the banking system, customer object may send a message named as **check balance** to the account object to get the response, i.e. bank balance. An Object Oriented system can be considered as **network of cooperating objects** which interact by sending messages to each other. Let's see in the *Figure 4*, how objects interact by sending messages to one another.

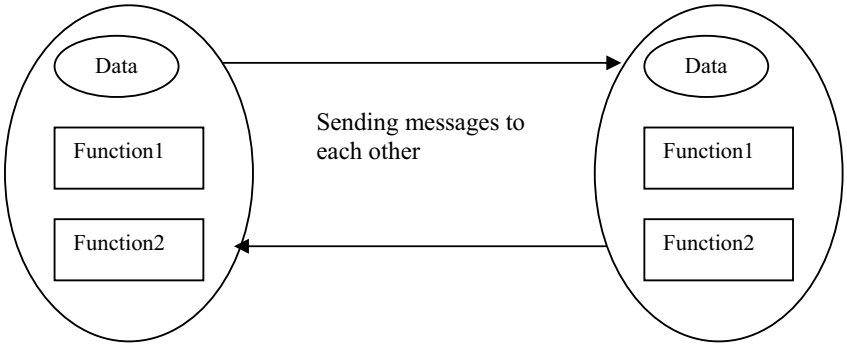


Figure 4: Message Passing

Classes

Objects of the similar type can be grouped together to form a class. Can you tell to which class **dog** belongs? Yes, of course, it belongs to the **animal** class. Now, let us concentrate on the creation of objects. This can be easily answered if we look at the way of creating any variable in common programming languages. Almost all computer languages have **built-in data types,** for example integer, character, real, boolean, etc. One can declare as many variables of any built-in type as needed in any

problem solution. In the similar way one can define many objects of the same class. You can take a class as a type created by a programmer.

A class serves as a **plan or template**. The programmer has to specify the entire set of data and functions for various operations on the data for an object as a user-defined type in the form of a class. In other words, **the programmer defines the object format and behavior by defining a class**. The compiler of that language does not know about this user-defined data type. The programmer has to define the data and functionality associated with it by designing a class.

Finally, defining the class doesn't create an object just as the existence of a built-in type integer doesn't create any variable. Once the class has been defined, you can create any number of objects belonging to that class.

A class is thus a *collection of objects of similar type*. For example, in a collection of potatoes each individual potato is an object and belongs to the class potato. Similarly, each individual car running on the road is an object, Collectively these cars are known as cars.

Data abstraction and encapsulation

The wrapping up of data and functions into a single unit is known as **encapsulation**. This is one of the strong features of the object oriented approach. The data is not directly accessible to the outside world and only the functions, which are wrapped in the class, can access it. Functions are accessible to the outside world. These functions provide the interface to access data. If one wants to modify the data of an object, s/he should know exactly what functions are available to interact with it. This insulation of the data from direct access by the program is known as **data hiding**.

Abstraction refers to the act of representing essential features without including the background details to distinguish objects/ functions from other objects/functions. In case of structured programming, **functional abstraction** was provided by telling, ***which task is performed by function*** **and** ***hiding how that task is performed***. A step further, in the Object Oriented approach, classes use the concept of **data abstraction**. With data abstraction, data structures can be used without having to be concerned about the exact details of implementation. As in case of built-in data types like integer, floating point, etc. The programmer only knows about the various operations which can be performed on these data types, but how these operations are carried out by the hardware or software is hidden from the programmer. Similarly in Object Oriented approach, classes act as ***abstract data types***. Classes are defined as a set of attributes and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.

Inheritance

Inheritance is the process by which objects of one class acquire the properties of objects of another class in the hierarchy. For example, the scooter is a type of the class two-wheelers, which is again a type of (or kind of) the class motor vehicles. As shown in the *Figure 5* the principle behind it is that the derived class shares common characteristics with the class from which it is derived.

New classes can be built from the existing classes. It means that we can add additional features to an existing class without modifying it. The new class is referred as **derived class** or **sub class** and the original class is known as **base class** or **super class**. Therefore, the concept of inheritance provides the idea of **reusability.** This inheritance mechanism allows the programmer to reuse a class that is made almost, but not exactly, similar to the required one by adding a few more features to it.

As shown in *Figure 5*, three classes have been derived from one base class. Feature A and Feature B of the base class are inherited in all the three derived classes. Also, each derived class has added its own features according to the requirement. Therefore, new classes use the concept of reusability and extend their functionality.

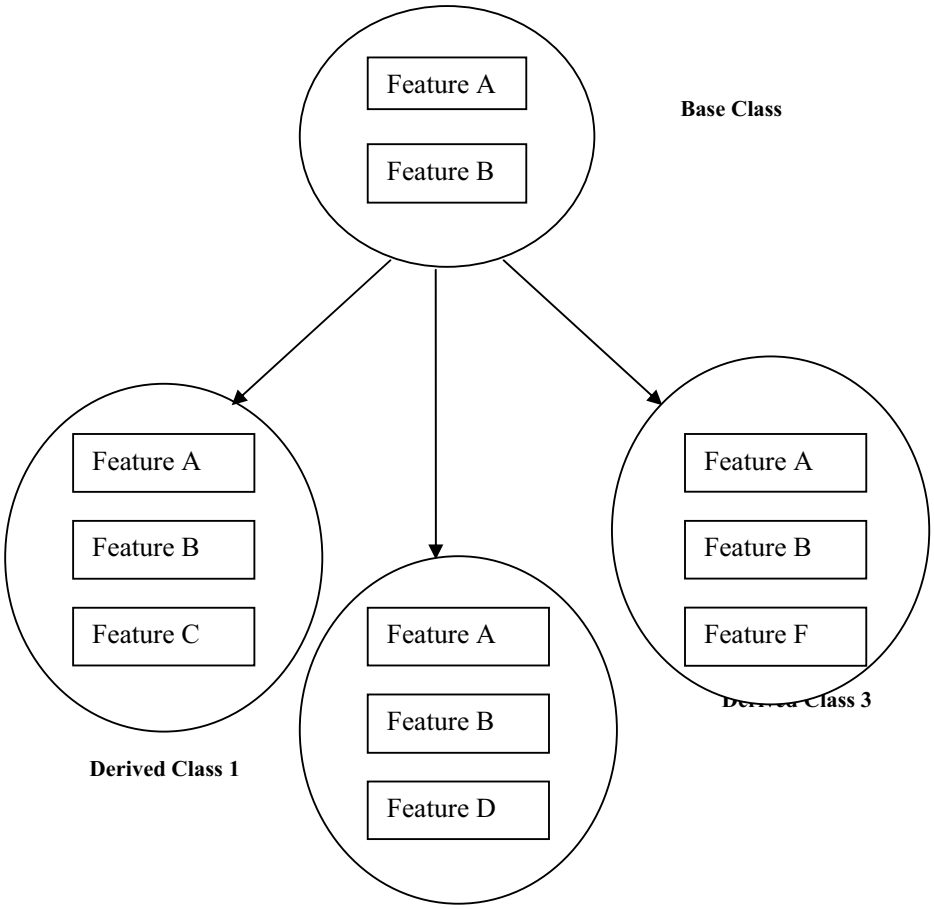


Figure 5: Inheritance

Polymorphism

Polymorphism means **the ability to take more than one form of the same property.** For example, consider an addition operation. It shows a different behavior in different types of data. For two numbers, it will generate a sum. The numbers may integers or float. Thus the addition for integers is different from the addition to floats.

An example is shown in *Figure 6,* where single function name, i.e. **draw** can be used to draw *different shapes*. The name is the same in all the classes but the functionality differs. This is known as function overriding, which is a type of polymorphism. We will discuss it in detail in our next unit.

In our example, we also used a function **"area"** which was inherited by all the three derived classes, i.e. **triangle**, **circle** and **rectangle**. But in the cases of the circle and the triangle, we override the function area because the data types and number of parameters varies.

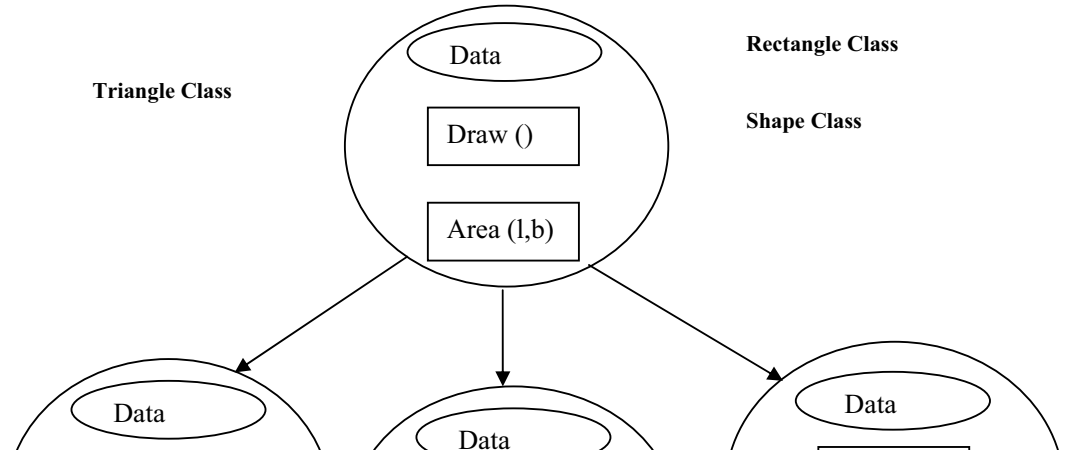


Figure 6: Polymorphism

Check Your Progress 1

1) What do you understand by structured programming?

……

……..……

…

……

……

……

……

2) What is the basic idea of Object Oriented approach?

……

……..……

…

……

……

……

……

3) Differentiate between Data abstraction and data hiding.

……

……..……

…

……

……

……

4) Differentiate between Inheritance and polymorphism.

……

……..……

…

……

……

……

……

1.5 COMPARISON OF OBJECT ORIENTED AND PROCEDURE-ORIENTED APPROACHES

Procedure-oriented approach

A program in a procedural language is *a list of instructions*. Each statement in the language tells the computer to do something. Get some input, do some computational task and finally display the output. This computational task is a function or procedure.

For small programs no other organizing principle is required. The programmer creates the list of instructions, and the computer carries them out. The primary focus is on functions. The program structure can be viewed as shown in *Figure 7.*

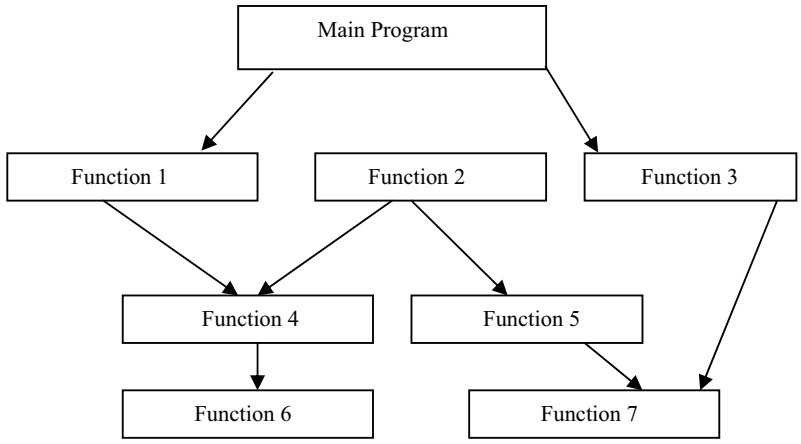


Figure 7: Procedure-Oriented Approach

All conventional programming languages like **Pascal, C, FORTRAN,** and **COBOL** used to model programs based on the procedure re-oriented at approach. In the procedure-oriented approach, the problem is divided into **subprograms** or **modules**. Then functions are defined for each subprogram. Each function can have its own **data** and **logic**. Information is passed between functions using parameters and global variables. Functions can have local variables that cannot be accessed outside the function.

The Programmer starts with thinking *"What do we have to do to solve this problem?* and then s/he does it. Typically it starts with a pseudo code, a flow chart or a **data flow diagram** and continuously refines the design into code. The concentration is more on development of functions. Procedural languages have certain properties, which give rise to some difficulties. The first and the foremost problem is the manner in which functions access global variables. Many important data items are placed as global so that they may be accessed by all the functions. Each function may have its

own local data. The relationship of data and functions in procedure-oriented approach can be viewed as shown in *Figure 8.*

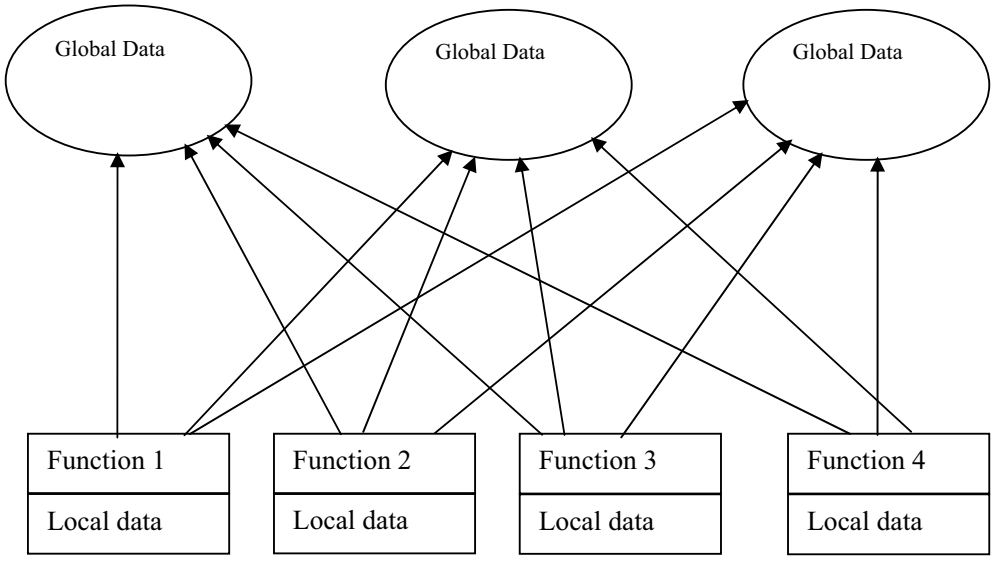


Figure 8: Functions and Global Variables

The following are the two major problems due to global variables:

- **Uncontrolled modification:** Due to these global variables, the risk of unwanted access and modification increases greatly. Any time global variables are available to any function, even to the functions which do not need it. That function may inadvertently access and modify the values without in any way violating the rules of the language. There is no way that this accidental misuse can be avoided and also no method by which we can enforce that the data definition and its use be confined to the functions which actually need them.

- **Problem of enhancement and maintainability:** In a large program it is difficult to keep track of what data is used by which function. If a programmer wants to add some new feature or to make some modification to the existing feature, there are chances that the new code he would add could modify some unprotected global data. If this modification is inappropriate, some part of the program, which is working fine, may stop to do so. Revising this may in turn may need some more modification. This may lead to an opportunity for errors to creep in.

The next issue is the manner in which procedural languages specify the user-defined data type. In these languages, the data type specifies the structure and the operations that can be performed on the variables of that type. For example, in C language, a built-in data type like *int* specifies the structure and the operations applicable to a variable of type *int*. But in these languages, it is not applicable for user-defined. In C language, a data type specifies only the structure of the variables of that type. The operations that are to be performed on a variable of this type are left unspecified. As a result, it may not be clear? What manner the variables are to be operated upon? It is left to the user program to decide the type of operations that are to be performed upon variables of that type.

For example we can specify the employee record using ***struct*** and using it. An array of employees is defined and it is possible to access employee information by accessing each element of the array. To have a richer specification, we can associate operations like ‘compute salary’, ‘print leave details’ etc. In the procedure; oriented approach, the programmer has to define these operations outside the structure.

The next issue is of **reusability**. Since procedural languages are strongly typed, their functions are highly dependent on the type of variables that are being used. This

property hampers reusability. *For example*, if a program for sorting were written for integers, it would not be able to sort real numbers. For that a new program has to be written.

Finally the serious drawback of the procedural approach is that it *does not model the real world problems very well*. The emphasis is more on functions that represents the action or activity and does not really correspond to the elements of the problem.

The major characteristics of the procedure-oriented approach are:

- More emphasis is on doing things.
- It is based on the problem at hand. Sequence or procedure or functionality is paramount.
- Most of the functions share global data which may lead to the following problems:
  - It will not produce software that is easy to maintain,
  - The risk of unwanted access and modification increases.
- It will not result in reusable software.
- It employs top-down approach in program design.
- Works well in small systems.

Object Oriented approach

The major factor, which leads to the development of this new approach i.e, Object Oriented approach is to resolve many problems encountered earlier in the procedural approach.

In this approach, we decompose a problem into a number of entities called objects and then build data and functions around these entities. The notion of "Object" comes into the picture. *'A collection of data and its operations is referred to as an object'*. Data is a vital element in the program development. Data is local to an object. This is encapsulated within an object and is not accessible directly from outside the object. These objects know how to interact with another object through the interface (a set of operations). The organization of data and functions in Object Oriented programs is shown in *Figure 9* given below.

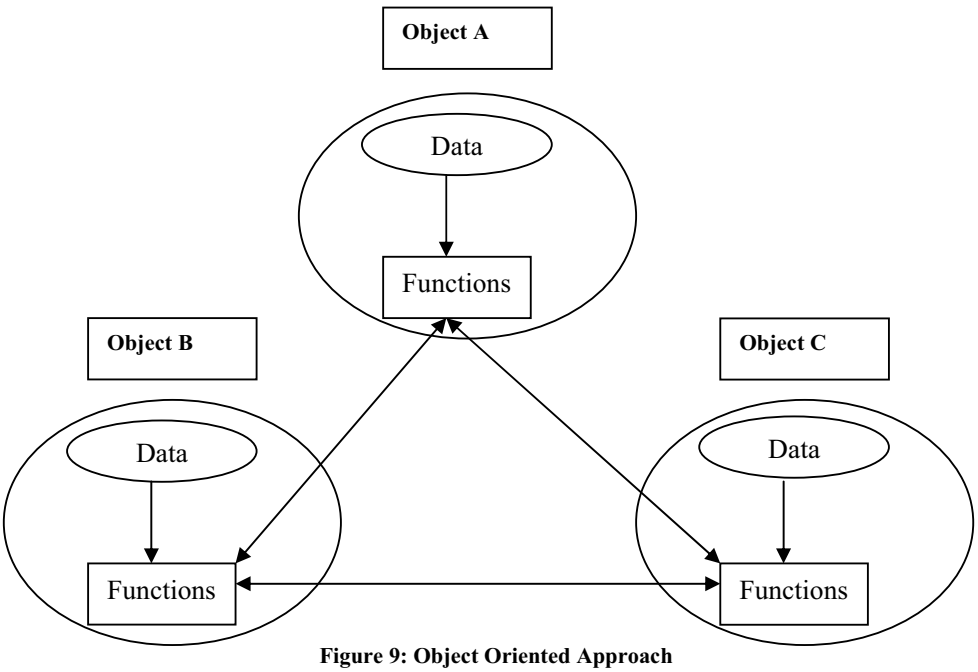


Figure 9: Object Oriented Approach

The salient features of Object Oriented programming are:

- More emphasis is on data rather than procedure.
- Programs are modularized into entities called *objects*.
- Data structures methods characterize the objects of the problem.
- Since the data is not global, there is no question of any operations other than those defined within the object, accessing the data. Therefore, there is no scope of accidental modification of data.
- It is easier to maintain programs. The manner in which an object implements its operations is internal to it. Therefore, any change within the object would not affect external objects. Therefore, systems built using objects are resilient to change.
- Object reusability, which can save many human hours of effort, is possible. An application developer can use objects like 'array', 'list', 'windows', 'menus', 'event' and many other components, which were developed by other programmers, in her program and thus reduce program development time.
- It employs bottom-up approach in program design.

1.6 BENEFITS OF OOPS

OOP offers several benefits to both the program developer and the user. The new technology provides greater programmer productivity, better quality of software and lesser maintenance cost. The major benefits are:

- **Ease in division of job:** Since it is possible to map objects of the problem domain to those objects in the program, the work can be easily partitioned based on objects.
- **Reduce complexity:** Software complexity can be easily managed.
- **Provide extensibility:** Object Oriented systems can be easily upgraded from small to large system.
- **Eliminate redundancy:** Through inheritance we can eliminate redundant code and extend the use of existing classes.
- **Saves development time and increases productivity:** Instead of writing code from scratch, solutions can be built by using standard working modules.
- **Allows building secure programs:** Data hiding principle helps programmer to build secure programs that cannot be accessed by code in other parts of the program.
- **Allows designing simpler interfaces:** Message passing techniques between objects allows making simpler interface descriptions with external systems.

Check Your Progress 2

1) State True or False.

   a) In the procedure-oriented approach, all data are shared by all functions. ☐

   b) One of the major characteristics of OOP is the division of programs into objects that represent real-world entities. ☐

   c) Object Oriented programming language permit reusability of the existing code. ☐

   d) Data is given a second-class status in procedural programming approach. ☐

   e) OOP languages permit functional as well as data abstraction ☐

2) Does procedure oriented language support the concept of class?

……

……..……

…

……

……

……

……

3) Give the reason of accessing data of a class through its functions only.

……

……..……

…

……

……

……

……

1.7 INTRODUCTION TO COMMON OO LANGUAGE

The language should support several of the OO Concepts to claim that they are object oriented. Depending on the features they support, *they can be classified into the following two categories:*

- Object-based programming languages,
- Object Oriented programming languages.

Object-based programming is the style of programming that primarily supports encapsulation and object identity. *Major features that are required for object-based programming are:*

- Data encapsulation
- Data hiding and access mechanisms
- Automatic initialization and clear-up objects
- Polymorphism.

Languages that support programming with objects are said to be object-based programming languages. They do not support inheritance and dynamic binding. Ada is a typical object-based programming language.

We have already introduced to you the concept of data encapsulation, data hiding, polymorphism and inheritance in previous sections of this unit. The access mechanism is implemented through various access specifiers such as public, protected, and private. The function of these specifiers is to decide the scope of a particular member within software. This provides the convenient way for the programmer to decide about the **accessibility** of a member for others.

The other issue is automatic initialization and clear-up objects. Automatic initialization means to give initial values to various variables that denote the state of the object at the time of object creation. This is implemented using ***constructors***. *Finally* when there is no use of the object, we can destroy it. At that time all the resources used by it must be cleared-up. This is implemented using ***destructors***. We will discuss about this in due course.

Let us discuss the concept of **dynamic binding**. Binding refers to the linking of objects to their properties or method call to the code to be executed in response to a call. It can be done at *compile time or run time*. Dynamic binding refers to the linking done at the run (executor) time. We will study this in detail in our coming units of the next blocks.

Object Oriented programming incorporates all of object-based programming features along with additional features such as *inheritance* and *dynamic binding*. Examples of Object Oriented languages are **C++, Smalltalk, object-Pascal,** and **Java**.

Use of a particular language depends on characteristics and requirements of an application, organizational impact of the choice and reuse of the existing programs. *Java is becoming the most widely used general purpose OOP language in the computer industry today.*

1.8 APPLICATIONS OF OOPs

Applications of OOPs are gaining importance. There is a lot of excitement and interest among software developers in using OOPs. The richness of the OOP environment will enable the software industry to improve not only the quality of the software systems but also its **productivity**. Object Oriented technology is certainly changing the way software engineers think, analyze, design and implement systems.

The most popular application using Object Oriented programming is the interface designing for window base systems. Real systems are more complex and contain many objects with a large number of attributes and methods of complex nature. OOP is useful in such cases because it can simplify a complex problem. The application area of OOP includes:

- Object Oriented databases
- Embedded systems
- Simulation and modeling
- Neural networks
- Decision support systems
- Office automation systems
- AI and expert systems
- CAD/CAM systems
- Internet solutions.

Check Your Progress 3

1) State True or False

a) Protecting data from access by unauthorized functions is called data hiding. ☐

b) Wrapping up of data of different types and functions into a unit is known as encapsulation. ☐

c) Polymorphism can be used in implementing inheritance. ☐

d) A Class permits us to build user-defined data types. ☐

e) Object Oriented approach cannot be used to create databases. ☐

2) Explain the advantage of dynamic binding.

……

…

……

……

……

……

……

3) Differentiate between object based and object oriented programming languages

……

……..……

……

……

.

1.9 SUMMARY

OOP is a new way of organizing and developing programs. It eliminates many pitfalls of the conventional programming approach. OOP programs are organized around objects, which contain data and functions that operate on that data. A class is a template for a number of objects. The object is an instance of a class. The major features of OOP are data abstraction, data encapsulation, inheritance and polymorphism. This new methodology increases programmer productivity, delivers better quality of software and lessens maintenance cost. Languages that support several OOP concepts include C++, Smalltalk, Object Pascal and Java.

1.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) In structured programming, a program is divided into functions or modules and each module has a clearly defined purpose and a defined interface to the other functions in the program. Dividing a program into functions and modules is one of the major characteristics of structured programming.

   Here we are least bothered about the data of the problem provided. Our main objective is to achieve control of the execution of program correctly.

2) In Object Oriented programming (OOP), the program is organized around the data being operated upon rather than the operations performed. The basic idea behind OOP is to combine both, data and its functions that operate on the data into a single unit called object.

3) In data abstraction, data structures are used without having to be concerned about the exact details of implementation.

   This insulation of the data from direct access by the other elements of the program is known as data hiding. It is achieved through classes in OOPs.

4) Inheritance is the process by which objects of one class acquire the properties of objects of another class in the hierarchy. By using inheritance, new classes can be built from the existing old classes. It means that we can add additional features to an existing class without modifying it. This inheritance mechanism allows the programmer to reuse a class that is almost, but not exactly, similar to the required one by adding a few more features to it.

Polymorphism means the ability to take more than one form with the same name. Using polymorphism we can have more than one function with the same name but with different functionalities.

Check Your Progress 2

1) False. b) True. c) True. d) True. e) True.

2) Yes procedural languages also support the concept of class, for example, type (data type of the language) is a class and is supported by procedural languages. You know C language support several data types. But procedural languages don't support the user-defined class that has data and functions together.

3) Accessing data of a class through its functions is in basic philosophy of object orientation. If data is not having restrictive access and open to all the principle of data hiding is violated and emphasis on data get reduced.

Check Your Progress 3

1) True. b) True. c) True. d) True. e) False.

2) It gives option of run-time selection of methods on the basis of current input during execution of program. Dynamic binding allows new objects and code to be interfaced with or added to a system without affecting existing code.

3) Object based languages support the notion of objects. Object Oriented languages support the concept of class and permit inheritance between classes.

UNIT 2 OBJECT-ORIENTED METHODOLOGY- 2

Structure **Page Nos.**

2.0 INTRODUCTION

Object-oriented is a philosophy for developing software. In this approach we try to solve the real-world problems using real-world ways, and carries the analogy down to the structure of the program. In simple words we can say the Object-Oriented Methodology is a method of programming in which independent *blocks of codes* or *objects* are built to interact with each other similar to our real-world objects. It is essential to understand the underlying concepts related to an object before you start writing objects and their methods. You need to understand **How objects and classes are related?** and **How objects communicate by using messages?**

OOP
⇩
Type-extensibility
+
Polymorphism

In this unit the concepts **behind** the object-oriented methodologies will be described, which form the **core** of Object-oriented languages like Java and C++.

We will start with a basic concept of *class and object*, then we will learn the concept of **abstraction** and **encapsulation** in relation to the object-oriented methodology. We will also discuss the very important concept of **inheritance**, which helps in *fast* and *efficient* development of programs. Finally we will discuss the concept of **polymorphism**.

2.1 OBJECTIVES

After going through this unit, you should be able to:

- define class and object;
- explain the concept of abstraction and encapsulation;
- explain the concept of inheritance, and
- explain method overriding and polymorphism.

2.2 CLASSES AND OBJECTS

Before we move further, in the very first instance let's define what is object-oriented programming.

Definition: OOP is an approach that provides a way of modularizing programs by creating partitioned memory blocks for both data and functions that can be used as templates for creating copies of such modules on demand.

This means that an object is considered to be a block of computer memory that stores **data** and a set of **operations** that can access the stored data. Since memory blocks are independent, the objects can be used in a variety of different programs without modifications. To understand this definition, it is important to understand the notion of objects and classes.

Now let us discuss the notion of objects and classes.

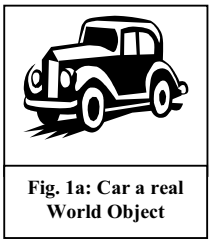


Fig. 1a: Car a real World Object

Objects

Objects are the key to understand object-oriented technology. You can look around and can see many examples of real-world objects like your car, your dog or cat, your table, your television set, etc. as shown in *Figure 1a and 1b*.

Fig. 1b: A real World Object

These real-world objects share two characteristics: They all have *state* and *behavior*. For example, **dogs** have state (name, color, breed, hungry) and behavior (barking, fetching, and wagging tail). **Cars** have state (current gear, current speed, front wheels, number of gears) and behavior (braking, accelerating, slowing down, changing gears).

Because Software objects are modeled after real-world objects, so in that they too have state and behavior. A software object maintains its state in one or more variables. A variable is an item of data named by an identifier. A software object reflects its behavior with the help of method implementation. A method is an implementation way of a function associated with an object.

Definition: *An object is a software bundle of variables and related methods*. You can represent real-world objects by using software objects. For example, you might have observed that real world dogs as software objects in an animation program or a real-world airplane as software object in the simulator program that controls an electronic airplane. You can also use software objects to model abstract concepts.

Variable is a symbol that can hold different values at different times.

For example, an *event* is a common object used in GUI window systems to represent the action of a user pressing a mouse button or a key on the keyboard. The illustration given in *Figure 2,* is a common visual representation of a software object:

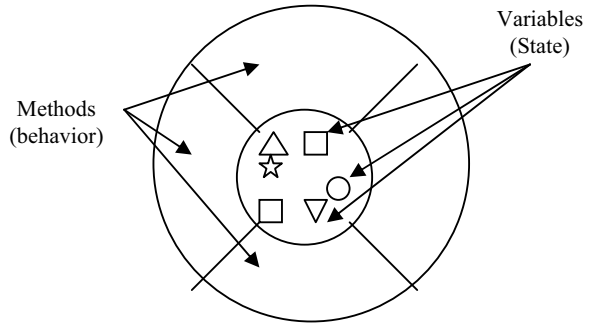
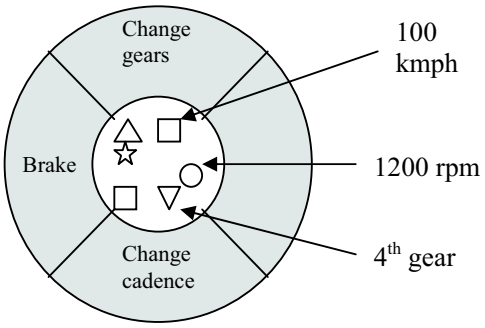


Figure 2: A Software Object

Everything that the software object *knows* (state) and *can do* (behavior) is expressed by the variables and the methods within that object as shown in *Figure 2*. For example, a software object that modeled your real-world **car** would have variables that indicated the car's *current state*: as its speed is 100 kmph, and its current gear is the 5th gear. These variables used to represent the object's state are formally known as **instance variables**, because they contain the state for a particular car object. In object-oriented terminology, a particular object is called an *instance* of the class to which it belongs. In *Figure 3* it *is* shown how a car object is modeled as a software object.



Instance variable: variables of a class, which may have a different value for each object of that class.

Figure 3: A Car Object

In addition to its variables, the software **car** would also have methods to brake, change the cadence, and change gears. These methods are formally known as instance methods because they impact or change the state of a particular car instance.

Class

In the real world, you often have many objects of the same kind. *For example*, *your car* is just one of many cars in the world. In object-oriented terminology, we say that your car object is an instance of the class known as car. Cars have some state (current gear, current cadence, front and rear wheels) and behavior (change gears, brake) in common. However, each car's state is independent of the other and can be different from other vehicles.

When building cars, manufacturers take advantage of the fact that cars share characteristics, building many cars from the same blueprint. It would be very inefficient to produce a new blueprint for every individual car manufactured. The blueprint is a template to create individual car objects.

A software blueprint for objects is called a class.

In object-oriented software, it is also possible to have many objects of the same kind that share characteristics: rectangles, employee records, video clips, and so on. Like the car manufacturers, you can take advantage of the fact that objects of the same kind are similar and you can create a blueprint for those objects.

Definition: *A class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.*

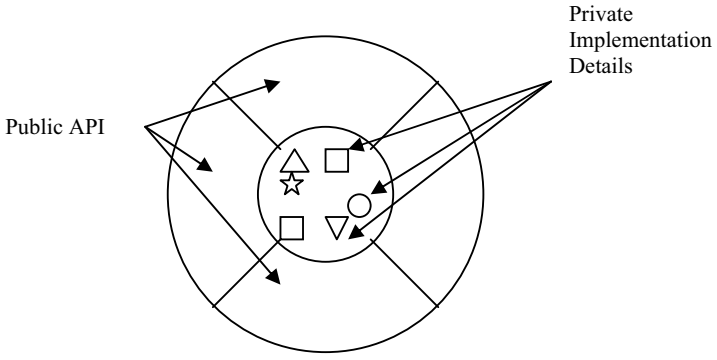


Figure 4: A Class Structure

The class for our car example would declare the instance variables necessary to contain the current gear, the current speed, and so on, for each car object. The class

would also declare and provide implementations for the instance methods that allow the driver to change gears, brake, and show the speed, as shown in *Figure 5.*

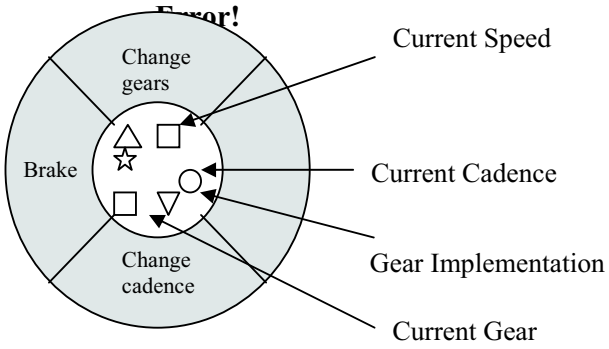


Figure 5: Structure of a "car" Class

After you've created the car class, you can create any number of car objects from the class. For example, we have created two different objects Your Car and My Car (as shown in *Figure 6)* from class car. When you create an instance of a class, the system allocates **enough memory** for the object and all its instance variables. Each instance gets its own copy of all instance variables defined in the class as shown in *Figure 6.*

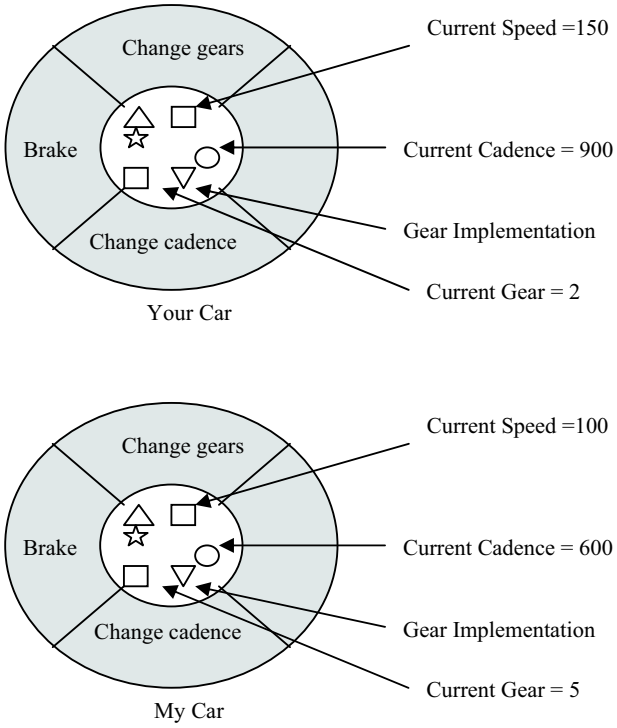


Figure 6: Instance variables of class

In addition to instance variables, **classes can define class variables**. A class variable contains information that is shared by all instances of the class. **For example**, suppose that all cars had the same number of gears (18) as shown in *Figure 7a*. In this case, defining an instance variable to hold the number of gears is inefficient; each instance would have its own copy of the variable, but the value would be the same for every instance. In such situations, you can define a **class variable** that contains the number of gears. All instances share this variable as shown in *Figure 7b*. If one object changes the variable, it changes for all other objects of that class. A class can

also declare class methods. You can invoke a class method directly from the class, whereas you must invoke instance methods in a particular instance.

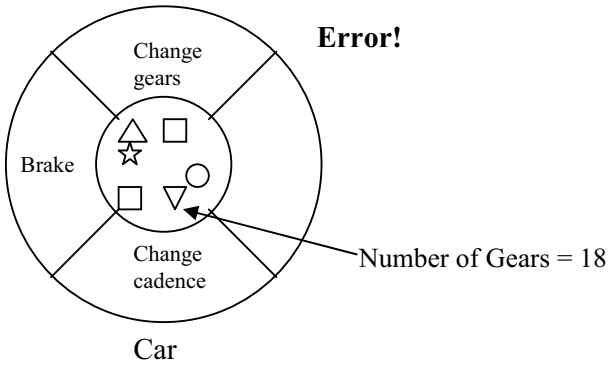


Figure. 7a: Class

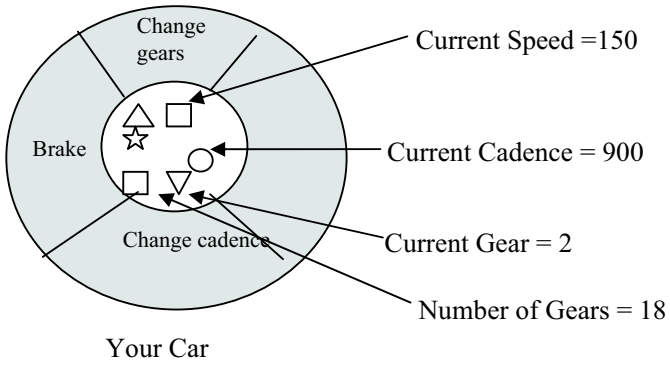


Figure 7b: Instance of a Class

Objects vs. Classes

You probably noticed that the illustrations of objects and classes look very similar. And indeed, the difference between classes and objects is often the source of some confusion. In the real world, it's obvious that classes are not themselves the objects they describe: A blueprint of a car is not a car. However, it's a little more difficult to differentiate classes and objects in software. This is partially because software objects are merely electronic models of real-world objects or abstract concepts in the first place. But it's also because the term "**object"** is sometimes used to refer to both classes and instances.

A single object alone is generally not very useful. Instead, an object usually appears as a component of a larger program or application that contains many other objects. Through the interaction of these objects, programmers achieve higher-order functionality and more complex behavior. Your car standing in the garage is just a product of steel and rubber. By itself the car is incapable of any activity. The car is useful only when another object (may be you) interacts with it (clutch, accelerator).

Software objects interact and communicate with each other by sending **messages** to each other. As shown in *Figure 8* when object **A** wants object **B** to perform one of **B's** methods, object **A** sends a message to object **B** for the same.

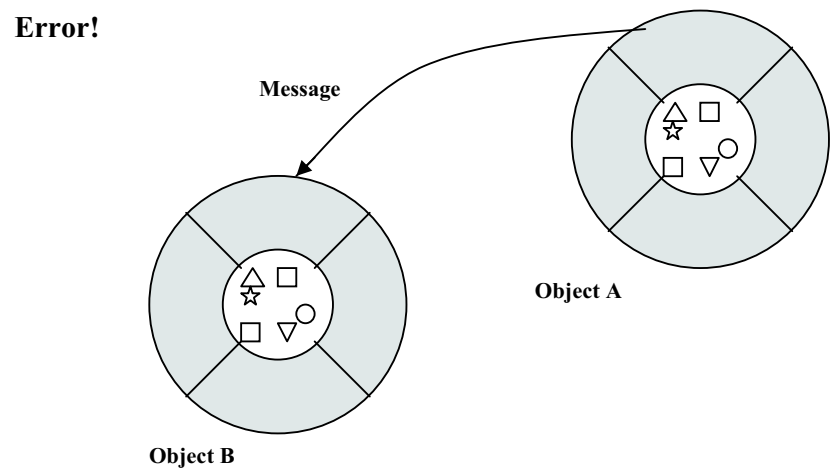


Figure 8: Message passing between objects

Sometimes, the receiving object needs more information so that it knows exactly what to do; for example, when you want to change gears on your car, you have to indicate which gear you want. This information is passed along with the message as *parameters*. The *Figure 9* shows the **three components** that comprise a message:

1. The object to which the message is addressed (Your Car)
2. The name of the method to perform (change Gears)
3. Any parameters needed by the method (lower Gear)

Change Gears (lowerGear)

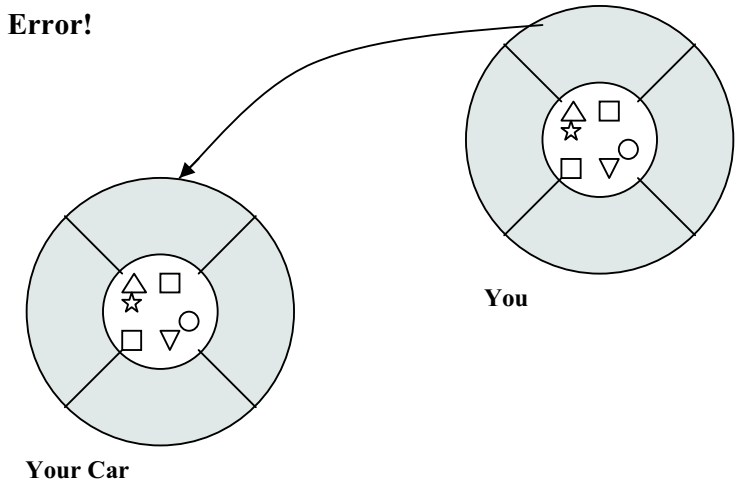


Figure 9: Components of message

These three components contain enough information for the receiving object to perform the desired method. No other information or context is required. Messages provide **two important benefits**.

- An object's behavior is expressed through its methods, so (aside from direct variable access) message passing supports all possible interactions between objects.
- Objects don't need to be in the same process or even on the same machine to send and receive messages back and forth to each other.

☞ Check Your Progress 1

1) What is instance variable? Two different objects of same class can have same value for an instance variable or not?

……

……

……

2) How object and class are associated with each other?

……

……

……

3) Why an object can be used in different programs without modification?

……

……

2.3 ABSTRACTION AND ENCAPSULATION

"*Encapsulation is used as a generic term for techniques, which realize data abstraction. Encapsulation therefore implies the provision of mechanisms to support both modularity and information hiding. There is therefore a one-to-one correspondence in this case between the technique of encapsulation and the principle of data abstraction" -- [Blair et al, 1991]*

The object diagrams show that the object's variables make up the center, or nucleus, of the object. Methods surround and hide the object's nucleus from other objects in the program. *Packaging* an object's variables within the protective custody of its methods is called **encapsulation**. This conceptual picture of an object-a nucleus of variables packaged within a protective membrane of methods-is an ideal representation of an object and is the ideal that designers of object-oriented systems strive for.

However, it's not the whole story. Often, for practical reasons, an object may wish to expose some of its variables to other objects or hide some of its methods from others. In the Java programming language, an object can specify one of four access levels for each of its variables and methods. This feature of Java we will study in Unit 2 of Block 2 of this course. The access level determines which other objects and classes can access the variables or methods of object. Encapsulating related variables and methods into a neat software bundle is a simple yet powerful idea that provides two primary benefits to software developers:

- **Modularity:** The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system. You can give your car to someone else, and it will still work.
- **Information hiding:** An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it. You don't need to understand the gear mechanism on your car to use it.

Therefore, a class is a way to bind the data and its associated methods together. It allows the data (and functions) to be hidden, if necessary, from external use.

Abstraction

"*A view of a problem that extracts the essential information relevant to a particular purpose and ignores the remainder of the information.*'' *-- [IEEE, 1983]*

Abstraction refers to the act of representing essential features without including the background details. All programming languages provide abstractions. Assembly language is a small abstraction of the underlying machine. Procedural languages that followed (such as FORTRAN, BASIC, and C) were abstractions of assembly language. These languages are big improvements over assembly language, but their primary abstraction still requires you to think in terms of the structure of the computer rather than the structure of the problem you are trying to solve. The programmer must establish the association between the machine model (in the "solution space", which is the place *where you're modeling that problem* i.e. computer) and the *model of the problem* that is actually being solved (in the "problem space", which is a place where the problem exists). A lot of effort is required to perform this mapping and it produces programs that are difficult to write and expensive to maintain.

The object-oriented approach provides tools for the programmer to represent elements in the problem space. This representation is general enough that the programmer is not constrained to any particular type of problem. Here we refer to the elements in the problem space and their representations in the solution space as "objects". Object-oriented programming (OOP) allows you to describe the problem in terms of the problem, rather than in terms of the solution or computer where the solution will be executed.

Abstract Data Types: A set of data values and associated operations that are defined independent of any particular implementation.

In object-oriented approach, classes use the concept of **data abstraction**. With data abstraction data structures can be used without having to be concerned about the exact details of implementation. As in case of **built-in data types** like integer, floating point etc. the programmer only knows about the various operations which can be performed using these data types, but how these operations are carried out by the hardware or software is hidden from the programmer.

Classes act as ***abstract data types***. Classes are defined as a set of abstract attributes and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.

When defining a class, we are creating a new *abstract data type* that can be treated like any other built-in data type. Generally, a class specification has two parts:

i. **Class declaration**
ii. **Class method definition**

The class declaration describes the type and scope of its members. The class method definitions describe how the class functions are implemented. We will study about them in detail in the later units of this block.

2.4 INHERITANCE

Now, you are conversant with classes, the building blocks of OOP. Now let us deal with another important concept called **inheritance**. Inheritance is probably the most powerful feature of object-oriented programming.

Inheritance is an ability to derive new classes from existing classes. A derived class is known as subclass which inherits the instance variables and methods of the super

class or base class, and can add some new instance variables and methods. [Katrin Becker 2002].

Generally speaking, objects are defined in terms of classes. You know a lot about an object by knowing its class. If I tell you that the object is a bicycle, you can easily tell that it has two wheels, a handle bar, and pedals.

Object-oriented systems take this a step further and allow classes to be defined in terms of other classes. For example, mountain bikes, racing bikes, and tandems are all kinds of bicycles as shown in *Figure 10.* In OOP, mountain bikes, racing bikes, and tandems are all subclass (i.e. derived class or child class) of the bicycle class. Similarly, the bicycle class is the superclass (i.e., base class or parent class) of mountain bikes, racing bikes, and tandems. This relationship you can see shown in the *Figure 10.*

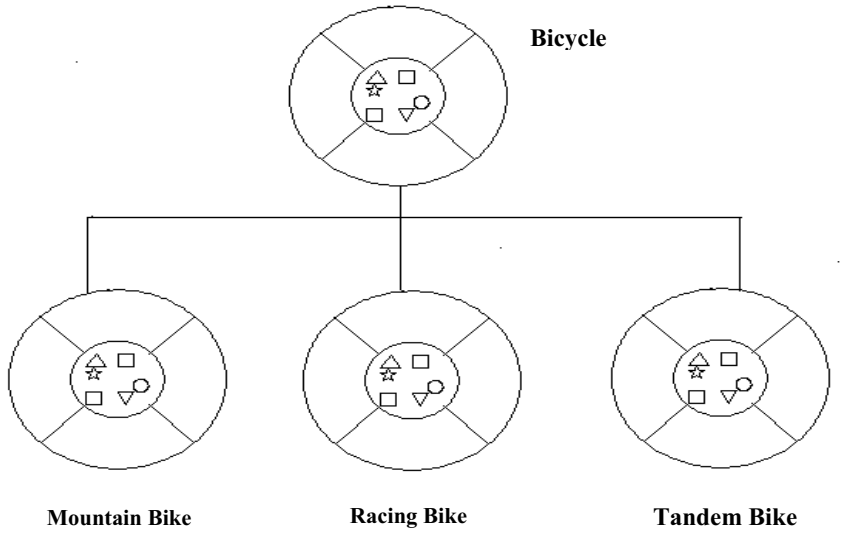


Figure 10: Superclass and Subclasses

Each subclass inherits state (in the form of variable declarations) from the superclass. Mountain bikes, racing bikes, and tandems share some states: cadence, speed, etc. Also, each subclass inherits methods from the superclass. Mountain bikes, racing bikes, and tandems share some behaviors: for example braking and changing pedaling speed.

However, subclasses are not limited to the state and behaviors provided to them by their superclass. Subclasses can add variables and methods to the ones they inherit from the superclass. Tandem bicycles have two seats and two sets of handle bars; some mountain bikes have an extra set of gears with a lower gear ratio.

Subclasses can also override inherited methods and provide specialized implementations for those methods. For example, if you had a mountain bike with an extra set of gears, you would override the change gears method so that the rider could use those new gears.

You are not limited to just one layer of inheritance. The inheritance tree, or class hierarchy can be as deep as needed. Methods and variables are inherited down through the levels. In general, the farther down in the hierarchy a class appears, the more specialized its behavior. Inheritance may have different forms as shown in *Figure 11*:

- **Single Inheritance (Fig. 11a)**: In this form, a subclass can have only one super class.
- **Multiple Inheritance (Fig. 11b)**: This form of inheritance can have several superclasses.
- **Multilevel Inheritance (Fig. 11c)**: This form has subclasses derived from another subclass. The example can be grandfather, father and child.
- **Hierarchical Inheritance (Fig. 11d)**: This form has one superclass and many subclasses. More than one class inherits the traits of one class. For example: bank accounts**.**

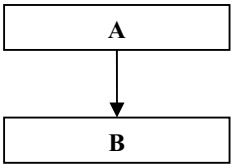


Figure 11a: Single Inheritance

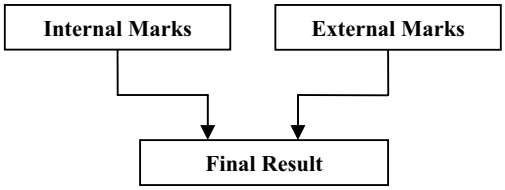


Figure 11b: Multiple-Inheritance

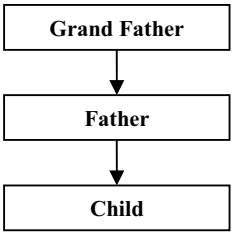


Figure 11c: Multilevel Inheritance

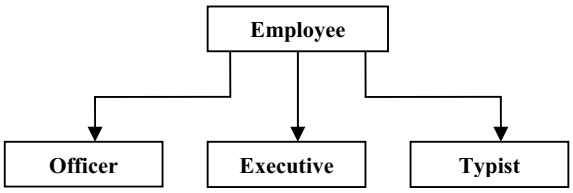


Figure 11d: Hierarchical Inheritance

Figure 11: Different forms of Inheritance

These forms of inheritance could be used for writing extensible programs. The direction of arrow indicates the direction of inheritance. *For example*, in case of single inheritance, traits of class **A** are inherited by class **B**.

☞ Check Your Progress 2

1) What is data abstraction and why classes are known as abstract data types?

……

……

……

2) What are the two main benefits provided by encapsulation to software developers?

……

……

……

3) What is inheritance? Differentiate between multilevel and multiple inheritance.

……

……

……

……

Benefits of Inheritance

- Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times. Once a superclass is written and debugged, it need not be touched again but at the same time can be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability.
- Programmers can implement superclasses called abstract classes that define "*generic*" behaviors. A class for which objects doesn't exist. Example: Further class has no object, but its derived classes-chair, table-that have objects. The abstract superclass defines and may partially implement the behavior, but much of the abstract class is undefined and unimplemented. These undefined and unimplemented behaviors fill in the details with specialized subclasses. Hence, Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program.

The **code reusability** helps in case of distributing class libraries. A programmer can use a class created by another person or company, and, without modifying it, derive other classes from it that are suited to particular programming situations.

2.5 METHOD OVERRIDING AND POLYMORPHISM

A significant addition made to the capabilities of functions or methods in OOP is that of ***method overloading***. With this facility the programmer can have multiple methods with the same name but their functionality changes according to the situation. Suppose a programmer wants to have a method for calculating the absolute value of a numeric argument. Now this argument can be any numeric data type like integer, float etc. Since the functionality remains the same, there is no need to create many methods with different names for different numeric data types. The OOP overcomes this situation by allowing the programmer to create methods with the same name like *abs*. This is called **function overloading**. The compiler will automatically call the required method depending on the type of the argument.

Similarly **operator overloading** is one of the most fascinating features of OOP. For example, consider an addition operation performed by **'+'** operator. It shows a different behavior in different instances, i.e. different types of data. For two numbers, it will generate a sum. Three types may be integer or float.

Let us consider the situation where we want a method to behave in a different manner than the way it behaves in other classes. In such a case we can use the facility of ***method overriding*** for that specific class. For example, in our earlier example of the last section, bicycle superclass and subclasses, we have a mountain bike with an extra set of gears. In that case we would override the "change gears" method of that subclass (i.e. Mountain Bike) so that the rider could use those new gears.

Polymorphism

After classes and inheritance, polymorphism is the next essential feature of OOP languages. ***Polymorphism*** allows one name to be used for several related but slightly different purposes. The purpose of polymorphism is to let one name be used to specify a general class of action. *Method overloading* is one kind of polymorphism. We have already dealt with this type of polymorphism. The other type of polymorphism simplifies the syntax of performing the same operation with the hierarchy of classes. Thus a programmer can use polymorphism to keep the *interface* to the classes clean;

he doesn't have to define unique method names for similar operations on each derived class.

Suppose we have three different classes called *rectangle, circle and triangle* as shown in *Figure 12*. Each class contains a ***draw*** method to draw the relevant shape on the screen. When you call a ***draw*** method through a function call, the required ***draw*** will be executed depending on the class, i.e., rectangle, circle or triangle.

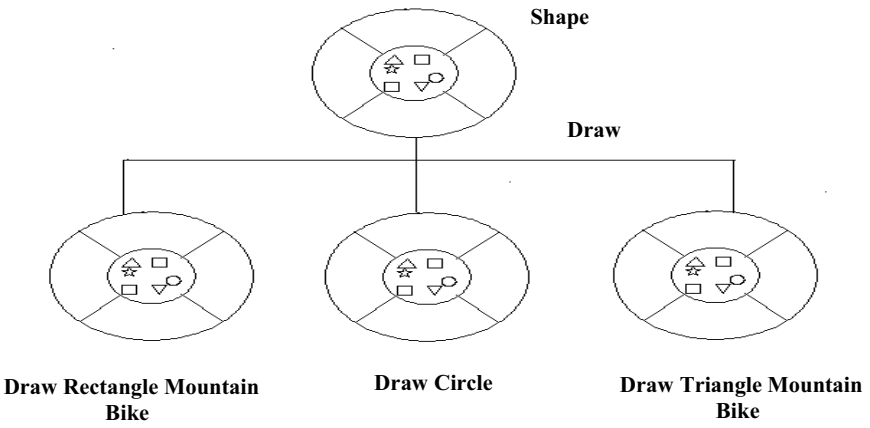


Figure 12: Polymorphism

Thus, Polymorphism plays an important role in allowing objects to have different internal structures (by method overloading or method overriding) but can share the same external interface.

☞ Check Your Progress 3

1) What are the benefits of inheritance?

……

……

……

2) Differentiate between method overloading and method overriding?

……

……

3) Explain message passing with an example.

……

……

2.6 SUMMARY

Object oriented programming takes a different approach to solving problems by using a program model that mirrors the real world problems. It allows you to easily decompose a problem into subgroups of related parts.

The most important feature of an object-oriented language is the object. An object is a logical entity containing *data* and *code* that manipulates that data. They are the basic run time entities. A class is a template that defines the variables and the methods common to all objects of a certain kind. Abstraction consists of focusing on the essential, inherent aspects of an entity and ignoring its accidental properties. Classes use the concept of abstraction and are hence known as **Abstract data type**

(ADT). **Encapsulation** is the mechanism by which data and methods are bound together within the object definition. It is the most important feature of a class. The data is insulated from direct access by the program and it is known as data hiding. **Inheritance** is the process by which an object can acquire the properties of another object in the hierarchy. The concept of inheritance also provides the idea of reusability. New classes can be built from the existing classes. Most of the OOP languages support **polymorphism**. It allows one name to be used for several related but slightly different purposes. It is available in the form of method or operator overloading and method overriding. Polymorphism is extensively used in implementing inheritance.

2.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Every object in the world has its state and behavior. For example, a student object may have name, programme of study, semester etc. The variables name, programmes of a study, semester are known as instance variables. The value of instance variables at a particular time to decide the state of an object.

   Instance variables of different objects of the same class may have same value for example two different student may have same name, but mind it that they are not same.

2) An object is a software bundle of variables and related methods. Everything that the software object knows (state) and can do (behavior) is expressed by the variables and the methods within that object. In object-oriented software, it's also possible to have many objects of the same kind that share characteristics. Therefore, a class is a blueprint, or prototype, that defines the variables and the methods common to all objects of a certain kind.

3) Objects are instance of a class are combination of variables and methods used to represent state and behaviour of objects. One the class is defined its object can be used any where with the properties which are defined for it in class without modification

Check Your Progress 2

1) With data abstraction, data structures can be used without having to be concerned about the exact details of implementation. Classes act as abstract data types as they are defined as a set of abstract attributes and functions to operate on these attributes. They encapsulate all the essential properties of the objects that are to be created.

2) Two main benefits of encapsulation are:

   a) Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Also, an object can be easily passed around in the system.

   b) Information hiding: An object has a public interface that other objects can use to communicate with it. The object can maintain private information and methods that can be changed at any time without affecting the other objects that depend on it.

3) Inheritance: It is the feature by which classes can be defined in terms of other classes. The class which feature is used in defining subclass is known as superclass. Subclasses are created by inheriting the state of super classes this can be seen as reasonability.

Inheritance is seen as generalization to specialization. In multiple inheritance attempt is made to define a specialized feature class which inherit the features of multiple classes simultaneously.

In multi level inheritence specialization is achieved step by step and the last class is the hierarchy is most specialized.

Check Your Progress 3

1) Inheritance offers the following benefits:

   a) Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times. Once a superclass is written and debugged, it need not be touched again but at the same time can be adapted to work in different situations. Reusing existing code saves time and money and increases a program's reliability.

   b) Programmers can implement superclasses called abstract classes that define "generic" behaviors. The abstract superclass defines and may partially implement the behavior, but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses. Hence, Inheritance can also help in the original conceptualization of a programming problem, and in the overall design of the program.

2) With method overloading, programmer can have multiple methods with the same name but their functionality changes according to the situations. Whereas method overriding is used in the situation where we want a method with the same name to behave in a different manner in different classes in the hierarchy.

3) Message Passing – Object communicate with each other by message passing when object A want to get some information from object B then A pass a message to object B, and object B in turn give the information.

   Let us take one example, when student want to get certain information from office file. In this situation three objects student, clerk, and office will come in picture.

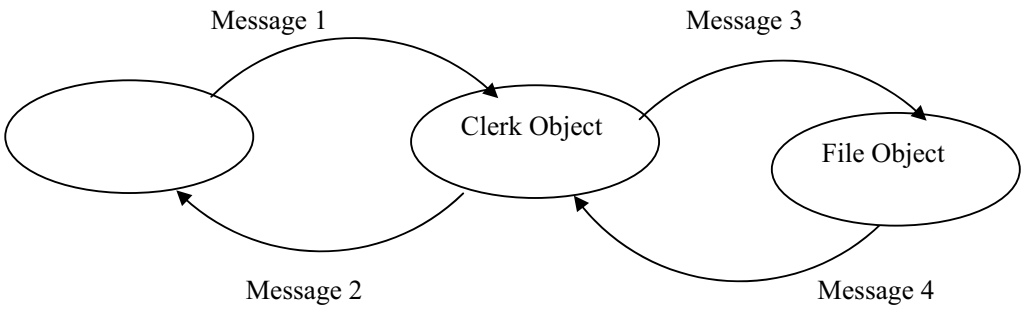


Figure 13: Message Passing

UNIT 3 JAVA LANGUAGE BASICS

Structure **Page Nos.**

3.0 INTRODUCTION

Java is an island in Indonesia, or you can say Java is type of coffee, but Java is most popular as a programming language. In **1991** Java Programming Language stepped into the world and got dominant status. In this unit you will learn the advantages and strength of Java, and what actually makes Java powerful and popular.

Broadly a Java program can be divided into two types, **Application** and **Applets.** The differences between the two are explained in this unit. The concept of Java virtual Machine makes this language platform independent. With this concept of Java virtual machine we will move towards the programming of Java language. You can learn the basic programming concept of Java and how you can compile and execute the Java Application and Applet programs. To improve your programming concepts, you can move towards the declaration and initiations of different data types and variables. Similar to C, Java also contains operators. Those are explained at the end of this unit.

3.1 OBJECTIVES

After going through this unit, you should be able to:

- describe the strength of Java language;
- explain Java virtual machine concept;
- differentiate between Java applet and Java application;
- write simple programs in Java;
- compile and execute Java applets and application programs;
- describe the data types in Java, and
- declare and initialize of variables in the programs.

3.2 INTRODUCTION TO JAVA

Java was developed by a team of computer professionals under the guidance of James Gosling at Sun Microsystems in 1991. They wanted to give a suitable name. Initially they called it **"oak"** after seeing an oak tree from their window, but after some weeks they were discussing another name and were sipping Java coffee, so one of them suggested the name **"Java".** Other than this there is no interesting story/reason about its name **Java.** The only reason I can say is that its designers wanted to give a

beautiful name to their beautiful language, just as all parents want to give a sweet name to their sweet child.

Java is a **simple** (similar to C/C++), **scalable** (easy to integrate), **object oriented** (able to program real life complexities)**, general purpose** programming language with powerful features, which can be used to develop a variety of applications from simple web animations to high-end business applications that program hand-held devices, microwave appliances, cross platform server applications, etc.

Java is a **strongly typed language**. This specification clearly distinguishes between the compile time errors that must be detected at compile time and those that occur at run time.

Generally a language is either compiled or interpreted, but Java is both compiled as well as interpreted. First a Java program is complied and comes in the form of **"Java byte code"** (which is an intermediate code). Then this Java byte code is run on interpreter to execute a program. This byte code is the actual power of Java to make it popular and dominating over other programming languages. We will discuss this topic in more detail in a later section of this unit.

3.2.1 Basic Features

In the last decade Java has become very popular. There are many reasons why Java is so popular and some of these reasons are explained here: **carefully read all the features of Java and try to realize its strength.**

Platform Independent

Java is Platform independent. The meaning of platform here may be confusing for you but actually this word is poorly defined. In the computer industry it typically means some combination of hardware and system software but here you can understand it as your operating system.

Java is compiled to an intermediate form called **Java byte-code** or simply byte code. A Java program never really executes immediately after compilation on the host machine. Rather, this special program called the Java interpreter or Java Virtual Machine reads the byte code, translates it into the corresponding host machine instructions and then executes the machine instruction. A Java program can run on any computer system for which a JVM (**Java Virtual Machine**) and some library routines have been installed. The second important part which makes Java portable is the elimination of hardware architecture dependent constructs. For example, Integers are always four bytes long and floating-point variables follow the IEEE 754.You don't need to worry that the interpretation of your integer is going to change if you move from one hardware to another hardware like Pentium to a PowerPC. You can develop the Java program on any computer system and the execution of that program is possible on any other computer system loaded with JVM. For example, you can write and compile the Java program on Windows 98 and execute the compiled program on JVM of the Macintosh operating system. The same concept is explained in Figure 1 given below.

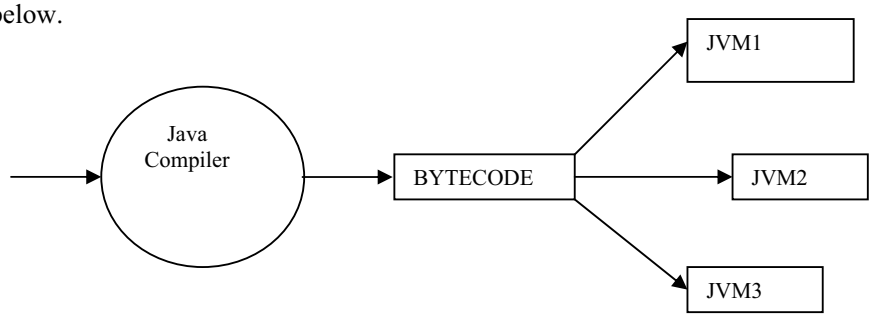


Figure 1: Compilation and execution of Java program

Object Oriented

As you know that in objects represent object-oriented languages data. Objects have two sections. The first is **Data** (instance variables) and the second is **methods**. Data represents what an object is. A method represents what an object does. The Data and methods are closely related to the real world structure and behavior of objects. Object oriented programming has a number of advantages like. Simpler to read program, efficient reuse of programming segments, robust and error-free code.

Java is a true object-oriented language, which provides a platform to develop an effective and efficient application and program real life complexities. Java does not allow methods without class, thus an application consists of only the object which makes it true OOl. Most of the Object-oriented concepts in Java are inherited from C++ which makes it easy for traditional programmers to understand it.

Easy to Learn

Java is easy to learn for programmers because it is (syntax) similar to C and C++ and most of the complex parts of C/C++ have been excluded including **operator overloading, multiple inheritance and pointers**. Approximately half of the bugs in C and C++ programs are related to memory allocation and de-allocation. Therefore the important addition in Java is automatic memory allocation and de-allocation.

Do not think Java is very simple, it is both a simple as well as complex language depending on how you use it because Java has a wide range of applications, simple to complex.

Robust

Java provides checking for possible problems at two levels, one at the compile time and the other at the run time, so programs are highly **reliable** and eliminate situations that are error-prone compared to C/C++. The best and worst features of C and C++ are **pointers** that help in direct manipulation of memory addresses. The power of pointers is as a great tool used by expert programmers for developing system software, driver, etc. But many times pointers are the main cause of **runtime errors** because of improper use of memory. Java eliminates pointer manipulation completely from the language, and therefore eliminates a large source of runtime errors. Java programmers need not remember to de-allocate memory in programs since there is a **garbage collection mechanism** which handles de-allocation of memory. It provides powerful a robust exception handling mechanism to deal with both expected and unexpected errors at run time.

Secure

Java is intended to work in networked and distributed environments by providing security. All the references to memory are **symbolic references,** meaning that the user is not aware where in the memory program is present, it totally depends on the JVM and machine on which the program is running. Each applet is loaded on its own memory space, which avoids the information interchange between applets.

Java applets can be executed in run time environment that restricts them from introducing **viruses**, deleting and modifying files in the host computer. The Java enabled web browser checks the byte code of applets to ensure that it should not do anything wrong before it will run the applet. Furthermore, Java is a **strongly typed language**, which means that variables should be declared and variables should not change types. Type casting are strictly limited highly **sensible, therefore** you can cast an **int** to a **long** or you can cast a **byte** to a **short** but you cannot cast an **int** to a boolean or an **int** to a **String**. The major security issue in today's software world is

BUGS. Unintended bugs are responsible for more data loss than data loss because of viruses. In Java it is easier to write bug-free code then in other languages.

Multi-threaded

Before answering what is multithreading, let me explain you what '**thread**' is. Simply, a thread is a program's path of execution. In your problems, when multiple events or actions need to occur at the same time, how you will handle it? For example, a program is not capable of drawing pictures when you keep pressing keys of the keyboard. The program gives its full attention to receiving the keyboard input and doesn't draw the picture properly.

The best solution to this problem is the execution of two or more sections of a program at the same time and this technique is known as multithreading. Multithreaded applications deliver their potent power by running many threads **concurrently** within a single program. A web browser, for instance, can print a file in the background while it downloads a page in one window and formats the page as it downloads. The ability of an individual program to do more than one thing at the same time is most efficiently implemented through threads.

Java is inherently multi-threaded, for example **garbage collection** subsystem runs as a low-priority thread. A single Java program can have many different threads executing independently and continuously, for example, different Java applets on the same web page can run together with getting equal time from the processor. Because multithreaded applications share data and all threads of an application exists in the same data space therefore to maintaining reliability is sometime difficult. To making easy the use of threads Java offers features for synchronization between threads.

Dynamic

Java was designed to adapt to an evolving environment, therefore the Java compiler is smart and dynamic. If you are compiling a file that depends on other non-compiled files, then the compiler will try to find and compile them also. The compiler can handle methods that are used before they're declared. It can also determine whether a source code has been changed since the last time it was compiled. In Java classes that were unknown to a program when it was compiled can still be loaded into it at runtime. For example, a web browser can load applets of other classes without recompilation.

High Performance

As we know in Java we have to first compile the program, then execute it using Java interpreter. In general, interpreters are slow, because an interpreter executes programs instruction by instruction while Java is a fast-interpreted language. Java has also been designed so that the run-time system can optimize their performance by compiling bytecode to native machine code on the fly (execute immediately after compilation). This is called "**just in time"** (JIT) compilation.

According to 'Sun' with JIT compilation, Java code can execute nearly as fast as native compiled code and maintain its transportability and security but array bounds checking are a problem of the natively compiled Java code. Many companies are an working on native-machine-architecture compilers for Java. These will produce an executable code that does not require a separate interpreter, and that is indistinguishable in speed from C++.

Java offers two flavors of programming, Java applets and Java application. Applets are small Java programs (mostly) that can be downloaded over a computer network and run from a web page by using a Java enabled browser like Netscape / Microsoft Internet Explorer. Applets used to add dynamic features like animation, sound etc. to web pages.

3.2.2 Java Virtual Machine Concepts

When a Java program is compiled it is converted to **byte code** which is then executed by the Java interpreter by translating the byte code into machine instructions.

Java interpreter is part of Java runtime environment. Byte code is an intermediate code independent of any machine and any operating system. Program in Java run time environment, which is used to interpret byte code, is called Java Virtual Machine (JVM). The Java compiler reads Java language source files, translates the source into Java byte codes, and places the byte codes into class files.

Any machine for which Java interpreter is available can execute this byte code. That's why Java is called **Machine independent and Architecture neutral**. *Figure 2* shows that Java compiler is accepting a Java program and producing its byte code. This byte code can be executed on any operating system (Window-98, Macintosh, Linux etc.) running on any machine with suitable Java interpreter of that machine.

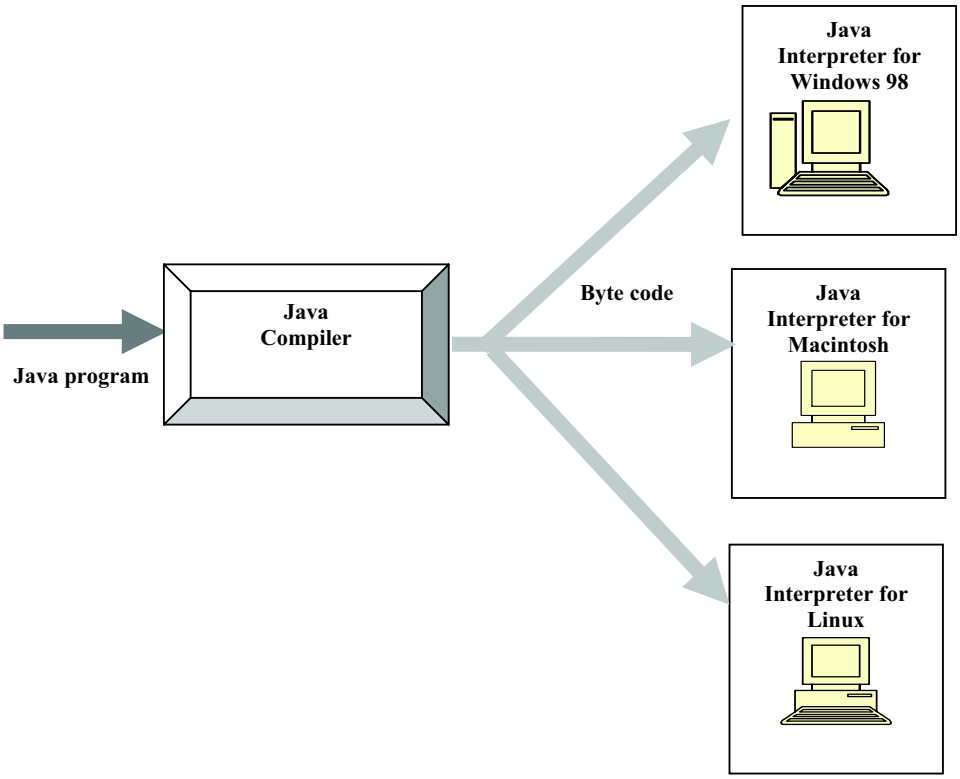


Figure 2: Java is Machine Independent and Architecture Neutral

The JVM plays the main role to making Java portable. It provides a **layer of abstraction** between the compiled Java program and the hardware platform and operating system. The JVM is central to Java's portability because compiled Java programs run on the JVM, independent of whatever hardware is used.

☞ Check Your Progress 1

1) What is the meaning of "virtual" in Java virtual machine?

……

……

2) How can you say that Java is a secure programming language?

……

……

3) What was the first name given to Java by its team?

……

……

In the next section let us see how you can develop a Java program on your own machine. After developing a program try to run it on different machines so that you can understand the meaning of machine Independence in practice.

3.2.3 A Simple Java Program

Java offers two flavors of programming, Java applets and Java application. It is important to understand the basics of both flavors. Let us taste both cups of Java programming one by one.

Java application Program

Let us write your first program with the file name "Application". This program, after successful compilation and run, prints the words "This is My First Java Program" on your display. This program is very basic but the goal of this program is not to teach you how to print words on your display but to tell you how to type, save, compile and execute Java programs. In this section you can learn many issues that can go wrong even if your source code is correct.

To create your own program you should follow the three steps given below. In the next section you will get an explanation of each step in detail:

1. First of all using any text editor, type Java source file of your program as given below.
2. Now compile the source file-using compiler named **Java c** that takes your source file and translates its statements into a bytecode file.
3. Run the program using interpreter named **Java** that takes your bytecode file and translates them into machine instructions that your computer can understand.

// Code start here…. don't include these line numbers in your program

```
1. /* This is my First Java Application Save the file as Application.Java: same as class
name */
2.  class Application
3.        {
4.            public static void main (String args[ ] )
5.                {
6.                    System.out.println("This is My First Java Application");
7.                } // main ends here
8.        }// Code ends here
```

Let us see the above program line by line.

Line 1. /* This is my First Java Program. Save this file as Application.Java same as class name */

Comments in Java are similar to C++. Everything between /* and */ is ignored by the compiler but Comments allow you to describe the details of the program. This is useful for developing the understandability in your program.

Line 2. class Application

Second line in the program defines a class named Application using a keyword class. After that, class definition is specified within curly braces. More about Java classes you can read in Unit 1 Block 1 of this course.

```
class Application
{
.
class definition
}
```
Line 4. public static void main (String args[ ])

Line number 3 is not important to discuss here as it contains only the separator. What use is the separator you will see later.

“public static void main (String args[ ])” This is the point from where the program will start the execution. This program starts the execution by calling main () method. In this line public, static, and void all are keywords. May be you are thinking of the meaning of ‘keyword’. Keywords are nothing but some reserved words, details you will see in the later section of this unit.

The public keyword is used to control the access of various class members. If member is public it can be accessed outside the class. So we have to declare main () as public because it has to be invoked by the code outside the class when program is executed. Static key word allows the main () method to be executed without creating an object of that class, and void means main () method does not return any value.

Separators define the structure of a program. The separators used in Application are parentheses, ( ), braces, { }, the period, ., and the semicolon, ;. Java contains six separators and all are widely used by programmers. Let us see what the proper meaning of each separator is.

Parentheses ( ), generally it encloses arguments in method definitions or calling and used to delimits test expressions control statements.

Braces { }, defines blocks of code and it also defines automatically initializes arrays. Square bracket [ ], declares array types.

Semicolons; are used to terminate statements.

Single coma “,” is use to separates successive identifiers in variable declarations.

Single dot “.” Selects a method from an object and separates package names from sub-package and class names. And in last “:” is used in loop labels.

Line 6. System.out.println(“This is My First Java Application.”);

This line prints the string “This is My First Java Application”. In this line println () function is used to display this line. The println () function accepts any string and display its value on console.

Typing and saving program

To write the program you need a text editor like Notepad, Brief, or vi or any other. You should not use a word processor like Microsoft Word because word processors save their files in a proprietary format and not in pure ASCII text. Now type the above program into a new file but remember to type it exactly as it written here because Java is case sensitive, so **System** is not the same as **system** and **CLASS** is not the same as **class**. But white space is meaningless except inside string literals.

Assume we are using Notepad in windows. Save this program in a file called Application.Java. Windows text editors Notepad add a three letter ".txt" extension to all the files saved without informing user. You may get unexpectedly file called " Application.Java.txt." because it in not “.Java” compiler will not compile this file. If your editor has this problem, you can change to better editor.

Compiling and running Application.Java

Before compiling ensure that your Java environment is correctly configured. To set PATH and CLASSPATH see MCS 025 ‘s Java Programming section.

```
C:> Javac Application.Java
C:> Java Application
This is My First Java Application
C:>
```

Java compiler named Javac that takes your source file and translates its instructions into a bytecode file. Java interpreter named Java that takes your bytecode file and translates them into instructions that your computer can understand. When you compile the program you need to use Application.Java but when you run the program you need not use Application.class.

A Simple Java Applet

As you know Java programs can be classified into applets, application and servlets. *Java servlets are similar programs like applets except they execute on servers side*. **Servlets** are Java programs answer to traditional CGI programming. They are programs that run on a Web server and build Web pages.

Now let us try to write a code for Java applet, When we write a program for applet, we must import two Java groups. Import is a process to tell the compiler where to find the methods of classes from library we will use in the program. These two important groups are Java.awt and Java.applet. Java.awt is called the Java abstract windows tool kit and Java.applet is the applet group.

To create an applet, which will display “Hello IGNOU”, follow the three steps that are used for creating Java application as given below.

1. Create a Java source file.
2. Compile the source file.
3. Run the program.

Write the following Java source code into text editor and save as MyFirstApplet.Java

```
// Code start here… don’t include these line numbers in your program
1. import Java.applet.Applet;
2. import Java.awt.Graphics;
3. public class MyFirstApplet extends Applet
    {
4.    public void paint (Graphics g)
      {
5.        g.drawString("Hello IGNOU !", 50, 25);
6     }
7   }  // please note in applets there is no ‘ main () ‘.
// code ends here….
```

Most of the symatx of this program is similar to previous the Java application program you have just completed. The important steps for developing basic Java applet are explained here step by step.

1. import Java.applet.Applet
2. import Java.awt.Graphics;

The above two lines import two important Java groups. Java.awt and Java.applet are necessary whenever you develop applets. java .awt is called the Java abstract windows tool kit and Java.applet is the applet group. Here in the code java.applet.Applet is required to create an applet and java.awt.Graphics is required to paint on the screen.

3. public class MyFirstApplet extends Applet

My First Applet is the executable class. It is public class and it extends applet class which means that a programmer builds the code on the standard Applet class.

4. public void paint (Graphics g)

In our MyFirstApplet class we have defined paint. Method inside which we are using object of Graphics class. Graphics class is available in Java, group Java.awt.

5. g.drawString ("Hello IGNOU !", 50, 25);

Now in this line drawString method is used to write "Hello Ignou!" message on the screen. It is a method of Graphic class. This drawString takes three arguments. First argument is a message and the other two are pixel positions on X-axis and Y-axis respectively, from where the printing of the string on the screen will begin.

Compiling and running MyFirstApplet.Java

When Java applet is ready, we can compile it using Java compiler named 'Javac'.

To compile this program write:
C:> Javac MyFirstApplet.Java

When compilation is over MyFirstApplet.class is created as an output of compilation. To execute the program you have to create a HTML file which includes the given applet class MyFirstApplet.class, and then run the applet using Web browser or applet viewer.

For example:

```
<HTML>
        <HEAD>
        <TITLE> A Simple Applet </TITLE>
        </HEAD>
        <BODY>
        Here is the output of my program:
        <APPLET CODE=" MyFirstApplet.class" WIDTH=150
        HEIGHT=50>
        </APPLET>
        </BODY>
</HTML>
```

When we open this HTML file using Internet Explorer, you will get window showing the following

Here, the string "Hello IGNOU!" is displayed inside a window of size 150 pixel wide and 50 pixel in height. You will find more detailed discussion on applet programming in Unit 1 Block 4 of this course.

After getting flavors of both application and applet programming you will be thinking that Java application programs and applets are similar but there are many differences between them. **What are these differences?** The differences between them are given below in *Table 1*.

Table 1: Main differences between Java applets and application programs

Java Application Programs	Java Applets
Java Application Program can be executed independently. Applications are executed at command line by Java.exe	Applet cannot be executed independently. Applets can only be executed inside a Java compatible container, such as a browser or appletviewer.
It contain main () method. It has a single point for entry to execution	It does not contain main () method, and does not have a single point of entry for execution.
Applications have no inherent security restrictions, and can perform read/write to files in local System.	Applets cannot perform read/write to files in local system This is to provide security.
Applications have no special support in HTML for embedding or downloading	Applets can be embedded in HTML pages and downloaded over the Internet

☞ Check Your Progress 2

1) Compile and Execute a Java Application that will display the statement " I am Learning Java".

……

……

2) Compile and run a Java Applet of WIDTH=250 and HEIGHT=100, applet will display a message " Soon I will write big programs in Java".

……

……

3) What are the different arguments of 'drawstring'?

……

……

Now you are able to write and execute your own simple programs but going in of programming concepts you need to know the different Keywords and data types available in Java. In the next two sections we will discuss different data types.

3.3 PRIMITIVE DATA TYPE AND VARIABLES

Java supports different primitive types given in *Table 2 where* each primitive data type *is given with its description, group and example.*

Table 2: Java Primitive Types

Data Type	Description	Example	Groups
Byte	Byte-length integer	10, 24	Integer
Short	Short integer	500, 786	
Int	Integer	89, 945, 37865	

Long	Long integer	89L, -945L	
Float	Single-precision floating point	89.5f, -32.5f,	Floating point numbers
Double	Double-precision floating point	89.5, -35.5, 87.6E45	
Char	A single character	‘c’, ‘9’, ‘t’	Characters
Boolean	A boolean value (0 or 1)	True or false	Boolean

In most of the languages, the format and size of primitive data types depend on the platform on which a code is running. But in Java programming language, size and format of its primitive data types is specified. So programmers need not worry about the system-dependencies while using data types. For example, an **int** is always 32 bits, regardless of the particular platform. This allows programs to be written that are guaranteed to run *without porting* on any machine architecture. While strictly specifying the size of an integer may cause a small loss of performance in some environments, it is necessary in order to achieve portability. All the primitive data types can be divided into four major groups as shown in *Table”.*

3.3.1 Java Keywords

A keyword or reserved word in Java has a special meaning and cannot be used as a user defined identifier, because they are used by the compiler and if you use them as variable names, compiler will generate errors. The following *Table 3* gives a list of Java keywords and their purpose in programming. These keywords cannot be used as variable name or function name in any Java program.

Table 3: List of Java keywords

Keyword	Purpose
abstract	It declares that a class or method is abstract
assert	Used to specify assertions
boolean	Declares that variable is Boolean type
break	Used to exit a loop before end of the loop is reached
byte	Declares that variable is byte type
case	To represent different conditions in switch statement
catch	Used for handling exceptions, i.e., capture the exceptions thrown by some actions
char	Declares that variable is character type
class	Signals the beginning of a class definition
const	This keyword is reserved by Java but now it is not in use
continue	Prematurely return to the beginning of a loop
default	Default action in switch statement
do	Begins a do while loop
double	Declares that variable is double type

else	Signals the code to be executed when if condition executes to false
extends	Specifies the class base from which the correct class is inherited
final	Declares that a class may not be extended or that a field or method may not be overridden
finally	Declares a block of code guaranteed to be executed
float	Declares a floating point variable
for	Start a for loop
goto	This keyword is reserved by Java but now it is not in use
if	Keyword to represent conditional statement
implements	Declares that this class implements the given interface
import	permits access to a class or group of classes in a package
Instance of	tests whether an object is an instance of a class
int	Declares an integer variable
interface	signals the beginning of an interface definition
long	Declares a long integer variable
native	Declares that a method that is implemented in native code
new	Allocates memory to an object dynamically
package	Defines the package to which this source code file belongs
private	Declares a method or member variable to be private
protected	Declares a class, method or member variable to be protected
public	Declares a class, method or member variable to be public
return	Returns a value from a method
short	Declares a short integer variable
static	Declares that a field or a method belongs to a class rather than an object
strictfp	To declare that a method or class must be run with exact IEEE 754 semantics
super	A reference to the parent of the current object
switch	Tests for the truth of various possible cases
synchronized	Indicates that a section of code is not thread-safe
this	A reference to the current object
throws	Declares the exceptions thrown by a method
transient	Data should not be serialized
try	Attempt an operation that may throw an exception

void	Declare that a method does not return a value
volatile	Warns the compiler that a variable changes asynchronously
while	Begins a while loop

It is important that Java is case-sensitive, so even though break is a keyword, Break is not a keyword at all.

For the C++ programmers case sensitivity in Java cause problems, but at the time of designing Java its designers decided to **make it case sensitive because** first of all this enhances the readability of cod, secondly it reduces the compilation time and increases the efficiency of compiler. All the Java technology keywords are in lower case. The words true, false, and null are reserved words. Keywords **'const'** and **'goto'** are currently not in use. In this list of Keywords the new Keywords have also been added, for exapmle **Java 1.2** adds the **strictfp** keyword to declare that a method or class must run with exact IEEE 754 semantics. **Java 1.4** adds the **assert** keyword to specify assertions. Each keyword has a specific and well-defined purpose. The following list explains the purpose and meaning of available keywords.

3.3.2 Integer and Floating Point Data Type

Integer & Floating-point numbers fall under Numeric datatypes. Java has six numeric datatypes that differ in size and precision of the numbers they can hold. The Size means how many bits are needed to represent the data type. The Range of a data type expresses the precision of numbers.

Integers

Integers are whole numbers. Depending on range, as given in *Table 4,* integer data type can be further divided into four categories:

Table 4: Four categories of Integer data types

Type	Values	Default	Size	Range
Byte	signed integers	0	8 bits	-128 to 127
Short	signed integers	0	16 bits	-32768 to 32767
Int	signed integers	0	32 bits	-2147483648 to 2147483647
Long	signed integers	0	64 bits	-9223372036854775808 to 9223372036854775807

Floating Point Numbers

A number containing a fractional part is called real number or floating point Number. As given in *Table 5*, Real Number can also be divided into two categories: float and double. But double is used where more accuracy is required for fractional part.

Table 5: Two categories of real numbers

Type	Values	Default	Size	Range

float	IEEE 754 floating point	0.0	32 bits	+/-1.4E-45 to +/- 3.4028235E+38, +/-infinity, +/-0, NAN
double	IEEE 754 floating point	0.0	64 bits	+/-4.9E-324 to +/- 1.7976931348623157E+308, +/-infinity, +/-0, NAN

3.3.3 Character and Boolean Types

Characters

A char is a single character that is a letter, a digit, a punctuation mark, a tab, a space or something similar. **A char** literal is a single one character enclosed in single quotes like
char myCharacter = ‘a’;
char doublequote = ‘ ” ’;

The character datatype, ***char***, holds a single character. Each character is a number or character code that belongs to a character set, an indexed list of symbols. For Example, **ASCII** (American Standard Code for Information Interchange) is a character set. In ASCII character set ranges from **0 to 127** and needs 8 bit to represent a character. Different attributes of character data type are given the *Table 6.*

Java uses the **Unicode** character set. Unicode defines a fully international character set that can represent all of the characters found in all human languages and writing systems around the world such as English, Arabic, Chinese, etc. Since there are a large number of languages, therefore a large set is a required and 8 bits, are not sufficient. This 16-bit character set fulfills the need. Thus, in Java **char** is a 16-bit type.

Table 6: Character data type

Type	Values	Default	Size	Range
Char	Unicode character	\u0000	16 bits	\u0000 to \uFFFF

Booleans

In Java’ Booleans are logical variables, which can contain the value either true or false. Any other value cannot be assigned to a Boolean variable. Different attributes of Boolean data type are given in *Table 7*.

Table 7: Boolean data type

Type	Values	Default	Size	Range
Boolean	True, False	False	1 bit used in 32 bit integer	NA

3.3.4 Declaring and Initialization Variables

Variables (Identifiers)

A variable is a basic unit of storage in a Java Program. Variables represent **memory location** in which value can be stored. Before using any variable, you have to declare it. After it is declared, you can then assign values to it (you can also declare and assign a value to a variable at the same time.)

Java actually has three kinds of variables:

- Instance variables
- Class variables
- Local Variables.

Instance variables are used to define the attributes of a particular object. Class Variables are similar to instance variables, except their values apply to all the instances of a class (and to the class itself) rather than having different values for each object. Local variables are declared and used inside methods, for example, for index counters in loops, temporary variables or to hold values that you need only inside the method definition itself.

Variable Declarations

Before using any variable, it must first be declared. A variable declaration specifies the datatype, the variable name and, optionally, the default value for the variable. A general variable declaration looks like the following:

datatype identifier [= default value] {, identifier [= defaultvalue] };
Consider the following examples for variable declaration:
byte b; short age;
boolean male;
Declare multiple variables of one type in one expression such as in the following example:
int age, enrollnum, numChildren;

Variable declarations can be put anywhere in your code, as long as they precede the first use of the variable. However, it is common practice to place the declarations at the top of each block of code. Variable names in Java can only start with a letter, an underscore (_), or a dollar sign ($). They cannot start with a number. After the first character, your variable names can include letters or numbers or a combination of both.

As you know, the Java language is case sensitive which implies that the variable x in lowercase is different from variable X in uppercase. In Java a variable name cannot start with a digit or hyphen. Variable names should help you understand what is happening in your program. Thus, it is useful to name your variables intelligently or according to its role in the program

The main restriction on the names you can give your variables is that they cannot contain any white space. There is no limit to the length of a Java variable name.

If you want to begin variable name with a digit, prefix underscore, with the name, e.g. _9cats. You can also use the underscore to act like a space in long variable names.

Variable Assignment and Initialization

Once you have declared the type of a variable, you can initialize it with some value.
variable name = some value.
For example, consider the following example:
int year;
year = 28; // here the value 28 is assigned to the variable ‘year’

Dynamic Initialization

We have seen how variables can be declared and assigned a value. However, we can declare and assign a value to the variable in a single statement. This is called **Dynamic initialization**. Variables can be dynamically initialized using any expression valid at the time the variable is declared. This can be made clearer by looking at the example code given below:
// Demonstrate dynamic initialization.

/* This Program calculates the hypotenuse of a triangle with height of the triangle, h base b, using formula: square root of [square of height h + square of base b] */

```
class DynamicInitialization   {
    public static void main (String args [ ] )
                        {
                           double  h = 3.0, b = 4.0;
                           double  c = Math.sqrt(h*h + b*b);   // here c is dynamically
                           initialized
                           System.out.println (" Hypotenuse is " + c);
                        }
                  )
```

Here, in this code, the variables h and b are declared and initialized to some values in a single statement. Then you find in the next line that another variable c, is declared as double and is assigned the output of an expression. The expression uses the sqrt() function which is one of the Java's built-in methods of the Math class.

The key point here is that the initialization expression may use any element valid at the time of initialization, including calls to methods, other variables, or literals.

Literals

Literals are nothing but pieces of Java code that indicate explicit values. For example "Hello IGNOU!" is a String literal. The double quote marks indicate to the compiler that this is a string literal. The quotes indicate the start and the end of the string, but remember that the quote marks themselves are not a part of the string. Similarly, Character Literals are enclosed in single quotes and it must have exactly one character. TRUE and FALSE are boolean literals that mean true and false. Number, double, long and float literals also exits there. See examples of all types of literals in the following *Table 8*.

Table 8: Examples of literals

Types of literal	Example
Number Literals	-45, 4L, 0777, 0XFF, 2.56F, 10e45, .36E-2
Boolean Literals	TRUE, FALSE
Character Literals	'a', '#', '3', \n, \\, \"
String Literals	"A string with a \t tab in it"
Double Literals	1.5, 45.6, 76.4E8
Long Literals	34L
Float Literals	45.6f, 76.4E8F, 1.5F

Constants

Constants are used for fixed values. Their value never changes during the program execution. To declare constants in Java keyword final is used. The following statement defines an integer constant x containing a value of 10.

```
final int x =10;
```

Identifiers

In simple words Identifiers are nothing but the names of variables, methods, classes, packages and interfaces. In the Application program, String, args, main and System.out.println are identifiers. Identifiers must be composed of letters, numbers, the underscore '_' and the dollar sign $. But identifiers should begin with a letter, the underscore or a dollar sign.

☞ Check Your Progress 3

1) Write a program in Java to calculate the area and the circumference of a circle. Show the use of 'final' keyword in your program

……

……

……

……

2) What are the kinds of variables in Java? What are their uses?

……

……

……

……

3) Why are the following are unacceptable as Java integer constant.

   a. 2,145

   b. –8.62

   c. 146E32

   d. 54-

4) Why is abstract keyword but Abstract is not?

……

……..……..

……

……

In the next section we will explore the different types of operators in Java with their meaning and uses in the programs.

3.4 JAVA OPERATORS

Arithmetic operators

You must be friendly with arithmetic expressions in mathematics like 'A – B'. In this expression A and B are operands and the subtraction sign ' – ' is the operator. The same terminology is also used here in the programming language. The following table lists the basic arithmetic operators provided by the Java programming language along with the description and uses. Here in the Table A and B are the operands. Operators

are divided into two categories **Binary and Unary.** Addition, subtraction, multiplication etc. are binary operators and applied only on two operands. Increment and decrement are unary operators and applied on single operand.

Table 9A: Description of arithmetic operators

Binary Operators		
Operator	**Use**	**Description**
+	A + B	Adds A and B
-	A - B	Subtracts B from A
*	A * B	Multiplies A by B
/	A / B	Divides A by B
%	A % B	Computes the remainder of dividing A by B
Unary Operators		
Operator	**Use**	**Description**
++ "Post-increment"	A++	Post-increment: The value is assigned before the increment is made, e.g. A = 1; B = A++; Then B will hold 1 and A will hold 2
-- "Post-decrement"	A--	Post-decrement: The value is assigned before the decrement is made, e.g. : A = 1; B = A--; Then B will hold 1 and A will hold 0.
++ "Pre-increment"	++A	Pre-increment: The value is assigned after the increment is made, e.g. A = 1; B = ++A; Then B will hold 2 and A will hold 2.
-- "Pre-decrement"	--A	Pre-decrement: The value is assigned after the decrement is made, e.g. A = 1; B = --A; Then B will hold 0 and A will hold 0.

Maybe you are thinking here that in the list we don't have any operator for exponentiation. If you want to do exponentiation you should import Java.lang class where math is subclass. Inside math class you have one function 'pow' similar to C++ which can be used for exponentiation. All the above operators can be used only on numeric values except for +, which is also used to concatenate strings.

Assignment Operators

The basic assignment operator '=' is used to assign value to the variables. For example A = B; in which we assign value of B to A. Similarly, let us see how to assign values to different data types.

int Integer = 100; float Float = 10.5; char Chararcter = 'S'; boolean aboolean = true;

With basic assignment operator, the Java programming language defines short cut assignment operators that allow you to perform arithmetic, shift, or bitwise operation with one operator.

Now let us see, if you want to add a number to a variable and assign the result back into the same variable, so you will write something like i = i + 2; but using shortcut operator '+=' You can shorten this statement, like i += 2. But remember, i = i + 2; and i += 2; both statements are the same for the compiler. As given in *Table10*, Java programming language provides different *short cut assignment operators*:

Table 10: Use of short cut assignment operators

Assignment Operators		
Operator	**Use**	**Equivalent to**
+=	A += B	A = A + B
-=	A -= B	A = A – B
*=	A *= B	A = A * B
/=	A /= B	A = A / B
%=	A %= B	A = A % B
&=	A &= B	A = A & B
|=	A |= B	A = A | B
^=	A ^= B	A = A ^ B
<<=	A <<= B	A = A << B
>>=	A >>= B	A = A >> B
>>>=	A >>>= B	A = A >>> B

Multiple Assignments

Multiple assignments are possible in the following format:
Identifier= Identifier B= Identifier C=…….=expression
In this case all the identifiers are assigned the value of expression. Lets see one example:

A=B=786+x+y+z; this is the same if you write: A=786+x+y+z; and B=786+x+y+z;

Relational Operators

Relational operators also called, comparison operators, are used to determine the relationship between two values in an expression. As given in *Table 11*, the return value depends on the operation.

Table 11: Use of relational operators

Relational operators		
Operator	**Use**	**Returns true if**
>	A > B	A is greater than B
>=	A >= B	A is greater than or equal to B

<	A < B	A is less than B
<=	A <= B	A is less than or equal to B
==	A == B	A and B are equal
!=	A != B	A and B are not equal

Boolean Operators

Boolean operators allow you to combine the results of multiple expressions to return a single value that evaluates to either true or false. *Table 12* describes the use of each boolean operator.

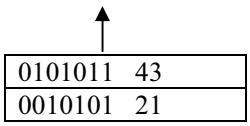
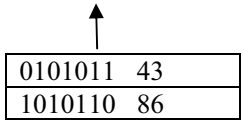
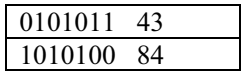
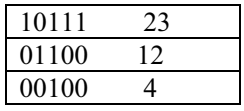
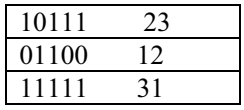
Table 12: Description of Boolean operators

Boolean Operators		
Operator	**Use**	**Description**
&&	A && B	Conditional AND :If both A and B are true, result is true. If either A or B are false, the result is false. But if A is false, B will not be evaluated. For example, (A > 4 && B <= 10) will evaluate to true if A is greater than 4, and B is less than or equal to 10.
||	A || B	Conditional OR: If either A or B are true, the result is true. But if A is true, B will not be evaluated. For example, (A > 10 || B > 10) will evaluate to true if either A or B are greater than 10.
!	! A	Boolean NOT: If A is true, the result is false. If A is false, the result is true.
&	A & B	Boolean AND: If both A and B are true, the result is true. If either A or B are false, the result is false and both A and B are evaluated before the test.
|	A | B	Boolean OR: If either A or B are true, the result is true. Both A & B are evaluated before the test.
^	A ^ B	Boolean XOR: If A is true and B is false, the result is true. If A is false and B is true, the result is true. Otherwise, the result is false. Both A and B are evaluated before the test.

Bitwise operators

As you know in computers data is represented in binary (1's and 0's) form. The binary representation of the number 43 is 0101011. The first bit from the right to left in the binary representation is the least significant bit, i.e. here value is 1. Each Bitwise operator allows you to manipulate integer variables at bit level. *Table 13* given below describes the use of bitwise operator with help of suitable example for each. Similarly, the use of class and object operators is given in *Table 14.*

Table 13: Description and use of Bitwise operators

Bitwise operators		
Operator	**Use**	**Description**
>>	A >> B	Right shift: Shift bits of A right by distance B, 0 is introduced to the vacated most significant bits, and the vacated least significant bits are lost. The following shows the number 43-shifted right once (42 >> 1). 
<<	A << B	Left shift: Shift bits of A left by distance B , the most significant bits are lost as the number moves left, and the vacated least significant bits are 0. The following shows the number 43 shifted left once (42 << 1). 
>>>	A >>> B	Right shift unsigned: Shift A to the right by B bits. Low order bits are lost. Zeros fill in left bits regardless of sign example.
~	~B	Bitwise complement: The bitwise Complemnt changes the bits. 1, into 0, and bit 0, to 1. The following shows the complement of number 43. 
&	A & B	Bitwise AND: AND is 1 if both the bits are 1 otherwise AND is 0. The bitwise AND is true only if both bits are set. Consider 23 & 12: 
|	A | B	Bitwise OR: The bitwise OR is true if either bits are set. Or if the bit is 1 if either of the two bits is 1, otherwise it is 0. Consider 23 | 12: 

^	A ^ B	Bitwise Exclusive OR: The bitwise Exclusive OR is true if either bits are set, but not both. XOR of the bits is 1 if one bit is 1 & other bit is 0, otherwise, it is 0. Consider 23 ^ 12: 10111 23 01100 12 11011 27

Class and Object Operators

Table 14: Class and Object Operators

Operator	Name	Description
instance of	Class Test Operator	The first operand must be an object reference. For example 'A instance of B', Returns true if A is an instance of B. Otherwise, it return false.
new	Class Instantiation	Creates a new object. For example: new A, in this A is either a call to a constructor, or an array specification.
"."	Class Member Access	It accesses a method or field of a class or object. For example A.B used for 'field access for object A' and A.B() used for 'method access for object A'
()	Method Invocation	For example: A(parameters) , Declares or calls the method named A with the specified parameters.
(type)	Object Cast	(type) A, in this example () operator Cost (convert) A to specific type. An exception will be thrown if the type of A is incompatible with specified type. In this type can be object or any primitive data type.

Other Operators

There are some other operators that cannot fit in the above categories but are very important for programmers. These operators are explained in *Table 15* given below.

Table 15: Other operators

Operator	Use	Description
?:	A ? B : C	If A is true, returns B. Otherwise, returns C.
[ ]	type []	Declares an array of unknown length, which contains type elements.
[ ]	type[A]	Creates and array with A elements. Must be used with the new operator.
[ ]	A[B]	Accesses the element at index B within the

		array A. Indices begin at 0 and extend through the length of the array minus one.
+	A+B	This binary operator concatenates one string to another. For example: String str1 = "IG"; String str2 = "NOU"; String str3 = str1 + str2 results in str3 holding "IGNOU".

☞ Check Your Progress 4

1) What is a literal? How many types of literals are there in Java?

……

……

2) What is synchronization and why is it important?

……

……

3) What is the difference between the Boolean & operator and the & operator?

……

……

……

……

4) What is the difference between a “compiler” and an “interpreter”?

……

……

……

3.5 SUMMARY

In this unit you learnt that Java was developed by SUN Microsystems in 1991 under the guidance of James Gosling. It’s an Object Oriented, general-purpose programming language. After its birth it became popular because of many reasons *like* security, robustness and multithreadedness but mainly because of its characteristic of *Architecture Neutral and platform independent*. The logic and magic behind its platform independence is “BYTECODE”. Java offers two flavors of programming, Java application and Java applet. Application can be executed independently while applet cannot be executed independently. When we compile a Java program we get Java bytecode (.class file) that can be executed on any other computer system, equipped with Java interpreter. To execute applets you can use applet viewer or explorer to run Java embedded HTML code.

You learnt the meaning of different Java keywords which have special meaning for compiler and cannot be used as a user defined identifier. Java supports eight primitive data types which can be classified into four categories. Each data type has its memory size and range of values it can store. Before using the Java variables in your program you should declare them and assign them some value. You learnt how to declare a variable dynamically with the help of an example. You have learnt different types of operators in the last section of this unit which are similar to C++ so you must be familiar with some of them. You must be tired after reading a long unit so now you can go and have a cup of coffee. In the next unit we will discuss about the classes and objects in Java.

3.6 SOLUTIONS /ANSWERS

Check Your Progress 1

1) Java Virtual Machine plays very important role in making Java portable and it executes the byte code generated by Java compiler. Java compiler translates Java program into a form called byte code. This byte code is kind of machine language for hypothetical machine or virtual machine. The name given 'virtual' is analysis to old system where compiler generates language which is understandable by hardware. In Java, compiler generates a language which is understandable by an imaginary machine which does not exist in hardware but is a software machine.

2) Java was designed for networking and a distributed environment so security was a major issue at the time of its development. Following are the two main characteristics that make Java secure.

- All the references to memory are 'symbolic references' which means that users do not know where the program is residing in memory and allocation of memory to program is totally handled by JVM of each machine.
- Java applets execute in runtime environment that restrict the intruder applet to spread virus and deleting and modifying files in host machine.

3) After developing Java language its team wanted to give a suitable name. They used to see an oak tree from their window and they thought to give the same name "Oak" to their new born language.

Check Your Progress 2

1) First of all using the text editor type Java source file of your program as given below. Save the file as Display Message.Java .

```
/* This is a program for simple display of message on output terminal */
class DisplayMessage
    {
     public static void main(String args[ ])
        {
            System.out.println("I am Learning Java ");
        }
    }
```

Now compile the source file-using compiler named Javac that takes your source file and translates its statements into a bytecode file.

C:> Javac DisplayMessage.Java

Run the program using interpreter named Java that takes your bytecode file and translates it them into instructions that our computer can understand.

C:> Java Display Message

2)

i. Write the following Java source code into text editor and save as BigProgram.Java

```
import Java.applet.Applet;
import Java.awt.Graphics;
public class BigProgram extends Applet {
public void paint(Graphics g) {
```

```
        g.drawString("Soon I will write big programs in Java ", 50, 25);
                                }
    .                                       }
```

ii. Compile the source file: Javac BigProgram.Java (This will generate BigProgram.class)

iii. Execute the Applet: include the code in HTML code.

```
HTML>
        <HEAD>
<TITLE> A Big Program </TITLE>
</HEAD>
<BODY>
Here is the output of my program:
<APPLET CODE=" BigProgram.class " WIDTH=250 HEIGHT=100>
</APPLET>
</BODY>
</HTML>
```

3) This drawString takes three arguments the first is message and the other two are pixal positions X-axis and Y-axis respectively.
For example; g.drawString("Hello IGNOU !", 50, 25);
Here, in the example," Hello IGNOU! Message is first argument, 50 and 25 are other two arguments that define X-position and Y-position where the message will be displayed.

Check Your Progress 3

1) If we get the error " Exception in thread "main"

Java.lang.NoClassDefFoundError", this means Java is unable to find bytecode file. Java tries to find bytecode file in current directory. So, if bytecode file is in C:\Java, we should change our current directory to that. The prompt should change to C:\Java>.

If still there is a problems, in this case may be CLASSPATH environment variable is not set up properly. You can set your CLASSPATH environment variable using DOS command like

C:\> SET CLASSPATH=C:\JDK\JAVA\CLASSES;c:\Java\lib\classes.zip

// program to calculate area of circle and circumference of circle.

```
class AreaTriangle   {
    public static void main (String args [ ] )
                        {
                           final double  pi = 3.14159;  // pi is defined as final here
                             double  radius = 4.0;
                             double  circumference = 2*pi*radius;   // here c is dynamically
                             initialized double  area = pi*radius*radius;
                             System.out.println ("circumference of circle of radius 4 unit is
                             " + circumference);
                             System.out.println ("Area of circle of radius 4 unit is " +area);
                    }
                   }
```

2) Java has three kinds of variables, namely, the instance variable, the local variable and the class variable.

- Local variables are used inside blocks as counters or in methods as temporary variables to store information needed by a single method.

- Instance variables are used to define attributes or the state of a particular object and are used to store information needed by multiple methods in the objects.

- Class variables are global to a class and to all the instances of the class. They are useful for communicating between different objects of the same class or keeping track of global states.

3) The given integer constants are unacceptable because:

   a. The comma is not allowed in the integer.
   b. The decimal point is not allowed in integers constants.
   c. The character E is not allowed in integer constant.
   d. The symbol '–' can appear in front of integer constant not in last or in-between.

4) Because Java is case-sensitive, so even though abstract is a keyword but Abstract is not a keyword at all.

Check Your Progress 4

1) A literal represents a value of a certain type where the type describes how that value behaves.

There are different types of literals namely number literals, character literals, boolean literals, and string literals.

2) With respect to multithreading, synchronization is the capability to control the access of multiple threads to shared resources. Without synchronization, it is possible for one thread to modify a shared object while another thread is in the process of using or updating that object's value. This often leads to significant errors.

3) If an expression involving the Boolean & operator is evaluated, both operands are evaluated. Then the && operator is applied to the operand. When an expression involving the & operator is evaluated, the first operand is evaluated. If the first operand returns a value of true then the second operand is evaluated. The && operator is then applied to the first and second operands. If the first operand evaluates to false, the evaluation of the second operand is skipped.

4) Compilers and interpreters have similar functions: They take a program written in some programming language and translate it into machine language. A compiler does the translation all at once. It produces a complete machine language program that can then be executed. An interpreter, on the other hand, just translates one instruction at a time, and then executes that instruction immediately. (Java uses a compiler to translate Java programs into Java Bytecode, which is a machine language for the imaginary Java Virtual Machine. An interpreter then executes Java Bytecode programs.)

UNIT 4 EXPRESSIONS, STATEMENTS AND ARRAYS

Structure **Page Nos.**

4.0 INTRODUCTION

Variables and operators, which you studied in the last unit, are basic building blocks of programs. Expressions are segments of code that perform some computations and return some values. Expressions are formed by combining literals, variables and operators. Certain expressions can be made into statements, which are complete units of execution. Statements are normally executed sequentially in the order in which they appear in the program. But there are number of situations where you may have to change the order of execution of statements based on certain conditions, or repeat a group of statements until certain specified condition(s) are met. Java language supports various control statements that can be put into the following categories: selection, iteration and jump. You will learn use of these in Java programming. An *array* is a data structure supported by all the programming languages that stores multiple values of the same type in a fixed length structure.

In this unit first we will cover about expressions and their evaluation. We will discuss about *operator precedence*. Then we will cover various kinds of statements supported in Java. At the end we will discuss arrays in Java language.

4.1 OBJECTIVES

After going through this unit, you should be able to:

- define expression;
- define and write the various kinds of statements;
- write programs using sequential, conditional, and iterative statements, and
- use array in programming.

4.2 EXPRESSIONS

Arithmetic expressions consist of operators, operands, parentheses, and function calls.

You studied in your school mathematics about mathematical expressions, made up of simpler terms or variables. Similarly in Java, expression is built from simpler expressions or variables using the operators. In this section we look at the various operators and show how they can be combined to generate expressions. **Expressions** perform the work of a program. They are used to compute and to assign values to

variables. Also, they help in controlling the execution flow of a program. The job of an expression is to perform the computation as indicated by the elements of the expression. Finally, it returns a value that is the result of the computation.

Definition: An *Arithmetic expression* is a series of variables, operators, and method calls that evaluates to a single value. It is constructed according to the syntax of the language.

As discussed in the previous unit, the application of operators returns a value, so that you can say the expression is nothing but a use of operators. For example if you write code "index++;" then '++' is an operator used with variable "index" and "index++;" is an expression.

The program given below shows some of the expressions given in **bold**:

```
…….
// primitive types
char Respond = 'Y';
boolean a Boolean = true;

// display all
System.out.println ("The largest byte value is " + largest Byte);
...

if (Character.isUpperCase (Respond))
{
   …….
}
```

Each of these expressions performs an operation and returns a value. As you can see, it is explained in *Table 1*.

Table 1: Arithmetic expression

Expression	Action	Value Returned
Respond = 'Y'	Assign the character 'Y' to the character variable Respond	The value of Respond after the assignment is ('Y')
"The largest byte value is " + largest Byte	Concatenate the string "The largest byte value is " and the value of largest Byte converted to a string	The resulting string: The largest byte value is 127
Character.isUpperCase (Respond)	Call the method is Upper Case	The return value of the method: true

In an expression, the data type of the value returned depends on the elements used. The expression **Respond = 'Y'** returns a character. Since both the operands Respond and 'Y' are characters, the use of assignment operator returns a character value. As you observe from the other expressions, an expression can return a boolean, a string, and so on.

The Java programming language allows programmers to construct **compound expressions** and statements from smaller expressions. It is allowed as long as the data types required by one part of the expression matches the data types of the other. Let us consider the following example of a compound expression:

```
x * y / z
```

In this example, the order in which the expression is evaluated is unimportant because the results of multiplication and division are *independent of order*. However, it is not true for all expressions. Let us consider another *example*, where addition and division operations are involved. Different results will be there depending on the *order of operation* i.e. whether you perform the addition or the division operation first:

```
x + y / 50     //ambiguous expression
```

The programmer can specify exactly how s/he wants an expression to be evaluated by using balanced parentheses **'('** and **')'**. For example s/he could write:

```
(x + y)/ 50   //unambiguous and recommended way
```

If the programmer doesn't explicitly indicate the order of the operations in a compound expression to be performed, it is determined by the **precedence** assigned to the operators. Operators with a higher precedence get evaluated first. Let us consider the example again; the division operator has a higher precedence than the addition operator. Thus, the two following statements are equivalent:

```
x + y / 50
x + (y / 50)
```

Therefore, it is suggested that while writing compound expressions, you should be explicit by using parentheses to specify which operators should be evaluated first. This will make the code easier to read and maintain.

Table 2 shows the operator precedence. The operators in this table are listed in order of the following precedence rule: the higher in the table an operator appears, the higher its precedence. In an expression, operators that have higher precedence are evaluated before operators that have relatively lower precedence. Operators on the same line have equal precedence.

Table 2: Operators precedence

Special operators	[ ] Array element reference Member selection (*params*) Function call
unary operators	++, --, +, -, ~, !
Creation or cast	new (*type*)*expression*
multiplicative	*, /, %
Additive	+, -
Shift	<<, >>, >>>
Relational	<, >, <=, >=, instance of
Equality	= =, !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|

logical AND	&&
logical OR	| |
Conditional	? :
Assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

When operators of same precedence appear in the same expression, some rule must govern which is evaluated first. All binary operators except for the assignment operators are evaluated in left-to-right order and assignment operators are evaluated right to left.

4.3 STATEMENTS

Statements in Java are equivalent to sentences in natural languages. A *statement* is a complete unit of execution. It is an executable combination of tokens ending with a semicolon (;) mark. A token can be any variable, constant, operator or an expression. By terminating the expression with a semicolon (;), the following types of expressions can be made into a statement

- Assignment expressions
- Any use of ++ or --
- Method calls
- Object creation expressions

These kinds of statements are called *expression statements*. Let us consider some examples of expression statements:

```
piValue = 3.141;                          //assignment statement
counter++;                                //increment statement

System.out.println(piValue);              //method call statement

Integer integerObject = new Integer(4);   //object creation statement
```

In addition to these kinds of expression statements, there are two other kinds of statements.

- A *declaration statement* declares a variable. Let us consider the following examples of declaration statements.
  ```
  Integer counter;
  double xValue = 91.224;
  ```
- A *control statement* regulates the flow of execution of statements based on the changes to the state of a program.

The while loop, **for** loop and the **if** statement are examples of control flow statements. This category of statements is considered in depth in coming sections.

Blocks

A *code **block*** is a group of two or more statements between balanced braces. It is a single logical unit that can be used anywhere. Even a single statement is allowed in this logical unit. The following listing shows a block that is target for if statement:

```
if (x < y)
{   //beginning of block
```

```
    System.out.println("The value of x is less than y" );
    x = 0;
}  // Ending of block
```

In this example, if x is less than y, both statements inside the block will be executed. Thus, both the statements inside the block form a logical unit. The key point here is that whenever you need to logically link two or more statements, you do so by creating a code block.

☞ Check Your Progress 1

1) What are the data types of the following expressions? Assuming that "i" is an integer data type.

   a) i > 0
   b) i = 0
   c) i++
   d) (float)i
   e) i == 0
   f) a String" + i

2) Consider the following expression:
   i--%5>0

   What is the result of the expression, assuming that the value of i is initially 10?

3) Modify the expression of Ex 2 so that it has the same result but is easier for programmers to read.

4.4 CONTROL STATEMENTS

The program is a set of statements, which are stored into a file. The interpreter executes these statements in a sequential manner, i.e., in the order in which they appear in the file. But there are situations where programmers want to alter this normal sequential flow of control. For example, if one wants to repeatedly execute a block of statements or one wants to conditionally execute statements. Therefore' one more category of statements called control flow statements is provided. In *Table 3*, different categories of control statements are defined:

Note: Although goto is a reserved word, currently the Java programming language does not support the goto statement.

Table 3: Control flow statements

Statement Type	Keyword
Selection	If-else, switch-case
Iteration	While, do-while, for
Jump	Break, continue, label;, return
Exception handling	Try-catch-finally, throw

We will cover the *first three* types of statements in the next three sections. The last statement type will be covered in Unit 4 Block 2 of this course.

4.5 SELECTION STATEMENTS

Java supports *two* selection statements: **if-else** and **switch**. These statements allow you to control the flow of execution based upon conditions known only during *run-time*.

The if / else Statements

The " **if** " statement enables the programmer to selectively execute other statements in the program, based on some condition. The code block governed by the " if " is executed if a condition is true. Generally, the simple form of it can be written like this:

```
if (expression)
{
   statement(s)
}
```

The following code block illustrates the ***if*** statement:

```
Public class If Example
 {
   public static void main (String[] args)
   {
      int x,y;
      x = 10; y = 50;
      if (x < y)
         System.out.println("x is less than y" );
      if (x > y)
         System.out.println ("x is greater than y");
      if (x = y)
         System.out.println ("x is equal to y");
    }
   }
```

If a programmer wants to perform a different set of statements when the expression is false, s/he can use the ***else*** statement.

Let us consider another example. The following code performs different actions depending on whether the user clicks the OK button or another button in an alert window. To do this, programmer can use **if** statement along with an **else** statement. The "*else code block"* is executed only-if "*if code block*" is false.

```
// response is either OK or CANCEL depending
// on the button pressed by the user
if (response == OK)
{
// code to perform OK action
}
else
{
// code to perform Cancel action
}
```

There is one more form of the else statement "**else if "**. It executes a statement based on another expression. "If statement" can have any number of companion *else if* statements but at the end, only one else part will be there. Let us consider the following example in which the program assigns a grade based on the value of a test score: an **A** for a score of 90% or above, a **B** for a score of 75% or above, and so on:

```
public class If Else Example
{
   public static void main (String[] args)
   {
      int test score = 78;
```

```
      char grade;
      if (test score >= 90)
       {
          grade = 'A';
       }
     else if (testscore >= 75)
      {
          grade = 'B';
       }
     else if (testscore >= 60)
     {
          grade = 'C';
      }
     else if (testscore >= 50)
    {
          grade = 'D';
    }
    else
   {
        grade = 'F';
    }
   System.out.println ("Grade = " + grade);
   }
}
```

The output from this program is:

Grade = B

The value of testscore can satisfy more than one of the expressions in the compound if statement: 78 >= 75 and 78 >= 60. At runtime, the system processes a compound if statement by evaluating the conditions. Once a condition is satisfied, the appropriate statements are executed (i.e. grade = 'B';), and control passes out of the "if statement" without evaluating the remaining conditions.

The?: Operator

The Java programming language also supports an operator"**?:**" known as ternary operator, this operator is a compact version of an *if statement*. It is used for making two-way decisions. It is a combination of ?and :, and takes three operands. It is popularly known as the conditional operator. The general form of use of the *conditional operator* is as follows:

Conditional expression **?** expression**:** expression2

The *conditional expression* is evaluated first. If the result is true, *expression1* is evaluated and is returned as the value of the conditional expression. Otherwise, *expression2* is evaluated and its value is returned.

```
if (x <= 10)
commission = 10 * x;
else
commission = 15 * x;
System.out.println("The Total commission is " + commission );
```

You could rewrite this statement using the?: operator:

```
commission =  (x <= 10)? (10 * x): (15 * x);
```

System.out.println ("The Total commission is " + commission);} + "case.");
When the conditional operator is used, the code becomes more *concise* and *efficient*.

The switch Statement

Java has a built-in multiway decision statement known as **switch** statement. The *switch* statement is used to conditionally perform statements based on an integer expression. Let us consider the following example. Using the *switch* statement, it displays the name of the month, based on the value of *month* variable.

```
public class SwitchExample1
{
   public static void main(String[] args)
   {
      int month = 11;
      switch (month)
        {
          case 1:  System.out.println("January"); break;
          case 2:  System.out.println("February"); break;
          case 3:  System.out.println("March"); break;
          case 4:  System.out.println("April"); break;
          case 5:  System.out.println("May"); break;
          case 6:  System.out.println("June"); break;
          case 7:  System.out.println("July"); break;
          case 8:  System.out.println("August"); break;
          case 9:  System.out.println("September"); break;
          case 10: System.out.println("October"); break;
          case 11: System.out.println("November"); break;
          case 12: System.out.println("December"); break;
        }
     }
}
```

The *switch* statement evaluates its expression and executes the appropriate *case* statement. Thus, in this case it evaluates the value of month and the output of the program is: November. It can be implemented by using an if statement:

```
int month = 11;
if (month == 1)
{
   System.out.println ("January");
}
 else if (month == 2)
{
   System.out.println("February");
}. . . // and so on
```

You can decide which statement (*if or switch*) to use, based on readability and other factors.

- An "**if statement**" can be used to make decisions based on ranges of values or conditions.
- A **switch** statement can make decisions based only on a single integer value. The value provided to each case statement must be unique.

It is important to notice the use of the **break** statement after each case in the switch statement. Each break statement terminates the enclosing **switch** statement, and the flow of control continues with the first statement following the switch block.

The **break** statements are necessary because without them, the case statements fall through. That is, without an explicit break, control will flow sequentially through subsequent case statements.

Let's consider the following example.

```
public class SwitchExample2
 {
   public static void main(String[] args)
   {
      int month = 2;
      int year = 2003;
      int numDays = 0;
      switch (month)
      {
         case 1:
         case 3:
         case 5:
         case 7:
         case 8:
         case 10:
         case 12:
            numDays = 31;
            break;
         case 4:
         case 6:
         case 9:
         case 11:
            numDays = 30;
               break;
         case 2:
            if ( ((year % 4 == 0) && !(year % 100 == 0))
                || (year % 400 == 0) )
               numDays = 29;
            else
               numDays = 28;
            break;
      }
      System.out.println("Number of Days = " + numDays);
   }
 }
```

The output from the above program is: Number of Days = 28

Now let us see how **break** is used to terminate loops in branching statements and the use of default statement at the end of the switch to handle all values that aren't explicitly handled by one of the case statements.

```
int month = 11;
. . .
switch (month)
{
   case 1:  System.out.println("January"); break;
   case 2:  System.out.println("February"); break;
   case 3:  System.out.println("March"); break;
   case 4:  System.out.println("April"); break;
   case 5:  System.out.println("May"); break;
   case 6:  System.out.println("June"); break;
   case 7:  System.out.println("July"); break;
   case 8:  System.out.println("August"); break;
```

```
   case 9:  System.out.println("September"); break;
   case 10: System.out.println("October"); break;
   case 11: System.out.println("November"); break;
   case 12: System.out.println("December"); break;
   default: System.out.println("It is not a valid month!"); break;
 }
```

Now let us see the iterative statements.

4.6 ITERATIVE STATEMENTS

Java iteration statements are **while**, **do-while** and **for**. These statements enable program execution to repeat one or more statements, i.e. they create **loops**. A loop repeatedly executes the same set of instructions until a termination condition is met.

The while and do-while Statements

A **while** statement is used to continually execute a block of statements provided a condition remains true. The general syntax of the *while* statement is:

```
while (expression)
{
  //body of loop
  statement
}
```

Here first, the while statement evaluates *expression*, which returns a boolean value. If the expression returns *true*, then the while statement executes the statement(s) in the body of the loop. The while statement continuously keeps on testing the *expression* and executing the code block until the expression returns *false*.

The example program shown below, displays the table of 6 unto 10 count. Every time while statement checks the value of count variable and displays the line for the table until the value of count becomes *greater than 10.*

```
public class While Example
{
   public static void main (String[] args)
   {
      int count = 1;
      int product;
     System.out.println (“Table of 6”);
      while (count <= 10)
      {
        product = 6 * count;
        System.out.println (“ 6 x “ + count + “ = “ + product);
        count++;
      }
   }
}
```

The do-while statement

The Java programming language also provides another statement which is similar to the while statement; the *do-while* statement. The general syntax of the do-while is:

```
do
{
   statement(s)
```

} while (*expression*);

The *do-while* evaluates the expression at the bottom instead of evaluating it at the top of the loop. Thus the code block associated with a *do-while* is executed at least once.

Here is the previous example program rewritten using **do-while**:

```
public class Do While Example
    {
   public static void main (String[ ] args)
   {
      int count = 1;
      int product;
      System.out.println("Table of 6");
      do
      {
         product = 6 * count;
         System.out.println(" 6 x " + count + " = " + product);
         count++;
      } while (count <= 10);
   }
}
```

The for Statement

The **for** statement provides a way to iterate over a range of values. The general form of the **for** statement can be expressed as:

```
for (initialization; termination; increment)
 {
   statement
 }
```

- The *initialization* is an expression that initializes the loop. It is executed only once at the beginning of the loop.
- The *termination* expression determines when to terminate the loop. This expression is evaluated at the top of iteration of the loop. When the expression evaluates to *false*, the loop terminates.
- The *increment* is an expression that gets invoked after each iteration through the loop. All these components are optional.

Often **for** loops are used to iterate over the elements in an array, or the characters in a string. The following example uses a **for** statement to iterate over the elements of an array and print them:

```
public class For Example
{
   public static void main (String[ ] args)
   {
      int[ ] array O fInts = { 33, 67, 31, 5, 122, 77,204, 82, 163, 12, 345, 23 };
      for (int i = 0; i < arrayOfInts.length; i++)
      {
       System.out.print (array O fInts [i] + " ");
      }
       System.out.println();
   }
}
```

The output of this program is: 33 67 31 5 122 77 204 82 163 12 345 23.

The programmer can declare a local variable within the *initialization* expression of a **for** loop. The scope of this variable extends from its declaration to the end of the block. It is recommended to declare the variable in the initialization expression in case the variable that controls a **for** loop is not needed outside the loop. Declaring the variables like i, j, and k within the for loop initialization expression limits their life span and reduces errors.

4.7 JUMP STATEMENTS

The Java language supports three jump statements:

- The break statement
- The continue statement
- The return statement.

These statements transfer control to another part of the program.

The break Statement

The **break** statement has two forms: *labeled* and *unlabeled.* A *label* is an identifier placed before a statement. The label is followed by a colon (:) like Statement Name: Java Statement;

Unlabeled Form: The unlabeled form of the break statement is used with *switch*. You can note in the given *example*, an **unlabeled break** terminates the enclosing switch statement, and flow of control transfers to the statement immediately following the switch. The unlabeled form of the break statement is *also used* to terminate a for, while, or do-while loop. Let us consider the following example program where a break is used in for loop:

```
public class BreakExample
  {
   public static void main (String[ ] args)
   {
      int [ ] arrayOfInts = {33, 67, 31, 5, 122, 77, 204, 82, 163, 12, 345, 23};
      int search for = 122;
      int i = 0;
      boolean foundIt = false;
      for (; i < arrayOfInts.length; i++)
      {
          if (arrayOfInts[i] == searchfor)
          {
             foundIt = true;
             break;
          }
      }
      if (foundIt)
     {
       System.out.println( searchfor + " at index " + i);
      }
     else
     {
       System.out.println(searchfor + "not in the array");
      }
   }
}
```

The **break** statement terminates the “for” loop when the value is found. The flow of control transfers to the statement following the enclosing for, which is the print statement at the end of the program.

The output of this program is:

122 at index 4

The unlabeled form of the break statement is used to terminate the innermost switch, for, while, or do-while structures.

Labeled Form: The labeled form terminates an **outer statement**, which is identified by the label specified in the break statement. Let us consider the following *example* program, which searches for a value in a two-dimensional array. When the value is found, a *labeled break* terminates the statement labeled search, which is the outer for loop:

```
public class BreakWithLabelExample
 {
   public static void main (String[ ] args)
   {
      int[][] arrayOfInts = { { 33, 67, 31, 5 },{ 122, 77, 204, 82 },{ 163, 12, 345, 23 }
};
      int searchfor = 122;
      int i = 0;
      int j = 0;
      boolean foundIt = false;
     search:
      for ( ; i < arrayOfInts.length; i++)
      {
          for (j = 0; j < arrayOfInts[i].length; j++)
            {
             if (arrayOfInts[i][j] == searchfor)
             {
                foundIt = true;
                break search;
             }
            }
      }
     if (foundIt)
      {
       System.out.println (search for + " at " + i + ", " + j);
      }
    else
     {
     System.out.println (search for + "not in the array");
     }
   }
}
```

The output of this program is: 122 at 1, 0

The break statement terminates the labeled statement; the flow of control transfers to the statement immediately following the labeled (terminated) statement.

The continue Statement

The continue statement is used to skip the current iteration of a for, while, or do-while loop.

The **unlabeled form** of continues statement skips the remainder of this iteration of the loop. It evaluates the boolean expression that controls the loop at the end. Let us take the following *example* program in which a string buffer is checking each letter.

If the current character is not '**s'**, the continue statement skips the rest of the loop and proceeds to the next character. If it is '**s'**, the program increments a counter.

```
public class Continue Example
  {
   public static void main (String[ ] args)
   {
          String Buffer search Str = new String Buffer ("she sell sea-shell on the sea
          shore");
      int max = search Str. length  ();
      int numOfS = 0;
      for (int i = 0; i < max; i++)
      {
         if (search Str. charAt(i) != 's') // we want to count only S's
            continue;
         numOfS++;
         }
      System.out.println("Found " + numOfS + " S's in the string:");
      System.out.println (search Str);
   }
}
```

Here is the output of this program:

```
Found 6 S's in the string.
she sell sea-shell on the sea shore
```

The **labeled form** of the continue statement skips the current iteration of an outer loop marked with the given label. The following *example* program uses *nested loops* to search a substring within another string. Two nested loops are required: one to iterate over the substring and one to iterate over the string being searched. This program uses the labeled form of continue to skip an iteration in the outer loop:

```
public class ContinueWithLabelExample
{
 public static void main (String[ ] args)
  {
  String searchMe = "Look for a substring in me";
  String substring = "sub";
  boolean foundIt = false;
  int max = searchMe.length() - substring.length();
 test:
  for (int i = 0; i <= max; i++)
  {
     int n = substring.length();
     int j = i;
     int k = 0;
     while (n-- != 0)
     {
        if (searchMe.charAt(j++) != substring.charAt (k++))
        {
          continue test;
        }
     }
          found It = true;
          break test;
  }
  System.out.println (found It? "Found it": "Didn't find it");
}
}
```

Here is the output from this program: Found it

The return Statement

The last of Java's jump statements is the **return** statement. It is used to *explicitly* return from the current method. The flow of control transfers *back to the caller* of the method. The return statement has **two forms**: one that returns a value and one that doesn't. To return a value, simply put the value (or an expression that calculates the value) after the *return* keyword:

return ++count;

The data type of the value returned by **return** must match the type of the method's declared return value. When a method is declared *void*, use the form of return that doesn't return a value like "return;"

☞ Check Your Progress 2

1) **State True or False for the followings**

a) The modulus operator (%) can be only used with integer operand. ☐

b) If a = 10 and b = 15, then the statement x = (a > b)? a: b; assigns the value 15 to x. ☐

c) In evaluating a logical expression of type BoolExp1 && BoolExp2 Both the Boolean expressions are not always evaluated. ☐

d) In evaluating the expression (x == y && a < b) the boolean expression x == y is evaluated first and then a < b is evaluated. ☐

e) The default case is always required in the switch selection structure. ☐

f) The break statement is required in the default case of a switch selection structure. ☐

g) A variable declared inside for the loop control cannot be referenced outside the loop. ☐

h) The following loop construct is valid ☐

```
int i = 10;
while (i)
{
Statements
}
```

i) The following loop construct is valid ☐

```
int i = 1; sum = 0;
do {Statements }
while ( i < 5 || sum < 20);
```

2) **Select appropriate answer for the followings:**

a) What will be the value of x , i and j after the execution of following statements?

```
int x , i , j;
i = 9;
j = 16;
x = ++i + j++
```

(i) x = 25, i = 9, j = 16
(ii) x = 26, i = 10, j = 17
(iii) x = 27, i = 9, j = 16
(iv) x = 27, i = 10, j = 17

b) If i and j are integer type variables, what will be the result of the expression

```
i % j
```

when i = 7 and j = 2?

(i) 0
(ii) 1
(iii) 2
(iv) None of the above

c) If i and j are integer type variables, what will be the result of the expression
i % j
when i = – 16 and j = –3 ?

(i) -1
(ii) 1
(iii) 5
(iv) None of the above

d) Consider the following code:

```
Char c = 'a';
Switch (c)
{
case 'a' :
System.out.println( "A");
case 'b' :
System.out.println( "B");
default :
System.out.println( "C");
}
```

3) Which of the following statements is True?

(i) Output will be A ☐
(ii) Output will be A followed by B ☐
(iii) Output will be A, followed by B, then followed by C ☐
(iv) Illegal code. ☐

4) What is wrong with the following program code:

```
Switch (x)
{
case 1:
m = 15;
n = 20;
case 2:
p = 25;
break;
y = m + n – p;
}
```

5) How is the if … else if combination more general than a switch statement?

4.8 ARRAYS

An *array* is an important data structure supported by all the programming languages. An array is a *fixed-length* structure that stores multiple values of the same type. You can group values of the same type within arrays. *For example* array name salary can be used to represent a set of salaries of a group of employees. Arrays are supported directly by the Java programming language but there is no array class.
The length of an array is established when the array is created (at runtime). After creation, an array is a fixed-length structure.

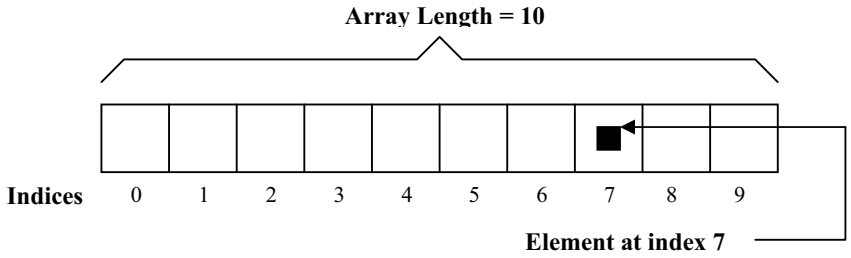


Figure 1: Example of an array

An *array element* is one of the values within an array and is accessed by its position within the array. Index counting in Java starts from 0. For example, to access the salary of 8th employee in array salary, it is represented by *salary[7]* as given in *Figure 1*.

Here is an example program that creates the array, puts some values in it, and displays the values.

```
public class ArraySample
{
public static void main(String[] args)
{
   int[] anArray;                  // declare an array of integers
   anArray = new int[10];          //create an array of integers
   // assign a value to each array element and print
   for (int i = 0; i < anArray.length; i++)
    {
     anArray[i] = i;
     System.out.print(anArray[i] + " ");
    }
   System.out.println();
}
}
```

Declaring a Variable to refer to an Array

The following line of code from the sample program declares an array variable:

Int [] an Array; // declare an array of integers

An array declaration has two components:

- Array's type
- Array's name.

An **array's type** is written *type* [], where *type* is the data type of the elements contained within the array, and [] indicates that this is an array. The example program uses **int []** and name of the array is an Array. Here, an Array will be used to hold integer data. Here are some other *declarations* for arrays that hold other types of data:

```
float[] marks;
boolean [] gender;
Object[] listOfObjects;
String[] NameOfStudents;
```

The *declaration* for an array variable does not allocate any memory to contain the array elements. The example program must assign a value to an Array before the name refers to an array.

Creating an Array

The programmer creates an array explicitly using Java's new operator. The next statement in the example program allocates an array with enough memory for ten integer elements and assigns the array to the variable an Array declared earlier.

An Array = new int[10]; // create an array of integers

In general, when creating an array, the programmer uses the new operator, followed by data type of the array elements, and then followed by the number of elements desired enclosed within square brackets ('[' and ']') like:

new element Type [array Size]

If the new statement were omitted from the example program, the compiler would print an error and compilation would fail.

Accessing an Array Element

Let us consider the following code, which assigns values to the array elements:

```
for (int i = 0; i < anArray.length; i++)
 {
anArray[i] = i;
System.out.print (anArray[i] + " ");
 }
```

It shows that to reference an array element, append square brackets to the array name. The value between the square brackets indicates (either with a variable or some other expression) the *index* of the element to access. As mentioned earlier in Java, array indices begin at 0 and end at the array length minus 1.

Getting the Size of an Array

You can get the size of an array, by writing "***arrayname*.length".** Here, length is not a method, it is a property provided by the Java platform for all arrays. The "for" loop in our example program iterates over each element of an Array, assigning values to its elements. The "for" loop uses the **anArray.length** to determine, when to terminate the "for" loop.

Array Initializers

The Java programming language provides one another way also for creating and initializing an array. Here is an example of this syntax:

boolean [] answers = {true, false, true, true, false};

The number of values provided between {and} determines the length of the array.

Multidimensional array in Java

In Java also you can take multidimensional arrays as arrays of arrays. You can declare a multidimensional array variable by specifying each additional index with a set of square brackets. For example

float two Dim [][]= new float [2][3];

This declaration will allocate a 2 by 3 array of float to two Dim. Java also provides the facility to specify the size of the remaining dimensions separately except the first dimension.

To understand the above stated concept see the program given below in which different second dimension size is allocated manually.

```
class Arra_VSize
{
public static void main (String args[])
{
        int i, j, k=0;
        int  twoDim [][] = new int [3][];
        twoDim[0] = new int[1];
        twoDim[1] = new int[2];
        twoDim[2] = new int[3];
        for ( i= 0; i <3 ; i++)
        {
          for (j = 0; j< i+1;  j++)
                {
                twoDim[i][j] = k +   k*3;
                k++;
                }
        }
  for ( i= 0; i <3 ; i++)
        {
          for (j = 0; j< i+1;  j++)
                System.out.print(twoDim[i][j] + " ");
                System.out.println();
        }
}
}
```

Output of this program

```
0
4 8
12 16 20
```

☞ Check Your Progress 3

1) Why should switch statement be avoided?

2) Answer the following questions using the sample code given below:

```
String[] touristResorts = {     "Whistler Blackcomb", "Squaw Valley",
"Ooty",  "Snowmass", "Flower Valley", "Taos"};
```

   a) What is the index of "Ooty" in the array?
   b) Write an expression that refers to the string Ooty within the array.
   c) What is the value of the expression touristResorts.length?
   d) What is the index of the last item in the array?
   e) What is the value of the expression touristResorts[4]?

3) The following program contains a bug. Find it and fix it.

```
//
// This program compiles but won't run successfully.
public class WhatHappens {
public static void main(String[] args) {
StringBuffer[] stringBuffers = new StringBuffer[10];
for (int i = 0; i < stringBuffers.length; i ++) {
stringBuffers[i].append("StringBuffer at index " + i);
}
}
}
```

4.9 SUMMARY

In this unit we first discussed about **expression**, which is a series of variables, operators, and method calls that evaluates to a single value. You can write compound expressions by combining expressions as long as the types required by all of the operators involved in the compound expression are correct. But remember Java platform evaluates the compound expression in the order dictated by *operator precedence* .We discussed three kinds of statements: expression statements, declaration statements, and control flow statements. The second important concept we discussed is that for controlling the flow of a program, Java has **three loop** constructs, Let us summarize these constructs:

Loop constructs

- Use the **while** statement to loop over a block of statements while a boolean expression remains true. The expression is evaluated at the *top of the loop*.
- Use the **do-while** statement to loop over a block of statements while a boolean expression remains true. The expression is evaluated at the *bottom of the loop*, so the statements within the do-while block execute at least once.
- The **for** statement loops over a block of statements and includes an initialization expression, a termination condition expression, and an increment expression.

Decision-Making Statements

- In the case of the basic **if** statement, a single statement block is executed if the boolean expression is true.
- In case of an **if** statement with a companion **else** statement, the if statement executes the first block if the boolean expression is true; otherwise, it executes the second block of code.
- Use **else if** to construct compound if statements.
- Use **switch** to make multiple-choice decisions based on a single integer value. It evaluates an integer expression and executes the appropriate case statement.

Jump Statements

Jump statements change the *flow of control* in a program to a labeled statement. You label a statement by placing a legal identifier (the label) followed by a colon (:) before the statement.

- Use the **unlabeled** form of the **break** statement to terminate the innermost switch, for, while, or do-while statement.
- Use the **labeled** form of the **break** statement to terminate an outer switch, for, while, or do-while statement with the given label.
- A **continue** statement terminates the current iteration of the innermost loop and evaluates the boolean expression that controls the loop.
- The **labeled** form of the **continue** statement terminates the current iteration of the loop with the given label.
- Use **return** to terminate the current method.

And in the end we discussed an **array,** which is a *fixed-length data structure* that can contain multiple objects of the same type. An element within an array can be accessed by its index. Indices begin at 0 and end at the length of the array.

4.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) a) i > 0 [boolean]
b) i = 0 [int]
c) i++ [int]
d) (float)i [float]
e) i == 0 [boolean]
f) "aString" + i [String]

2) false

3) (i-- % 5) > 0

Check Your Progress 2

1) State True or false

a) False
b) True
c) True
d) True
e) False
f) False
g) False
h) False
i) True

2) (a) (ii)
(b) (ii)
(c) (i)
(d) (ii)

3) Statement y = m + n – p; is unreachable.

4) The switch statement must be controlled by a single integer control variable and each case section must correspond to a single constant value for the variable. The if … else if combination allows any kind of condition after each if. Consider the following example:

```
…
if (income < 50000)
   System.out.println ( “Lower income group, Tax is nil”);
else if (income < 60000)
   System.out.println ( “Lower Middle income group, Tax is below 1000”);
else if (income < 120000)
   System.out.println ( “Middle income group, Tax is below 19000 ”);
else
   System.out.println ( “Higher income group”);
```

5) It refers to the way the switch statements executes its various case sections. Each statement that follows the selected case section will be executed unless a break statement is encountered.

Check Your Progress 3

1) Sometimes it is easy to fall through accidentally to an unwanted case while using switch statement. It is advisable to use `if/else` instead.

2)

a) 2.
b) Tourist Resorts[2].
c) 6.
d) 5.
e) Flower Valley

3)

The program generates a Null Pointer Exception on line 6. The program creates the array, but does not create the string buffers, so it cannot append any text to them. The solution is to create the 10 string buffers in the loop with new String Buffer() as follows:

```
public class ThisHappens
{
public static void main(String[] args)
{
StringBuffer[] stringBuffers = new StringBuffer[10];

for (int i = 0; i < stringBuffers.length; i ++)
{
stringBuffers[i] = new StringBuffer();
tringBuffers[i].append("StringBuffer at index " + i);
}
}
}
```

UNIT 2 INHERITANCE AND POLYMORPHISM

Structure **Page Nos.**

2.0 INTRODUCTION

In Object Oriented Programming, one of the major feature is reusability. You are already introduced to the reusability concept in Unit 2 of Block1 of this course. Reusability in Java is due to the inheritance. Inheritance describes the ability of one class or object to possess characteristics and functionality of another class or object in the hierarchy. Inheritance can be supported as "run-time" or "compile-time" or both. Here we are discussing inheritance with the Java programming language, and in our discussion we will generally refer to compile-time inheritance.

In this unit we will discuss importance of inheritance in programming, concept of superclass and subclass, and access controls in Java programming language. You will see through example programs how methods are overridden, and how methods of a super class are accessed by subclass. You will also see how multilevel inheritance is implemented, use of abstract classes, and demonstration of polymorphism with the help of example program.

2.1 OBJECTIVES

After going through this unit you will be able to:

- explain the need of inheritance;
- write programs to demonstrate the concept of super class and sub class;
- describe access control in Java;
- explain method overriding and its use in programming;
- incorporate concept of multilevel inheritance in programming, and
- define abstract classes and show use of polymorphism in problem solving.

2.2 INHERITANCE BASICS

Inheritance is a language property specific to the object-oriented paradigm. Inheritance is used for a unique form of code-sharing by allowing you to take the implementation of any given class and build a new class based on that implementation. Let us say class B, starts by inheriting all of the data and operations defined in the class A. This new subclass can extend the behaviour by adding additional data and new methods to operate on it. Basically during programming

inheritance is used for extending the existing property of a class. In other words it can be said that inheritance is "from generalization- to-specialization". In this by using general class, class with specific properties can be defined.

Now let us take the example of Employee class, which is declared as:

```
class BankAccount
{
data members
member functions
}
```

data members and member functions of BankAccount class are used to display characteristics of Withdraw, Deposit, getBalance of objects of a BankAccount class.

Now suppose you want to define a SavingAccount class. In the SavingAccount class definitely you will have basic characteristics of a Bankaccount class mentioned above. In your SavingAccount class you can use the properties of BankAccount class, without any modification in them. This use of properties of BankAccount class in Saving Account class is called inheriting property of BankAccount class into SavingAccount class.

To inherit a class into another class **extends** keyword is used. For example, SavingAccount class will inherit BankAccount class as given below.

```
class SavingAccount extends BankAccount
{
data members
ember functions
}
```

In this example, SavingAccount declares that it inherits or "extends" BankAccount.

Now let us see what is superclass and subclass.

Superclass and Subclass

The superclass of a class A is the class from which class A is derived. In programming languages like C++, allow deriving a class from multiple classes at a time. When a class inherits from multiple super classes, the concepts is known as multiple inheritance. Java doesn't support multiple inheritance. If there is a need to implement multiple inheritance. It is realized by using interfaces. You will study about Interfaces in Unit 4 of this Block.

A class derived from the superclass is called the subclass. Sometime-superclass is also called parent class or base class and subclass is called as child class or derived class. The subclass can reuse the data member and methods of the superclass that were already implemented and it can also extend or replace the behaviour in the superclass by overriding methods. Subclass can have its own data members and member functions.

You can see in this example program, the Employee class is used for tracking the hours an employ worked for along with the hourly wages, and the "attitude" which gives you a rough measure of their activeness or for what percentage of time they are actually productive.

```
public class Employee
{
protected double attitude;
protected int numHoursPerWeek, wagePerHour;
public Employee(int wage, int hours, double att) // constructor
{
```

```
wagePerHour = wage;
numHoursPerWeek = hours;
attitude = att;
}
public double getProductivity()
{
return numHoursPerWeek*attitude;
}
public double getTeamProductivity()
{
return getProductivity();
}
public int WeekSalary()
{
return wagePerHour*numHoursPerWeek;
}
}
```

If you look closely you will observe that Employee class possesses the very basic characteristics of an employee. So think quickly about different type of employees! Of course you can think about employees with special characteristics, for example, Manager Engineer, Machine-man etc. You are right Subclass of Employee, will have properties of Employee class as well as some more properties.

For example, if you take a class Manager (Subclass of Employee class) class. The Manager is a more specialized kind of Employee. An important point to note is that Manager represents ***a*** relationship with Employee. A Manager is a particular type of employee, and it has all the properties of Employee class plus some more property specific to it. Manager overrides the team productivity method to add the work done by the employees working under him/her and adds some new methods of its own dealing with other properties, which reflect characteristics of typical Managers. For example, annoying habits, taking employees under her/him, preparing report for employees, etc.

```
public class Manager extends Employee
{
// subclass of Employee
public Manager(int wage, int hours, double att, Employee underling)
{
super(wage, hours, att); // chain to our superclass constructor
 }
public double getTeamProductivity()
{
// Implementation here
}
public int askSalary()
{
  // Implementation here
}
public void addUnderling(Employee anUnderling)
{
// Implementation here
}
public void removeUnderling(Employee anUnderling)
{
// Implementation here
}
```

In the incomplete program above you can see that how inheritance is supporting in cremental development. Basically in the above program (this program is incomplete, you can write code to complete it) an attempt has been made to introduce a new code, without causing bugs in the existing code.

2.3 ACCESS CONTROL

You can see that the terms super, public and protected are used in previous programs. Can you tell what is the role of these terms in programs? Right, public and protected are access controller, used to control the access to members (data members and member functions) of a class, and super is used in implementing inheritance.

Now you can see how access control is used in Java programs.

Controlling Access to Members of a Class

One of the objectives of having access control is that classes can protect their member data and methods from getting accessed by other objects. Why is this important? Well, consider this. You're writing a class that represents a query on a database that contains all kinds of secret information; say student's records or marks obtained by a student in final examination.

In your program you will have certain information and queries contained in the class. Class will have some publicly accessible methods and variables in your query object, and you may have some other queries contained in the class simply for the personal use of the class. These methods support the operation of the class but should not be used by objects of another type. In other words you can say–you've got secret information to protect.

How can you protect it?

Ok in Java, you can use access specifiers to protect both variables and methods of a class when you declare them. The Java language supports four distinct access specifiers for member data and methods: private, protected, public, and if left unspecified, package.

The following chart shows the access level permitted by each specifier.

Specifier	class	subclass	package	world
Private	X			
Protected	X	X*	X	
Public	X	X	X	X
Package	X		X	

The first column indicates whether the class itself has access to the members defined by the access specifier. As you can see, a class always has access to its own members.

The second column indicates whether subclasses of the class (regardless of which package they are in) have access to the member.

The third column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member.

The fourth column indicates whether all classes have to the member.

Note that the protected/subclass intersection has an '*'. This particular case has a special association with inheritance implementation. You will see in the next section of this unit how protected specifier is used in inheritance. Package will cover in the next unit of this Block.

☞ Check Your Progress 1

1) What is the advantage of inheritance? How can a class inherit the properties of any other class in Java?

……

……

2) Explain the need of access specifiers.

……

……

3) When is private specifier used in a program?

……

……

Let's look at each access level in more detail.

private

Private is the most restrictive access level. A private member is accessible only to the class in which it is defined. You should use this access to declare members that you are going to use within the class only. This includes variables that contain information if it is accessed by an outsider could put the object in an inconsistent state, or methods, if invoked by an outsider, could jeopardize the state of the object or the program in which it is running. You can see private members like secrets you never tell anybody.

To declare a private member, use the private keyword in its declaration. The following class contains one private member variable and one private method:

```
class First
{
private int MyPrivate; // private data member
private void privateMethod() // private member function
{
System.out.println("Inside privateMethod");
}
}
```

Objects of class First can access or modify the MyPrivate variable and can invoke privateMethod.Objects of other than class First cannot access or modify MyPrivate variable and cannot invoke privateMethod . For example, the Second class defined here:

```
class Second {
void accessMethod() {
First a = new First();
a. MyPrivate = 51;      // illegal
a.privateMethod();      // illegal
}
}
```

cannot access the MyPrivate variable or invoke privateMethod of the object of First.

If you are attempting to access a method to which it does not have access in your program, you will see a compiler error like this:

```
Second.java:12: No method matching privateMethod()
found in class First.
a.privateMethod();          // illegal
1 error
```

One very interesting question can be asked, "whether one object of class First can access the private members of another object of class First". The answer to this question is given by the following example. Suppose the First class contained an instance method that compared the current First object (this) to another object based on their iamprivate variables:

```
class Alpha
{
private int MyPrivate;
boolean isEqualTo (First anotherObject)
{
if (this. MyPrivate == anotherobject. MyPrivate)
return true;
else
return false;
}
}
```

This is perfectly legal. Objects of the same type have access to one another's private members. This is because access restrictions apply at the class or type level (all instances of a class) rather than at the object level.

Now let us discuss protected specifier.

protected

Protected specifiers allows the class itself, subclasses, and all classes in the same package to access the members. You should use the protected access level for those data members or member functions of a class, which you can be accessed by subclasses of that class, but not unrelated classes. You can see protected members as family secrets–you don't mind if the whole family knows, and even a few trusted friends but you wouldn't want any outsiders to know. A member can be declared protected using keyword protected.

```
public class Student
{
protected int age;
public  String name;
protected void protectedMethod()
{
System.out.println("protectedMethod");
}
}
```

You will see the use of protected specifier in programs discussed in next sections of this block.

public

This is the easiest access specifier. Any class, in any package, can access the public members of a class's. Declare public members only if you want to provide access to a

member by every class. In other words you can say if access to a member by outsider cannot produce undesirable results the member may be declared public.
To declare a public member, use the keyword public. For example,

```
public class Account
{
public String name;
protected String Address;
protected int Acc_No;
public void publicMethod()
{
System.out.println("publicMethod");
}
}
class Saving_Account
{
void accessMethod()
{
Account a = new Account();
String MyName;
a.name =  MyName;       // legal
a.publicMethod();       // legal
}
}
```

As you can see from the above code snippet, Saving_Account can legally inspect and modify the name variable in the Account class and can legally invoke publicMethod also.

Member Access and Inheritance

Now we will discuss uses of **super** keyword in Java programming.

There are two uses of super keyword.

1. It is used for calling superclass constructor.
2. It is used to access those members of superclass that are hidden by the member of subclass (How a subclass can hide member of a superclass?).

Can you tell why subclass is called constructor of superclass?

An Object of class is created by call constructor to initialize its data member! Now if you create an object of a subclass you will call a suitable constructor of that subclass to initialize its data members. Can you tell how those data members of the parent class, which subclass is inheriting will be initialized? Therefore, to initialize superclass (parent class) data member, superclass constructor is called in subclass constructor.

To call a superclass constructor write super **(argument-list)** in subclass constructor and this should be the very first statement in the subclass constructor. This argument list includes the arguments needed by superclass constructor. Since the constructor can be overloaded, super () can be called using any form defined by the superclass. In case of constructor overloading in superclass, which of the constructors will be called is decided by the number of parameters or the type of parameter passed in super( ).

Now let us take one example program to show how subclass constructor calls superclass constructor.

```
class  Student
{
```

```
String name;
String address;
int age;
Student( String a, String b, int c)
{
name = a;
address = b;
age = c;
}
void display( )
{
System.out.println("*** Student Information ***");
Sstem.out.println("Name : "+name+"\n"+"Address:"+address+"\n"+"Age:"+age);
}
}
class PG_Student extends Student
{
int age;
int  percentage;
String course;
PG_Student(String a, String b, String c, int d , int e)
{
super(a,b,d);
course = c;
percentage = e;
age = super.age;
}
void display()
{
super.display();
System.out.println("Course:"+course);
}
}
class Test_Student
{
public static void main(String[] args)
{
 Student std1 = new Student("Mr. Amit Kumar" , "B-34/2 Saket J Block",23);
PG_Student pgstd1 = new  PG_Student("Mr.Ramjeet ", "241- Near Fast Lane Road
Raipur" ,"MCA", 23, 80);
std1.display();
pgstd1.display();
}
}
```

Output:

```
*** Student Information ***
Name : Mr. Amit Kumar
Address:B-34/2 Saket J Block
Age:23
*** Student Information ***
Name : Mr.Ramjeet
Address:241- Near Fast Lane Road Raipur
Age:23
Course:MCA
```

In the above program PG_Student class is derived from Student class. PG_Student class constructor has called constructor of Student class. One interesting point to note in this program is that both Student and PG_Student classes have variable named age.

When PG_Student class will inherit class Student, member data **age** of Student class will be hidden by the member data **age** of PG_Student class. To access member data **age** of Student class in PG_Student class, **super.age** is used. Whenever any member of a superclass have the same name of the member of subclass, it has to be accessed by using super keyword prefix to it.

2.4 MULTILEVEL INHERITANCE

Now let us discuss about multilevel. In program given below it is soon that how multilevel inheritance is implemented.

Order of Constructor Calling in Multilevel Inheritance

When the object of a subclass is created the constructor of the subclass is called which in turn calls constructor of its immediate superclass. For example, if we take a case of multilevel inheritance, where class B inherits from class A. and class C inherits from class B. You can see the output of the example program given below, which show the order of constructor calling.

```
//Program
class A
{
A()
{
System.out.println("Constructor of Class A has been called");
}
}
class B extends A
{
B()
{
super();
System.out.println("Constructor of Class B has been called");
}
}
class C extends B
{
C()
{
super();
System.out.println("Constructor of Class C has been called");
}
}
class Constructor_Call
{
public static void main(String[] args)
{
System.out.println("------Welcome to Constructor call Demo------");
C objc = new C();
}
}
```

Output:

```
------Welcome to Constructor call Demo------
Constructor of Class A has been called
Constructor of Class B has been called
Constructor of Class C has been called
```

2.5 METHOD OVERRIDING

You know that a subclass extending the parent class has access to all the non-private data members and methods its parent class. Most of the time the purpose of inheriting properties from the parent class and adding new methods is to extend the behaviour of the parent class. However, sometimes, it is required to modify the behaviour of parent class. To modify the behaviour of the parent class overriding is used.

Some important points that must be taken care while overriding a method:

i. An overriding method (largely) replaces the method it overrides.
ii. Each method in a parent class can be overridden at most once in any one of the subclass.
iii. Overriding methods must have exactly the same argument lists, both in type and in order.
iv. An overriding method must have exactly the same return type as the method it overrides.
v. Overriding is associated with inheritance.

The following example program shows how member function area () of the class Figure is overridden in subclasses Rectangle and Square.

```
class Figure
{
double sidea;
double sideb;
Figure(double a, double b)
{
sidea = a;
sideb = b;
}
Figure(double a)
{
sidea = a;
sideb = a;
}
double area( )
{
System.out.println("Area inside figure is Undefined.");
return 0;
}
}
class Rectangle extends Figure
{
Rectangle( double a , double b)
{
super ( a, b);
}
double area ( )
{
System.out.println("The Area of Rectangle:");
return sidea*sideb;
}
}
class Squre extends Figure
{
Squre( double a )
{
```

```
super (a);
}
double area( )
{
System.out.println("Area of Squre: ");
return sidea*sidea;
}
}
class Area_Overrid
{
public static void main(String[] args)
{
Figure f = new  Figure(20.9, 67.9);
Rectangle r = new  Rectangle( 34.2, 56.3);
Squre s = new Squre( 23.1);
System.out.println("***** Welcome to Override Demo ******");
f.area();
System.out.println(" "+r.area());
System.out.println(" "+s.area());
}
}
```

Output:

```
***** Welcome to Override Demo ******
Area inside figure is Undefined.
The Area of Rectangle:
 1925.46
Area of Squre:
 533.61
```

In most of the object oriented programming languages like C++ and Java, a reference parent class object can be used as reference to the objects of derived classes. In the above program to show overriding feature object of *Figure class* can be used as reference to objects of Rectangle and Square class. Above program with modification(shown in bold) in Area_Override class will be as :

```
class Area_Overrid
{
public static void main(String[] args)
{
Figure f = new  Figure(20.9, 67.9);
Rectangle r = new  Rectangle( 34.2, 56.3);
Squre s = new Squre( 23.1);
System.out.println("***** Welcome to Override Demo ******");
f.area();
f= r;
System.out.println(" "+f.area());
f = s;
System.out.println(" "+f.area());
}
}
```

☞ Check Your Progress 2

1) State True/False for the following statements:

T	F

i. One object can access the private member of the object of the same class. ☐

ii A subclass cannot call the constructor of its super class. ☐

iii A public variable in a package cannot be accessed from other package. ☐

2) Explain the use of super keyword in Java programming.

……

……

……..

3) How is method overriding implemented in Java? Write the advantage of method overriding.

……

……

……..

2.6 ABSTRACT CLASSES

As seen from the previous examples, when we extending an existing class, we have a choice whether to redefine the methods of the superclass. Basically a superclass has common features that are shared by subclasses. In some cases you will find that superclass cannot have any instance (object) and such of classes are called **abstract classes**. Abstract classes usually contain **abstract methods.** Abstract method is a method signature (declaration) without implementation. Basically these abstract methods provide a common interface to different derived classes. Abstract classes are generally used to provide common interface derived classes. You know a superclass is more general than its subclass(es). The superclass contains elements and properties common to all of the subclasses. Often, the superclass will be set up as an *abstract* class, which does not allow objects of its prototype to be created. In this case only objects of the subclass are created. To do this the reserved word *abstract* is included (prefixed) in the class definition.

For example, the class given below is an abstract class.

```
public abstract class Player  // class is abstract
{
private String name;
public Player(String  vname)
{
name=vname;
}
public String getName()        // regular method
{
return (name);
}
public abstract void Play();
//  abstract method: no implementation
}
```

Subclasses *must* provide the method implementation for their particular meaning. If the method statements is one provided by the superclass, it would require overriding in each subclass. In case you forget to override, the applied method statements may be inappropriate.

Now can you think what to do if you have to force that derived classes must redefine the methods of superclass?

The answer is very simple Make those methods abstract.

In case attempts are made to create objects of abstract classes, the compiler doesn't allow and generates an error message. If you are inheriting in a new class from an abstract class and you want to create objects of this new class, you must provide definitions to all the abstract methods in the superclass. If all the abstract methods of super class are not defined in this new class this class also will become abstract.

Is it possible to have an abstract class without abstract method? Yes, you can have. Can you think about the use of such abstract classes? These types of classes are defined in case it doesn't make any sense to have any abstract methods, in the class and yet you want to prevent an instance of that class.

Inheritance represent, "is–a" relationship between a subclass and a superclass. In other words, you can say that every object of a subclass is also a superclass object with some additional properties. Therefore, the possibility of using a subclass object in place of a superclass object is always there. This concept is very helpful in implementing polymorphism.

Now we will discuss polymorphism one of the very important features of object oriented programming, called polymorphism supported by Java programming language.

2.7 POLYMORPHISM

Polymorphism is the capability of a *method* to do different things based on the object through which it is invoked or object it is acting upon. For example method *find _area* will work definitely for Circle object and Triangle object In Java, the type of actual object always determines method calls; object reference type doesn't play any role in it. You have already used two types of polymorphism (overloading and overriding) in the previous unit and in the current unit of this block. Now we will look at the third: *dynamic method binding*. Java uses Dynamic Method Dispatch mechanism to decide at run time which overridden function will be invoked. Dynamic Method Dispatch mechanism is important because it is used to implement runtime polymorphism in Java. Java uses the principle: "a super class object can refer to a subclass object" to resolve calls to overridden methods at run time.

If a superclass has method that is overridden by its subclasses, then the different versions of the overridden methods are invoked or executed with the help of a superclass reference variable.

Assume that three subclasses (Cricket_Player Hockey_Player and Football_Player) that derive from Player abstract class are defined with each subclass having its own Play() method.

```
abstract class Player  // class is abstract
{
private String name;
public Player(String nm)
{
name=nm;
}
public String getName()        // regular method
{
return (name);
```

```
}
public abstract void Play();
//  abstract method: no implementation
}
class Cricket_Player extends Player
{
Cricket_Player( String  var)
{
}
public void Play()
{
System.out.println("Play Cricket:"+getName());
}
}
class Hockey_Player extends Player
{
Hockey_Player( String  var)
{
}
public void Play()
{
System.out.println("Play Hockey:"+getName());
}
}
class Football_Player extends Player
{
Football_Player( String  var)
{
}
public void Play()
{
System.out.println("Play Football:"+getName());
}
}
public class PolyDemo
{
public static void main(String[] args)
{
Player ref;                    // set up var for an Playerl
Cricket_Player  aCplayer = new Cricket_Player("Sachin");  // makes specific objects
Hockey_Player aHplayer = new Hockey_Player("Dhanaraj");
Football_Player aFplayer = new Football_Player("Bhutia");
// now reference each as an Animal
ref = aCplayer;
ref.Play();
ref = aHplayer;
ref.Play();
ref = aFplayer;
ref.Play();
}
}
```

Output:

```
Play Cricket:Sachin
Play Hockey:Dhanaraj
Play Football:Bhutia
```

Notice that although each method is invoked through ref, which is a reference to player class (but no player objects exist), the program is able to resolve the correct

method related to the subclass object at runtime. This is known as dynamic (or late) method binding.

2.8 FINAL KEYWORD

In Java programming the *final* key word is used for three purposes:

i. Making constants
ii. Preventing method to be overridden
iii. Preventing a class to be inherited

The final keyword (as discussed in Unit 3 of Block1 of this course) is a way of marking a variable as "read-only". Its value is set once and then cannot be changed.

For example, if year is declared as
final int year = 2005;

Variable year will be containing value 2005 and cannot take any other value after words.

The final keyword can also be applied to methods, with similar semantics: i.e. the definition will not change. You cannot override a final method in subclasses, this means the definition is the “final” one. You should define a method final when you are concerned that a subclass may accidentally or deliberately redefine the method (override).

If you want to prevent a class to be inherited, apply the final keyword to an entire class definition. For example, if you want to prevent class Personal to be inherited further, define it as follows:

```
final   class Personal
{
// Data members
//Member functions
}
```
Now if you try to inherit from class Personal

```
class   Sub_Personal extend Personal
```

It will be illegal. You cannot derive from Personal class since it is a final class.

☞ Check Your Progress 3

1) Explain the advantage of abstract classes.

……

……

……

2) Write a program to show polymorphism in Java.

……

……

……

……

3) What are the different uses of final keyword in Java?

……

……

……

2.9 SUMMARY

In this unit the concept of inheritance is discussed. It is explained how to derive classes using extends keyword in Java. To control accessibility of data members three access specifiers: privatepublic, and protected-are discussed. How to write programs using concept of inheritance, method overriding, need of abstract classes is also explained. The concept of polymorphism is explained with the help of a programming example. In the last section of the unit the use of final keyword in controlling inheritance is discussed.

2.10 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Inheritance is used for providing reusability of pre-existing codes. For example, there is an existing class A and you need another class B which has class A properties as well as some additional properties. In this situation class B may inherit from class A and there is no need to redefine the properties common to class A and class B. In Java a class is inherited by any other class using *extends* keyword.

2) Access specifiers are used to control the accessibility of data members and member functions of class. It helps classes to prevent unwanted exposure of members (data and functions) to outside world.

3) If some data members of a class are used in internal operations only and there is no need to provide access of these members to outside world. Such member data should be declared private. Similarly, those member functions Which are used for internal communications/operations only should be declared private.

Check Your Progress 2

1)

i. True
ii. False
iii. False

2) Java's super keyword is used for two purposes.

i. To call the constructors of immediate superclass.
ii. To access the members of immediate superclass.

When constructor is defined in any subclass it needs to initialize its superclass variables. In Java using **super()** superclass constructor is called **super()** must be the first executable statement in subclass constructor. Parameters needed by superclass constructor are passed in ***super (parameter_list)***. The super keyword helps in conflict resolution in subclasses in the situation of " when members name in superclass is *same as* members name in subclass and the members of the superclass to be called in subclass".

super.member; // member may be either member function or member data

3) Java uses Dynamic Method Dispatch, one of its powerful concepts to implement method overriding. Dynamic Method Dispatch helps in deciding the version of the method to be executed. In other words to identify the type of object on which method is invoked.

Overriding helps in:

- Redefining inherited methods in subclasses. In redefinition declaration should be identical, code may be different. It is like having another version of the same product.
- Can be used to add more functionality to a method.
- Sometimes class represent an abstract concept (i.e. abstract class). In this case it becomes essential to override methods in derived class of abstract class.

Check Your Progress 3

The advantage of abstract classes.

1) Any abstract class is used to provide a common interface to different classes derived from it. A common interface gives a feeling (understanding) of commonness in derived classes. All the derived classes override methods of abstract class with the same declaration. Abstract class helps to group several related classes together as subclass, which helps in keeping a program organized and understandable.

2)

```
// Program to show polymorphism in Java.
abstract class Account
{
public String Name;
public int Ac_No;
Account(String nm,int an )
{
Name=nm;
Ac_No= an;
}
abstract  void getAc_Info( );
}
class Saving_Account extends Account
{
private int min_bal;
Saving_Account( String na, int an, int bl)
{
super(na,an);
min_bal= bl;
}
void getAc_Info()
{
System.out.println(Name +"is having Account Number  :"+Ac_No);
System.out.println("Minimum Balance in Saving Account :"+Ac_No +"is
Rs:"+min_bal);
}
}
class Current_Account extends Account
{
private int min_bal;
Current_Account( String na, int an, int bl)
```

```
{
super(na,an);
min_bal= bl;
}
void getAc_Info()
{
System.out.println(Name +"is having Account Number  :"+Ac_No);
System.out.println("Minimum Balance in Current Account :"+Ac_No +"  is
Rs:"+min_bal);
}
}
public class AccountReference
{
public static void main(String[] args)
{
Account ref;                    // set up var for an Animal
Saving_Account s  = new Saving_Account("M.P.Mishra", 10001,1000);
Current_Account c  = new Current_Account("Naveen ", 10005,15000);
ref =s;
ref.getAc_Info();
ref =c;
ref.getAc_Info();
}
}
```

Output:

M.P.Mishrais having Account Number: 10001
Minimum Balance in Saving Account: 10001 is Rs: 1000
Naveen is having Account Number: 10005
Minimum Balance in Current Account: 10005 is Rs: 15000

3) Final keyword of Java is used for three things:

   i. To declare a constant variable.
   ii. To prevent a method to be overridden (to declare a method as final).
   iii. To prevent a class to be inherited (to declare a class as final).

UNIT 3 PACKAGES AND INTERFACES

Structure **Page Nos.**

3.0 INTRODUCTION

Till now you have learned how to create classes and use objects in problem solving. If you have several classes to solve a problem, you have to think about a container where you can keep these classes and use them. In Java you can do this by defining packages. Also, Java provides a very rich set of package covering a variety of areas of applications including basic I/O, Networking, and Databases. You will use some of these packages in this course. In this unit you will learn to create your own packages, and use your own created package in programming.

In this unit we will also discuss *interface,* one of Java's useful features. If we take Object Orientation into consideration an *interface in Java* is a subset of the public methods of a class. The *implementation* of a class is the code that makes up those methods. In Java the meaning of interface is different from the meaning in Object Orientation. An *interface* is just a specification of a set of *abstract methods*. If a class implements the interface, then it is essential for the class to provide an implementation for all of the abstract methods in the interface. As we have discussed in Unit 2 of this block that Java does not provide multiple inheritance. You can use interfaces to overcome the limitation of non-availability of multiple inheritance, because in Java a class can implement many interfaces. In this unit you will learn to create interfaces and implement them in classes.

3.1 OBJECTIVES

After going through this unit you will be able to:

- explain what is a package in Java;
- set CLASSPATH variable;
- using user created packages in programs;
- define Interface;
- explain the need of Java interfaces in programming;
- implement interfaces in classes, and
- use interfaces to store constant variables of programs.

3.2 PACKAGE

Java programmer creates packages to partition classes. Partitioning of classes helps in managing the program. The package statement is used to define space to store classes. In Java you can write your own package. Writing packages is just like you write any other Java program. You just have to take care of some points during writing packages, which are given below.

A *package* is a collection of related classes and interfaces providing access protection and namespace management.

There must be not more than one public class per file.
All files in the package must be named name_of_class.Java where name_of_class is the name of the single public class in the file.
The very first statement in each file in the package, before any import statements or anything put the statement package *myPackageName*;

Now let us see how to define a package.

3.2.1 Defining Package

You can define package for your own program
package PackageName;

This statement will create a package of the name **PackageName**.You have always to do one thing that is **.class** files created for the classes in package-let us say in package **PackageName–** must be stored in a directory named **PackageName**. In other words you can say that the directory name must exactly match the package name.

You can also create a hierarchy of packages. To do this you have to create packages by separating them using period(.).Generally a multilevel package statement look likes:
package MyPak1[.MyPak2[.MyPak3]];

For example, if a package is declared as:
package Java.applet;

Then it will be stored in directory *Java\applet* in Windows. If the same has to be stored in Unix then it will be stored in directory *Java/applet*.

There is one system environment variable named as CLASS PATH. This variable must be set before any package component takes part in programs. Now we will discuss CLASSPATH.

3.2.2 CLASSPATH

CLASSPATH is an environment variable of system. The setting of this variable is used to provide the root of any package hierarchy to Java compiler.
Suppose you create a package named **MyPak**. It will be stored in **MyPak** directory.Now let us say class named **MyClass** is in **MyPak**.You will store **MyClass.Java** in **MyPak** directory . To complie **MyClass.Java** you have to make **MyPak** as current directory and ***MyClass.class*** will be stored in **MyPak**.

Can you run **MyClass.class** file using Java interpreter from any directory? No it is not so, because **MyClass.class** is in package **MyPak,** so during execution you will to refer to package hierarchy. For this purpose you have to set CLASSPATH variable for setting the top of the hierarchy. Once you set CLASSPATH for MyPak , it can be used from any directory.

For example, if /home/MyDir/Classes is in your CLASSPATH and your package is called **MyPak1**, then you would make a directory called **MyPak** in /home/MyDir/Classes and then put all the **.class** files in the package in /home/MyDir/Classes/MyPak1.

Now let us see an example program to create a package.

```
//program
package MyPack;
class Student
{
 String Name;
int Age;
String Course;
Student( String n, int a, String c)
{
Name = n;
Age = a;
Course = c;
}
void Student_Information()
{
System.out.println("Name of the Student :"+ Name);
System.out.println("Age of the Student :"+Age);
System.out.println("Enrolled in Course  :"+Course);
}
}
class PackTest
{
public static void main( String args[])
{
Student  Std1 = new Student("Rajeev",19, “MCA”);
Std1.Student_Information();
}
}
```

Output:
Name of the Student :Rajeev
Age of the Student :19
Enrolled in Course: “MCA”

3.2.3 Packages Naming

Space conflicts will arise if two pieces of code declare the same name. The java runtime system internally keeps track of what belongs to each package. For example, suppose someone decides to put a Test class in **myJava.io**, it won’t conflict with a Test class defined in **myJava.net** package. Since they are in different packages Java can tell them apart. Just as you tell Mohan Singh apart from Mohan Sharma by their last names, similarly Java can tell two Test classes apart by seeing their package names.

But this scheme doesn’t work if two different classes share the same package name as well as class name. It is not unthinkable that two different people might write packages called **myJava.io** with a class called Test.You can ensure that package names do not conflict with each other by prefixing all your packages in the hierarchy. More about conflict resolution we will discuss in Section 3.4.1 of this Unit.

Java provides various classes and interfaces that are members of various packages that bundle classes by function: fundamental classes are in **Java.lang** classes for reading and writing (input and output) are in Java.io,for applet programming **Java.applet** and so on.

Now you are aware of the need of putting your classes and interfaces in packages. Here we again look at a set of classes and try to find why we need to put them in a package.

Suppose that you write a group of classes that represent a collection of some graphic objects, such as circles, rectangles, triangles, lines, and points. You also write an interface, MoveIt is implemented to check if graphics objects can be moved from one location to another with the help of mouse or not..

We are using interface here which is discussed in Section 3.5 of this unit.

Now let us see the code snippet given below:

```
//code snippet
public abstract class Graphic
{
   . . .
}
//for Circle, MoveIt is an interface
public class Circle extends Graphic implements MoveIt
{
. . .
}
//for Triangle, MoveIt is an interface
public class Triangle extends Graphic implements MoveIt
{
. . .
}
//interface MoveIt
public interface MoveIt
{
  // method to check movement of graphic objects.
}
```

If you closely observe this basic structure of the code snippet given above, you will find four major reasons to have these classes and the interface in a package. These reasons are:

i. Any programmers can easily determine that these classes and interfaces are related.
ii. Anybody can know where to find classes and interfaces that provide graphics-related functions.
iii. The names of your classes won't conflict with class names in other packages, because the package creates a new namespace.
iv. Your classes within the package can have unrestricted access to one another, yet still restrict access for classes outside the package.

Now let us create a package named graphics for the above program structure.

This code will appear in the source file Circle.Java and puts the Circle class in the graphics package.

```
package graphics;
public class Circle extends Graphic implements MoveIt
{
. . .
}
```

The Circle class is a public member of the graphics package. Similarly include the statement in Rectangle.Java, triangle.Java, and so on.

```
package graphics;

public class Rectangle extends Graphic implements MoveIt
{
. . .
}
```

☞ Check Your Progress 1

1) What is import? Explain the need of importing a package.

……

……

……

2) Write a program to create a package named AccountPack with class BankAccount in it with the method to show the account information of a customer.

……

……

Now we will discuss the accessibility of the classes and interfaces defined inside the packages.

3.2 ACCESSIBILITY OF PACKAGES

The scope of the *package* statement is the *entire* source file. So all classes and interfaces defined in Circle.Java and Rectangle.Java are also members of the graphics package. You must take care of one thing, that is, if you put multiple classes in a single source file, only one class *may be public*, and it must share the name of the source files. Remember that only public package members are accessible from outside the package.

If you do not use a package statement, your class or interface are stored in a *default package*, which is a package that has no name. Generally, the default package is only for small or temporary applications or when you are just beginning the development of small applications. For projects developments, you should put classes and interfaces in some packages. Till now all the example programs you have done are created in default package.

One of the major issues is how the classes and interfaces of packages can be used in programs. Now we will discuss using members of packages.

3.4 USING PACKAGE MEMBERS

Only public package members are accessible outside the package in which they are defined. To use a public package member from outside its package, one or more of the following things to be done:

1. Refer to the member by its long (qualified) name.
2. Import the package member.

3. Import an entire package.

Each of this is appropriate for different situations, as explained below:

1) Referring to a Package Member by Name

So far, in the examples in this course we have referred to classes by their simple names. You can use a package member's simple name if the code you are writing is **in the same package as that member** or if the member's package has been **imported**. However, if you are trying to use a member from a different package and that package has not been imported, you must use the member's qualified name, which includes the package name.

For example, this is the qualified name for the Rectangle class declared in the graphics package in the previous example:

graphics.Rectangle

You can use this long name to create an instance of graphics. Rectangle like this:
graphics. Rectangle myRect = new graphics.Rectangle();

It is okay to use long names if you have to use a name once. But if you have to write graphics. Rectangle, several times, you would not definitely like it. Also, your code would get very messy and difficult to read. In such cases, you can just import the member.

2) Importing a Package Member

To import a specific member into the current file, you have to put an import statement at the beginning of the file before defining any class or interface definitions. Here is how you would import the **Circle** class from the **graphics** package created in the previous section:

import graphics.Circle;

Now you can refer to the Circle class by its simple name:
Circle myCircle = new Circle(); // creating object of Circle class

This approach is good for situations in which you use just a few members from the graphics package. But if you have to use many classes and interfaces from a package, you should import the entire package.

3) Importing an Entire Package

To import all the classes and interfaces contained in a particular package, use the import statement with the asterisk (*).For example to import a whole graphics package write statement:

import graphics.*;

Now you can refer to any class or interface in the graphics package by its short name as:
Circle myCircle = new Circle();
Rectangle myRectangle = new Rectangle();

The asterisk in the import statement can be used only to specify all the classes within a package, as shown here. It cannot be used to match only a subset of the classes in a

package. With the import statement, you can import only a single package member or an entire package.

For your convenience, the Java runtime system automatically imports two entire packages: the **Java.lang package** and the **current package** by default.

Resolving Name Conflicts

There may be situations in which a member in one package shares the same name with a member in another package and both packages are imported. In these situations you must refer to each member by its *qualified name*. For example, the previous example defined a class named Rectangle in the **graphics** package. The Java.awt package also contains a Rectangle class. If both ***graphics*** and ***Java.awt*** have been imported, in this case the following is ambiguous:
Rectangle rect;

In such a situation, you have to be more specific and use the member's qualified name to indicate exactly which Rectangle class you want, for example, if you want Rectangle of graphics package then write:

graphics. Rectangle rect; //rect is object of graphics. Rectangle class

And if you Rectangle class of Java.awt package then write:

Java.awt. Rectangle rect; //rect is object of Java.awt. Rectangle class

☞ Check Your Progress 2

1) Write two advantages of packages.

……

……

……

2) Write a program to show how a package member is used by name.

……

……

……

3) When does name conflict arise in package use? How to resolve it?

……

……

……

Java does not support multiple inheritance. But if you have some implementation which needs solutions similar to multiple inheritance implementation. This can be done in Java using interfaces. Now let us see how to define and implement interfaces.

3.5 INTERFACES

Interfaces in Java look similar to classes but they don't have instance variables and provides methods that too without implementation. It means that every method in the

interface is strictly a declaration.. All methods and fields of an interface must be public.

Interfaces are used to provide methods to be implemented by class(es).A class implements an interface using ***implements*** clause. If a class is implementing an interface it has to define all the methods given in that interface. More than one interface can be implemented in a single class. Basically an interface is used to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. As Java does not support multiple inheritance, interfaces are seen as the alternative of multiple inheritance, because of the feature that multiple interfaces can be implemented by a class.

An interface is like a class with nothing but abstract methods and final, static fields. All methods and fields of an interface must be public.

Interfaces are useful because they:

- Provide similarities among unrelated classes without artificially forcing a class relationship.
- Declare methods that one or more classes are expected to implement.
- Allow objects from many different classes which can have the same type. This allows us to write methods that can work on objects from many different classes, which can even *be in different inheritance hierarchies.*

Now let us see how interfaces are defined.

Defining an Interface

Though an interface is similar to a class, there are several restrictions to be followed:

- An interface does not have instance variable.
- Every method of an interface is abstract.
- All the methods of an interface are automatically public.

Figure 1 shows that an interface definition has two components:

i. The interface declaration
ii. The interface body.

The interface declaration declares various attributes of the interface such as its name and whether it extends another interface. The interface body contains the constant and method declarations within that interface.

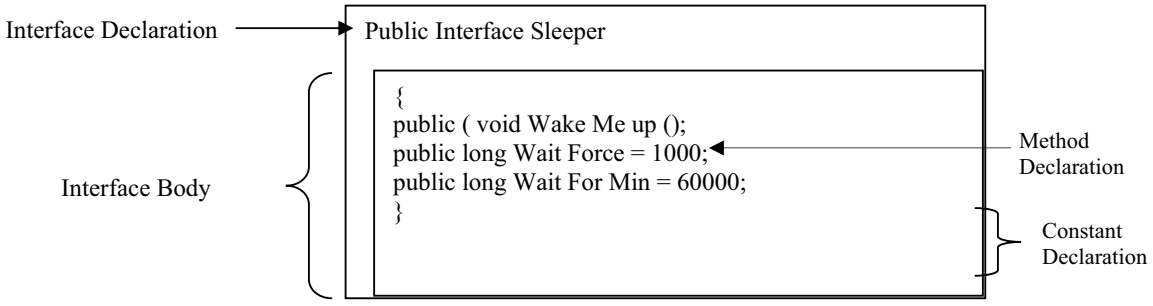


Figure 1: Interface Declaration

Interface defining is similar to defining a class. You can define an interface as given below:

```
access_specifier interface  Name_of_Interface
{
return_type method1(patameters);
return_type method2(patameters);
return_type method3(patameters);
 ……
return_type methoN(patameters);
type   variable_Name1;
type   variable_Name2;
}
```

In this declaration access_specifier is either public or not given. If no specifier is given then interface is available only to the members of the package in which it is declared. If specifier is given as public, then it is available to all. As you can see, variables also can be declared within inside of interface but they are *final* and *static* by default. Classes implementing this interface cannot change these variables. All the methods and variables are by default *public,* if interface is *public* otherwise they are visible to the package in which interface is declared.

Unlike a class, a interface can be added to a class that is already a subclass of another class. A single interface can apply to members of many different classes. For example you can define a Calculate interface with the single method calculateInterest().

```
public interface Calculate
{
public double calculateInterest();
}
```

Now you can use this interface on many different classes if you need to calculate interest. It would be inconvenient to make all these objects derive from a single class. Furthermore, each different type of system is likely to have a different means of calculating the interest. It is better to define a Calculate interface and declare that each class implements Calculate.

Many different methods can be declared in a single interface. The method of an interface may be overloaded. Also in an interface may extend other interfaces just as a class extend or subclass another class.

3.6 IMPLEMENTING INTERFACES

As discussed earlier an interface can be implemented by a class using implements clause. The class implementing an interface looks like

```
access_specifier class className  implements interfaceName
{
//class body consisting of methods definitions of interface
}
```

To declare a class that implements an interface, include an implements clause in the class declaration. Your class can implement more than one interface (the Java platform supports multiple interface inheritance), so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class. Remember that when a class implements an interface, it is essentially signing a contract with the interface that the class must provide method implementations for all of the methods declared in the interface and its superinterfaces. If a class does not implement all the methods declared in the interface the class must be declared abstract. The method signature (the name and the number and type of arguments) for the method in the class must match the method signature as it appears in the interface.

The interface :

```
public interface Sleeper
{
   public void wakeUpMe();
   public long WaitForSec= 1000;
   public long WaitForMin=60000;
}
```

declares two constants that are useful arguments to letMeSleep. All constant values defined in an interface are implicitly public, static, and final. The use of these modifiers on a constant declaration in an interface is discouraged as a matter of style. Any class can use an interface's constants from the name of the interface, like this: Sleeper.WaitForSec
Classes that implement an interface can treat the constants as they were inherited. This is why ControlClock can use WaitForsec directly when calling letMeSleep:

```
public class ControlClock extends Applet implements Sleeper
{
  . . .
public void wakeUpMe()
{
     repaint();
     clock.letMeSleepr(this, WaitForSec);
}
}
```

Note: Member declarations in an interface does not allow for some declaration modifiers; you may not use transient, volatile, or synchronized in a member declaration in an interface. Also, you may not use the private and protected specifiers when declaring members of an interface.

Again let us take example of interface Calculate and implement it in a class:

```
public interface Calculate
{
public double calculateInterest();
}
```

In the following program interface Calculate is implemented by SavingAccount class.

```
//program
interface Calculate
{
public double calculateInterest();
}
class SavingAccount implements Calculate
{
int Ac_No;
int Amount;
double Rate_of_Int;
int time;
SavingAccount( int a , double b, int c, int d)
{
Ac_No = a;
Rate_of_Int = b;
time = c;
Amount = d;
}
void DisplayInt()
```

```
{
System.out.println("Interest for Account No. "+Ac_No+" is Rs "+calculateInterest());
}
public double calculateInterest( )
{
return( Amount*Rate_of_Int*time/100);
}
}
public class TestIterface
{
public static void main( String args[])
{
SavingAccount S = new SavingAccount( 1010,4.5,5,5000);
S.DisplayInt();
}
}
```

Output:
Interest for Account No. 1010 is Rs 1125.0

Interface and Inheritance

One interface can inherit another interface using **extends** keyword. This is syntactically the same as inheriting classes. Let us say one interface A is inhering interface B. At the same time a class C is implementing interface A. Then class C will implement methods of both the interfaces A and B.

See the program given below to show inheritance of interfaces.

```
//program
interface One
{
void MethodOne( );
}
interface Two extends One
{
public void MethodTwo();
}
class interfaceInherit implements Two
{
public void MethodOne()
{
System.out.println("Method of  interface One is implemented");
}
public void MethodTwo()
{
System.out.println("Method of  interface Two is implemented");
}
}
class Test
{
public static void main( String args[])
{
System.out.println("Interface Inheritance Demonstration");
interfaceInherit object = new interfaceInherit() ;
object.MethodOne();
object.MethodTwo();
}
}
```

Output:

```
Interface Inheritance Demonstration
Method of  interface One is implemented
Method of  interface Two is implemented.
```

3.7 INTERFACE AND ABSTRACT CLASSES

Abstract classes and interfaces carry similarity between them that methods of both should be implemented but there are many differences between them. These are:

- A class can implement more than one interface, but an abstract class can only subclass one class.
- An abstract class can have non-abstract methods. All methods of an interface are implicitly (or explicitly) abstract.
- An abstract class can declare instance variables; an interface cannot.
- An abstract class can have a user-defined constructor; an interface has no constructors.
- Every method of an interface is implicitly (or explicitly) public.
- An abstract class can have non-public methods.

3.8 EXTENDS AND IMPLEMENTS TOGETHER

By convention the implements clause follows the extends clause, if it exists.
You have seen that the Control Clock class implements the Sleeper interface, which contains a single method declaration for the wakeUpMe method.Here Applet class is extended first then Sleeper interface is implemented.

```
public class Control Clock extends Applet implements Sleeper
{
. . .
public void wakeUpMe()
{
// update the display
}
}
```

Again I remind you that when a class implements an interface, it is essentially signing a contract. The class must provide method implementations for all of the methods declared in the interface and its superinterfaces. The method signature (the name and the number and type of arguments) for the method in the class must match the method signature as it appears in the interface.

☞ Check Your Progress 3

1) What is an interface?

……

……

2) Write a program to show how a class implements two interfaces.

……

……

……

3) Show through a program that fields in an interface are implicitly static and final and methods are automatically public.

……

……

……

3.9 SUMMARY

In this unit the concepts of packages and interfaces are discussed. This unit explains the need of packages, how to create package, and the need of setting CLASSPATH variable is discussed. It is also discussed how members of packages are imported in a program and how to remove name conflicts if any. In this unit we have also discussed the need of Interfaces, how interfaces are declared, how an interface is implemented by a class. In the last two sections of this unit we have discussed differences between abstract classes and interfaces and explained how a class may inherit from a class and implement interfaces.

3.10 SOLUTIONS/ANSWERS

Check your Progress 1

1) A package is a collection of related classes and interfaces, providing access protection and naming space management.

To define a package keyword package is used. The code statements:

```
package MyOwnPackage;
// class and interfaces
public class A
{
// code
}
 public interface I
{
//methods declaration
}
```

define a package named MyOwnPackage.

2) CLASSPATH is a system variable used to provide the root of any package hierarchy to Java compiler.

To set CLASSPATH see Java programming section of MCSL–025 course .

3)

```
//program
package AccountPaack;
class Account
{
String Name;
int AccountNo;
String Address;
Account( String n, int a, String c)
{
```

```
Name = n;
AccountNo = a;
Address = c;
}
void Accout_Information()
{
System.out.println("Name of the Account holder :"+ Name);
System.out.println("Account Number:"+AccountNo);
System.out.println("Address of Account holder  :"+Address);
}
}
class AccoutTest
{
public static void main( String args[])
{
Account  AcH1 = new Account("Rajeev",110200123, "28-K Saket, New Delhi");
Account  AcH2 = new Account("Naveen",110200113, "D–251,Sector-55, Noida");
AcH1.Accout_Information();
AcH2.Accout_Information();
}
}
```

Output:
Name of the Account holder :Rajeev
Account Number:110200123
Address of Account holder :28-K Saket, New Delhi
Name of the Account holder :Naveen
Account Number:110200113
Address of Account holder :D–251,Sector-55, Noida

Check your Progress 2

1) import is a Java keyword used to include a particular package or to include specific classes or interfaces of any package.
import keyword is used to:

- either include whole package by statement
import PackageName.*;
- or to import specific class or interface by statement
import PackageName.ClassName(or InterfaceName).*;

2) Naming conflict arise when one package share the same name with a member another package and both packages are imported.
Naming conflict can be resolved using qualified name of the members to be imported.

Check your Progress 3

1) Interfaces can be seen as a group of methods declaration that provides basic functionality to classes that share common behavior. Java allows a class to implement multiple interfaces and by this Java try to fill the gap of not supporting multiple inheritance. In Java a class implementing an interface must have to all the methods of in that interface otherwise this class has to be abstract class.

2)

```
//program to implement two interfaces in a single class
interface First
```

```
{
void MethodOne( );
int MyValue1 = 100;
}
interface Second
{
public void MethodTwo();
int MyValue2 = 200;
}
class interfaceIn implements First,Second
{
public void MethodOne()
{
System.out.println("Method of  interface First is implemented with
value:"+MyValue1);
}
public void MethodTwo()
{
System.out.println("Method of  interface Second is implemented with
value:"+MyValue2);
}
}
class TwoInterfTest
{
public static void main( String args[])
{
interfaceIn object = new interfaceIn() ;
object.MethodOne();
object.MethodTwo();
}
}
```

Output:
Method of interface First is implemented with value:100
Method of interface Second is implemented with value:200

3)

```
//program
interface MyInterface
{
void MyMethod( );
int MyValue1 = 500;
}
class MyinterfaceImp implements MyInterface
{
public void MyMethod()
{
System.out.println("Method of  interface MyInterface is implemented with
value:"+MyValue1);
//MyValue1 += 50;
}
}
class InterfaceTest
{
public static void main( String args[])
{
MyinterfaceImp object = new MyinterfaceImp() ;
object.MyMethod();//myMethod is by default public.
}
}
```

Output:
Method of interface MyInterface is implemented with value:500

If you remove comment (//) from statement "//MyValue1 += 50;" and compile this program you will get :

---------- Compile ----------
InterfaceTest.java:12: cannot assign a value to final variable MyValue1
MyValue1 += 50;
^
1 error
This error is because of attempt to modify final variable MyValue1.

UNIT 4 EXCEPTIONS HANDLING

Structure **Page Nos.**

4.0 INTRODUCTION

During programming in languages like c, c++ you might have observed that even after successful compilation some errors are detected at runtime. For handling these kinds of errors there is no support from programming languages like c, c++. Some error handling mechanisms like returning special values and setting flags are used to determine that there is some problem at runtime.

In C++ programming language there is a very basic provision for exception handling. Basically exception handlings provide a safe escape route from problem or clean-up of error handling code.

In Java exception handling is the only semantic way to report error .In Java exception is an object, which describes error condition, occurs in a section of code. In this unit we will discuss how exceptions are handled in Java, you will also learn to create your own exception classes in this unit.

4.1 OBJECTIVES

After going through this unit you will be able to:

- describe exception;
- explain causes of exceptions;
- writing programs with exceptions handling;
- use built–in exceptions;
- create your own exception classes.

4.2 EXCEPTION

An exceptional condition is considered as a problem, which stops program execution from continuation from the point of occurrence of it. Exception stops you from continuing because of lack of information to deal with the exception condition. In other words it is not known what to do in specific conditions.

If the system does not provide it you would have to write your own routine to test for possible errors. You need to write a special code to catch exceptions before they cause an error.

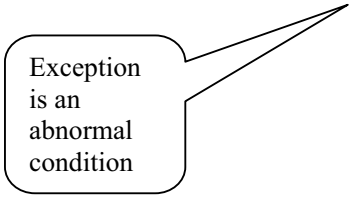
If you attempt in a Java program to carry out an illegal operation, it does not necessarily halt processing at that point. In most cases, the **JVM** sees for the possibility of *catching* the problem and recovering from it.

If the problems are such which can be caught and recovery can be provided, then we say the problems are **not fatal**, and for this the term ***exception*** is used rather than **error**.

Now let us see what to do if exceptions occur.

Causes of Exception

Exception arises at runtime due to some abnormal condition in program for example when a method. For division encounters an abnormal condition that it can't handle itself, i.e. "divide by zero," then this method may *throw* an exception.



If a program written in Java does not follow the rule of Java language or violates the Java execution environment, constraints exception may occur. There may be a manually generated exception to pass on some error reports to some calling certain methods.

If an exception is caught, there are several things that can be done:

i. Fix the problem and try again.
ii. Do something else instead to avoid the problem.
iii. Exit the application with System.exit()
iv. Rethrow the exception to some other method or portion of code.
v. Throw a new exception to replace it.
vi. Return a default value (a non-void method: traditional way of handling exceptions).
vii. Eat the exception and return from the method (in a void method). In other words don't give importance to the exception .
viii. Eat the exception and continue in the same method (Rare and dangerous. Be very careful if you do this).

You should give due care to exceptions in program. Programmers new to programming almost always try to ignore exceptions. Do not simply avoid dealing with the exceptions. Generally you should only do this if you can logically guarantee that the exception will never be thrown or if the statements inside exception checking block do not need to be executed correctly in order for the following program statements to run).

Now let us see how exceptions are handled in Java.

4.3 HANDLING OF EXCEPTION

Exceptions in Java are handled by the use of these five keywords**: try, catch, throw, throws, and finally**. You have to put those statements of program on which you want to monitor for exceptions, in **try** block. If any exceptions occur that will be catched using catch. Java runtime system automatically throws system-generated exceptions.

The throw keyword is used to throw exceptions manually.

4.3.1 Using try catch

Now let us see how to write programs in Java, which take care of exceptions handling.
See the program given below:

```
//program
public class Excep_Test
```

```
{
public static void main(String[] args)
{
int data[] = {2,3,4,5};
System.out.println("Value at : " + data[4]);
}
}
```

Output:

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

At runtime this program has got ArrayIndexOutOfBoundsException.This exception occurs because of the attempt to print beyond the size of array.

Now let us see how we can catch this exception.

To catch an exception in Java, you write a try block with one or more catch clauses. Each catch clause specifies one exception type that it is prepared to handle. The *try* block places a fence around the code that is under the watchful eye of the associated catchers. If the bit of code delimited by the *try* block *throws* an exception, the associated *catch* clauses will be examined by the Java virtual machine. If the virtual machine finds a *catch* clause that is prepared to handle the thrown exception, the program continues execution starting with the first statement of that *catch* clause, and the catch block is used for executing code to handle exception and graceful termination of the program.

```
public class Excep_Test
{
public static void main(String[] args)
{
try
{
int data[] = {2,3,4,5};
System.out.println("Value at : " + data[4]);
}
catch( ArrayIndexOutOfBoundsException e)
{
System.out.println("Sorry you are trying to print beyond the size of data[]");
}
}
}
```

Output:

```
Sorry you are trying to print beyond the size of data[]
```

In this program you can observe that after the occurrence of the exception the program is not terminated. Control is transferred to the *catch* block followed by *try* block.

4.3.2 Catching Multiple Exceptions

Sometimes there may be a chance to have multiple exceptions in a program. You can use multiple catch clauses to catch the different kinds of exceptions that code can throw. If more than one exception is raised by a block of code, then to handle these exceptions more than one catch clauses are used. When an exception is thrown, different catch blocks associated with try block inspect in order and the first one whose type (the exception type passed as argument in catch clause) matches with the exception type is executed This code snippet will give you an idea how to catch multiple exceptions.

```
//code snippet
```

```
try
{
// some code
}
catch (NumberFormatException e)
{
//Code to handle NumberFormatException
}
catch (IOException e)
{
// Code to handle IOException
}
catch (Exception e)
{
//  Code to handle exceptions than NumberFormatException and IOException
}
finally // optional
{
//Code in this block always executed even if  no exceptions
}
```

Now let us see the program given below :

```
//program
public class MultiCatch
{
public static void main(String[] args)
{
int repeat ;
try
{
repeat = Integer.parseInt(args[0]);
}
catch (ArrayIndexOutOfBoundsException e)
{
// pick a default value  for repeat
repeat = 1;
}
catch (NumberFormatException e)
{
// print an error message
System.err.println("Usage:  repeat as count" );
System.err.println("where repeat is the number of times to say Hello Java" );
System.err.println("and given as an integer like 2 or 5" );
return;
}
for (int i = 0; i < repeat; i++)
{
System.out.println("Hello");
}
}
}
```

Output:

```
Hello
```

Output of the above program is “Hello”. This output is because of no argument is passed to program and exception “ArrayIndexOutOfBoundsException” occurred. If pass some non-numeric value is as argument to this program you will get some other output. Check what output you are getting after passing “Java” as argument.

It is important to ensure that something happens upon exiting a block, no matter how the block is exited. It is the programmer's responsibility to ensure what should happen. For this finally clause is used.

Check Your Progress 1

1) What is an exception? Write any three actions that can be taken after an exception occurs in a program.

……

……

……

…….

…….

.

2) Is the following code block legal?

```
try
{
...
}
finally
{
...
}
```

……

……

……

…….

…….

3) Write a program to catch more than two exceptions.

……

……

……

…….

4.3.3 Using finally clause

There are several ways of exiting from a block of code (the statements between two matching curly braces). Once a **JVM** has begun to execute a block, it can exit that block in any of several ways.

It could, for example simply exit after reaching the closing curly brace.
It could encounter a break, continue, or return statement that causes it to jump out of the block from somewhere in the middle.

If an exception is thrown that isn't caught inside the block, it could exit the block while searching for a catch clause.
Let us take an example, of opening a file in a method. You open the file perform needed operation on the file and most importantly you want to ensure that the file

gets closed no matter how the method completes. In Java, this kind of desires are fulfilled with the help of finally clause.

A finally clause is included in a program in the last after all the possible code to be executed. Basically finally block should be the last block of execution in the program.

```
//program
public class Finally_Test
{
public static void main(String[] args)
{
try
{
System.out.println("Hello " + args[0]);
}
catch (ArrayIndexOutOfBoundsException e)
{
System.out.println("Hello, You are here after ArrayIndexOutOfBoundsException");
}
finally
{
System.out.println("Finally you have to reach here");
}
}
}
```

Output:
Hello, You are here after ArrayIndexOutOfBoundsException
Finally you have to reach here

Note: At least one clause, either *catch* or *finally*, must be associated with each *try* block. In case you have both *catch* clauses and a *finally* clause with the same *try* block, you must put the *finally* clause after all the *catch* clauses.

Now let us discuss the types of exceptions that occur in Java.

4.4 TYPES OF EXCEPTIONS

Exceptions in Java are of two kinds, ***checked*** and **unchecked**. Checked exceptions are so called because both the Java compiler and the **JVM** check to make sure that the rules of Java are obeyed. Problems causes to checked exceptions are: Environmental error that cannot necessarily be detected by testing; e.g, disk full, broken socket, database unavailable, etc. Checked exceptions must be handled at compile time. Only checked exceptions need appear in throws clauses. Problems such as Class not found, out of memory, no such method, illegal access to private field, etc, comes under virtual machine error.

Unchecked exceptions

Basically, an unchecked exception is a type of exception for that you option that handle it, or ignore it. If you elect to ignore the possibility of an unchecked exception, then, as a result of that your program will terminate. If you elect to handle an unchecked exception that occur then the result will depend on the code that you have written to handle the exception. Exceptions instantiated from **RuntimeException** and its subclasses are considered as *unchecked* exceptions.

Checked exceptions

Checked exceptions are those that cannot be ignored when you write the code in your methods. “According to Flanagan, the exception classes in this category represent routine abnormal conditions that should be anticipated and caught to prevent program termination.”

All exceptions instantiated from the **Exception** class, or from subclasses, of **Exception** other than **RuntimeException** and its subclasses, must either be:

(i) Handled with a **try** block followed by a **catch** block, or

(ii) Declared in a **throws** clause of any method that can throw them

The conceptual difference between checked and unchecked exceptions is that checked exceptions signal **abnormal conditions** that you have to deal with. When you place an exception in a throws clause, it ***forces*** to invoke your method to deal with the exception, either by catching it or by declaring it in their **own throws** clause. If you don't deal with the exception in one of these two ways, your class will not compile

4.4.1 Throwable class Hierarchy

All exceptions that occur in Java programs are a subclass of built–in class **Throwable**. Throwable class is top of the exception class hierarchy. Two classes **Exception** and **Error** are subclass of Throwable class. Exception class is used to handle exceptional conditions. Error class defines those exceptions which are not expected by the programmer to handle.

Exception class and its Subclasses in Throwable class Hierarchy

```
class java.lang.Object
 |
  + class java.lang.Throwable
                  |
                + class java.lang.Exception
                   |
                   + class java.awt.AWTException
                   |
                   +class java.lang.ClassNotFoundException
                   |
                   + class java.lang.CloneNotSupportedException
                   |
                   + class java.io.IOException
                   |
                   + class java.lang.IllegalAccessException
                   |
                   + class java.lang.InstantiationException
                   |
                   + class java.lang.InterruptedException
                   |
                   + class java.lang.NoSuchMethodException
                   |
                   + class java.lang.NoSuchMethodException
                    |
                   + class java.lang.RuntimeException
                   |
                   + class java.lang.ArithmeticException
                   |
                   + class java.lang.ArrayStoreException
                   |
                   + class java.lang.ClassCastException
                   |
                   + class java.util.EmptyStackException
                   |
```

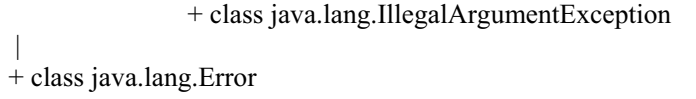


Figure 1: Throwable Class Partial Hierarchy

Now let us see in Table 1 some exceptions and their meaning.

Table1: Exceptions and Their Meaning

ArithmeticException	Division by zero or some other kind of arithmetic problem
ArrayIndexOutOfBoundsException	An array index is less than zero or greater than or equal to the array's length
FileNotFoundException	Reference to a file that cannot be found
IllegalArgumentException	Calling a method with an improper argument
IndexOutOfBoundsException	An array or string index is out of bounds
NullPointerException	Reference to an object that has not been instantiated
NumberFormatException	Use of an illegal number format, such as when calling a method
StringIndexOutOfBoundsException	A String index is less than zero or greater than or equal to the String's length

4.4.2 Runtime Exceptions

Programming errors that should be detected in testing for example index out of bounds, null pointer, illegal argument, etc. are known as runtime exception. Runtime exceptions do need to be handled (They are of the types that can be handled), but errors often cannot be handled.

Most of the runtime exceptions (members of the RuntimeException family) are thrown by the Java virtual machine itself. These exceptions are usually an indication of software bugs. You know problems with arrays, such as ArrayIndexOutOfBoundsException, or passed parameters, such as IllegalArgumentException, also could happen just about anywhere in a program. When exceptions like these are thrown, you have to fix the bugs that caused them to be thrown.

Sometimes you have to decide whether to throw a checked exception or an unchecked runtime exception. In this case you must look at the abnormal condition you are signalling. If you are throwing an exception to indicate an improper use of your class, then here you are signalling a software bug. In this case the class of exception you throw probably should descend from RuntimeException, which will make it unchecked. Otherwise, if you are throwing an exception to indicate not a software bug but an abnormal condition, then you have to deal with it every time your method is used. In other words your exception should be checked.

The *runtime* exceptions are not necessarily to be caught. You don't have to put a try-catch for runtime exceptions, for example, every integer division operation to catch a divide by zero or around every array variable to watch for whether it is going out of bounds. But it is better if you handle possible runtime exceptions. If you think there is a reasonable chance of such exceptions occurring.

Check Your Progress 2

1) Explain the exception types that can be caught by the following code.

```
try
{
//operations
}
catch (Exception e)
 {
//exception handling.
}
```

……

……

……

2) Write a partial program to show the use of finally clause.

……

……

……

3) Is there anything wrong with this exception handing code? Will this code compile?

```
try
{
//operation code...
}
catch (Exception e)
{
//exception handling
}
catch (ArithmeticException ae)
{
 // ArithmeticException handling
}
```

……

……

……

4) Differentiate between checked and unchecked exceptions.

……

……

……

4.5 THROWING EXCEPTIONS

If you don't want explicitly catching handle an exception and want to declare that your method throws the exception. This passes the responsibility to handle this exception to the method that invokes your method. This is done with the **throws** keyword.

Let us see the code snippet given below:

```
//code snippet
public static void copy (InputStream in, OutputStream out)
throws IOException
{
byte[] Mybuf = new byte[256];
while (true)
{
int bytesRead = in.read(Mybuf);
if (bytesRead == -1) break;
out.write(Mybuf, 0, bytesRead);
}
}
```

In this code snippet copy method throws IOException, and now the method invoke copy method is responsible for handling IOException.

Sometime single method may have to throw more than one type of exception. In this case the exception classes are just separated by commas. For example in the code:

```
public MyDecimal
public divide(MyDecimal value, int roundMode) throws ArithmeticException,
IllegalArgumentException
```

The divide method throws two exceptions ArithmeticException, and IllegalArgumentException.

See the program given below written for showing how IllegalArgumentException exceptions are thrown if arguments are not passed properly.

```
//program
class MyClock
{
int hours ,minutes, seconds;
public MyClock(int hours, int minutes, int seconds)
{
if (hours < 1 || hours > 12)
{
throw new IllegalArgumentException("Hours must be between 1 and 12");
}
if (minutes < 0 || minutes > 59)
{
throw new IllegalArgumentException("Minutes must be between 0 and 59");
}
if (seconds < 0 || seconds > 59)
{
throw new IllegalArgumentException("Seconds must be between 0 and 59");
}
this.hours   = hours;
this.minutes = minutes;
this.seconds = seconds;
}
public MyClock(int hours, int minutes)
{
this(hours, minutes, 0);
}
public MyClock(int hours)
{
this(hours, 0, 0);
}
```

```
}
public class ThrowTest
{
public static void main( String args [])
{
try
{
MyClock clock = new MyClock(12, 67,80);
}
catch( IllegalArgumentException e)
{
System.out.println("IllegalArgumentException is caught....");
}
}
}
```

Output:
IllegalArgumentException is caught....

Now let us see how own exception subclasses can be written.

4.6 WRITING EXCEPTION SUBCLASSES

Most of the exception subclasses inherit all their functionality from the superclass and they serve the purpose of exception handling. Sometimes you will need to create your own exceptions types to handle specific situations that arises in your applications. This you can do quite easily by just defining subclass of **Exception** class. Exception class does not define any method of its own .Of course it inherits methods of **Throwable** class. If you inherit Exception you have the method of Throwable class available for your class. If needed you can override the methods of Throwable class.

Some methods of Throwable class:

Throwable FillInStackTrce(): Fills in the execution stack trace.
String GetMessage() : Returns the detail message string of this throwable.
String ToString() : Returns a short description of this throwable.

Now see the program given below to create exception subclass which throws exception if argument passed to MyException class method compute has value greater that 10.

```
//program
MyException extends Exception
{
private int Value;
MyException (int a)
{
Value = a;
}
public String toString()
{
return ("MYException [for Value="+ Value +"]");
}
}
class ExceptionSubClassDemo
{
static void compute(int a) throws MyException
{
System.out.println("Call compute method ("+a+")");
```

```
if(a>10)
throw new MyException(a);
System.out.println("Normal Exit of compute method[for Value="+a +"]");
}
public static void main(String args[])
{
try
{
compute(5);
compute(25);
}
catch(MyException e)
{
System.out.println("MyException Caught :"+e);
}
}
}
```

Output:
Call compute method (5)
Normal Exit of compute method [for Value=5]
Call compute method (25)
MyException Caught: MYException [for Value=25]

Embedding information in an exception object

You can say that when an exception is thrown by you ,it is like performing a kind of structured go-to from the place in your program where an abnormal condition was detected to a place where it can be handled. The Java virtual machine uses the class of the exception object you throw to decide which catch clause, if any, should be allowed to handle the exception.

But you cannot take an exception just as a transfer control from one part of your program to another, it also transmits information. As mentioned earlier the exception is a full-fledged object that you can define yourself. Also you can embed information about the abnormal condition in the object before you throw it. The catch clause can then get the information by querying the exception object directly.

The Exception class allows you to specify a String detail message that can be retrieved by invoking getMessage() of Throwable class on the exception object. In your program at the time of defining it you can give the option of specifying a detail message like this:

```
class NotAcceptableValuException extends Exception
{
NotAcceptableValuException
{
}
NotAcceptableValuException (String msg)
{
super(msg);
}
}
```

Given the above declaration of NotAcceptableValuException, now you can create an object of NotAcceptableValuException in one of two ways:

new NotAcceptableValuException ()
new NotAcceptableValuException ("This Value is not Acceptable.")
Now a catch clause can query the object for a detail string, like this code snippet:
class MyValue

```
{
public void serveCustomers()
{
try
{
// operations
}
catch (NotAcceptableValuException e)
{
System.out.println (“NotAcceptableValuException Caught:”+e);
}
}
}
```

If during operations NotAcceptableValuException is generated and throw it will be handled by catch then statement
System.out.println (“NotAcceptableValuException Caught:”+e);

will print:
NotAcceptableValuException Caught:" This Value is not Acceptable." provided NotAcceptableValuException object is created as:
new NotAcceptableValuException ("This Value is not Acceptable.") ;

Check Your Progress 3

1) Explain how you can throw an exception from a method in Java.

……

……

……

…….

2) Write a program to create your own exception subclass that throws exception if the sum of two integers is greater that 99.

……

……

……

……
…

3) Dry run the following program and show the output:

```
//program
class MyStack
{
private int MaxSize;
private int size;
private Object[] ob1;
public MyStack(int cap)
{
ob1 = new Object[cap];
MaxSize = cap;
size = 0;
}
public void push(Object o)  throws StackException
{
if (size == MaxSize)
```

```
throw new StackException("overflow");
ob1[size++] = o;
}
public Object pop() throws StackException
{
if (size <= 0)
throw new StackException("underflow");
return ob1[--size];
}
public Object top()  throws StackException
{
if (size <= 0)
throw new StackException("underflow");
return ob1[size-1];
}
public int size()
{
return this.size;
}
}
class StackException extends Exception
{
StackException()  {}
StackException(String msg)
{
super(msg);
}
}
class StackTest
{
public static void main(String[] args)
{
MyStack s = new MyStack(5);
System.out.println("***** Welcome to Stack operations ****");
Test(s);
}
public static void Test(MyStack s)
{
try
{
s.push("Hi");
s.push("Java");
s.push(new Float(1.4));
s.push("Good for all");
s.push("Learn it");
s.push("Now"); // error!
}
catch(StackException se)
{
System.out.println(se);
}
try
{
System.out.println("Top of MyStack : " + s.top());
System.out.println("Popping data ...");
while (s.size() > 0)
System.out.println(s.pop());
}
catch(StackException se)
{
```

```
// This should never happen:
throw new InternalError(se.toString());
}
}
}
```

……

……

……

4.7 SUMMARY

This unit discusses how Java goes to great lengths to help you deal with error conditions. In this unit we have discussed how Java's exception mechanisms give a structured way to perform a go-to from the place where an error occurs to the code that knows how to handle the error. In this unit we have discussed different causes of exception, using try, catch, finally, throw and throws clauses in exception handling. This unit deals with ways to handle error conditions in a structured, methodical way. This unit discusses types of exceptions, Throwable class hierarchy, and explains how to write own exception subclasses.

4.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) An exception is a condition or a problem, due to which program stops execution from the point of occurrence of this condition or problem. If an exception has occurred the following things can be done:

   i. Fix the problem and try again.
   ii Exit the application with System.exit ()
   iii. Rethrow the exception to some other method or portion of code.

2) Yes, it is legal. It is not necessary for a try statement to have a catch statement if it has a finally statement. If the code in the try statement has multiple exit points and no catch clauses are associated to the code, then code in the finally statement is executed no matter how the try block is exited.

3)

```
//program
public class TwoCatch_Test
{
public static void main(String[] args)
{
try
{
int i = 1;
int data[] = {2,3,4,5};
System.out.println("Value at : " + data[2]);
i = i/(i-i);
}
catch( ArrayIndexOutOfBoundsException e)
{
System.out.println("Sorry you are trying to print beyond the size of data[]");
}
catch(ArithmeticException e)
{
System.out.println("Divide By 0 :"+e);
```

```
}
}
}
```

Output:
Value at: 4
Divide By 0: java.lang.ArithmeticException: / by zero

Check Your Progress 2

1) This code will handle catch exceptions of type Exception; therefore, it will catch any exception. This can be a poor implementation because you are losing valuable information about the type of exception being thrown and making your code less efficient. It is better to handle different exceptions on the basis of their actual type, which may be ArithmeticException, or IllegalArgumentException or anything else

2)
```
//Code snippet to explain finally
public void MyMethod(File file) throws Exception
{
FileInputStream stream = new FileInputStream(file);
try
{
// process stream object contents
}
finally
{
// No matter what happen in try block this statement will execute.
stream.close();
}
}
```
the code snippet given above give a hint that some file related operations are performed , in this code finally block will make sure in all the situations that object stream get closed.

3) This first handler catches exceptions of type Exception; therefore, it catches any exception, including ArithmeticException. In this situation the second handler could never be reached.

4) An unchecked exception is a type of exception that doesn't force you to handle it. It is optional to handle it, or ignore it. Exceptions instantiated from **RuntimeException** and its subclasses are considered unchecked exceptions. Checked exceptions are those exceptions that cannot be ignored during you write the code in your methods. The conceptual difference between checked and unchecked exceptions is that checked exceptions signal abnormal conditions that you have to deal with, and it is not the case with unchecked exceptions.

Check Your Progress 3

1) A method in Java can throw exception using throws keyword. In code snippet below MyMethod() throws an ArrayOutOfBoundsExceptio:
```
//code snippet
class TestTrows
{
void MyMethod() throws ArrayOutOfBoundsException
{
 // operation code
throw ArrayOutOfBoundsException("My ArrayOutOfBoundsException");
}
```

2) //program

```
class MySum extends Exception
{
int Sum;
MySum( int a , int b)
{
Sum = a+ b;
}
public String toString()
{
return ("MYException [for Sum="+ Sum + "]");
}
}
class ExceptionSumDemo
{
static void SumIt(int a, int b ) throws MySum
{
int Sum;
Sum= a+b;
System.out.println("Call SumIt ("+a+","+b+")");
if(Sum>99)
throw new MySum(a,b);
System.out.println("Normal Exit from  SumIt method [for Sum="+Sum +"]");
}
public static void main(String args[])
{
try
{
SumIt(20,50);
SumIt(90,11);
}
catch(MySum e)
{
System.out.println("MyException Caught :"+e);
}
}
}
```

Output:

```
Call SumIt (20,50)
Normal Exit from  SumIt method [for Sum=70]
Call SumIt (90,11)
MyException Caught :MYException [for Sum=101]
```

3) Output:

```
***** Welcome to Stack operations ****
StackException: overflow
Top of MyStack : Learn it
Popping data...
Learn it
Good for all
1.4
Java
Hi
```

UNIT 1 MULTITHREADED PROGRAMMING

Structure **Page Nos.**

1.0 INTRODUCTION

A thread is single sequence of execution that can run independently in an application. This unit covers the very important concept of multithreading in programming. Uses of thread in programs are good in terms of resource utilization of the system on which application(s) is running. Multithreaded programming is very useful in network and Internet applications development. In this unit you will learn what is multithreading, how thread works, how to write programs in Java using multithreading. Also, in this unit will be explained about thread-properties, synchronization, and interthread communication.

1.1 OBJECTIVES

After going through this unit, you will be able to:

- describe the concept of multithreading;
- explain the Java thread model;
- create and use threads in program;
- describe how to set the thread priorities;
- use the concept of synchronization in programming, and
- use inter-thread communication in programs.

1.2 MULTITHREADING: AN INTRODUCTION

Multithreaded programs support more than one concurrent thread of execution. This means they are able to simultaneously execute multiple sequences of instructions. Each instruction sequence has its own unique flow of control that is independent of all others. These independently executed instruction sequences are known as *threads*. Your PC has only a single CPU; you might ask how it can execute more than one thread at the same time? In single processor systems, only a single thread of execution occurs at a given instant. But multiple threads in a program increase the utilization of CPU.

The CPU quickly switches back and forth between several threads to create an illusion that the threads are executing at the same time. You know that single-processor systems support logical concurrency only. Physical concurrency is not supported by it. Logical concurrency is the characteristic exhibited when multiple threads execute

with separate, independent flow of control. On the other hand on a multiprocessor system, several threads can execute at the same time, and physical concurrency is achieved.

The advantage of multithreaded programs is that they support logical concurrency. Many programming languages support multiprogramming, as it is the logically concurrent execution of multiple programs. For example, a program can request the operating system to execute programs A, B and C by having it spawn a separate process for each program. These programs can run in a concurrent manner, depending upon the multiprogramming features supported by the underlying operating system. Multithreading differs from multiprogramming. Multithreading provides concurrency within the content of a single process. But multiprogramming also provides concurrency between processes. Threads are not complete processes in themselves. They are a separate flow of control that occurs within a process.

A process is the operating system object that is created when a program is executed

In *Figure1a and Figure 1 b* the difference between multithreading and multiprogramming is shown.

Multiprogramming:

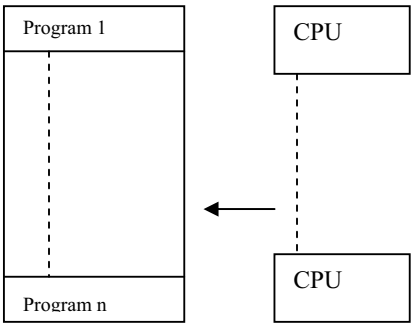


Figure 1a: Multiprogramming

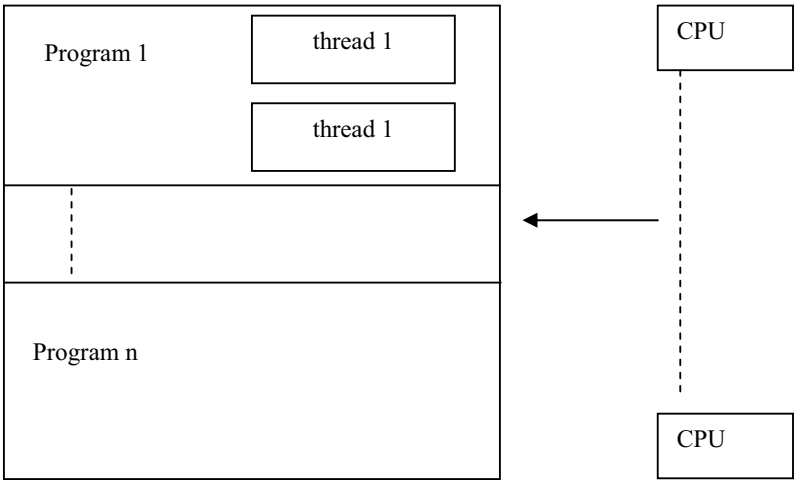


Figure 1b: Multithreading

Now let us see the advantages of multithreading.

Advantages of Multithreading

The advantages of multithreading are:

i. Concurrency can be used within a process to implement multiple instances of simultaneous services.
ii. Multithreading requires less processing overhead than multiprogramming because concurrent threads are able to share common resources more efficiently.

A multithreaded web server is one of the examples of multithreaded programming. Multithreaded web servers are able to efficiently handle multiple browser requests. They handle one request per processing thread.

iii. Multithreading enables programmers to write very efficient programs that make maximum use of the CPU. Unlike most other programming languages, Java provides built-in support for multithreaded programming. The Java run-time system depends on threads for many things. Java uses threads to enable the entire environment to be synchronous.

Now let us see how Java provides multithreading.

1.3 THE MAIN THREAD

When you start any Java program, one thread begins running immediately, which is called the ***main thread*** of that program. Within the main thread of any Java program you can create other 'child' threads. Every thread has its lifetime. During programming you should take care that the main thread is the last thread to finish execution because it performs various shutdown actions. In fact for some older JVM's if the main thread finishes before a child thread, then the Java run-time system may *hang*.

The main thread is created automatically when your program is started. The main thread of Java programs is controlled through an object of ***Thread*** *class*.

You can obtain a reference to Thread class object by calling the method current Thread( ), which is a static method:
static Thread currentThread( )

By using this method, you get a reference to the thread in which this method is called. Once you have a reference to the main thread, you can control it just like other threads created by you. In coming sections of this unit you will learn how to control threads.

Now let us see the program given below to obtain a reference to the main thread and the name of the main thread is set to MyFirstThread using setName (String) method of *Thread* class

```
//program
class Thread_Test
{
public static void main (String args [ ])
{
try
{
Thread threadRef  = Thread.currentThread( );
System.out.println("Current Thread :"+ threadRef);
System.out.println ("Before changing the name of main thread : "+ threadRef);
//change the internal name of the thread
threadRef.setName("MyFirstThread");
System.out.println ("After changing the name of main thread : "+threadRef);
}
```

```
catch (Exception e)
{
System.out.println("Main thread interrupted");
  }
}
}
```

Output:

```
Current Thread: Thread[main,5,main]
Before changing the name of main thread: Thread[main,5,main]
After changing the name of main thread: Thread[MyFirstThread,5,main]
```

In output you can see in square bracket ([]) the name of the thread, thread priority which is 5, by default and main is also the name of the group of threads to which this thread belongs. A thread group is a data structure that controls the state of a collection of threads as a whole. The particular run-time environment manages this process. You can see that the name of the thread is changed to MyFirstThread and is displayed in output.

☞ Check Your Progress 1

1) How does multithreading take place on a computer with a single CPU?

……

……

……

2) State the advantages of multithreading.

……

……

……

……..

3) How will you get reference to the main thread?

……

……

……

……..

4) What will happen if the main thread finishes before the child thread?

……

……

……

……..

The Java run-time system depends on threads for many activities, and all the class libraries are designed with multithreading in mind. Now let us see the Java thread model.

1.4 JAVA THREAD MODEL

The benefit of Java's multithreading is that a thread can pause without stopping other parts of your program. A paused thread can restart. A thread will be referred to as dead when the processing of the thread is completed. After a thread reaches the dead state, then it is not possible to restart it.

The thread exists as an object; threads have several well-defined states in addition to the dead states. These state are:

Ready State

When a thread is created, it doesn't begin executing immediately. You must first invoke start ( ) method to start the execution of the thread. In this process the thread scheduler must allocate CPU time to the Thread. A thread may also enter the ready state if it was stopped for a while and is ready to resume its execution.

Running State

Threads are born to run, and a thread is said to be in the running state when it is actually executing. It may leave this state for a number of reasons, which we will discuss in the next section of this unit.

Waiting State

A running thread may perform a number of actions that will cause it to wait. A common example is when the thread performs some type of input or output operations.

As you can see in *Figure 2* given below, a thread in the waiting state can go to the ready state and from there to the running state. Every thread after performing its operations has to go to the dead state.

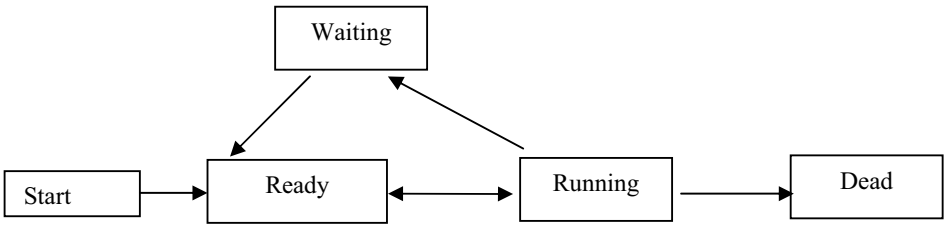


Figure 2: The Thread States

A thread begins as a ready thread and then enters the running state when the thread scheduler schedules it. The thread may be prompted by other threads and returned to the ready state, or it may wait on a resource, or simply stop for some time. When this happens, the thread enters the waiting state. To run again, the thread must enter the ready state. Finally, the thread will cease its execution and enter the dead state.

The multithreading system in Java is built upon the **Thread Class**, its methods and its companion interface, **Runnable**. To create a new thread, your program will either *extend Thread Class* or *implement the Runnable interface*. The Thread Class defines several methods that help in managing threads.

For example, if you have to create your own thread then you have to do one of the following things:

1)

```
class MyThread extends Thread
 {
  MyThread(arguments) // constructor
   {
   }//initialization
   public void run()
   {
   // perform operations
   }
   }
```

Write the following code to create a thread and start it running:

```
    MyThread p = new MyThread(arguments);
    p.start();
```

2)

```
    class MyThread implements Runnable
    {
    MyThread(arguments)
   {
    //initialization
   }
   public void run()
  {
  // perform operation
  }
  }
```

The Thread Class Constructors

The following are the Thread class constructors:

Thread()
Thread(Runnable target)
Thread (Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, Runnable target, String name, long stackSize)
Thread(ThreadGroup group, String name)

Thread Class Method

Some commonly used methods of Thread class are given below:

static Thread **currentThread**() Returns a reference to the currently executing thread object.

String **getName**() Returns the name of the thread in which it is called

int **getPriority**() Returns the Thread's priority

void **interrupt**() Used for Interrupting the thread.

static boolean **interrupted**() Used to tests whether the current thread has been interrupted or not.

boolean **isAlive**() Used for testing whether a tread is alive or not.

boolean **isDaemon**() Used for testing whether a thread is a daemon thread or not.

void **setName**(String NewName) Changes the name of the thread to NewName

void **setPriority**(int newPriority) Changes the priority of thread.

static void **sleep**(long millisec) Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.

void **start**() Used to begin execution of a thread .The Java Virtual Machine calls the run method of the thread in which this method is called.

String **toString**() Returns a string representation of thread.String includes the thread's name, priority, and thread group.

static void **yield**() Used to pause temporarily to currently executing thread object and allow other threads to execute.

static int **activeCount**() Returns the number of active threads in the current thread's thread group.

void **destroy**() Destroys the thread without any cleanup.

Creating Threads

Java provides native support for multithreading. This support is centered on the Java.lang.Thread class, the Java.lang.Runnable interface and methods of the Java.lang.object class. In Java, support for multithreaded programming is also provided through synchronized methods and statements.

The thread class provides the capability to create thread objects, each with its own separate flow of control. The thread class encapsulates the data and methods associated with separate threads of execution and enables multithreading to be integrated within Java's object oriented framework. The minimal multithreading support required of the Thread Class is specified by the java.lang.Runnable interface. This interface defines a single but important method run.
public void run( )

This method provides the entry point for a separate thread of execution. As we have discussed already Java provides two approaches for creating threads. In the first approach, you create a subclass of the Thread class and override the run( ) method to provide an entry point for the Thread's execution. When you create an instance of subclass of Thread class, you invoke start( ) method to cause the thread to execute as an independent sequence of instructions. The start( ) method is inherited from the Thread class. It initializes the thread using the operating system's multithreading capabilities, and invokes the run( ) method.

Now let us see the program given below for creating threads by inheriting the Thread class.

```
//program
class MyThreadDemo extends Thread
{
public String MyMessage [ ]=
{"Java","is","very","good","Programming","Language"};
MyThreadDemo(String s)
{
```

```
super(s);
}
public void run( )
{
String name = getName( );
for ( int i=0; i < MyMessage.length;i++)
{
Wait( );
System.out.println (name +":"+ MyMessage [i]);
}
}
void Wait( )
{
try
{
 sleep(1000);
}
catch (InterruptedException e)
{
System.out.println (" Thread is Interrupted");
}
}
}
class ThreadDemo
{
public static void main ( String args [ ])
{
MyThreadDemo          Td1= new MyThreadDemo("thread 1:");
MyThreadDemo          Td2= new MyThreadDemo("thread 2:");
Td1.start ( );
Td2.start ( );
boolean isAlive1 =  true;
boolean  isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
 isAlive1= false;
 System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
 isAlive2= false;
 System.out.println ("Thread 2 is dead");
}
 }
 while(isAlive1 || isAlive2);
}
}
```

Output:

```
thread 1::Java
thread 2::Java
thread 1::is
thread 2::is
thread 1::very
thread 2::very
thread 1::good
```

```
thread 2::good
thread 1::Programming
thread 2::Programming
thread 1::Language
thread 2::Language
Thread 1 is dead
Thread 2 is dead
```

This output shows how two threads execute in sequence, displaying information on the console. The program creates two threads of execution, Td 1 and Td2. The threads display the "Java","is","very","good","Programming","Language" message word by word, while waiting a short amount of time between each word. Because both threads share the console window, the program's output identifies which thread wrote to the console during the program's execution.

Now let us see how threads are created in Java by implementing the java.lang.Runnable interface. The Runnable interface consists of only one method, i.e, run ( )method that provides an entry point into your threads execution.

In order to run an object of class you have created, as an independent thread, you have to pass it as an argument to a constructor of class Thread.

```
//program
class MyThreadDemo  implements  Runnable
{
public String MyMessage [ ]=
{"Java","is","very","good","Programming","Language"};
String name;
public MyThreadDemo(String s)
{
name = s;
}
public void run( )
{
for ( int i=0; i < MyMessage.length;i++)
{
Wait( );
System.out.println (name +":"+ MyMessage [i]);
}
}
void Wait( )
{
try
{
 Thread.currentThread().sleep(1000);
}
catch (InterruptedException e)
{
System.out.println (" Thread is Interrupted");
}
}
}
class ThreadDemo1
{
public static void main ( String args [ ])
{
Thread  Td1= new Thread( new MyThreadDemo("thread 1:"));
Thread  Td2= new Thread(new MyThreadDemo("thread 2:"));
Td1.start ( );
```

```
Td2.start ( );
boolean isAlive1 =  true;
boolean  isAlive2 = true;
do
{
if (isAlive1 && ! Td1.isAlive( ))
{
 isAlive1= false;
 System.out.println ("Thread 1 is dead");
}
if (isAlive2 && !Td2.isAlive( ))
{
 isAlive2= false;
 System.out.println ("Thread 2 is dead");
}
 }
 while(isAlive1 || isAlive2);
}
}
```

Output:

```
thread 1::Java
thread 2::Java
thread 1::is
thread 2::is
thread 1::very
thread 2::very
thread 1::good
thread 2::good
thread 1::Programming
thread 2::Programming
thread 1::Language
Thread 1 is dead
thread 2::Language
Thread 2 is dead
```

This program is similar to previous program and also gives same output. The advantage of using the Runnable interface is that your class does not need to extend the thread class. This is a very helpful feature when you create multithreaded applets. The only disadvantage of this approach is that you have to do some more work to create and execute your own threads.

☞ Check Your Progress 2

1) Sate True/False for the following statements:

	T	F
i. Initial state of a newly created thread is Ready state.	☐	
ii. Ready, Running, Waiting and Dead states are thread states.	☐	
iii. When a thread is in execution it is in Waiting state.	☐	
iv. A dead thread can be restarted.	☐	
v. sart() method causes an object to begin executing as a separate thread.	☐	
vi. run( ) method must be implemented by all threads.	☐	

2) Explain how a thread is created by extending the Thread class.

……

……

……

……

……

3) Explain how threads are created by implementing Runnable interface

……

……

……

……

……

Java provides methods to assign different priorities to different threads. Now let us discuss how thread priority is handled in Java.

1.5 THREAD PRIORITIES

Java assigns to each *thread* a *priority***.** Thread priority determines how a thread should be treated with respect to others. Thread priority is an integer that specifies the relative priority of one thread to another. A thread's priority is used to decide which thread. A thread can voluntarily relinquish control. Threads relinquish control by explicitly yielding, sleeping, or blocking on pending Input/ Output operations. In this scenario, all other threads are examined, and the highest- priority thread that is ready to run gets the chance to use the CPU.

A higher-priority thread can pre-empt a low priority thread. In this case, a lower-priority thread that does not yield the processor is forcibly pre-empted. In cases where two threads with the same priority are competing for CPU cycles, the situation is handled differently by different operating systems. In Windows 98, threads of equal priority are time-sliced automatically in round-robin fashion. In case of Solaris, threads of equal priority must voluntarily yield control to their peers. If they don't the other threads will not run.

Java thread class has defined two constants MIN_PRIORITY and MAX_PRIORITY. Any thread priority lies between MIN_PRIORITY and MAX_PRIORITY.Currently, the value of MIN_PRIORITY is 1 and MAX_PRIORITY is 10.

The priority of a thread is set at the time of creation. It is set to the same priority as the Thread that created it. The default priority of a thread can be changed by using the setPriority( ) method of Thread class .

Final void setPriority (int Priority_Level) where Priority_Level specifies the new priority for the calling thread. The value of Priority_Level must be within the range MIN_PRIORITY and MAX_PRIORITY.

To return a thread to default priority, specify Norm_Priority, which is currently 5. You can obtain the current priority setting by calling the getPriority( ) method of thread class.
final int getPriority( ).

Most operating systems support one of two common approaches to thread scheduling.

Pre-emptive Scheduling: The highest priority thread continues to execute unless it dies, waits, or is pre-empted by a higher priority thread coming into existence.

Time Slicing: A thread executes for a specific slice of time and then enters the ready state. At this point, the thread scheduler determines whether it should return the thread to the ready state or schedule a different thread.

Both of these approaches have their advantages and disadvantages. Pre-emptive scheduling is more predictable. However, its disadvantage is that a higher priority thread could execute forever, preventing lower priority threads from executing. The time slicing approach is less predictable but is better in handling selfish threads. Windows and Macintosh implementations of Java follow a time slicing approach. Solaris and other Unix implementations follow a pre-emptive scheduling approach.

Now let us see this example program, it will help you to understand how to assign thread priority.

```
//program
class Thread_Priority
{
public static void main (String args [ ])
{
 try
{
Thread Td1  = new  Thread("Thread1");
Thread Td2  = new  Thread("Thread2");
System.out.println ("Before any change in default priority:");
System.out.println("The Priority of "+Td1.getName() +" is "+ Td1.getPriority());
System.out.println("The Priority of "+Td1.getName() +" is "+ Td2.getPriority());
//change in priority
Td1.setPriority(7);
Td2.setPriority(8);
System.out.println ("After changing in Priority:");
System.out.println("The Priority of "+Td1.getName() +" is "+ Td1.getPriority());
System.out.println("The Priority of "+Td1.getName() +" is "+ Td2.getPriority());
}
catch ( Exception e)
{
  System.out.println("Main thread interrupted");
 }
}
}
```

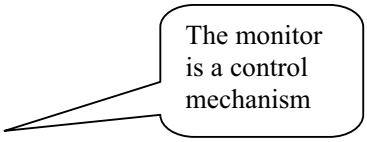
Output:
Before any change in default priority:
The Priority of Thread1 is 5
The Priority of Thread1 is 5
After changing in Priority:
The Priority of Thread1 is 7
The Priority of Thread1 is 8

Multithreaded programming introduces an asynchronous behaviour in programs. Therefore, it is necessary to have a way to ensure synchronization when program need it. Now let us see how Java provide synchronization.

1.6 SYNCHRONIZATION IN JAVA

If you want two threads to communicate and share a complicated data structure, such as a linked list or graph, you need some way to ensure that they don't conflict with each other. In other words, you must prevent one thread from writing data while another thread is in the middle of reading it. For this purpose, Java implements model of ***interprocess synchronizations.*** You know that once a thread enters a monitor, all other threads must wait until that thread exists in the monitor. Synchronization support is built in to the Java language.

There are many situations in which multiple threads must share access to common objects. There are times when you might want to coordinate access to shared resources. For example, in a database system, you would not want one thread to be updating a database record while another thread is trying to read from the database. Java enables you to coordinate the actions of multiple threads using synchronized methods and synchronized statements. Synchronization provides a simple monitor facility that can be used to provide mutual-exclusion between Java threads.



Once you have divided your program into separate threads, you need to define how they will communicate with each other. Synchronized methods are used to coordinate access to objects that are shared among multiple threads. These methods are declared with the *synchronized* keyword. Only one synchronized method at a time can be invoked for an object at a given point of time. When a synchronized method is invoked for a given object, it acquires the monitor for that object.In this case no other synchronized method may be invoked for that object until the monitor is released. This keeps synchronized methods in multiple threads without any conflict with each other.

When a synchronized method is invoked for a given object, it tries to acquire the lock for that object. If it succeeds, no other synchronized method may be invoked for that object until the lock is released. A lock is automatically released when the method completes its execution and returns. A lock may also be released when a synchronized method executes certain methods, such as wait( ).

Now let us see this example which explains how synchronized methods and object locks are used to coordinate access to a common object by multiple threads.

```
//program

class Call_Test
{
synchronized void callme (String msg)
{
//This prevents other threads from entering call( ) while another thread is using it.
System.out.print ("["+msg);
try
{
Thread.sleep (2000);
}
catch ( InterruptedException e)
{
System.out.println ("Thread is Interrupted");
}
System.out.println ("]");
}
}

class Call implements Runnable
```

```
{
String msg;
Call_Test ob1;
Thread t;
public Call (Call_Test tar, String s)
{
System.out.println("Inside caller method");
ob1= tar;
msg = s;
t = new Thread(this);
t.start();
}
public void run( )
{
ob1.callme(msg);
}
}

class Synchro_Test
{
public static void main (String args [ ])
{
Call_Test T= new Call_Test( );
Call ob1= new Call (T, "Hi");
Call ob2= new Call (T, "This ");
Call ob3= new Call (T, "is");
Call ob4= new Call (T, "Synchronization");
Call ob5= new Call (T, "Testing");
try
{
ob1.t.join( );
ob2.t.join( );
ob3.t.join( );
ob4.t.join( );
ob5.t.join( );
}
catch ( InterruptedException e)
{
System.out.println (" Interrupted");
}
}
}
```

Output:

```
Inside caller method
Inside caller method
Inside caller method
Inside caller method
Inside caller method
[Hi]
[This ]
[is]
[Synchronization]
[Testing]
```

If you run this program after removing synchronized keyword, you will find some output similar to:

```
Inside caller method
```

```
Inside caller method
Inside caller method
Inside caller method
[Hi[This [isInside caller method
[Synchronization[Testing]
]
]
]
]
```

This result is because when in program sleep( ) is called , the callme( ) method allows execution to switch to another thread. Due to this, output is a mix of the three message strings.

So if at any time you have a method, or group of methods, that manipulates the internal state of an object in a multithreaded situation, you should make these methods synchronized. It helps in avoiding conflict.

An effective means of achieving synchronization is to create synchronized methods within classes. But it will not work in all cases, for example, if you want to synchronize access to object's of a class that was not designed for multithreaded programming or the class does not use synchronized methods. In this situation, the *synchronized statement* is a solution. Synchronized statements are similar to synchronized methods. It is used to acquire a lock on an object before performing an action.

The synchronized statement is different from a synchronized method in the sense that it can be used with the lock of any object and the synchronized method can only be used with its object's (or class's) lock. It is also different in the sense that it applies to a statement block, rather than an entire method. The syntax of the synchronized statement is as follows:

```
synchronized (object)
{
// statement(s)
}
```

The statement(s) enclosed by the braces are only executed when the current thread acquires the lock for the object or class enclosed by parentheses.

Now let us see how interthread communication takes place in java.

1.7 INTERTHREAD COMMUNICATION

To avoid wastage of precious time of CPU, or to avoid polling, Java includes an interprocess communication mechanism via the *wait( ), notify( ),* and *notifyAll( )* methods. These methods are implemented as final methods, so all classes have them. These three methods can be called only from within a synchronized method.

wait( ): tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll().
notify( ): Wakes up a single thread that is waiting on this object's monitor.
notifyAll( ): Wakes up all threads that are waiting on this object's monitor, the highest priority Thread will be run first.

These methods are declared within objects as following:
final void wait( ) throws Interrupted Exception

final void notify( )
final void notifyAll( )

These methods enable you to place threads in a waiting pool until resources become available to satisfy the Thread's request. A separate resource monitor or controller process can be used to notify waiting threads that they are able to execute.

Let us see this example program written to control access of resource using wait() and notify () methods.

```
//program
class WaitNotifyTest implements Runnable
{
WaitNotifyTest ()
{
Thread th = new Thread (this);
th.start();
}
synchronized void notifyThat ()
{
System.out.println ("Notify the threads waiting");
this.notify();
}
synchronized public void run()
{
try
{
System.out.println("Thead is waiting....");
this.wait ();
}
catch (InterruptedException e){}
System.out.println ("Waiting thread notified");
}
}
class runWaightNotify
{
public static void main (String args[])
{
WaitNotifyTest wait_not = new WaitNotifyTest();
Thread.yield ();
wait_not.notifyThat();
}
}
```

Output:
Thead is waiting....
Notify the threads waiting
Waiting thread notified

☞ Check Your Progress 3

1) Explain the need of synchronized method

……

……

……

……

2) Run the program given below and write output of it:

```
//program
class One
{
public static void main (String arys[])
{
Two t = new Two( );
t. Display("Thread is created.");
t.start();
}
}
class Two extends Thread
{
public void Display(String s)
{
System.out.println(s);
}
public void run ()
{
System.out.println("Thread is running");
}
}
```

3) What is the output of the following program:

```
//program
class ThreadTest1
{
public static void main(String arrays [ ])
{
Thread1 th1 = new Thread1("q1");
Thread1 th2 = new Thread1("q2");
th1.start ();
th2.start ();
}
}
class Thread1 extends Thread
{
public void run ()
{
for (int i = 0; i <3 ;++i)
{
try
{
System.out.println("Thread: "+this.getName()+" is sleeping");
this.sleep (2000);
}
catch (InterruptedException e)
{
System.out.println("interrupted");
}
}
}
public Thread1(String s)
{
super (s);
}
}
```

4) Run this program and explain the output obtained

```
//program
public class WaitNotif
{
int i=0;
public static void main(String argv[])
{
WaitNotif ob = new WaitNotif();
ob.testmethod();
}
public void testmethod()
{
while(true)
{
try
{
wait();
}
catch (InterruptedException e)
{
System.out.println("Interrupted Exception");
}
}
}//End of testmethod
}
```

1.8 SUMMARY

This unit described the working of multithreading in Java. Also you have learned what is the main thread and when it is created in a Java program. Different states of threads are described in this unit. This unit explained how threads are created using Thread class and Runnable interface. It explained how thread priority is used to determine which thread is to execute next. This unit explains concept of synchronization, creating synchronous methods and inter thread communication. It is also explained how object locks are used to control access to shared resources.

1.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1) In single CPU system, the task scheduler of operating system allocates executions time to multiple tasks. By quickly switching between executing tasks, it creates the impression that tasks executes concurrently.

2) i Provide concurrent execution of multiple instances of different services.
   ii Better utilization of CPU time.

3) To get reference of main thread in any program write:
   Thread mainthreadref = Thread.currentThread();
   As first executable statement of the program.

4) There is a chance that Java runtime may hang.

Check Your Progress 2

1) i True
   ii True

iii False
iv False
v True
vi True

2) A thread can be constructed on any object that implements Runnable interface. If you have to create a thread by implementing Runnable interface, implement the only method run() of this interface. This can be done as:

```
class  MyThread implements Runnable
{
 //Body
public void run()// this is method of Runnable interface
{
//method body
}
}
```

3) One way to create a thread is, to create a new class that extends Thread class. The extended class must override the run method of Thread class. This run() method work as entry point for the newly created thread. Also in extended class must be start method to start the execution. This can be done as:

```
class  My Thread extends Thread
{
MyThread()
{
super("Name of Thread");
System.out.println("This is a new Thread");
Start();
}
public void run() // override Thread class run() method
{
//body
}
}
```

Check Your Progress 3

1) If there is a task to be performed by only one party at a time then some control mechanism is required to ensure that only one party does that activity at a time. For example if there is a need to access a shared resource with ensuring that resource will be used by only one thread at a time, then the method written to perform this operation should be declared as synchronized. Making such methods synchronized work in Java for such problem because if a thread is inside a synchronized method al other thread looking for this method have to wait until thread currently having control over the method leaves it.

2) Output:

Thread is created.
Thread is running

3) Output:

Thread: q1 is sleeping
Thread: q2 is sleeping
Thread: q1 is sleeping
Thread: q2 is sleeping

```
Thread: q1 is sleeping
Thread: q2 is sleeping
```

4) Output:

```
java.lang.IllegalMonitorStateException
at java.lang.Object.wait(Native Method)
at java.lang.Object.wait(Object.java:420)
at WaitNotif.testmethod(WaitNotif.java:16)
at WaitNotif.main(WaitNotif.java:8)
```

Exception in thread "main"

This exception occurs because of wait and notify methods are not called within synchronized methods.

UNIT 2 I/O IN JAVA

Structure **Page Nos.**

2.0 INTRODUCTION

Input is any information that is needed by a program to complete its execution. *Output* is any information that the program must convey to the user. Input and Output are essential for applications development.

To accept input a Java program opens a stream to a data source, such as a file or remote socket, and reads the information serially. Whether reading data from a file or from a socket, the concept of serially reading from, and writing to, different data sources is the same. For that very reason, it is essential to understand the features of top-level classes (Java.io.Reader, Java.io.Writer).

In this unit you will be working on some basics of I/O (Input–Output) in Java such as Files creation through *streams* in Java code. A stream is a linear, sequential flow of bytes of input or output data. Streams are written to the file system to create files. Streams can also be transferred over the Internet.

In this unit you will learn the basics of Java streams by reviewing the differences between byte and character streams, and the various stream classes available in the Java.io package. We will cover the standard process for standard Input (Reading from console) and standard output (writing to console).

2.1 OBJECTIVES

After going through this unit you will be able to:

- explain basics of I/O operations in Java;
- use stream classes in programming;
- take inputs from console;
- write output on console;
- read from files, and
- write to files.

2.2 I/O BASICS

Java input and output are based on the use of *streams*, or sequences of bytes that travel from a source to a destination over a communication path. If a program is writing to a

stream, you can consider it as a stream's *source*. If it is reading from a stream, it is the stream's *destination*. The communication path is dependent on the type of I/O being performed. It can consist of memory-to-memory transfers, a file system, a network, and other forms of I/O.

Streams are powerful because they abstract away the details of the communication path from input and output operations. This allows all I/O to be performed using a common set of methods. These methods can be extended to provide higher-level custom I/O capabilities.

Three streams given below are created automatically:

- System.out - standard output stream
- System.in - standard input stream
- System.err - standard error

An **InputStream** represents a stream of data from which data can be read. Again, this stream will be either directly connected to a device or else to another stream.

An **OutputStream** represents a stream to which data can be written. Typically, this stream will either be directly connected to a device, such as a file or a network connection, or to another output stream.

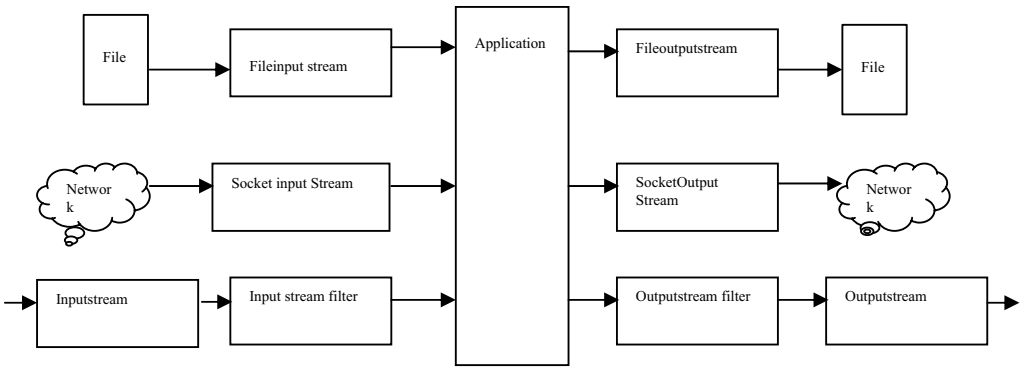


Figure 1: I/O stream basics

Java.io package

This package provides support for basic I/O operations. When you are dealing with the Java.io package some questions given below need to be addressed.

- What is the file format: text or binary?
- Do you want random access capability?
- Are you dealing with objects or non-objects?
- What are your sources and sinks for data?
- Do you need to use filtering (You will know about it in later section of this unit)?

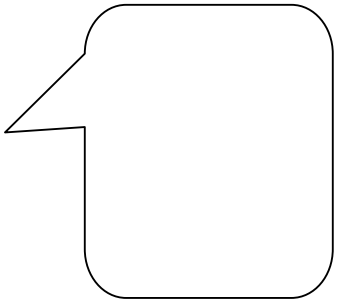
For example:

- If you are using binary data, such as integers or doubles, then use the InputStream and OutputStream classes.

- If you are using text data, then the Reader and Writer classes are right.

Exceptions Handling during I/O

Almost every input or output method throws an exception. Therefore, any time you do an I/O operation, the program needs to catch exceptions. There is a large hierarchy of I/O exceptions derived from IOException class. Typically you can just catch IOException, which catches all the derived class exceptions. However, some exceptions thrown by I/O methods are not in the IOException hierarchy, so you should be careful about exception handling during I/O operations.



2.3 STREAMS AND STREAM CLASSES

The Java model for I/O is entirely based on streams.

There are two types of streams: byte streams and character streams.

Byte streams carry integers with values that range from 0 to 255. A diversified data can be expressed in byte format, including numerical data, executable programs, and byte codes – the class file that runs a Java program.

Character Streams are specialized type of byte streams that can handle only textual data.

Most of the functionality available for byte streams is also provided for character streams. The methods for character streams generally accept parameters of data type *char,* while *byte* streams work with *byte* data types. The names of the methods in both sets of classes are almost identical except for the suffix, that is, character-stream classes end with the suffix Reader or Writer and byte-stream classes end with the suffix InputStream and OutputStream.

For example, to read files using character streams use the Java.io.FileReader class, and for reading it using byte streams use Java.io.FileInputStream.

Unless you are writing programs to work with binary data, such as image and sound files, use readers and writers (character streams) to read and write information for the following reasons:

- They can handle any character in the Unicode character set (while the byte streams are limited to ISO-Latin-1 8-bit bytes).
- They are easier to internationalize because they are not dependent upon a specific character encoding.
- They use buffering techniques internally and are therefore potentially much more efficient than byte streams.

Now let us discuss byte stream classes and character stream classes one by one.

2.3.1 Byte Stream Classes

Java defines two major classes of byte streams: InputStream and OutputStream. To provide a variety of I/O capabilities subclasses are derived from these InputStream and OutputStream classes.

InputStream class

The InputStream class defines methods for reading bytes or arrays of bytes, marking locations in the stream, skipping bytes of input, finding out the number of bytes available for reading, and resetting the current position within the stream. An input stream is automatically opened when created. The close() method can explicitly close a stream.

Methods of InputStream class

The basic method for getting data from any InputStream object is the read()method. public abstract int read() throws IOException: reads a single byte from the input stream and returns it.

public int read(byte[] bytes) throws IOException: fills an array with bytes read from the stream and returns the number of bytes read.

public int read(byte[] bytes, int offset, int length) throws IOException: fills an array from stream starting at position offset, up to length bytes. It returns either the number of bytes read or -1 for end of file.

public int available() throws IOException: the readmethod always blocks when there is no data available. To avoid blocking, program might need to ask ahead of time exactly how many bytes can safely read without blocking. The available method returns this number.

public long skip(long n): the skip() method skips over n bytes (passed as argument of skip()method) in a stream.

public synchronized void mark (int readLimit): this method marks the current position in the stream so it can backed up later.

OutputStream class

The OutputStream defines methods for writing bytes or arrays of bytes to the stream. An output stream is automatically opened when created. An Output stream can be explicitly closed with the close() method.

Methods of OutputStream class

public abstract void write(int b) throws IOException: writes a single byte of data to an output stream.

public void write(byte[] bytes) throws IOException: writes the entire contents of the bytes array to the output stream.

public void write(byte[] bytes, int offset, int length) throws IOException: writes length number of bytes starting at position offset from the bytes array.

The Java.io package contains several subclasses of InputStream and OutputStream that implement specific input or output functions. Some of these classes are:

- FileInputStream and FileOutputStream: Read data from or write data to a file on the native file system.
- PipedInputStream and PipedOutputStream : Implement the input and output components of a pipe. Pipes are used to channel the output from one program (or thread) into the input of another. A PipedInputStream must be connected to a PipedOutputStream and a PipedOutputStream must be connected to a PipedInputStream.
- ByteArrayInputStream and ByteArrayOutputStream : Read data from or write data to a byte array in memory.

ByteArrayOutputStream provides some additional methods not declared for OutputStream. The reset() method resets the output buffer to allow writing to restart at

the beginning of the buffer. The size() method returns the number of bytes that have been written to the buffer. The write to () method is new.

- SequenceInputStream: Concatenate multiple input streams into one input stream.

- StringBufferInputStream: Allow programs to read from a StringBuffer as if it were an input stream.

Now let us see how Input and Output is being handled in the program given below: this program creates a file and writes a string in it, and reads the number of bytes in file.

```
// program for I/O
import Java.lang.System;
import Java.io.FileInputStream;
import Java.io.FileOutputStream;
import Java.io.File;
import Java.io.IOException;
public class FileIOOperations {
public static void main(String args[]) throws IOException {
 // Create output file test.txt
FileOutputStream outStream = new FileOutputStream("test.txt");
String s = "This program is for Testing I/O Operations";
for(int i=0;i<s.length();++i)
outStream.write(s.charAt(i));
outStream.close();
// Open test.txt for input
FileInputStream inStream = new FileInputStream("test.txt");
int inBytes = inStream.available();
System.out.println("test.txt has "+inBytes+" available bytes");
byte inBuf[] = new byte[inBytes];
int bytesRead = inStream.read(inBuf,0,inBytes);
System.out.println(bytesRead+" bytes were read");
System.out.println(" Bytes read are: "+new String(inBuf));
inStream.close();
File f = new File("test.txt");
f.delete();
}
}
```

Output:
test.txt has 42 available bytes
42 bytes were read
Bytes read are: This program is for Testing I/O Operations.

Filtered Streams

One of the most powerful aspects of streams is that one stream can chain to the end of another. For example, the basic input stream only provides a read()method for reading bytes. If you want to read strings and integers, attach a special data input stream to an input stream and have methods for reading strings, integers, and even floats.

The FilterInputStream and FilterOutputStream classes provide the capability to chain streams together. The constructors for the FilterInputStream and FilterOutputStream take InputStream and OutputStream objects as parameters:

```
public FilterInputStream(InputStream in)
public FilterOutputStream(OutputStream out)
```

FilterInputStream has four filtering subclasses, -Buffer InputStream, Data

InputStream, LineNumberInputStream, and PushbackInputStream.
BufferedInputStream class: It maintains a buffer of the input data that it receives.

This eliminates the need to read from the stream's source every time an input byte is needed.

DataInputStream class: It implements the DataInput interface, a set of methods that allow objects and primitive data types to be read from a stream.

LineNumberInputStream class: This is used to keep track of input line numbers.

PushbackInputStream class: It provides the capability to push data back onto the stream that it is read from so that it can be read again.

The FilterOutputStream class provides three subclasses -BufferedOutputStream, DataOutputStream and Printstream.

BufferedOutputStream class: It is the output class analogous to the BufferedInputStream class. It buffers output so that output bytes can be written to devices in larger groups.

DataOutputStream class: It implements the DataOutput interface. It provides methods that write objects and primitive data types to streams so that they can be read by the DataInput interface methods.

PrintStream class: It provides the familiar print() and println() methods.

You can see in the program given below how objects of classes FileInputStream, FileOutputStream, BufferedInputStream, and BufferedOutputStream are used for I/O operations.

```
//program
import Java.io.*;
public class StreamsIODemo
{
  public static void main(String args[])
  {
  try
    {
     int a = 1;
     FileOutputStream fileout = new FileOutputStream("out.txt");
     BufferedOutputStream buffout = new BufferedOutputStream(fileout);
     while(a<=25)
     {
     buffout.write(a);
     a = a+3;
     }
     buffout.close();
     FileInputStream filein = new FileInputStream("out.txt");
     BufferedInputStream buffin = new BufferedInputStream(filein);
     int i=0;
     do
     {
     i=buffin.read();
     if (i!= -1)
     System.out.println(" "+ i);
     } while (i != -1) ;
```

```
buffin.close();
}
 catch (IOException e)
{
System.out.println("Eror Opening a file" + e);
}
}
}
```

Output:

```
1
4
7
10
13
16
19
22
25
```

2.3.2 Character Stream Classes

Character Streams are defined by using two class **Java.io.Reader and Java.io.Writer** hierarchies.

Both Reader and Writer are the abstract parent classes for character-stream based classes in the Java.io package. Reader classes are used to read 16-bit character streams and Writer classes are used to write to 16-bit character streams. The methods for reading from and writing to streams found in these two classes and their descendant classes (which we will discuss in the next section of this unit) given below:

```
int read()
int read(char cbuf[])
int read(char cbuf[], int offset, int length)
int write(int c)
int write(char cbuf[])
int write(char cbuf[], int offset, int length)
```

Specialized Descendant Stream Classes

There are several specialized stream subclasses of the Reader and Writer class to provide additional functionality. For example, the BufferedReader not only provides buffered reading for efficiency but also provides methods such as "readLine()" to read a line from the input.

The following class hierarchy shows a few of the specialized classes in the Java.io package:

Reader

- BufferedReader
- LineNumberReader
- FilterReader
- PushbackReader
- InputStreamReader
- FileReader
- StringReader

Writer:

Now let us see a program to understand how the read and write methods can be used.

```
import Java.io.*;
public class ReadWriteDemo
  {
  public static void main(String args[]) throws Exception
  {
   FileReader fileread = new FileReader("StrCap.Java");
   PrintWriter printwrite = new PrintWriter(System.out, true);
   char c[] = new char[10];
   int read = 0;
   while ((read = fileread.read(c)) != -1)
     printwrite.write(c, 0, read);
   fileread.close();
   printwrite.close();
  }
  }
```

Output:

```
class StrCap
{
public static void main(String[] args)
{
StringBuffer StrB = new StringBuffer("Object Oriented Programming is possible in
Java");
String  Hi = new  String("This morning is very good");
System.out.println("Initial Capacity of StrB is :"+StrB.capacity());
System.out.println("Initial length of StrB is :"+StrB.length());
//System.out.println("value displayed by the expression Hi.capacity() is:
"+Hi.capacity());
System.out.println("value displayed by the expression Hi.length() is: "+Hi.length());
System.out.println("value displayed by the expression Hi.charAt() is:
"+Hi.charAt(10));
 }
}
```

Note: Output of this program is the content of file StrCap.Java.You can refer unit 3 of this block to see StrCap.Java file.

☞ Check Your Progress 1

1) What is stream? Differentiate between stream source and stream destination.

……

……

……

……

2) Write a program for I/O operation using BufferedInputStream and BufferedOutputStream

……

……

……

3) Write a program using FileReader and PrintWriter classes for file handling.

……

……

……

2.4 THE PREDEFINED STREAMS

Java automatically imports the Java.lang package. This package defines a class called **System**, which encapsulates several aspects of run-time environment. **System** also contains three predefined stream objects, **in, out, and err.** These **objects** are declared as public and static within System. This means they can be used by other parts of your program without reference to your **System** object.

Access to standard input, standard output and standard error streams are provided via public static System.in, System.out and System.err objects. These are usually automatically associated with a user's keyboard and screen.

System.out refers to standard output stream. By default this is the console.
System.in refers to standard input, which is keyboard by default.
System.err refers to standard error stream, which also is console by default.
System.in is an object of InputStream. **System.out and System.err** are objects of **PrintStream.** These are byte streams, even though they typically are used to read and write characters from and to the console.

The predefined PrintStreams, out, and err within system class are useful for printing diagnostics when debugging Java programs and applets. The standard input stream System.in is available, for reading inputs.

2.5 READING FROM AND WRITING TO, CONSOLE

Now we will discuss how you can take input from console and see the output on console.

Reading Console Input

Java takes input from console by reading from System.in. It can be associated with these sources with reader and sink objects by wrapping them in a reader object. For System.in an InputStreamReader is appropriate. This can be further wrapped in a BufferedReader as given below, if a line-based input is required.
BufferedReader br = new BufferedReader (new InputStreamReader(System.in))

After this statement br is a character-based stream that is linked to the console through Sytem.in

The program given below is for receiving console input in any of your Java applications**.**

```
//program
import Java.io.*;
class ConsoleInput
 {
  static String readLine()
   {
     StringBuffer response = new StringBuffer();
     try
```

```
        {
        BufferedInputStream buff = new BufferedInputStream(System.in);
        int in = 0;
        char inChar;
        do
        {
          in = buff.read();
          inChar = (char) in;
          if (in != -1)
          {
           response.append(inChar);
           }
          } while ((in != 1) & (inChar != '\n'));
          buff.close();
          return response.toString();
         }
        catch (IOException e)
           {
          System.out.println("Exception: " + e.getMessage());
          return null;
          }
    }
    public static void main(String[] arguments)
      {
       System.out.print("\nWhat is your name? ");
       String input = ConsoleInput.readLine();
       System.out.println("\nHello, " + input);
      }
      }
```

Output:

C:\JAVA\BIN>Java ConsoleInput

What is your name? Java Tutorial

Hello, Java Tutorial

Writing Console Output

You have to use System.out for standard Output in Java. It is mostly used for tracing the errors or for sample programs. These sources can be associated with a writer and sink objects by wrapping them in writer object. For standard output, an OutputStreamWriter object can be used, but this is often used to retain the functionality of print and println methods. In this case the appropriate writer is PrintWriter. The second argument to PrintWriter constructor requests that the output will be flushed whenever the println method is used. This avoids the need to write explicit calls to the flush method in order to cause the pending output to appear on the screen. PrintWriter is different from other input/output classes as it doesn't throw an IOException. It is necessary to send check Error message, which returns true if an error has occurred. One more side effect of this method is that it flushes the stream. You can create PrintWriter object as given below.

PrintWriter pw = new PrintWriter (System.out, true)

The ReadWriteDemo program discussed earlier in this section 2.3.2 of this unit is for reading and then displaying the content of a file. This program will give you an idea how to use FileReader and PrintWriter classes.

You may have observed that close()method is not required for objects created for standard input and output. You should have a question in mind–whether to use System.out or PrintWriter? There is nothing wrong in using System.out for sample programs but for real world applications PrintWriter is easier.

2.6 READING AND WRITING FILES

The streams are most often used for the standard input (the keyboard) and the standard output (theCRT display). Alternatively, input can arrive from a disk file using "input redirection", and output can be written to a disk file using "output redirection".

I/O redirection is convenient, but there are limitations to it. It is not possible to read data from a file using input redirection *and* receive user input from the keyboard at same time. Also, it is not possible to read or write multiple files using input redirection. A more flexible mechanism to read or write disk files is available in Java through its *file streams*.

Java has two file streams – the *file reader stream* and the *file writer stream*. Unlike the standard I/O streams, file stream must explicitly "open" the stream before using it.

Although, it is not necessary to close after operation is over, but it is a good practice to "close" the stream.

Reading Files

Let's begin with the FileReader class. As with keyboard input, it is most efficient to work through the BufferedReader class. If input is text to be read from a file, let us say "input.txt," it is opened as a file input stream as follows:

```
BufferedReader inputFile=new BufferedReader(new FileReader("input.txt"));
```

The line above opens, input.txt as a FileReader object and passes it to the constructor of the BufferedReader class. The result is a BufferedReader object named inputFile.

To read a line of text from input.txt, use the readLine() method of the BufferedReader class.

```
String s = inputFile.readLine();
```

You can see that input.txt is *not* being read using input redirection. It is explicitly opened as a file input stream. This means that the keyboard is still available for input. So, user can take the name of a file, instead of "hard coding".

Once you are finished with the operations on file, the file stream is closed as follows:
inputFile.close();

Some additional file I/O services are available through Java's File class, which supports simple operations with filenames and paths. For example, if fileName is a string containing the name of a file, the following code checks if the file exists and, if so, proceeds only if the user enters "y" to continue.

```
File f = new File(fileName);
if (f.exists())
{
System.out.print("File already exists. Overwrite (y/n)? ");
if(!stdin.readLine().toLowerCase().equals("y"))
return;
}
```

See the program written below open a text file called input.txt and to count the number of lines and characters in that file.

```
//program
import Java.io.*;
public class FileOperation
{
public static void main(String[] args) throws IOException
{
// the file must be called ‘input.txt’
String s = “input.txt”
File f = new File(s);
//check if file exists
if (!f.exists())
{
System.out.println("\'" + s + "\' does not exit!");
return;
}
// open disk file for input
BufferedReader inputFile = new BufferedReader(new FileReader(s));
// read lines from the disk file, compute stats
String line;
int nLines = 0;
int nCharacters = 0;
while ((line = inputFile.readLine()) != null)
{
nLines++;
nCharacters += line.length();
}
// output file statistics
System.out.println("File statistics for \'" + s + "\'...");
System.out.println("Number of lines = " + nLines);
System.out.println("Number of characters = " + nCharacters);
inputFile.close();
}
}
```

Output:
File statistics for ‘input.txt’…
Number of lines = 3
Number of characters = 7

Writing Files

You can open a file output stream to which text can be written. For this use the FileWriter class. As always, it is best to buffer the output. The following code sets up a buffered file writer stream named outFile to write text into a file named output.txt.

```
PrintWriter outFile = new PrintWriter(new BufferedWriter(new
FileWriter("output.txt"));
```

The object outFile, is an object of PrintWriter class, just like System.out. If a string, s, contains some text, to be written in “output.text”. It is written to the file as follows:

```
outFile.println(s);
```

When finished, the file is closed as:

```
outFile.close();
```

FileWriter constructor can be used with two arguments, where the second argument is a boolean type specifying an "append" option. For example, the expression new

FileWriter("output.txt", true) opens “output.txt” as a file output stream. If the file currently exists, subsequent output is appended to the file.

One more possibility is of opening an existing *read-only* file for writing. In this case, the program terminates with an “access is denied” exception. This should be caught and dealt within the program.

```
import Java.io.*;
class FileWriteDemo
{
public static void main(String[] args) throws IOException
{
// open keyboard for input
BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
String s = "output.txt";
// check if output file exists
File f = new File(s);
if (f.exists())
{
  System.out.print("Overwrite " + s + " (y/n)? ");
   if(!stdin.readLine().toLowerCase().equals("y"))
   return;
 }
  // open file for output
 PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter(s)));
 System.out.println("Enter some text on the keyboard...");
 System.out.println("(^z to terminate)");
// read from keyboard, write to file output stream
String s2;
while ((s2 = stdin.readLine()) != null)
outFile.println(s2);
// close disk file
outFile.close();
}
}
```

Output:

```
Enter some text on the keyboard...
(^z to terminate)
hello students ! enjoying Java Session
^Z
```

Open out.txt you will find

“hello students ! enjoying Java Session” is stored in it.

☞ Check Your Progress 2

1) Which class may be used for reading from console?

……

……

2) Object of which class may be used for writing on console.

……

……

3) Write a program to read the output of a file and display it on console.

……

……

……

2.7 THE TRANSIENT AND VOLATILE MODIFIERS

Object serialization is very important aspect of I/O programming. Now we will discuss about the serializations.

Object serialization

It takes all the data attributes, writes them out as an object, and reads them back in as an object. For an object to be saved to a disk file it needs to be converted to a serial form. An object can be used with streams by implementing the serializable interface. The serialization is used to indicate that objects of that class can be saved and retrieved in serial form. Object serialization is quite useful when you need to use object persistence. By object persistence, the stored object continues to serve the purpose even when no Java program is running and stored information can be retrieved in a program so it can resume functioning unlike the other objects that cease to exist when object stops running.

DataOuputStreams and DataInputStreams are used to write each attribute out individually, and then can read them back in on the other end. But to deal with the entire object, not its individual attributes, store away an object or send it over a stream of objects. Object serialization takes the data members of an object and stores them away or retrieves them, or sends them over a stream.

ObjectInput interface is used for input, which extends the DataInput interface, and ObjectOutput interface is used for output, which extends DataOutput. You are still going to have the methods readInt(), writeInt() and so forth. ObjectInputStream, which implements ObjectInput, is going to read what ObjectOutputStream produces.

Working of object serialization

For ObjectInput and ObjectOutput interface, the class must be serializable. The serializable characteristic is assigned when a class is first defined. Your class must implement the serializable interface. This marker is an interface that says to the Java virtual machine that you want to allow this class to be serializable. You don't have to add any additional methods or anything.

There exists, a couple of other features of a serializable class. First, it has a zero parameter constructor. When you read the object, it needs to be able to construct and allocate memory for an object, and it is going to fill in that memory from what it has read from the serial stream. The static fields, or class attributes, are not saved because they are not part of an object.

If you do not want a data attribute to be serialized, you can make it transient. That would save on the amount of storage or transmission required to transmit an object. The transient indicates that the variable is not part of the persistent state of the object and will not be saved when the object is archived. Java defines two types of modifiers **Transient** and **Volatile.**

The volatile indicates that the variable is modified asynchronously by concurrently running threads.

Transient Keyword

When an object that can be serialized, you have to consider whether all the instance variables of the object will be saved or not. Sometimes you have some objects or sub objects which carry sensitive information like password. If you serialize such objects even if information (sensitive information) is private in that object if can be accessed

from outside. To control this you can turn off serialization on a field- by-field basis using the transient keyword.

See the program given below to create a login object that keeps information about a login session. In case you want to store the login data, but without the password, the easiest way to do it is to implements **Serializable** and mark the **password** field as **transient**.

```
//Program
import Java.io.*;
import Java.util.*;
public class SerialDemo implements Serializable
{
 private Date date = new Date();
 private String username;
 private transient String password;
 SerialDemo(String name, String pwd)
 {
  username = name;
  password = pwd;
 }
 public String toString()
  {
  String pwd = (password == null) ? "(n/a)" : password;
  return "Logon info: \n   " + "Username: " + username +
    "\n   Date: " + date +  "\n   Password: " + pwd;
  }
 public static void main(String[] args)
 throws IOException, ClassNotFoundException
  {
  SerialDemo a = new SerialDemo("Java", "sun");
  System.out.println( "Login is = " + a);
  ObjectOutputStream 0 = new ObjectOutputStream( new
  FileOutputStream("Login.out"));
  0.writeObject(a);
  0.close();
  // Delay:
  int seconds = 10;
  long t = System.currentTimeMillis()+ seconds * 1000;
  while(System.currentTimeMillis() < t)
    ;
  // Now get them back:
  ObjectInputStream in = new ObjectInputStream(new
  FileInputStream("Login.out"));
  System.out.println("Recovering object at " + new Date());
  a = (SerialDemo)in.readObject();
  System.out.println( "login a = " + a);
  }
}
```

Output:

```
Login is = Logon info:
Username: Java
Date: Thu Feb 03 04:06:22 GMT+05:30 2005
Password: sun
Recovering object at Thu Feb 03 04:06:32 GMT+05:30 2005
login a = Logon info:
Username: Java
```

Date: Thu Feb 03 04:06:22 GMT+05:30 2005
Password: (n/a)

In the above exercise Date and Username fields are ordinary (not **transient**), and thus are automatically serialized. However, the **password** is **transient**, and so it is not stored on the disk. Also the serialization mechanism makes no attempt to recover it. The **transient** keyword is for use with **Serializable** objects only.

Another example of transient variable is an object referring to a file or an input stream. Such an object must be created anew when it is part of a serialized object loaded from an object stream

```
// Donot serialize this field
private transient FileWriter outfile;
```

Volatile Modifier

The volatile modifier is used when you are working with multiple threads. The Java language allows threads that access shared variables to keep private working copies of the variables. This allows a more efficient implementation of multiple threads. These working copies need to be reconciled with the master copies in the shared (main) memory only at prescribed synchronization points, namely when objects are locked or unlocked. As a rule, to ensure that shared variables are consistently and reliably updated, a thread should ensure that it has exclusive use of such variables by obtaining a lock and conventionally enforcing mutual exclusion for those shared variables. Only variables may be volatile. Declaring them so indicates that such methods may be modified asynchronously.

2.8 USING INSTANCE OF NATIVE METHODS

You will often feel the requirement that a Java application *must* communicate with the environment outside of Java. This is, perhaps, the main reason for the existence of native methods. The Java implementation needs to communicate with the underlying system – such as an operating system (as Solaris or Win32, or) a Web browser, custom hardware, (such as a PDA, Set-top-device,) etc. Regardless of the underlying system, there must be a mechanism in Java to communicate with that system. Native methods provide a simple clean approach to providing this interface between. Java and non-Java world without burdening the rest of the Java application with special knowledge.

Native Method is a method, which is not written in Java and is outside of the JVM in a library. This feature is not special to Java. Most languages provide some mechanism to call routines written in another language. In C++, you must use the extern “C” statement to signal that the C++ compiler is making a call to C functions.

To declare a native method in Java, a method is preceded with native modifiers much like you use the public or static modifiers, but don’t define any body for the method simply place a semicolon in its place.

For example:

```
public native int meth();
```

The following class defines a variety of native methods:

```
public class IHaveNatives
{
native public void Native1(int x) ;
native static public long Native2();
native synchronized private float Native3( Object o ) ;
```

```
native void Native4(int[] ary) throws Exception ;
}
```

Native methods can be static methods, thus not requiring the creation of an object (or instance of a class). This is often convenient when using native methods to access an existing C-based library. Naturally, native methods can limit their visibility with the public, private, protected, or unspecified *default* access.

Every other Java method modifier can be used along with native, except *abstract*. This is logical, because the native modifier implies that an implementation exists, and the abstract modifier insists that there is no implementation.

The Following program is a simple demonstration of native method implementation.

```
class ShowMsgBox
 {
  public static void main(String [] args)
   {
    ShowMsgBox app = new ShowMsgBox();
    app.ShowMessage("Generated with Native Method");
    }
  private native void ShowMessage(String msg);
   {
    System.loadLibrary("MsgImpl");
   }
   }
```

☞ Check You Progress 3

1) What is Serialization?

……

……

……..

……

2) Differentiate between Transient & Volatile keyword.

……

……

……..

……

3) What are native method?

……

……

……..

……

……

……

2.9 SUMMARY

This unit covers various methods of I/O streams-binary, character and object in Java. This unit briefs that input and output in the Java language is organised around the concept of streams. All input is done through subclasses of InputStream and all output is done through subclasses of OutputStream. (Except for RandomAccessFile). We have covered how various streams can be combined together to get the added functionality of standard input and stream input. In this unit you have also learned the operations of reading from a file and writing to a file. For this purpose objects of FileReader and FileWriter classes are used.

2.10 SOLUTIONS/ ANSWERS

Check Your Progress 1

1) A stream is a sequence of bytes of undetermined length that travel from one place to another over a communication path.
Places from where the streams are picked–up are known as stream source. A source may be a file, input device or a network place-generating stream.
Places where streams are received or stored are known as stream destination. A stream destination may be a file, output device or a network place ready to receive stream.

2) This IO program is written using FileInputStream, BufferedInputStream , FileOutputStream, and BufferedOutputStream classes.

```
import java.io.*;
public class IOBuffer
{
  public static void main(String args[])
  {
  try
     {
    int x = 0;
    FileOutputStream FO = new FileOutputStream("test.txt");
    BufferedOutputStream BO = new BufferedOutputStream(FO);
    //Writing Data in BO
    while(x<=25)
    {
     BO.write(x);
      x = x+2;
    }
    BO.close();
    FileInputStream FI = new FileInputStream("test.txt");
    BufferedInputStream BI= new BufferedInputStream(FI);
    int i=0;
    do
    {
      i=BI.read();
      if (i!= -1)
      System.out.println(" " +i);
     } while (i != -1) ;
    BI.close();
    }
    catch (IOException e)
     {
```

```
            System.out.println("Eror Opening a file" + e);
           }
         }
       }
```

3) This program is written using FileReader and PrintWriter classes.

```
import java.io.*;
public class FRPW
{
public static void main(String args[]) throws Exception
    {
    FileReader FR = new FileReader("Intro.txt");
    PrintWriter PW = new PrintWriter(System.out, true);
    char c[] = new char[10];
    int read = 0;
    while ((read = FR.read(c)) != -1)
    PW.write(c, 0, read);
    FR.close();
    PW.close();
   }
  }
```

Check Your Progress 2

1) InputStream class

2) PrintStream class

3) This program Reads the content from Intro.txt file and print it on console.

```
import java.io.*;
public class PrintConsol
{
 public static void main(String[] args)
 {
  try
   {
   FileInputStream FIS = new FileInputStream("Intro.txt");
    int n;
    while ((n = FIS.available()) > 0)
    {
    byte[] b = new byte[n];
    int result = FIS.read(b);
    if (result == -1) break;
    String s = new String(b);
    System.out.print(s);
     } // end while
    FIS.close();
   } // end try
   catch (IOException e)
   {
   System.err.println(e);
   }
  System.out.println();
 }
}
```

Check Your Progress 3

1) Serialization is a way of implementation that makes objects of a class such that they are saved and retrieved in serial form. Serialization helps in making objects persistent.

2) Volatile indicates that concurrent running threads can modify the variable asynchronously. Volatile variables are used when multiple threads are doing the work. Transient keyword is used to declare those variables whose value need not persist when an object is stored.

3) Native methods are those methods, which are not written, in Java but they communicate to with Java applications to provide connectivity or to attach some systems such as OS, Web browsers, or PDA etc.

UNIT 3 STRINGS AND CHARACTERS

Structure **Page Nos.**

3.0 INTRODUCTION

In programming we use several data types as per our needs. If you observe throughout your problem solving study next to numeric data types character and strings are the most important data type that are used. In many of the programming languages strings are stored in arrays of characters (*e.g.* C, C++, Pascal, ...). However, in Java strings are a separate object type, called String. You are already using objects of String type in your programming exercises of previous units. A string is a set of character, such as "Hi". The Java platform contains three classes that you can use when working with character data: Character, String and StringBuffer. Java implements strings as object of String class when no change in the string is needed, and objects of StringBuffer class are used when there is need of manipulating of in the contents of string. In this unit we will discuss about different constructors, operations like concatenation of strings, comparison of strings, insertion in a string etc. You will also study about character extraction from a string, searching in a string, and conversion of different types of data into string form.

3.1 OBJECTIVES

After going through this unit you will be able to:

- explain Fundamentals of Characters and Strings;
- use different String and StringBuffer constructors;
- apply special string operations;
- extract characters from a string;
- perform such string searching & comparison of strings;
- data conversion using valueOf ( ) methods, and
- use StringBuffer class and its methods.

3.2 FUNDAMENTALS OF CHARACTERS AND STRINGS

Java provides three classes to deal with characters and strings. These are:

Character: Object of this class can hold a single character value. This class also defines some methods that can manipulate or check single-character data.

String: Objects of this class are used to deal with the strings, which are unchanging during processing.

String Buffer: Objects of this class are used for storing those strings which are expected to be manipulated during processing.

Characters

An object of Character type can contain only a single character value. You use a Character object instead of a primitive char variable when an object is required. For example, when passing a character value into a method that changes the value or when placing a character value into a Java defined data structure, which requires object. Vector is one of data structure that requires objects.

Now let us take an example program to see how Character objects are created and used.

In this program some character objects are created and it displays some information about them.

```
//Program
public class CharacterObj
{
public static void main(String args[])
{
Character Cob1 = new Character('a');
Character Cob2= new Character('b');
Character Cob3 = new Character('c');
int difference = Cob1.compareTo(Cob2);
if (difference == 0)
System.out.println(Cob1+"   is equal to  "+Cob2);
else
System.out.println(Cob1+"  is not equal to  "+Cob2);
System.out.println("Cob1 is " + ((Cob2.equals(Cob3)) ? "equal" : "not equal")+ " to
Cob3.");
}
}
```

Output:
a is not equal to b
Cob1 is not equal to Cob3.

In the above CharacterObj program following constructors and methods provided by the Character class are used.

Character (char): This is the Character class only constructor. It is used to create a Character object containing the value provided by the argument. But once a Character object has been created, the value it contains cannot be changed.

Compare to (Character): This instance method is used to compare the values held by two character objects. The value of the object on which the method is called and the value of the object passed as argument to the method.

For example, the statement

int difference = Cob1.compareTo(Cob2), in the above program.

This method returns an integer indicating whether the value in the current object is greater than, equal to, or less than the value held by the argument.

Equals (Character): This instance method is used to compare the value held by the current object with the value held by another (This method returns true if the values held by both objects are equal) object passed as on argument.

For example, in the statement

Cob2.equals(Cob3)
value held by object **Cob2** and the value held by **Cob3** will be compared.

In addition to the methods used in program CharacterObj, methods given below are also available in Character class.

To_String (): This instance method is used to convert the object to a string. The resulting string will have one character in it and contains the value held by the character object.

Char_Value(): An instance method that returns the value held by the character object as a primitive char value.

IsUpperCase (char): This method is used to determine whether a primitive char value is uppercase or not. It returns true if character is in upper case.

String and StringBuffer Classes

Java provides two classes for handling string values, which are **String** and

Can you tell why two String Classes?

The String class is provided for strings whose value will not change. For example, in program you write a method that requires string data and it is not going to modify the string.

The StringBuffer class is used for strings that will be modified. String buffers are generally used for constructing character data dynamically, for example, when you need to store information, which may change.

Content of strings do not change that is why they are more efficient to use than string buffers. So it is good to use String class wherever you need **fixed-length** objects that are **immutable (**size cannot be altered).

For example, the string contained by myString object given below cannot be changed.
String myString = “This content cannot be changed!”.

☞ Check Your Progress 1

1) Write a program to find the case (upper or lower) of a character.

……

……

2) Explain the use of equal () method with the help of code statements.

……

……

……

3) When should StringBuffer object be preferred over String object?

……

……

……

……

3.3 THE STRING CLASS

Now let us discuss String class in detail. There are many constructors provided in Java to create string objects. Some of them are given below.

public String(): Used to create a String object that represents an empty character sequence.

public String(String value): Used to create a String object that represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the string passed as an argument.

public String(char value[]): New object of string is created using the sequence of characters currently contained in the character array argument.

public String(char value[], int offset, int count): A new string object created contains characters from a sub-array of the character array argument.

public String(byte bytes[], String enc) throws Unsupported Encoding Exception: Used to construct a new String by converting the specified array of bytes using the specified character encoding.

public String(byte bytes[], int offset, int length): Used to create an object of String by converting the specified sub-array of bytes using the platform's default character encoding.

public String(byte bytes[]): Used to create an object of String by converting the specified array of bytes using the platform's default character encoding.

public String(StringBuffer buffer) : Used to create an object of String by using existing StringBuffer object which is passed as argument.

String class provides some important methods for examining individual characters of the strings, for comparing strings, for searching strings, for extracting sub-strings, and for creating a copy of a string with all characters translated to uppercase or to lowercase.

The Java language also provides special support for the string concatenation operator (+), and for conversion of other objects to strings.

Note: When you create a String object, and, if it has the same **value** as another object, Java will point both object references to the same memory location.

3.4 STRING OPERATIONS

String class provides methods for handling of String objects. String class methods can be grouped in index methods, value Of () methods and sub-string methods. Methods of these groups are discussed below:

Index Methods

Methods	Return
public int length()	Return the length of string.
public int indexOf(int ch)	Return location of first occurrence of ch in string, if ch don't exist in string return –1.
public int indexOf(int ch, int fromIndex)	Return location of occurrence of ch in string after fromIndex, if ch don't exist in string return –1.
public int lastIndexOf(int ch)	Return location of last occurrence of ch in string, if ch location does not exist in string return –1.
public int lastIndexOf(int ch, int fromIndex)	Return last of occurrence of ch in string after location fromIndex, if ch does not exist in string after location fromIndex return –1.
public int indexOf(String str)	Return location of first occurrence of substring str in string, if str does not exist in string return –1.
public int indexOf(String str, int fromIndex)	Return location of first occurrence of substring str in after location fromIndex string, if str does not exist in string return –1.
public int lastIndexOf(String str)	Return location of last occurrence of substring str in string , if str does not exist in string return –1.
public int lastIndexOf(String str, int fromIndex)	Return location of last occurrence of substring str in after location from Index string, if str does not exist in string return –1.

You can see in the example program given below, in which the index methods are used. This program will help you to know the use of index methods.

```
public class Index_Methods_Demo
{
public static void main(String[] args)
{
String  str =  "This is a test string";
System.out.println(" The length of str is :"+ str.length());
System.out.println("Index of 't' in str is:"+str.indexOf('t'));
```

```
// Print first occurrence of character t in string
System.out.println("First occurrence of 't' after  13 characters in the str:"+
str.indexOf('t',12));
// Print first occurrence of character t in string after first 13 characters
System.out.println("Last occurrence of 'z' in the str:"+ str.lastIndexOf('z'));
// Print Last  occurrence of character z in string : See output and tell
// what is printed because 'z' is not a character in str.
System.out.println("First occurrence of substring :is substring of string  str:"+
str.indexOf("is"));
// Print first occurrence of substring "is" in str
System.out.println("Last occurrence of substring :ing in the str:"+
str.lastIndexOf("ing"));
// Print first occurrence of substring "ing" in string str
System.out.println("First occurrence of substring :this after  11 characters in the str:"+
str.indexOf("this",10));
// Print first occurrence of substring "this" after first 11 characters in string str
}
}
}
```

Output:

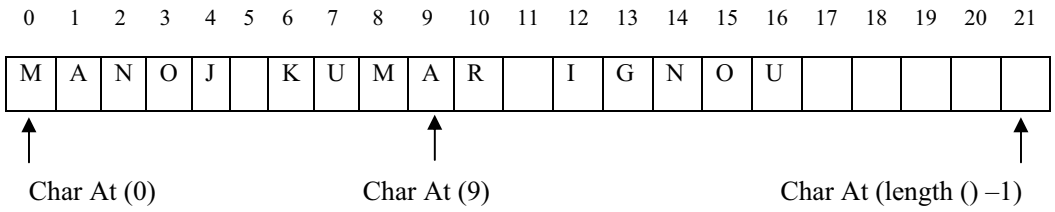
```
The length of str is :21
Index of 't' in str is:10
First occurrence of 't' after  13 characters in the str:13
Last occurrence of 'z' in the str:-1
First occurrence of substring :is substring of string  str:2
Last occurrence of substring :ing in the str:18
First occurrence of substring :this after  11 characters in the str:-1
```

In the output of the above given program you can see that if a character or substring does not exist in the given string then methods are returning: –1.

Now let us see some substring methods, comparisons methods, and string modifying methods provided in string class. These methods are used to find location of a character in string, get substrings, concatenation, comparison of strings, string reasons matching, case conversion etc.

Substring Methods

public char charAt(int index)	Return character at position i in string.



Methods	Returns
public byte[] getBytes()	Return byte array of string characters
public String substring(int beginIndex)	Return substring from index beginIndex to the end of string
public String substring	Return substring from index

(int beginIndex, int endIndex)	beginIndex to the endIndIndex of string
public String concat(String str)	Return a string which have sting on which this method is called Concatenated with str
public char[] toCharArray()	Return character array of string

Comparisons Methods

public boolean equals(Object str)	Return true if strings contain the same characters in the same order.
public boolean equalsIgnoreCase(String aString)	Return true if both the strings, - contain the same characters in the same order, and ignore case in comparison.
public int compareTo(String aString)	Compares to aString returns <0 if a String< sourceString 0 if both are equal and return >0 if aString>sourceString ; this method is case sensitive and used for knowing whether two strings are identical or not.
public boolean regionMatches (int toffset, String other, int offset, int len)	Return true if both the string have exactly same symbols in the given region.
public boolean startsWith (String prefix, int toffset)	Return true if prefix occurs at index toffset.
public boolean startsWith(String prefix)	Return true if string start with prefix.
public boolean endsWith(String suffix)	Return true if string ends with suffix.

Methods for Modifying Strings

public String replace(char oldChar, char newChar)	Return a new string with all oldChar replaced by newChar
public String toLowerCase()	Return anew string with all characters in lowercase
public String toUpperCase()	Return anew string with all characters in uppercase.
public String trim()	Return a new string with whitespace deleted from front and back of the string

In the program given below some of the string methods are used. This program will give you the basic idea about using of substring methods, comparisons methods, and string modifying methods.

```
public class StringResult
{
public static void main(String[] args)
{
String original = " Develop good software   ";
String sub1 = "";
String  sub2 = "Hi";
int index = original.indexOf('s');
sub1 =original.substring(index);
System.out.println("Substring sub1 contain: " + sub1);
System.out.println("Original contain: " + original+".");
original= original.trim();
System.out.println("After trim original contain: " + original+".");
System.out.println("Original contain: " + original.toUpperCase());
if (sub2.startsWith("H"))
System.out.println("Start with H: Yes");
else
System.out.println("Start with H: No");
sub2 = sub2.concat(sub1);
System.out.println("sub2 contents after concatenating sub1:"+sub2);
System.out.println("sub2 and  sub1 are equal:"+sub2.equals(sub1));
}
}
```

Output:

```
---------- Run ----------
Substring sub1 contain: software
Original contain:  Develop good software.
After trim original contain: Develop good software.
Original contain: DEVELOP GOOD SOFTWARE
Start with H: Yes
sub2 contents after concatenating sub1:Hisoftware
sub2 and  sub1 are equal:false
```

Now let us see valueOf() methods, these methods are used to convert different type of values like integer, float double etc. into string form. ValueOf () method is overloaded for all simple types and for Object type too. ValueOf() methods returns a string equivalent to the value which is passed in it as argument.

3.5 DATA CONVERSION USING VALUE OF( ) METHODS

public static String valueOf(boolean b)	Return string representation of the boolean argument
public static String valueOf(char c)	Return string representation of the character argument
public static String valueOf(int i)	Return string representation of the integer argument
public static String valueOf(long l)	Return string representation of the long argument
public static String valueOf(float f)	Return string representation of the float argument

public static String valueOf(double d)	Return string representation of the double argument
public static String valueOf(char data[])	Return string representation of the character array argument
public static String valueOf(char data[], offset, int count)	Return string representation of a int specific sub array of the character array argument

In the program given below you can see how an integer data is converted into a string type. This program also uses one very important operator '+' used for concatenating two strings.

```
class String_Test
{
public static void main(String[] args)
{
String s1 = new String("Your age is: ");
String s2 = new String();
int age = 28;
s2 = String.valueOf(age);
s1 = s1+s2+ " years";
System.out.println(s1);
}
}
```

Output:
Your age is: 28 years

☞ Check Your Progress 2

1) Write a program to find the length of string "Practice in programming is always good". Find the difference between first and last occurrence of 'r' in this string.

……

……

……

2) Write a program which replaces all the occurrence of 'o' in string "Good Morning" with 'a' and print the resultant string.

……

……

……

3) Write a program, which takes full path (directory path and file name) of a file and display Extension, Filename and Path separately for example, for input"/home/mem/index.html", output is Extension = html
Filename = index Path = /home/mem

……

……

……

……

You have seen that String class objects are of fixed length and no modification can be done in the content of strings except replacement of characters. If there is need of strings, which can be modified, then StrinBuffer class object should be used. Now let us discuss about StringBuffer class in detail.

3.6 STRINGBUFFER CLASS AND METHODS

StringBuffer: It is a peer class of String that provides much of the functionality of strings. In contrast to String objects, a StringBuffer object represents growable and writeable character sequences. In the strings created by using StringBuffer you may insert characters and sub-strings in the middle or append at the end. Three constructors of StringBuffer class are given below:

StringBuffer() Constructs a string buffer with no characters in it and an initial capacity of 16 characters.

StringBuffer (int size) constructs a string buffer with no characters in it and an initial capacity specified by the length argument.

StringBuffer (String str) Constructs a string buffer so that it represents the same sequence of characters as the string argument; in other words, the initial contents of the string buffer is a copy of the argument string.

Methods of String Buffer Class

In addition to functionality of String class methods, SrtingBuffer also provide methods for modifying strings by inserting substring, deleting some content of string, appending string and altering string size dynamically. Some of the key methods in StringBuffer are mentioned below:

public int **length**() Returns the length (number of characters)of the string buffer on which it is called.

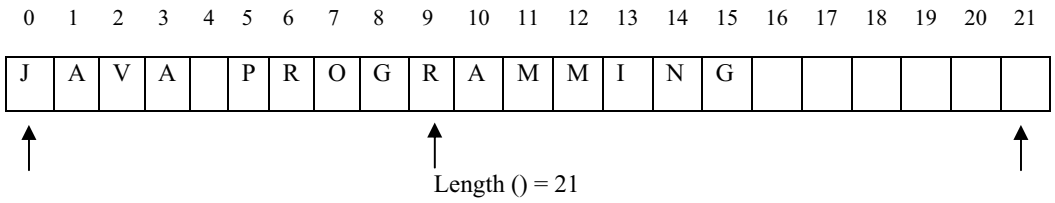


Figure 1: StringBuffer

public int **capacity**() — Returns the total allocated capacity of the String buffer. The capacity is the amount of storage available for characters can be inserted.

public void **ensureCapacity** (int minimumCapacity) — Used to ensures the capacity of the buffer is at least equal to the specified minimum capacity. If the current capacity of the string buffer on which this method is called is less than the argument value, then a new internal buffer is allocated with greater capacity and the new capacity is the larger than

i) The minimumCapacity argument.

ii) Twice the old capacity, plus 2.

	If you pass the minimumCapacity argument a nonpositive, value, then this method takes no action and simply returns.
public void **setLength**(int newLength)	Used to set the length of the string buffer on which it is called, if new length is less than the current length of the string buffer, the string buffer is truncated to contain exactly the number of characters given by the newLength argument. If the newLength argument is greater than or equal to the current length, string buffer is appended with sufficient null characters so that length becomes equal to the new Length argument. Obviously the newLength argument must be greater than or equal to 0.
public void **setCharAt**(int index, char ch)	Set character ch at the position specified by index in the string buffer on which it is called. This alters the character sequence that is identical to the old character sequence, except that now string buffer contain the character ch at position index and existing characters at index and after that in the string buffer are shifted right by one position. The value of index argument must be greater than or equal to 0, and less than the length of this string buffer.
public StringBuffer **append**(String newStr)	Appends the newStr to the string buffer. The characters of the string newStr are appended, to the contents of this string buffer increasing the length of the string buffer on which this method is called by the length of the newStr. Public.
StringBuffer **append** (StringBuffer strbuf)	Appends the characters of the strbuf to the string buffer, this results in increasing the length of the StringBuffer object on which this method is called. The method ensureCapacity is first called on this StringBuffer with the new buffer length as its argument (This ensures that the storage of this StringBuffer is adequate to contain the additional characters being appended).
public StringBuffer **append**(int i)	Appends the string representation of the int argument to the string buffer

	on which it is called. The argument is converted to a string by using method String.valueOf, and then the characters of converted string are then appended to the string buffer.
public StringBuffer **insert** (int offset, String str)	Insert the string str into the string buffer on which it is called at the indicated offset. It moves up characters originally after the offset position are shifted. Thus it increase the length of this string buffer by the length of the argument. The offset argument must be greater than or equal to 0, and less than or equal to the length of this string buffer.
public StringBuffer **insert** (int offset,char[] str)	Inserts the string representation of the array argument character into the string buffer on which this method is called. Insertion is made in the string buffer at the position indicated by offset. The length of this string buffer increases by the length of the argument
public StringBuffer **insert** ch (int offset, char ch)	Inserts the string representation of into the string buffer at the position indicated by offset. The length of this string buffer increases one.
public StringBuffer **reverse**()	Return the reverse of the sequence of the character sequence contained in the string buffer on which it is called. If n is the length of the old character sequence, in the string buffer just prior to execution of the reverse method. Then the character at index k in the new character sequence is equal to the character at index n-k-1 in the old character sequence.
public StringBuffer **delete**(int start, int end)	This method is used to remove total end–start +1 characters starting from location start and up to including index end.
public StringBuffer **deleteCharAt** (int index)	Removes the character at the specified position in the string buffer on which this method is called.
public StringBuffer **replace** (int start, int end, String str)	Replaces the characters of the string buffer from start to end by the characters in string str.

Now let us see use of some of the methods like capacity, append etc. of StringBuffer. As you know capacity method differs from length method, in that it returns the amount of space currently allocated for the StringBuffer, rather than the amount of

space used. For example, in the program given below you can see that capacity of the StringBuffer in the reverseMe method never changes, while the length of the StringBuffer increases by one in each of the iteration of loop.

```
class ReversingString
  {
        public String reverseMe(String source)
        {
       int i, len = source.length();
       StringBuffer dest = new StringBuffer(len);
        System.out.println("Length of dest:"+dest.length());
        System.out.println("Capacity of dest:"+dest.capacity());
       for (i = (len - 1); i >= 0; i--)
       dest.append(source.charAt(i));
        System.out.println("Capacity of dest:"+dest.capacity());
        System.out.println("Length of dest:"+dest.length());
       return dest.toString();
      }
      }
        public class ReverseString
        {
        public static void main ( String args[])
        {
        ReversingString R = new RString();
        String myName = new String("Mangala");
        System.out.println("Length of myName:"+myName.length());
           System.out.println(" myName:"+myName);
        System.out.println("reverseMe call:"+R.reverseMe(myName));
        }
```

Output:

```
Length of myName:7
myName:Mangala
Length of dest:0
Capacity of dest:7
Capacity of dest:7
Length of dest:7
Reverse call:alagnaM
```

Note:

i. In the reverseMe () method of above StringBuffer's to String () method is used to convert the StringBuffer to a String object before returning the String.

ii. You should initialize a StringBuffer's capacity to a reasonable first guess, because it minimizes the number of times memory to be allocated for the situation when the appended character causes the size of the StringBuffer to grow beyond its current capacity. Because memory allocation is a relatively expensive operation, you can make your code more efficient by initializing a StringBuffer's capacity to a reasonable first guess size.

Now let us take one more example program to see how to insert data into the middle of a StringBuffer. This is done with the help of one of StringBufffer's insert methods.

```
class  InsertTest
{
public static void main ( String args[])
{
StringBuffer MyBuffer = new StringBuffer("I got class in B.Sc.");
MyBuffer.insert(6, "First ");
System.out.println(MyBuffer.toString());
}
```

}
Output:
I got First class in B.Sc.

☞ Check Your Progress 3

1) List any two operations that you can perform on object StringBuffer but cannot perform on object of String class.

……

……

……

……

……

2) Write a program to answer the following:

i. Find the initial capacity and length of the following string buffer:
StringBuffer StrB = new StringBuffer("Object Oriented Programming is possible in Java");

ii. Consider the following string:

String Hi = "This morning is very good";
What is the value displayed by the expression Hi.capacity()?
What is the value displayed by the expression Hi.length()?
What is the value returned by the method call Hi.charAt(10)?

……

……

……

……

……

3) Write a program that finds initials from your full name and displays them.

……

……

……

……

……

3.7 SUMMARY

Character and strings are the most important data type. Java provides Character, String, and StringBuffer classes for handling characters, and strings. String class is used for creating the objects which are not to be changed. The string objects for which contents are to be changed StringBuffer class, is used. Character class provides various methods like compareTo, equals, isUpperCase.String class provides methods for index operations, substring operations, and a very special group of valueOf methods. Comparison methods are also provided in String class.String class methods

like replace, toLowerCase, trim are available for minor modifications though, in general string classes is not used for dynamic Strings. StringBuffer objects allow to insert character or substring in the middle or append it to the end. StringBuffer also allows deletion of a substring of a string.

3.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1)
```
//Program to test whether the content of a Character object is Upper or Lower
public class CharTest
{
public static void main(String args[])
{
Character MyChar1 = new Character('i');
Character MyChar2 = new Character('J');
//Test for MyChar1
if (MyChar1.isUpperCase(MyChar1.charValue()))
System.out.println("MyChar1 is in Upper Case: "+MyChar1);
else
System.out.println("MyChar1 is in Lower Case: "+MyChar1);
// Test for MyChar2
if (MyChar2.isUpperCase(MyChar2.charValue()))
System.out.println("MyChar2 is in Upper Case: "+MyChar2);
else
System.out.println("MyChar2 is in Lower Case: "+MyChar2);

}
}
```
Output:
MyChar1 is in Lower Case: i
MyChar2 is in Upper Case: J

2) This instance method is used to compare the value held by the current object with the value held by another object. For example let Ch1 and Ch2 be two objects of Character class then
Ch1.equals(Ch2);
method returns true if the values held by both objects are equal, otherwise false.

3) StringBuffer object should be given preference over String objects if there is a need of modification (change may take place) in the content. These include flexibility of increase in size of object.

Check Your Progress 2

1)
```
class  StringLen
{
public static void main(String[] args)
{
String MyStr = new String(" Practice in programming is always good");
System.out.println("The length of MyStr is :"+MyStr.length());
int i = MyStr.lastIndexOf("r")- MyStr.indexOf("r");
System.out.println("Difference between first and last occurence of 'r' in MyStr
is :"+ i);

}
```

```
}
```
Output:
The length of MyStr is: 39
Difference between first and last occurence of 'r' in MyStr is: 15

2)
```
//program
class ReplaceStr
{
public static void main(String[] args)
{
String S1 = new String("Good Morning");
System.out.println("The old String is:"+S1);
String S2= S1.replace('o', 'a');
System.out.println("The new String after rplacement of 'o' with 'a' is:"+S2);
}
}
```
Output:
The old String is:Good Morning
The new String after replacement of 'o' with 'a' is: Good Marning

3)
```
// It is assumed that  fullPath has a directory path, filename, and extension.
class Filename
{
  private String fullPath;
  private char pathSeparator, extensionSeparator;
  public Filename(String str, char separ, char ext)
  {
     fullPath = str;
     pathSeparator = separ;
     extensionSeparator = ext;
  }
  public String extension()
  {
     int dot = fullPath.lastIndexOf(extensionSeparator);
     return fullPath.substring(dot + 1);
   }
  public String filename()
  {
     int dot = fullPath.lastIndexOf(extensionSeparator);
     int separ = fullPath.lastIndexOf(pathSeparator);
     return fullPath.substring(separ + 1, dot);
  }
  public String path()
  {
     int separ = fullPath.lastIndexOf(pathSeparator);
     return fullPath.substring(0, separ);
  }
}
// Main method  is in FilenameDemo class
  public class FilenameDemo1
  {
   public static void main(String[] args)
   {
     Filename myHomePage = new Filename("/HomeDir/MyDir/MyFile.txt",'/', '.');
     System.out.println("Extension = " + myHomePage.extension());
     System.out.println("Filename = " + myHomePage.filename());
     System.out.println("Path = " + myHomePage.path());
```

```
    }
}
```

Output:

```
Extension = txt
Filename = MyFile
Path = /HomeDir/MyDir
```

Note:

In this program you can notice that extension uses dot + 1 as the argument to substring. If the period character is the last character of the string, then dot + 1 is equal to the length of the string which is one larger than the largest index into the string (because indices start at 0). However, substring accepts an index equal to but not greater than the length of the string and interprets it to mean "the end of the string."

Check Your Progress 3

1) i. Insertion of a substring in the middle of a string.

   ii. Reverse the content of a string object.

2)
```
class StrCap
{
public static void main(String[] args)
{
StringBuffer StrB = new StringBuffer("Object Oriented Programming is
possible in Java");
String  Hi = new  String("This morning is very good");
System.out.println("Initial Capacity of StrB is :"+StrB.capacity());
System.out.println("Initial length of StrB is :"+StrB.length());

//System.out.println("value displayed by the expression Hi.capacity() is:
"+Hi.capacity());
System.out.println("value displayed by the expression Hi.length() is:
"+Hi.length());
System.out.println("value displayed by the expression Hi.charAt() is:
"+Hi.charAt(10));
}
}
```

Output:

```
Initial Capacity of StrB is :63
Initial length of StrB is :47
value displayed by the expression Hi.length() is: 25
value displayed by the expression Hi.charAt() is: n
```

Note: The statement "System.out.println("value displayed by the expression Hi.capacity() is: "+Hi.capacity());" is commented for successful execution of program because capacity method is mot available in String class.

3)
```
public class NameInitials
{
public static void main(String[] args)
{
String myNameIs = "Mangala Prasad Mishra";
StringBuffer myNameInitials = new StringBuffer();
System.out.println("The name is : "+myNameIs);
// Find length of name given
int len = myNameIs.length();
```

```
for (int i = 0; i < len; i++)
                    {
if (Character.isUpperCase(myNameIs.charAt(i)))
                    {
myNameInitials.append(myNameIs.charAt(i));
}
}
System.out.println("Initials of the name: " + myNameInitials);
}
}
```

Output:

The name is: Mangala Prasad Mishra
Initials of the name: MPM

UNIT 4 EXPLORING JAVA I/O

Structure **Page Nos.**

4.0 INTRODUCTION

Input is any information that is needed by your program to complete its execution and *Output* is information that the program must produce after its execution. Often a program needs to bring information from an external source or to send out information to an external destination. The information can be anywhere in file. On disk, somewhere on network, in memory or in other programs. Also the information can be of any type, i.e., object, characters, images or sounds. In Java information can be stored, and retrieved using a communication system called streams, which are implemented in Java.io package.

The Input/output occurs at the program interface to the outside world. The input-output facilities must accommodate for potentially wide variety of operating systems and hardware. It is essential to address the following points:

- Different operating systems may have different convention how an end of line is indicated in a file.
- The same operating system might use different character set encoding.
- File naming and path name conventions vary from system to system.

Java designers have isolated programmers from many of these differences through the provision of package of input-output classes, java.io. Where data is stored objects of the reader classes of the java.io package take data, read from the files according to convention of host operating system and automatically converting it.

In this unit we will discuss how to work on streams. The dicussion will be in two directions: pulling data into a program over an input stream and sending data from a program using an output stream.

In this unit you will work with I/O classes and interfaces, streams File classes, Stream tokenizer, Buffered Stream, Character Streams, Print Streams, and Random Access Files.

4.1 OBJECTIVES

After going through this unit you will be able to:

- explain Java I/O hierarchy;

- handle files and directories using File class;
- read and write using I/O stream classes;
- differentiate between byte stream, character stream and text stream;
- use stream tokenizer in problem solving;
- use Print stream objects for print operations, and
- explain handling of random access files.

4.2 JAVA I/O CLASSES AND INTERFACES

The package java.io provides two sets of class hierarchies-one for reading and writing of bytes, and the other for reading and writing of characters. The InputStream and OutputStream class hierarchies are for reading and writing bytes. Java provides class hierarchies derived from Reader and Writer classes for reading and writing characters.

A character encoding is a scheme for internal representation of characters.

Java programs use 16 bit Unicode character encoding to represent characters internally. Other platforms may use a different character set (for example ASCII) to represent characters. The reader classes support conversions of Unicode characters to internal character storage.

Classes of the *java.io* hierarchy

Hierarchy of classes of Java.io package is given below:

- InputStream
  - FilterInputStream
    - BufferedInputStream
    - DataInputStream
    - LineNumberInputStream
    - PushbackInputStream
  - ByteArrayInputStream
  - FileInputStream
  - ObjectInputStream
  - PipedInputStream
  - SequenceInputStream
  - StringBufferInputStream
- OutputStream
  - FilterOutputStream
    - BufferedOutputStream
    - DataOutputStream
    - PrintStream
  - ByteArrayOutputStream
  - FileOutputStream
  - ObjectOutputStream
  - PipedOutputStream
- Reader
  - BufferedReader
    - LineNumberReader
  - CharArrayReader
  - FilterReader
    - PushbackReader
  - InputStreamReader
    - FileReader
  - PipedReader
  - StringReader

- Writer
  - BufferedWriter
  - CharArrayWriter
  - FilterWriter
  - OutputStreamWriter
    - FileWriter
  - PipedWriter
  - PrintWriter
  - StringWriter
- File
  - RandomAccessFile
  - FileDescriptor
  - FilePermission
  - ObjectStreamClass
  - ObjectStreamField
  - SerializablePermission
  - StreamTokenizer.

Interfaces of Java.io

Interfaces in Java.io are given below:

- DataInput
- DataOutput
- Externalizable
- FileFilter
- FilenameFilter
- ObjectInput
- ObjectInputValidation
- ObjectOutput
- ObjectStreamConstants
- Serializable

Each of the Java I/O classes is meant to do one job and to be used in combination with other to do complex tasks. For example, a BufferedReader provides buffering of input, nothing else. A FileReader provides a way to connect to a file. Using them together, you can do buffered reading from a file. If you want to keep track of line numbers while reading from a character file, then by combining the File Reader with a LineNumberReader to serve the purpose.

4.3 I/O STREAM CLASSES

I/O classes are used to get input from any source of data or to send output to any destination. The source and destination of input and output can be file or even memory. In Java input and output is defined in terms of an abstract concept called stream. A Stream is a sequence of data. If it is an input stream it has a source. If it is an output stream it has a destination. There are two kinds of streams: byte stream and character stream. The java.io package provides a large number of classes to perform stream IO.

The File Class

The File class is not I/O. It provides just an identifier of files and directories. Files and directories are accessed and manipulated through java.io.file class. So, you always remember that creating an instance of the File class does not create a file in the

operating system. The object of the File class can be created using the following types of File constructors:

File MyFile = new File("c:/Java/file_Name.txt");
File MyFile = new File("c:/Java", "file_Name.txt");
File MyFile = new File("Java", "file_Name.txt");

The first type of constructor accepts only one string parameter, which includes file path and file name, here in given format the file name is file_Name.txt and its path is c:/Java.

In the second type, the first parameter specifies directory path and the second parameter denotes the file name. Finally in the third constructor the first parameter is only the directory name and the other is file name.
Now let us see the methods of the file class.

Methods

boolean exists() : Return true if file already exists.
boolean canWrite() : Return true if file is writable.
boolean canRead() : Return true if file is readable.
boolean isFile() : Return true if reference is a file and false for directories references.
boolean isDirectory(): Return true if reference is a directory.
String getAbsolutePath() : Return the absolute path to the application

You can see the program given below for creating a file reference.

```
//program
import java.io.File;
class FileDemo
        {
        public static void main(String args[])
        {
        File f1 = new File ("/testfile.txt");
        System.out.println("File name : " + f1.getName());
        System.out.println("Path :  " + f1.getPath());
        System.out.println("Absolute Path : " + f1.getAbsolutePath());
        System.out.println(f1.exists() ? "Exists" : "doesnot exist");
        System.out.println( f1.canWrite()?"Is writable" : "Is not writable");
        System.out.println(f1.canRead() ? "Is readable" :"Is not readable");
        System.out.println("File Size : " + f1.length() + " bytes");
        }
   }
```

Output: (When testfile.txt does not exist)
File name: testfile.txt
Path: \testfile.txt
Absolute Path: C:\testfile.txt
doesnot exist
Is not writable
Is not readable
File Size: 0 bytes
Output: (When testfile.txt exists)
File name : testfile.txt
Path : \testfile.txt
Absolute Path : C:\testfile.txt
Exists
Is writable

Is readable
File Size: 17 bytes

4.3.1 Input and Output Stream

Java Stream based I/O builts upon four abstract classes: InputStream, OutputStream, Reader and Writer. An object from which we can *read a sequence of bytes* is called an *input stream*. An object from which we can *write sequence of byte* is called *output stream*. Since byte oriented streams are inconvenient for processing. Information is stored in Unicode characters that inherit from abstract *Reader and Writer* superclasses.

All of the streams: The readers, writers, input streams, and output streams are automatically opened when created. You can close any stream explicitly by calling its close method. The garbage collector implicitly closes it if you don't close. It is done when the object is no longer referenced. Also, the *direction of flow* and *type* of data is not the matter of concern; algorithms for sequentially reading and writing data are basically the same.

Reading and Writing IO Stream

Reader and InputStream define similar APIs but for different data types. You will find, that both Reader and InputStream provide methods for marking a location in the stream, skipping input, and resetting the current position. *For example,* Reader and Input Stream defines the methods for reading characters and arrays of characters and reading bytes and array of bytes respectively.

Methods

int read() : reads one byte and returns the byte that was read, or –1 if it encounters the end of input source.

int read(char chrbuf[]): reads into array of bytes and returns number of bytes read or –1 at the end of stream.
int read(char chrbuf[], int offset, int length): chrbuf – the array into which the data is read. offset - is offset into chrbuf where the first byte should be placed. len - maximum number of bytes to read.

Writer and OutputStream also work similarly. Writer defines these methods for writing characters and arrays of characters, and OutputStream defines the same methods but for bytes:

Methods

int write(int c): writes a byte of data
int write(char chrbuf[]) : writes character array of data.
int write(byte chrbuf[]) writes all bytes into array b.
int write(char chrbuf[], int offset, int length) : Write a portion of an array of characters.

☞ **Check Your Progress 1**

1) What is Unicode? What is its importance in Java?

……

……

2) Write a program which calculates the size of a given file and then renames that file to another name.

……

……

3) Write a program which reads string from input device and prints back the same on the screen.

……

……

4.3.2 Input Stream and Output Stream hierarchy

There are two different types of Streams found in Java.io package as shown in Figure 1a OutputStream and InputStream. Figure 1b shows the further classification of Inputstream.

The following classes are derived from InputStream Class.

InputStream: Basic input stream.
StringBufferInputStream: An input stream whose source is a string.
ByteArrayInputStream: An input stream whose source is a byte array.
FileInputStream: An input stream used for basic file input.

FilterInputStream: An abstract input stream used to add new behaviour to existing input stream classes.

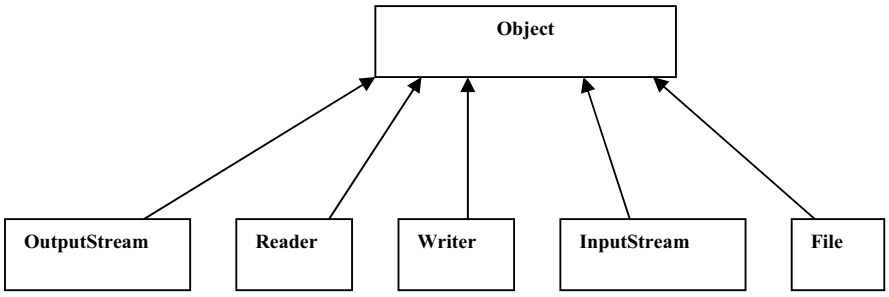


Figure 1(a): Types of stream in Java I/O

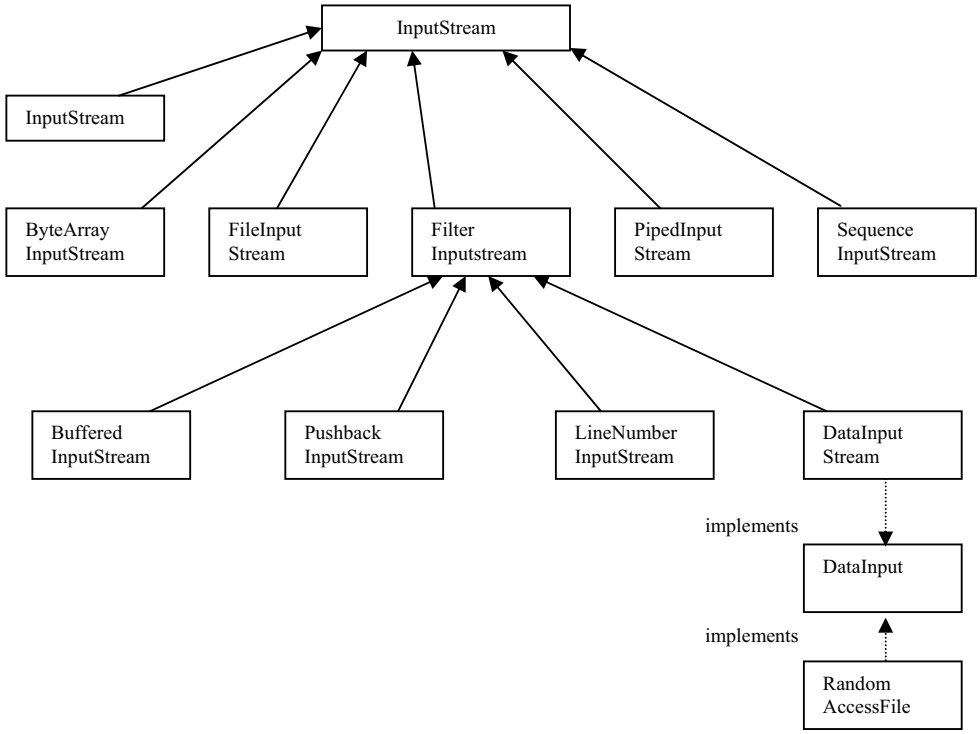


Figure 1(b): Input Stream Class

PipedInputStream: An input stream used for inter-thread communication.
SequenceInputStream: An input stream that combines two other input streams.
BufferedInputStream: A basic buffered input stream.
PushbackInputStream: An input stream that allows a byte to be pushed back onto the stream after the byte is read.
LineNumberInputStream: An input stream that supports line numbers.
ataInputStream: An input stream for reading primitive data types.
RandomAccessFile: A class that encapsulates a random access disk file.

Output streams are the logical counterparts to input streams and handle writing data to output sources. *In Figure2* the hierarchy of output Stream is given. The following classes are derived from outputstream.

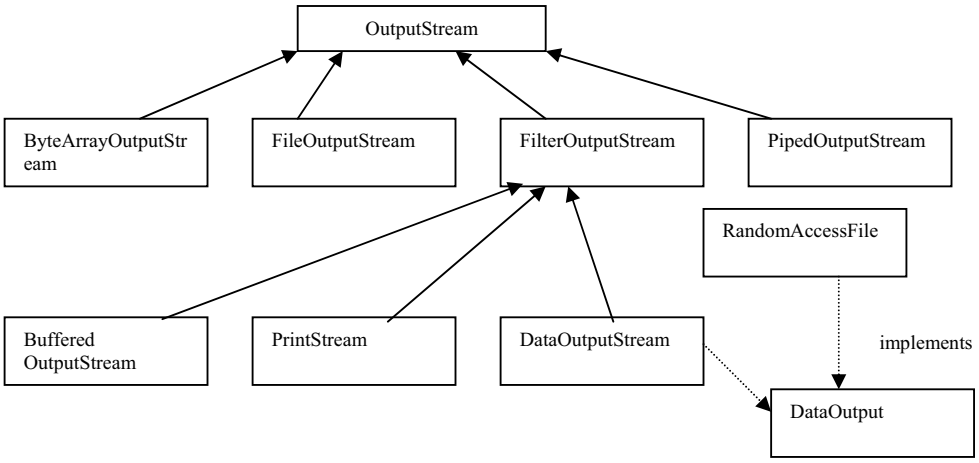


Figure 2: Output Stream Class Hierarchy

OutputStream: The basic output stream.
ByteArrayOutputStream: An output stream whose destination is a byte array.
FileOutputStream: Output stream used for basic file output.
PipedOutputStream: An output stream used for inter-thread communication.
BufferedOutputStream: A basic buffered output stream.
PrintStream: An output stream for displaying text.
DataOutputStream: An output stream for writing primitive data types.
FilterOutputStream: An abstract output stream used to add new behaviour to existing output stream classes.

Layering byte Stream Filter

Some-times you will need to combine the two input streams. in Java. It is known as **filtered streams** *(i.e., feeding an existing stream to the constructor of another stream)*. You should continue layering stream constructor until you have access to the functionality you want.

FileInputStream and FileOutputStream give you input and output streams attached to a disk file. For example giving file name or full path name of the file in a constructor as given below:

FileInputStream fin = new FileInputStream("Mydat.dat");

Input and output stream classes only support reading and writing on byte level, DataInputStream has a method that could read numeric. FileInputStream has no method to read numeric type and the DataInputStream has no method to get data from the file. If you have to read the numbers from file, first create a FileInputStream and pass it to DataInputStream, as you can see in the following code:

FileInputStream fin = new FileInputStream ("Mydat.dat");

```
DataInputStream din = new DataInputStream (fin)
doubles = din.readDouble();
```

If you want buffering, and data input from a file named Mydata.dat then write your code like:

```
DataInputStream din = new DataInputStream(new BufferedInputStream
                                          (new FileInputStream("employee.dat")));
```

Now let us take one example program in which we open a file with the binary FileOutputStream class. Then wrap it with DataOutput Stream class. Finally write some data into this file.

```
//program
import java.io.*;
public class BinFile_Test
{
    public static void main(String args[])
    {
    //data array
    double data [] = {1.3,1.6,2.1,3.3,4.8,5.6,6.1,7.9,8.2,9.9};
    File file = null;
    if (args.length > 0 ) file = new File(args[0]);
    if ((file == null) || !(file.exists()))
    {
    // new file created
    file = new File("numerical.txt");
    }
    try
    {
     // Wrap the FileOutput Stream with DataOutputStream to obtain its writeInt()
     method
    FileOutputStream fileOutput = new FileOutputStream(file);
    DataOutputStream dataOut    = new DataOutputStream(fileOutput);
    for (int i=0; i < data.length;i++)
    dataOut.writeDouble(data[i]);
    }
    catch (IOException e)
    {
     System.out.println("IO error is there " + e);
    }
    }
    }
```

Output:

You can see numeric.txt created in your system you will find something like:
?ôÌÌÌÌÍ?ù™™™™™™š@

Reading from a Binary File

Similarly, we can read data from a binary file by opening the file with a **FileInputStream** Object.

Then we wrap this with a **DataInputStream class** to obtain the many **readXxx()** methods that are very useful for reading the various primitive data types.

```
import java.io.*;
public class BinInputFile_Appl
```

```
{
  public static void main(String args[])
  {
   File file = null;
   if (args.length > 0 ) file = new File(args[0]);
   if ((file == null) || !file.exists())
   {
    file = new File("numerical.dat");
    }
    try
    {    // Wrap the FileOutput Stream with DataInputStream so that we can use its
     readDouble() method
     FileInputStream fileinput = new FileInputStream(file);
     DataInputStream dataIn    = new DataInputStream(fileinput);
      int i=1;
     while(true)
     {
      double d = dataIn.readDouble();
       System.out.println(i+".data="+d);
      }
    }
    catch (EOFException eof)
     {
     System.out.println("EOF Reached " + eof);
      }
    catch (IOException e)
     {
     System.out.println(" IO erro" + e);
      }
      }
      }
```

Output:

```
C:\JAVA\BIN>Java BinInputFile_Appl
1.data=1.3
1.data=1.6
1.data=2.1
1.data=3.3
1.data=4.8
1.data=5.6
1.data=6.1
1.data=7.9
1.data=8.2
1.data=9.9
```

Reading/Writing Arrays of Bytes

Some time you need to read/write some bytes from a file or from some network place. For this purpose classes Bytes Array InputStream and ByteArrayOutputStream are available.

Constructors of ByteArrayInputStream

ByteArrayInputStream(byte[] input_buf, int offset, int length)
ByteArrayInputStream(byte[] input_buf)

In the first constructor input_buf is the input buffer, offset indicates the position of the first byte to read from the buffer and length tells about the maximum number of bytes to be read from the buffer. Similarly for simple purpose another constructor is defined which uses input_buf its buffer array and also contain information to read this array.

For example in the code given below ByteArrayInputStream creates two objects input1 and input2, where input1 contains all letters while input2 will contain only first three letters of input stream (“abc”).

```
String tmp = “abcdefghijklmnopqrstuvwxyz”
byte b[] =  tmp.getBytes();
ByteArrayInputStream input1 = new ByteArrayInputStream(b);
ByteArrayInputStream input2 = new ByteArrayInputStream(b,0,3);
```

ByteArrayOutputStream is an output stream that is used to write byte array. This class has two constructors- one with no parameter and another with an integer parameter that tells the initial size of output array.

```
ByteArrayOutputStream( )
ByteArrayOutputStream( int buf_length)
```

In the following piece of code you can see how ByteArrayOutputStream provides an output with an array of bytes.

```
ByteArrayOutputStream f = new ByteArrayOutputStream();
String tmp = “abcdefghijklmnopqrstuvwxyz”
byte b[] =  tmp.getBytes();
f,write(b);
```

PushbackInputStream is an input stream that can unwrite the last byte read

You often need to peek at the next byte to see whether it is a value that you expect. Use **PushbackInputStream** for this purpose. You can use this class to unwrite the last byte read is given in the following piece of code:

```
PushbackInputStream pbin = new PushbackInputStream
(new BufferedInputStream(new FileInputStream (“employee.dat”)));
```

You can read the next byte, as follow

```
int b = pbin.read();
    if (b != ‘< ‘) pbin.read.unread(b);
```

Read() and Unread() are the only methods that apply to pushback input stream. If you want to look ahead and also read numbers, you need to combine both a pushback inputstream and a data. Input. stream reference.

```
    DatainputStream din = new DataInputStream
(PushBackInputStream pbin = new PushBackInputStream
(new bufferedInputStream (newFileInputStream(“employee.dat”))));
```

4.4 TEXT STREAMS

Binary IO is fast but it is not readable by humans. For example, if integer 1234 is saved as binary, it is written as a sequence of bytes **00 00 04 D2** (in hexadecimal notation). In text format it is saved as the string “1234”. Java uses its own character-encoding. This may be single byte, double byte or a variable byte scheme based on host system. If the Unicode encoding is written in text file, then the resulting file will unlikely be human readable. To overcome this Java has a set of stream filters that bridges the gap between Unicode encoded text and character encoding used by host. All these classes are descended from abstract *Reader and Writer classes.*

Reader and Writer classes

You may know that Reader and Writer are abstract classes that define the java model of streaming characters. These classes were introduced to support 16 bit Unicode

characters in java 1.1 and onwards and provide Reader/Writer for java 1.0 stream classes.

java 1.0	**java 1.1**
InputStream	Reader
OutputStream	Writer
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferInputStream	StringReader
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

For *filter classes,* we also have corresponding *Reader* and *Writer* classes as given below:

FilterInputStream	FilterReader
FilterOutputStream	FilterWriter (abstract)
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter
DataInputStream	DataInputStream BuffedReader (readLine())
PrintStream	PrintWriter
LineNumberInputStream	LineNumberReader
StreamTokenizer	StreamTokenizer
PushBackInputStream	PushBackReader

In addition to these classes java 1.1 also provides two extra classes which provide a bridge between byte and character streams. For reading bytes and translating them into characters according to specified character encoding we have InputStreamReader while for the reverse operation translating characters into bytes according to a specified character encoding java 1.1 provides OutputStreamWriter.

Reading Text from a File

Let us see how to use a FileReader stream to read character data from text files. In the following example read () method obtains one character at a time from the file. We are reading bytes and counting them till it meets the EOF (end of file). Here FileReader is used to connect to a file that will be used for input.

```
// Program
import java.io.*;
public class TextFileInput_Appl
{
public static void main(String args[])
{
File file = null;
        if (args.length > 0 ) file= new File(args[0]);
        if (file == null || !file.exists())
        {
        file=new File("TextFileInput_Appl.Java");
        }
        int numBytesRead=0;
        try
{
int tmp =0 ;
FileReader filereader = new FileReader (file);
while( tmp != -1)
```

```
{
tmp= filereader.read();
if (tmp != -1) numBytesRead++;
}
}
catch (IOException  e)
{
System.out.println(" IO erro" + e);
}
System.out.println(" Number of bytes read : " + numBytesRead);
System.out.println(" File.length()= "+file.length());
}
}
```

For checking whether we read all the bytes of the file “TextFileInput_Appl.Java” in this program we are comparing both, length of file and the bytes read by read( ) method. You can see and verify the result that both values are 656.

Output:

```
Number of bytes read: 656
File.length()= 656
```

Writing Text to a file

In the last example we read bytes from a file. Here let us write some data into file. For this purpose Java provides FileWriter class. In the following example we are demonstrating you the use of FileWriter. In this program we are accepting the input data from the keyboard and writes to a file a "textOutput.txt". To stop writing in file you can press ctrl+C which will close the file.

```
// Program Start here
import java.io.*;
public class TextFileOutput_Appl
{
  public static void main(String args[])
  {
   File file = null;
   file = new File("textOutput.txt");
    try
    {
          int tmp= 0;
          FileWriter filewriter = new FileWriter(file);
          while((tmp=System.in.read()) != -1)
    {
    filewriter.write((char) tmp);
    }
    filewriter.close();
    }
    catch (IOException e)
    {
    System.out.println(" IO erro" + e);
    }
    System.out.println(" File.length()="+file.length());
   }
  }
Output:
C:\Java\Block3Unit3> Java TextFileOutput_Appl
Hi, test input start here.
^Z
```

```
File.length()=28
C:\Java\Block3Unit3>
```

4.5 STREAM TOKENIZER

Stream Tokenizer is introduced in JDK 1.1 to improve the wc() [Word Count] method, and to provide a better way for pattern recognition in input stream. (do you know how wc() mehod is improved by Stream Tokenizer ?) during writing program. Many times when reading an input stream of characters, we need to parse it to see if what we are reading is a word or a number or something else. This kind of parsing process is called “tokenizing”. A “token” is a sequence of characters that represents some abstract values stream. Tokenizer can identify members, quoted string and many other comment styles.

Stream Tokenizer Class is used to break any input stream into tokens, which are sets of bits delimited by different separator. A few separators already added while you can add other separators. It has two contributors given below:

There are 3 instance variable as defined in streamtokenizer **nval**,**sval** and **ttype.** nval is public double used to hold number, sval is public string holds word, and ttype is pubic integer to indicates the type of token. nextToken() method is used with stream tokenizer to parse the next token from the given input stream. It returns the value of token type **ttype**, if the token is word then ttype is equal to TT_WORD. There are some other values of ttype also which you can see in Table 1 given below. You can check in the given program below that we are using TT_EOF in the same way as we have used end of file (EOF) in our previous example. Now let me tell you how wc() is improved, different token types are available stream tokenizer which we can use to parse according to our choice, which were limitation of wc() method also.

Table 1: Token Types

Token Types	Meaning
TT_WORD	String word was scanned (The string field **sval** contains the word scanned)
TT_NUMBER	A number was scanned (only decimal floating point number)
TT_EOL	End – of - line scanned
TT_EOF	End – of –file – scanned

```
// Program Start here
import java.io.*;
public class tokenizer
{
  public static void main(String args[])
  {
     String sample="this 123 is an 3.14 \n simple test";
      try
      {
      InputStream  in;
      in = new StringBufferInputStream(sample);
      StreamTokenizer parser = new StreamTokenizer(in);
      while(parser.nextToken() != StreamTokenizer.TT_EOF)
      {
       if (parser.ttype == StreamTokenizer.TT_WORD)
          System.out.println("A word"+parser.sval);
       else if (parser.ttype == StreamTokenizer.TT_NUMBER)
          System.out.println("A number: "+parser.nval);
```

```
        else if (parser.ttype == StreamTokenizer.TT_EOL)
        System.out.println("EOL");
        else if (parser.ttype == StreamTokenizer.TT_EOF)
        throw new IOException("End of FIle");
        else
        System.out.println("other" +(char) parser.ttype);
       }
       }
       catch (IOException e)
       {
       System.out.println(" IO erro" + e);
       }
    }
}
```

Output:
A word: this
A number: 123
A word: is
A word: an
A number: 3.14
A word: simple
A word: test

After passing the input stream, each token is identified by the program and the value of token is given in the output of the above program.

4.6 SERIALIZATION

You can read and write text, but with Java you can also read and write objects to files. You can say object I/O is serialization where you read/write state of an object to a byte stream. It plays a very important role sometimes when you are interested in saving the state of objects in your program to a persistent storage area like file. So that further you can De-Serialize to restore these objects, whenever needed. I have a question for you, why is implementing of serialization very difficult with other language and it is easy with Java?

In other languages it may be difficult, since objects may contain references to another objects and it may lead to circular references. Java efficiently implements serialization, though we have to follow different Conditions; we can call them, Conditions for serializability.

If you want to make an object serialized, the class of the object must be declared as public, and must implement serialization. The class also must have a no argument constructer and all fields of class must be serializable.

Java gives you the facility of serialization and deserialization to save the state of objects. There are some more functions which we often used on files like compression and encryption techniques. We can use these techniques on files using Externalizable interface. For more information you can refer to books given in references.

☞ Check Your Progress 2

1) Why is PushbackInputStream is used?

……

……

2) Write a program to print all primitive data values into a File using FileWriter stream wrapped with a Printwriter.

……

……

3) How would you append data to the end of a file? Show the constructor for the class you would use and explain your answer. .

……

……

4.7 BUFFERED STREAM

What is Buffer? Buffer is a temporary storage place where the data can be kept before it is needed by the program that reads or writes data. By using buffer, you can get data without always going back to the original source of data. In JAVA, these Buffers allow you to do input/output operations on more than a byte at a time. It increases the performance, and we can easily manipulate, skip, mark or reset the stream of byte also.

Buffered Stream classes manipulate sequenced streams of byte data, typically associated with binary format files. For example, binary image file is not intended to be processed as text and so conversion would be appropriate when such a file is read for display in a program. There are two classes for bufferstream BufferInputStream and BufferedOutputStream.

A *Buffered input stream* fills a buffer with data that hasn't been handled yet. When a program needs this data, it looks to the buffer first before going to the original stream source. A *Buffered input stream* is created using one of the two constructors:

BufferedInputStream (Input_Stream) – Creates a buffered input stream for the specified Input Stream object.

BufferedInputStream (Input_Stream, int) – Creates a specified buffered input stream with a buffer of int size for a specified Inputstream object.

The simplest way to read data from a buffered input stream is to call its read () method with no argument, which returns an integer from 0 to 255 representing the next byte in the stream. If the end of stream is reached and no bytes are available, –1 is returned. A *Buffered output stream* is created using one of the two constructors:

BufferedOutputStream(OutputStream) – Creates a buffered output stream for the specified Output Stream object.

BufferedOutputStream (OutputStream, int) – Creates a buffered output stream with a buffer of int size for the specified outputstream object.

The output stream's write (int) method can be used to send a single byte to the stream. write (byte [], int, int) method writes multiple bytes from the specified byte array. It specifies the starting point, and the number of bytes to write. When data is directed to a buffered stream, it is not output to its destination until the stream fills or buffered stream *flush()* method is used.

In the following example, we have explained a program for you, where we are creating buffer input stream from a file and writing with buffer output stream to the file.

```
// program
import java.lang.System;
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.SequenceInputStream;
import java.io.IOException;
public class BufferedIOApp
{
public static void main(String args[]) throws IOException
{
SequenceInputStream f3;
FileInputStream f1 = new FileInputStream("String_Test.Java");
FileInputStream f2 = new FileInputStream("textOutput.txt");
f3 = new SequenceInputStream(f1,f2);
BufferedInputStream inStream = new BufferedInputStream(f3);
BufferedOutputStream outStream = new BufferedOutputStream(System.out);
inStream.skip(500);
boolean eof = false;
int byteCount = 0;
while (!eof)
{
int c = inStream.read();
if(c == -1) eof = true;
else
{
outStream.write((char) c);
++byteCount;
}
}
String bytesRead = String.valueOf(byteCount);
bytesRead+=" bytes were read\n";
outStream.write(bytesRead.getBytes(),0,bytesRead.length());
System.out.println(bytesRead);
inStream.close();
outStream.close();
f1.close();
f2.close();
}
}
```

Character Streams

Character streams are used to work with any text files that are represented by ASCII characters set or Unicode set. Reading and writing file involves interacting with peripheral devices like keyboard printer that are part of system environment. As you know such interactions are relatively slow compared to the speed at which CPU can operate. For this reason *Buffered reader and Buffered Writer* classes are used to overcome from difficulties posed by speed mismatch.

When a read method of a *Buffered Reader* object, is invoked it might actually read more data from the file then was specifically asked for. It will hold on to additional data read from the file that it can respond to the next read request promptly without making an external read from the file.
Similarly *Buffered Writer* object might bundle up several small write requests in its own internal buffer and only make a single external write to file operation when it considers that it has enough data to make the external write when its internal buffer is full, or a specific request is made via call to its *flush* method. The external write occurs.

It is not possible to open a file directly by constructing a Buffered Reader or Writer Object. We construct it from an existing FileReader or FileWriter object. It has an advantage of not duplicating the file- opening functionality in many different classes

The following constructors can be used to create a BufferedReader:
BufferedReader (Reader) – Creates a buffered character stream associated with the specified Reader object, such as FileReader.

BufferedReader (Reader, int) – Creates a buffered character stream associated with the specified Reader object, and with a buffered of int size.

Buffered character string can be read using read() and read(char[],int,int). You can read a line of text using readLine() method. The readLine() method returns a String object containing the next line of text on the stream, not including the characters or characters that represent the end of line. If the end of stream is reached the value of string returned will be equal to **null.**

In the following program we have explained the use of BufferedReader where we can calculate statistic (No. of lines, No. of characters) of a file stored in the hard disk named “text.txt”.

```
// Program
import Java.io.*;
public class FileTest
{
public static void main(String[] args) throws IOException
{
String s = "textOutput.txt";
File f = new File(s);
if (!f.exists())
{
System.out.println("\'" + s + "\' does not exit. Bye!");
return;
}
// open disk file for input
BufferedReader inputFile = new BufferedReader(new FileReader(s));
// read lines from the disk file, compute stats
String line;
int nLines = 0;
int nCharacters = 0;
while ((line = inputFile.readLine()) != null)
{
nLines++;
nCharacters += line.length();
}
// output file statistics
System.out.println("File statistics for \'" + s + "\'...");
System.out.println("Number of lines = " + nLines);
System.out.println("Number of characters = " + nCharacters);
inputFile.close();
}
}
```

Output:
File statistics for 'textOutput.txt'...
Number of lines = 1
Number of characters = 26

4.8 PRINT STREAM

A *PrintStream* is a grandchild of OutputStream in the Java class hierarchy–it has methods implemented that print lines of text at a time as opposed to each character at a time. It is used for producing formatted output.

The original intent of *PrintStream* was to print all of the primitive data types and String objects in a viewable format. This is different from DataOutputStream, whose goal is to put data elements on a stream in a way that DataInputStream can portably reconstruct them.

Constructors

PrintStream (OutputStream out): PrintStream(OutputStream out) creates a new print stream. This stream will not flush automatically; returns a reference to the new PrintStream object.

PrintStream(OutputStream out, boolean autoFlush): Creates a new print stream. If autoFlush is true, the output buffer is flushed whenever (a) a byte array is written, (b) one of the println() methods is invoked, or (c) a newline character or byte ('\n') is written; returns a reference to the new PrintStream object.

Methods

flush(): flushes the stream; returns a void

print(char c): prints a character; returns a void

println(): terminates the current line by writing the line separator string; returns a void

println(char c): prints a character and then terminates the line; returns a void

Two constructors of PrintStream are deprecated by java 1.1 because PrintStream does not handle Unicode character and is thus not conveniently internationalized. Deprecating the constructors rather than the entire class allows existing use of System.out and Sytem.err to be compiled without generating deprecation warnings. Programmers writing code that explicitly constructs a PrintStream object will see deprecating warning when the code is compiled. Code that produces textual output should use the new PrintWriter Class, which allows the character encoding to be specified or default encoding to be accepted. System.out and System.err are the *only* PrintStream object you should use in programs.

PrintWriter

To open a file output stream to which text can be written, we use the PrintWriter class. As always, it is best to buffer the output. The following sets up a buffered file writer stream named outFile to write text into a file named save.txt. The Java statement that will construct the required PrintWriter object and its parts is:

PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter("save.txt"));

The object outFile, above, is of the PrintWriter class, just like System.out. If a string, s, contains some text, it is written to the file as follows:

outFile.println(s);

When finished, the file is closed as expected:

outFile.close();

In the following program we are reading text from the keyboard and writing the text to a file called junk.txt.

```
//program
import Java.io.*;
class FileWrite
{
public static void main(String[] args) throws IOException
{
        BufferedReader stdin = new BufferedReader(new
        InputStreamReader(System.in));
        String s = "junk.txt";
        File f = new File(s);
        if (f.exists())
        {
        System.out.print("Overwrite " + s + " (y/n)? ");
        if(!stdin.readLine().toLowerCase().equals("y"))
        return;
        }
        // open file for output
        PrintWriter outFile = new PrintWriter(new BufferedWriter(new
        FileWriter(s)));
        System.out.println("Enter some text on the keyboard...");
        System.out.println("(^z to terminate)");
        String s2;
        while ((s2 = stdin.readLine()) != null)
        outFile.println(s2);
        outFile.close();
   }
   }
```

Output:
Enter some text on the keyboard...
(^z to terminate)
How are you? ^z

4.9 RANDOM ACCESS FILE

The character and byte stream are all sequential access streams whose contents must be read or written sequentially. In contrast, Random Access File lets you randomly access the contents of a file. The *Random Access File* allows files to be accessed at a specific point in the file. They can be opened in read/write mode, which allows updating of a current file.

The *Random Access File* class is not derived from InputStream and OutputStream. Instead it implements DataInput and DataOutput and thus provides a set of methods from both DataInputStream and DataOutputStream.

An object of this class should be created when access to binary data is required but in non-sequential form. The same object can be used to both read and write the file.

To create a new Random Access file pass the name of file and mode to the constructor. The mode is either “r” (read only) or “rw” (read and write)
An *IllegalArgumentException* will be thrown if this argument contains anything else.

Following are the two constructors that are used to create RandomAccessFile:

RandomAccessFile(String s, String mode) throws IOException and s is the file name and mode is “r” or “rw”.
RandomAccessFile(File f, String mode) throws IOException and f is an File object, mode is ‘r’ or ‘rw’.

Methods

long length() : returns length of bytes in a file
void seek(long offset) throws IOException: position the file pointer at a particular point in a file. The new position is current position plus the offset. The offset may be positive or negative.
long getFilePointer() throws IOException : returns the index of the current read write position within the file.
void close() throws IOException: close fine and free the resource to system.

To explain how to work with random access files lets write a program, here in the following program we are writing a program to append a string “I was here” to all the files passed as input to program.

```
import Java.io.*;
public class AppendToAFile
{
  public static void main(String args[])
  {
   try
    {
    RandomAccessFile raf = new RandomAccessFile("out.txt","rw");
   // Position yourself at the end of the file
    raf.seek(raf.length());
   // Write the string into the file.Note that you must explicitly handle line breaks
    raf.writeBytes("\n I was here\n");
   }
   catch (IOException e)
   {
    System.out.println("Eror Opening a file" + e);
   }
  }
  }
```

After executing this program: “I was here” will be appended in out.txt file.

☞ Check Your Progress 3

1) Write a program to read text from the keyboard and write the text to a file called junk.txt.

……

……

……

……

……

2) Create a file test.txt and open it in read write mode.Add 4 lines to it and randomly read from line 2 to 4 and line 1.

……

……

……

3) Write a program using FileReader and FileWriter, which copy the file f1.text one character at a time to file f2.text.

……

……

……

4.10 SUMMARY

This unit covered the Java.io package. Where you start with two major stream of Java I/O, the InputStream and OutputStream, then learnt the standard methods available in each of these classes. Next, you studied other classes that are in the hierarchy derived from these two major classes and their method to allow easy implementation of different stream types in Java.

In the beginning of this unit we studied File class and its methods for handling files and directories. Further we have looked at the different classes of the Java.io package. We have worked with streams in two directions: pulling data over an input stream or sending data from a program using an output stream. We have used byte stream for many type of non-textual data and character streams to handle text. Filters were associated with streams to alter the way information was delivered through the stream, or to alter the information itself.

Let us recall all the uses and need of the classes we just studied in this unit. Classes based upon the InputStream class allow byte-based data to be input and similarly OutputStream class allows the output. The DataInputStream and DataOutputStream class define methods to read and write primititve datatype values and strings as binary data in host independent format. To read and write functionality with streams Inputstreamreader and outputstreamwriter class is available in Java.io package. The RandomAccessFile class that encapsulates a random access disk file allows a file to be both read and written by a single object reference. Reading functionality for system.in can be provided by using input stream reader. Writing functionality for system.out can be provided by using Print writing. The stream tokenizer class simplifies the parsing of input files which consist of programming language like tokens.

4.11 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Unicode is a 16-bit code having a large range in comparison to previous ASCII code, which is only 7 bits or 8 bits in size. Unicode can represent all the characters of all human languages. Since Java is developed for designing Internet applications, and worldwide people can write programs in Java, transformation of one language to another is simple and efficient. Use of Unicode also supports platform independence in Java.

2)

```
//program
import Java.io.*;
class rename
{
public static void main(String args[])
{
File f1 = new File("out1.txt");
File f2 = new File("out2.txt");
```

```
if(f1.exists())
{
System.out.println(f1 + " File exists");
System.out.println("size of file is: " + f1.length() + "bytes" );
f1.renameTo(f2);
System.out.println("Rename file is: " + f2);
System.out.println("deleting file : " + f1);
f1.delete();
}
else
System.out.println("f1" + "file is not existing");
}
}
```

Output:
out1.txt File exists
size of file is: 22bytes
Rename file is: out2.txt
deleting file : out1.txt

3)

```
//program
import Java.io.*;
class ReadWrite
{
 public static void main(String args[] )throws IOException
 {
 byte[] c= new byte[10];
 System.out.println("Enter a string of 10 character");
 for (int i=0;i < 10;i++)
 c[i]=(byte)System.in.read();
 System.out.println("Following is the string which you typed: ");
 System.out.write(c);
 }
}
```

Output:
C:\Java\Block3Unit3>Java ReadWrite
Enter a string of 10 character
Hi how is life
Following is the string which you typed:
Hi how is

Check Your Progress 2

1) PushbackInputStream is an input stream that can unread bytes. This is a subclass of FilterInputStream which provides the ability to unread data from a stream. It maintains a buffer of unread data that is supplied to the next read operation. It is used in situations where we need to read some unknown number of data bytes that are delimited by some specific byte value. After reading the specific byte program can perform some specific task or it can terminate.

2) This program use FileWriter stream wrapped with a Printwriter to write binary data into file.

```
//program
import Java.io.*;
public class PrimitivesToFile_Appl
{
 public static void main(String args[])
 {
```

```
        try
        {
        FileWriter filewriter = new FileWriter("prim.txt");
        // Wrap the PrintWriter with a PrintWriter
        // for sending the output t the file
        PrintWriter printWriter = new PrintWriter(filewriter);
        boolean b=true;
        byte    by=127;
        short    s=1111;
        int      i=1234567;
        long     l=987654321;
        float    f=432.5f;
        double   d=1.23e-15;
        printWriter.print(b);
        printWriter.print(by);
        printWriter.print(s);
        printWriter.print(i);
        printWriter.print(l);
        printWriter.print(f);
        printWriter.print(d);
        printWriter.close();
        }
        catch (Exception e)
        {
        System.out.println("IO erro" + e);
    }
   }
 }
```

Output:

File prim.txt will contain : true1271111123456798765432143 2.51.23E-15

3) First let us use the Filewriter and BufferedWriter classes to append data to the end of the text file. Here is the FileWriter constructor, you pass in true value in second argument to write to the file in append mode:

```
FileWriter writer = new FileWriter (String filename, boolean append);
```

An alternate answer is to use RandomAccessFile and skip to the end of the file and start writing

```
RandomAccessFile file = new  RandomAccessFile(datafile,"rw");
file.skipBytes((int)file.length()); //skip to the end of the file
file.writeBytes("Add this text to the end of datafile"); //write at the end the of the file
file.close();
```

Check Your Progress 3

1)

```
        //program
        import Java.io.*;
        class FileWriteDemo
        {
        public static void main(String[] args) throws IOException
        {
        // open keyboard for input

BufferedReader stdin = new BufferedReader(new InputStreamReader(System.in));
//output file 'junk.txt'
```

```
String s = "junk.txt";
File f = new File(s);
if (f.exists())
{
System.out.print("Overwrite " + s + " (y/n)? ");
if(!stdin.readLine().toLowerCase().equals("y"))
return;
}
PrintWriter outFile = new PrintWriter(new BufferedWriter(new FileWriter(s)));
System.out.println("Enter some text on the keyboard...");
System.out.println("(^z to terminate)");
String s2;
while ((s2 = stdin.readLine()) != null)
outFile.println(s2);
outFile.close();
}
}
```

2)

```
//program
import Java.lang.System;
import Java.io.RandomAccessFile;
import Java.io.IOException;
public class RandomIOApp
{
 public static void main(String args[]) throws IOException
{
RandomAccessFile file = new RandomAccessFile("test.txt","rw");
file.writeBoolean(true);
file.writeInt(123456);
file.writeChar('j');
file.writeDouble(1234.56);
file.seek(1);
System.out.println(file.readInt());
System.out.println(file.readChar());
System.out.println(file.readDouble());
file.seek(0);
System.out.println(file.readBoolean());
file.close();
}
}
Output:
123456
j
1234.56
true
```

3)

```
//program
import Java.io.*;
public class CopyFile
{
public static void main(String[] args) throws IOException,
FileNotFoundException
{
File inFile = inFile = new File("out2.txt");
FileReader in = new FileReader(inFile);
File outFile = new File("f2.txt");
FileWriter out = new FileWriter(outFile);
```

```
// Copy the file one character at a time.
int c;
while ((c = in.read()) != -1)
out.write(c);
in.close();
out.close();
System.out.println("The copy was successful.");
}
}
```

Output:

The copy was successful.

UNIT 1 APPLETS

Structure **Page Nos.**

1.0 INTRODUCTION

You are already familiar with several aspects of Java applications programming discussed in earlier blocks. In this unit you will learn another type of Java programs called Java Applets. As you know in Java you can write two types of programmes,– Applications and Applets. Unlike a Java application that executes from a command window, an Applet runs in the ***Appletviewer (a test utility for Applets that is included with the J2SDK)*** or a World Wide Web browser such as Microsoft Internet Explorer or Netscape Communicator.

In this unit you will also learn how to work with *event driven programming*. You are very familiar with Windows applications. In these application environments program logic doesn't flow from the top to the bottom of the program as it does in most procedural code. Rather, the operating system collects *events* and the program responds to them. These events may be mouse clicks, key presses, network data arriving on the Ethernet port, or any from about two dozen other possibilities. The operating system looks at each event, determines what program it was intended for, and places the event in the appropriate program's *event queue*.

Every application program has an *event loop*. This is just a while loop which loops continuously. On every pass through the loop the application retrieves the next event from its event queue and responds accordingly.

1.1 OBJECTIVES

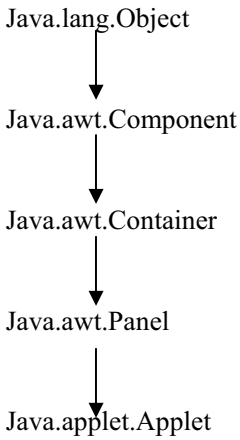
Our objective is to introduce you to Java Applets. After going through this unit, you will be able to:

- describe Java Applets and its Importance;
- explain Applet architecture;
- compile and Execute Java Applet programs;
- use Event Handling constructs of Java programs, and
- apply various HTML tags to execute the Applet programs.

1.2 THE APPLET CLASS

Java is an Object Oriented Programming language, supported by various classes. The Applet class is packed in the *Java. Applet* package which has several interfaces. These interfaces enable the creation of Applets, interaction of Applets with the browser, and playing audio clips in Applets. In Java 2, class Javax.swing. JApplet is used to define an Applet that uses the *Swing GUI components*.

As you know, in Java class hierarchy Object is the base class of Java.lang package. The Applet is placed into the hierarchy as follows:



Now let us see what you should do using Java Applet and what you should not do using it.

Below is the list given for Do's and Don'ts of Java Applets:

Do's

- Draw pictures on a web page
- Create a new window and draw the picture in it.
- Play sounds.
- Receive input from the user through the keyboard or the mouse.
- Make a network connection to the server from where the Applet is downloaded, and send to and receive arbitrary data from that server.

Don'ts

- Write data on any of the host's disks.
- Read any data from the host's disks without the user's permission. In some environments, notably Netscape, an Applet cannot read data from the user's disks even with permission.
- Delete files
- Read from or write to arbitrary blocks of memory, even on a non-memory-protected operating system like the MacOS
- Make a network connection to a host on the Internet other than the one from which it was downloaded.
- Call the native API directly (though Java API calls may eventually lead back to native API calls).
- Introduce a virus or Trojan horse into the host system.

Now we will discuss different components in Applet architecture.

1.3 APPLET ARCHITECTURE

Java Applets are essentially Java programs that run within a web page. Applet programs are Java classes that extend the Java.Applet.Applet class and are embedded by reference within a HTML page. You can observe that when Applets are combined with HTML, they can make an interface more dynamic and powerful than with HTML alone. While some Applets do nothing more than scroll text or play animations, but by incorporating these basic features in web pages you can make them dynamic. These dynamic web pages can be used in an enterprise application to view or manipulate data coming from some source on the server. For example, an

Applet may be used to browse and modify records in a database or control runtime aspects of some other application running on the server.

Besides the class file defining the Java Applet itself, Applets can use a collection of utility classes, either by themselves or archived into a JAR file. The Applets and their class files are distributed through standard HTTP requests and therefore can be sent across firewalls with the web page data. Applet code is refreshed automatically each time the user revisits the hosting web site. Therefore, keeps full application up to date on each client desktop on which it is running.

Since Applets are extensions of the Java platform, you can reuse existing Java components when you build web application interface with Applets. As we'll see in example programs in this unit, we can use complex Java objects developed originally for server-side applications as components of your Applets. In fact, you can write Java code that can operate as either an Applet or an application.

In Figure 1, you can see that using Applet program running in a Java-enabled web browser you can communicate to the server.

Basic WWW/Applet Architecture

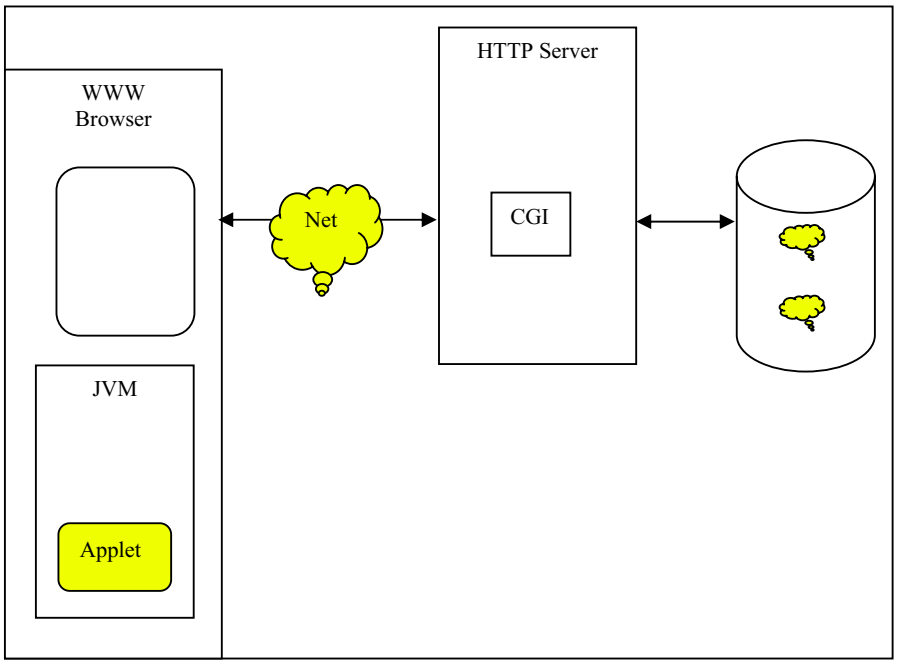


Figure1: Applet Architecture

Check Your Progress 1

1) Explain what an Applet is.

……

……

……

……

……

…

……

…

……
…

2) Explain of Applets security.

……
……
……
……

3) What are the various ways to execute an Applet?

……
……
……
……

4) Why do you think that OOPs concepts are important in the current scenario?

……
……
……
……

Now we will discuss the Applet methods around which Applet programming moves.

1.4 AN APPLET SKELETON: INITIALIZATION AND TERMINATION

In this section we will discuss about the Applet life cycle. If you think about **the Basic Applet Life Cycle, the points listed below should be thought of:**

1. The browser reads the HTML page and finds any <APPLET> tags.
2. The browser parses the <APPLET> tag to find the CODE and possibly CODEBASE attribute.
3. The browser downloads the. **Class** file for the Applet from the URL (Uniform Resource Locator) found in the last step.
4. The browser converts the raw bytes downloaded into a Java class, that is a Java.lang.Class object.
5. The browser instantiates the Applet class to form an Applet object. This requires the Applet to have a no-args constructor.
6. The browser calls the Applet's init () method.
7. The browser calls the Applet's start () method.
8. While the Applet is running, the browser passes all the events intended for the Applet, like mouse clicks, key presses, etc. to the Applet's handle Event () method.
9. The default method paint() in Applets just draw messages or graphics (such as lines, ovals etc.) on the screen. Update events are used to tell the Applet that it needs to repaint itself.
10. The browser calls the Applet's stop () method.

11. The browser calls the Applet's destroy () method.

In brief, you can say that, all Applets use their five following methods:

```
public void init ();
public void start();
public void paint();
public void stop();
public void destroy();
```

Every Applet program use these methods in its life cycle. Their methods are defined in super class, Java.applet. Applet (It has others too, but right now I just want to talk about these five.

In the super class, these are simply do-nothing methods. Subclasses may override these methods to accomplish certain tasks at certain times. For example,

```
public void init() {}
```

init () method is used to read parameters that were passed to the Applet via <PARAM> tags because it's called exactly once when the Applet starts up. If there is such need in your program you can override init() method. Since these methods are declared in the super class, the Web browser can invoke them when it needs, without knowing in advance whether the method is implemented in the super class or the subclass. This is a good example of polymorphism.

A brief description of **init (), start (), paint (), stop (), and destroy ()** methods are given below.

init () method: The init() method is called *exactly once* in an Applet's life, when the Applet is first loaded. It's normally used to read PARAM tags, start downloading any other images or media files you need, and to set up the user interface. Most Applets have init () methods.

start () method: The start() method is called at *least once* in an Applet's life, when the Applet is started or restarted. In some cases it may be called more than once. Many Applets you write will not have explicit start () method and will merely inherit one from their super class. A start() method is often used to start any threads the Applet will need while it runs.

paint () method: The task of paint () method is to draw graphics (such as lines, rectangles, string on characters on the screen).

stop() method: The stop () method is called at least once in an Applet's life, when the browser leaves the page in which the Applet is embedded. The Applet's start () method will be called if at some later point the browser returns to the page containing the Applet. In some cases the stop () method may be called multiple times in an Applet's life. Many Applets you write will not have explicit stop () methods and will merely inherit one from their super class. Your Applet should use the stop () method to pause any running threads. When your Applet is stopped, it *should* not use any CPU cycles.

destroy () method: The destroy() method is called exactly once in an Applet's life, just before the web browser unloads the Applet. This method is generally used to perform any final clean-up. For example, an Applet that stores state on the server might send some data back to the server before it is terminated. Many Applet programs generally don't have explicit destroy () methods and just inherit one from their super class.

Let us take one example to explain the use of the methods explained above. For example in a video Applet, the init () method might draw the controls and start loading the video file. The start () method would wait until the file was loaded, and then start playing it. The stop () method would pause the video, but not rewind it. If the start () method were called again, the video would pick up from where it left off; it would not start over from the beginning. However, if destroy () were called and then init (), the video would start over from the beginning.

The point to note here is, if you run an Applet program using Appletviewer, selecting the restart menu item calls stop () and then start (). Selecting the Reload menu item calls stop (), destroy (), and init (), in the order.

Note 1: The Applet start () and stop () methods are not related to the similarly named methods in the Java.lang.Thread class, you have studied in block 3 unit 1.

Note 2: i) Your own code may occasionally invoke star t() and stop(). For example, it is customary to stop playing an animation when the user clicks the mouse in the Applet and restart it when s/he clicks the mouse again.

ii) Now we are familiar with the basic needs and ways to write Applet program. You can see a simple Applet program given below to say Hello World.

```
import Java.applet.Applet;
import Java.awt.Graphics;
public class HelloWorldApplet extends Applet
 {
  public void paint(Graphics g)
   {
    g.drawString("Hello world!", 50, 25);
    }
}
```

If you observe, that this Applet version of Hello World is a little more complicated than to write an application program to say Hello World, it will take a little more effort to run it as well.

First you have to type in the source code and save it into file called HelloWorldApplet.Java.
Compile this file in the usual way.
If all is well a file called HelloWorldApplet.class will be created.
Now you need to create an HTML file that will include your Applet. The following simple HTML file will do.

```
<HTML>
<HEAD>
<TITLE> Hello World </TITLE>
</HEAD>
<BODY>
This is the Applet:<P>
<Applet code="HelloWorldApplet" width="150" height="50">
</Applet>
</BODY>
</HTML>
```

Save this file as HelloWorldApplet.html in the same directory as the HelloWorldApplet.class file.

When you've done that, load the HTML file into a Java enabled browser like Internet Explorer or Sun's Applet viewer. You should see something like below, though of course the exact details depend on which browser you use.
If the Applet is compiled without error and produced a HelloWorldApplet.class file, and yet you don't see the string "Hello World" in your browser chances are that the class file is in the wrong place. Make sure HelloWorldApplet.class is in the same directory as HelloWorld.html.

Also make sure that you're using a version of Netscape or Internet Explorer which supports Java. Not all versions do.

In any case Netscape's Java support is less than the perfect, so if you have trouble with an Applet, the first thing to try is load it into Sun's Applet Viewer. If the Applet Viewer has a problem, then chances are pretty good the problem is with the Applet and not with the browser.

According to Sun "*An Applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application. The Applet class provides a standard interface between Applets and their environment.*"

The output of the program will be like:

You know GUI programming is heavily dependent on events like mouse click, button pressed, key pressed, etc. Java also supports event handling. Now let us see how these events are handled in Java.

Check Your Progress 2

1) What is the sequence of interpretation, compilation of a Java Applet?

……

……

……

2) How can you re-execute an Applet from Appletviewer?

……

……

……

……

3) Give in brief the description of an Applet life cycle.

……

……

……

……

……

…

……

…

1.5 HANDLING EVENTS

You are leaving for work in the morning and someone rings the doorbell….
That is an event!

In life, you encounter events that force you to suspend other activities and respond to them immediately. In Java, events represent all actions that go on between the user and the application. Java's Abstract Windowing Toolkit (AWT) communicates these actions to the programs using events. When the user interacts with a program let us say by clicking a command button, the system creates an event representing the action and delegates it to the event-handling code within the program. This code determines how to handle the event so the user gets the appropriate response.

Originally, JDK 1.0.2 applications handled events using an inheritance model. A container sub class inherits the action () and handle Event () methods of its parent and handled the events for all components it contained. For instance, the following code would represent the event handler for a panel containing an OK and a Cancel button:

```
public boolean handleEvent(Java.awt.Event e)
{
        if (e.id == Java.awt.Event.ACTION_EVENT)
        {
 if (e.target == buttonOK)
 {
 buttonOKPressed();
 }
 else if (e.target == buttonCancel)
 (
                buttonCancelPressed();
 )
}
 return super.handleEvent(e);
}
```

The problem with this method is that events cannot be delivered to specific objects. Instead, they have to be routed through a universal handler, which increases complexity and therefore, weakens your design.

But Java 1.1 onward has introduced the concepts of the ***event delegation*** model. This model allows special classes, known as "adapter classes" to be built and be registered with a component in order to handle certain events. Three simple steps are required to use this model:

1. Implement the desired listener interface in your adapter class. Depending on what event you're handling, a number of listener interfaces are available. These include: ActionListener, WindowListener, MouseListener, MouseMotionListener, ComponentListener, FocusListener, and ListSelectionListener.

2. Register the adapter listener with the desired component(s). This can be in the

form of an add XXX Listener () method supported by the component for example include add ActionListener (), add MouseListener (), and add FocusListener ().

3. Implement the listener interface's methods in your adapter class. It is in this code that you will actually handle the event.

The event delegation model allows the developer to separate the component's display (user interface) from the event handling (application data) which results in a cleaner and more object-oriented design.

Components of an Event: Can be put under the following categories.

1. ***Event Object:*** When the user interacts with the application by clicking a mouse button or pressing a key an event is generated. The Operating System traps this event and the data associated with it. For example, info about time at which the event occurred, the event types (like keypress or mouse click etc.). This data is then passed on to the application to which the event belongs.

   You must note that, in Java, objects, which describe the events themselves, represent events. Java has a number of classes that describe and handle different categories of events.

2. ***Event Source:*** An event source is the object that generated the event, for example, if you click a button an ActionEvent Object is generated. The object of the ActionEvent class contains information about the event (button click).

3. ***Event-Handler:*** Is a method that understands the event and processes it. The event-handler method takes the Event object as a parameter. You can specify the objects that are to be notified when a specific event occurs. If the event is irrelevant, it is discarded.

The four main components based on this model are ***Event classes, Event Listeners, Explicit event handling and Adapters***.

Let me give you a closer look at them one by one.

Event Classes: The EventObject class is at the top of the event class hierarchy. It belongs to the **Java.util** package. While most of the other event classes are present in Java.awt.event package.

The get Source () method of the EventObject class returns the object that initiated the event.

The getId () method returns the nature of the event. For example, if a mouse event occurs, you can find out whether the event was a click, a press, a move or release from the event object.

AWT provides two conceptual types of events: **Semantic and Low-level events**.

Semantic event **:** These are defined at a higher-level to encapsulate the semantics of user interface component's model.

Now let us see what the various semantic event classes are and when they are generated:

An **ActionEvent** object is generated when a component is activated.

An **Adjustment Event** Object is generated when scrollbars and other adjustment elements are used.

A **Text Event** object is generated when text of a component is modified.

An **Item Event** is generated when an item from a list, a choice or checkbox is selected.

Low-Level Events: These events are those that represent a low-level input or windows-system occurrence on a visual component on the screen.

The various low-level event classes and what they generate are as follows:

- A **Container Event** Object is generated when components are added or removed from container.
- A **Component Event** object is generated when a component is resized moved etc.
- A **Focus Event** object is generated when component receives focus for input.
- A **Key Event** object is generated when key on keyboard is pressed, released etc.
- A **Window Event** object is generated when a window activity, like maximizing or close occurs.
- A **Mouse Event** object is generated when a mouse is used.
- A **Paint Event** object is generated when component is painted.

Event Listeners: An object delegates the task of handling an event to an **event listener**. When an event occurs, an event object of the appropriate type (as explained below) is created. This object is passed to a **Listener**. The listener must **implement the interface** that has the method for event handling. A component can have multiple listeners, and a listener can be removed using **remove Action Listener ()** method. You can understand a listener as a person who has listened to your command and is doing the work which you commanded him to do so.

As you have studied about interfaces in Block 2 Unit 3 of this course, an **Interface** contains constant values and method declaration. The Java.awt.event package contains definitions of all event classes and listener interface. The **semantic listener interfaces** defined by AWT for the above-mentioned semantic events are:

- ActionListener
- AjdustmentListener
- ItemListener
- TextListener

The **low-level event listeners** are as follows:

- ComponentListener
- ContainerListener
- FocusListener
- KeyListener
- MouseListener
- MouseMotionListener
- WindowsListener.

Action Event using the ActionListener interface: In *Figure 2* the Event Handling Model of AWT is given. This figure illustrates the usage of ActionEvent and ActionListener interface in a Classic Java Application.

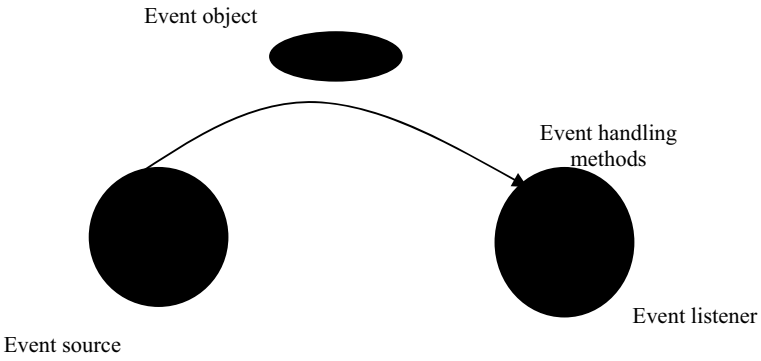


Figure 2: Event Handling Model of AWT

You can see in the program given below that the Button Listener is handling the event generated by pressing button “Click Me”. Each time the button “Click Me” is pressed; the message “The Button is pressed” is printed in the output area. As shown in the output of this program, the message. “The Button is pressed” is printed three times, since “Click Me” is pressed thrice.

```
import Java.awt.event.*;
import Java.awt.*;
public class MyEvent extends Frame
{
  public MyEvent()
   {
         super("Window Title: Event Handling");
        Button b1;
        b1 = new Button("Click Me");
        //getContentPane().add(b1);
        add(b1);
        Button Listener listen = new Button Listener();
        b1.add Action Listener(listen);
             set Visible (true);
              set Size (200,200);
      }
   public static void main (String args[])
   {
        My Event event = new My Event();
     }
}; // note the semicolon
  class Button Listener implements ActionListener
   {
        public void action Performed (ActionEvent evt)
          {
               Button source1 = (Button) evt. get Source ();
               System. out. print ln ("The Button is Pressed");
           }
}
```

Output of the program:

The Button is Pressed
The Button is Pressed
The Button is Pressed

How does the above Application work? The answer to this question is given below in steps.

1. The execution begins with the main method.
2. An Object of the MyEvent class is created in the main method, by invoking the constructor of the MyEvent class.
3. Super () method calls the constructor of the base class and sets the title of the window as given, “Windows Title: Event Handling”.
4. A button object is created and placed at the center of the window.
5. Button object is added in the event.
6. A Listener Object is created.
7. The addAction Listener () method registers the listener object for the button.
8. SetVisible () method displays the window.
9. The Application waits for the user to interact with it.
10. Then the user clicks on the button labeled “Click Me”: The “ActionEvent” event is generated. Then the ActionEvent object is created and delegated to the registered listener object for processing. The Listener object contains the actionPerformed () method which processes the ActionEvent In the actionPerformed () method, the reference to the event source is retrieved using getSource () method. The message “The Button is Pressed” is printed.

1.6 HTML APPLET TAG

Till now you have written some Applets and have run them in the browser or Appletviewer. You have used basic tags needed for running an Applet program. Now you will learn some more tag that contains various attributes.

Applets are embedded in web pages using the <APPLET> and </APPLET> tags. APPLET elements accept The. *class* file with the CODE attribute. The CODE attribute tells the browser where to look for the compiled .class file. It is relative to the location of the source document.

If the Applet resides somewhere other than the same directory where the page lives on, you don’t just give a URL to its location. Rather, you have to point at the CODEBASE.

The CODEBASE attribute is a URL that points at the directory where the *.class* file is. The CODE attribute is the name of the *.class* file itself. For instance if on the HTML page in the previous section had you written

```
<APPLET CODE="HelloWorldApplet" CODEBASE="classes"
WIDTH="200" HEIGHT="200">
</APPLET>
```

then the browser would have tried to find HelloWorldApplet.class in the classes subdirectory inside in directory where the HTML page that included the Applet is contained. On the other hand if you had written

```
<APPLET CODE="HelloWorldApplet"
CODEBASE="http://www.mysite.com/classes" WIDTH="200" HEIGHT="200">
</APPLET>
```

then the browser would try to retrieve the Applet from http://www.mysite.com/classes/HelloWorldApplet.class regardless of where the HTML page is residing.

If the Applet is in a non-default package, then the full package qualified name must be used. For example,

```
<APPLET CODE="mypackage. HelloWorldApplet"
CODEBASE="c:\Folder\Java\" WIDTH="200" HEIGHT="200">
</APPLET>
```

The HEIGHT and WIDTH attributes work exactly as they do with IMG, specifying how big a rectangle the browser should set aside for the Applet. These numbers are specified in pixels.

Spacing Preferences

The <APPLET> tag has several attributes to define how it is positioned on the page. The ALIGN attribute defines how the Applet's rectangle is placed on the page relative to other elements. Possible values include LEFT, RIGHT, TOP, TEXTTOP, MIDDLE, ABSMIDDLE, BASELINE, BOTTOM and ABSBOTTOM. This attribute is optional.

You can specify an HSPACE and a VSPACE in pixels to set the amount of blank space between an Applet and the surrounding text. The HSPACE and VSPACE attributes are optional.

```
<Applet code="HelloWorldApplet"   width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10>
</APPLET>
```

The ALIGN, HSPACE, and VSPACE attributes are identical to the attributes used by the <IMG> tag.

Alternate Text

The <APPLET> has an ALT attribute. An ALT attribute is used by a browser that understands the APPLET tag but for some reason cannot play the Applet. For instance, if you've turned off Java in Netscape Navigator 3.0, then the browser should display the ALT text. The ALT tag is optional.

```
<Applet code="HelloWorldApplet"
CODEBASE="c:\Folder\classes" width="200" height="200"
ALIGN="RIGHT" HSPACE="5" VSPACE="10"
ALT="Hello World!">
</APPLET>
```

ALT is not used by browsers that do not understand <APPLET> at all. For that purpose <APPLET> has been defined to require a closing tag, </APPLET>.

All raw text between the opening and closing <APPLET> tags is ignored by a Java capable browser. However, a non-Java-capable browser will ignore the <APPLET> tags instead and read the text between them. For example, the following HTML fragment says Hello World!, both with and without Java-capable browsers.

```
<Applet code="HelloWorldApplet" width=200 height=200
ALIGN=RIGHT HSPACE=5 VSPACE=10
ALT="Hello World!">
Hello World!<P>
</APPLET>
```

Naming Applets

You can give an Applet a name by using the NAME attribute of the APPLET tag. This allows communication between different Applets on the same Web page.

```
<APPLET CODE="HelloWorldApplet" NAME="Applet1"
CODEBASE=”c:\Folder\Classes”  WIDTH="200" HEIGHT="200"
align="right" HSPACE="5" VSPACE="10"
ALT="Hello World!">
Hello World!<P>
</APPLET>
```

JAR Archives

HTTP 1.0 uses a separate connection for each request. When you’re downloading many small files, the time required to set up and tear down the connections can be a significant fraction of the total amount of time needed to load a page. It would be better if you could load all the HTML documents, images, Applets, and sounds in a page in one connection.

One way to do this without changing the HTTP protocol is to pack all those different files into a single archive file, perhaps a zip archive, and just download that.

We aren’t quite there yet. Browsers do not yet understand archive files, but in Java 1.1 Applets do. You can pack all the images, sounds; and.class files that an Applet needs into one JAR archive and load that instead of the individual files. Applet classes do not have to be loaded directly. They can also be stored in JAR archives.

To do this you use the ARCHIVES attribute of the APPLET tag.

```
<APPLET CODE="HelloWorldApplet" WIDTH="200" HEIGHT="100"
ARCHIVES="HelloWorld.jar">
<hr>
Hello World!
<hr>
</APPLET>
```

In this example, the Applet class is still HelloWorldApplet. However, there is no HelloWorldApplet.class file to be downloaded. Instead the class is stored inside the archive file HelloWorld.jar.

Sun provides a tool for creating JAR archives with its JDK 1.1 onwards.

For Example

```
% jar cf HelloWorld.jar *.class
```

This puts all the .class files in the current directory in a file named "HelloWorld.jar". The syntax of the jar command is similar to the Unix tar command.

The Object Tag

HTML 4.0 deprecates the <APPLET> tag. Instead you are supposed to use the <OBJECT> tag. For the purposes of embedding Applets, the <OBJECT> tag is used almost exactly like the <APPLET> tag except that the class attribute becomes the classid attribute. For example,

```
<OBJECT classid="MyApplet" CODEBASE=”c:\Folder\Classes” width=200
height=200 ALIGN=RIGHT HSPACE=5 VSPACE=10>
</OBJECT>
```

The <OBJECT> tag is also used to embed ActiveX controls and other kinds of active content. It has a few additional attributes to allow it to do that. However, for the purposes of Java you don’t need to know about these.

The <OBJECT> tag is supported by Netscape and Internet Explorer. It is not supported by earlier versions of these browsers. <APPLET> is unlikely to disappear anytime soon in the further.

You can support both by placing an <APPLET> element inside an <OBJECT> element like this:

```
<OBJECT classid="MyApplet" width="200" height="200">
<APPLET code="MyApplet" width="200" height="200">
</APPLET>
</OBJECT>
```

You will notice that browsers that understand <OBJECT> will ignore its content while the browsers will display its content.

PARAM elements are the same for <OBJECT> as for <APPLET>.

Passing Parameters to Applets

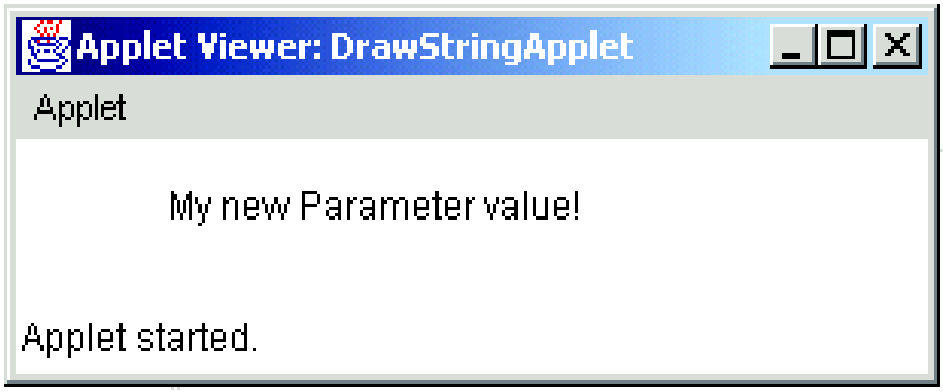
Parameters are passed to Applets in NAME and VALUE attribute pairs in <PARAM> tags between the opening and closing APPLET tags. Inside the Applet, you read the values passed through the PARAM tags with the getParameter() method of the Java.Applet.Applet class.

The program below demonstrates this with a generic string drawing Applet. The Applet parameter "Message" is the string to be drawn.

```
import Java.Applet.*;
import Java.awt.*;
public class DrawStringApplet extends Applet {
  private String str = "Hello!";

  public void paint(Graphics g) {
   String str = this.getParameter("Message");
   g.drawString(str, 50, 25);

  }

}
```

You also need an HTML file that references your Applet. The following simple HTML file will do:

```
<HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>
<BODY>
This is the Applet:<P>
<APPLET code="Draw String Applet" width="300" height="50">
<PARAM name="Message" value="My new Parameter value!">

</APPLET>
</BODY>
</HTML>
```



Of course you are free to change “My new Parameter value!” to a “message” of your choice. You only need to change the HTML, not the Java source code. PARAMs let you customize Applets without changing or recompiling the code.

This Applet is very similar to the HelloWorldApplet. However rather than hardcoding the message to be printed it’s read into the variable str through get parameter () method from a PARAM in the HTML.

You pass get Parameter() a string that names the parameter you want. This string should match the name of a <PARAM> tag in the HTML page. getParameter() returns the value of the parameter. All values are passed as strings. If you want to get another type like an integer, then you’ll need to pass it as a string and convert it to the type you really want.

The <PARAM> HTML tag is also straightforward. It occurs between <APPLET> and </APPLET>. It has two attributes of its own, NAME and VALUE. NAME identifies which PARAM this is. VALUE is the value of the PARAM as a String. Both should be enclosed in double quote marks if they contain white space.

An Applet is not limited to one PARAM. You can pass as many named PARAMs to an Applet as you like. An Applet does not necessarily need to use all the PARAMs that are in the HTML. You can be safely ignore additional PARAMs.

Processing An Unknown Number Of Parameters

Sometimes the parameters are not known to you, in that case most of the time you have a fairly good idea of what parameters will and won’t be passed to your Applet or perhaps you want to write an Applet that displays several lines of text. While it would be possible to cram all this information into one long string, that’s not too friendly to authors who want to use your Applet on their pages. It’s much more sensible to give each line its own <PARAM> tag. If this is the case, you should name the tags via some predictable and numeric scheme. For instance in the text example the following set of <PARAM> tags would be sensible:

```
<PARAM name="param1" value="Hello Good Morning">
```

<PARAM name="param2" value="Here I am ">

Check Your Progress 3

1) Is it possible to access the network resources through an Applet on the browser?

……

……

……

……

……

……

……

2) Write a program in which whenever you click the mouse on the frame, the coordinates of the point on which the mouse is clicked are displayed on the screen.

……

……

……

……

……

……

……

……

……

……

3) Write an Applet program in which you place a button and a textarea. When you click on button, in text area Your name and address is displayed. You have to take your name and address using < PARAM>.

……

……

……

……

…….

.

……

…….

1.7 SUMMARY

In this unit we have discussed Applet programs. Applets are secure programs that run inside a web browser and it's a subclass of Java.Applet. Applet.or its an instance of subclass of Java.Applet. Applet goes through five phases such as:

start(), init(),paint(),stop() and destroy(). paint() is one of the three methods that are guaranteed to be called automatically for you when any Applet begins execution.

These three methods are init, start and paint, and they are guaranteed to be called in that order. These methods are called from Appletviewer or browser in which the Applet is executing .The <Applet> tag first attribute or indicates the file containing the compiled Applet class. It specifies height and width of the Applet tag to process an event you must register an event listener and implement one or more event handlers. The use of event listeners in event handling is known as Event Delegation Model.

You had seen various events and event listeners associated with each component. The information about a GUI event is stored in an object of a class that extends AWT Event.

1.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) An Applet is a small program that is intended not to be run on its own, but rather to be embedded inside another application. The Applet class provides a standard interface between Applets and their environment. It is a secure program that runs inside a web browser. Applet can be embedded in an HTML page. Applets differ from Java applications in the way that they are not allowed to access certain resources on the local computer, such as files and serial devices (modems, printers, etc.), and are prohibited from communicating with most other computers across a network. The common rule is that an Applet can only make an Internet connection to the computer from which the Applet was sent.

2) You can surf the web without worrying that a Java Applet will format your hard disk or introduce a virus into your system. In fact you must have noticed that Java Applets and applications are much safer in practice than code written in traditional languages. This is because even code from trusted sources is likely to have bugs. However Java programs are much less susceptible to common bugs involving memory access than are programs written in traditional languages like C. Furthermore, the Java *runtime environment* provides a fairly robust means of trapping bugs before they bring down your system. Most users have many more problems with bugs than they do with deliberately malicious code. Although users of Java applications aren't protected from out and out malicious code, they are largely protected from programmer errors.Applets implement additional security restrictions that protect users from malicious code too. This is accomplished through the Java.lang.SecurityManager class. This class is sub classed to provide different security environments in different virtual machines. Regrettably implementing this additional level of protection does somewhat restrict the actions an Applet can perform.

Applets are not allowed to write data on any of the host's disks or read any data from the host's disks without the user's permission. In some environments, notably Netscape, an Applet cannot read data from the user's disks even with permission.

3) An Applet is a Java program that runs in the Appletviewer or a World Wide Web browser such as Netscape Communicator or Microsoft Internet Explorer. The Applet viewer (or browser) executes an Applet when a Hypertext Markup Language (HTML) document containing the Applet is opened in the Applet viewer (or web browser).

4) Object Oriented Programming (OOP) models real world objects with software counterparts. It takes advantage of class relationships where objects of a certain class have the same characteristics. It takes advantage of inheritance relationships where newly created classes are derived by inheriting characteristics of existing classes; yet contain unique characteristics of their

own. Object Oriented concepts implemented through Applet programming can communicate one place to other through web pages.

Check Your Progress 2

1) A Java program is first compiled to bytecode which is stored in a'.class file'. This file is downloaded as an Applet to a browser, which then interprets it by converting into machine code appropriate to the hardware platform on which the Applet program is running.

2) In the Appletviewer, you can execute an Applet again by clicking the Applet viewer's Applet menu and selecting the **Reload** option from the menu. To terminate an Applet, click the Appletviewer's Applet menu and select the **Quit** option.

3) When the Applet is executed from the Appletviewer, The Appletviewer only understands the <Applet> and </Applet> HTML tags, so it is sometimes referred to as the "minimal browser".(It ignores all other HTML tags). The starting sequence of method calls made by the Appletviewer or browser for every Applet is always init (), start () and paint () – this provides a start-up sequence of method calls as every Applet begins execution.

The stop () method would pause the Applet, However, destroy () will terminate the application.

Check Your Progress 3

1) There is no way that an Applet can access network resources. You have to implement a Java Query Sever that will be running on the same machine from where you are receiving your Applet (that is Internet Server). Your Applet will communicate with that server which fulfill all the requirement of the Applet. You communicate between Applet and Java query server using sockets Remote Method Invokation (RMI), this may be given preference because it will return the whole object.

2)

```
import Java.Applet.*;
import Java.awt.*;
import Java.awt.event.*;
 public class TestMouse1
{
public static void main (String[] args)
{
    Frame f = new Frame("TestMouseListener");
    f.setSize(500,500);
    f.setVisible(true);
    f.addMouseListener(new MouseAdapter()
   {
     public void mouseClicked(MouseEvent e)
    {
      System.out.println("Mouse clicked: ("+e.getX()+","+e.getY()+")");
    }
    });
}
}
```

3)

DrawStringApplet1.Java file:

```
import Java.Applet.*;
import Java.awt.*;
```

```
public class DrawStringApplet1 extends Applet
{
   public void paint(Graphics g)
  {
    String str1 = this.getParameter("Message1");
    g.drawString(str1, 50, 25);
    String str2 = this.getParameter("Message2");
    g.drawString(str2, 50, 50);
   }
 }
```

DrawStringApplet1.html file:

```
<HTML>
<HEAD>
<TITLE> Draw String </TITLE>
</HEAD>
<BODY>
<APPLET code="DrawStringApplet1" width="300" height="250">
<PARAM name="Message1" value="M. P. Mishra">
<PARAM name="Message2" value="SOCIS, IGNOU, New Delhi-68">
</APPLET>
</BODY>
</HTML>
```

Compile DrawStringApplet1.Java file then run DrawStringApplet1.html file either in web browser or through Appletviewer.

UNIT 2 GRAPHICS AND USER INTERFACES

Structure **Page Nos.**

2.0 INTRODUCTION

We all wonder that on seeing 2D or 3D movies or graphics, even on the Internet, we see and admire good animation on various sites. The java technology provides you the platform to develop these graphics using the Graphics class. In this unit you have to overview several java capabilities for drawing two-dimensional shapes, colors and fonts. You will learn to make your own interface, which will be user-interactive and friendly. These interfaces can be made either using java AWT components or java SWING components. In this unit you will learn to use some basic components like button, text field, text area, etc.

Interfaces using GUI allow the user to spend less time trying to remember which keystroke sequences do what and allow spend more time using the program in a productive manner. It is very important to learn how you can beautify your components placed on the canvas area using FONT and COLOR class. For placing components on different layouts knowing use of various Layout managers is essential.

2.1 OBJECTIVES

Our objective is to give you the concept of making the look and feel attractive. After going through this unit, you will be able to:

- explain the principles of graphical user interfaces;
- differentiate between AWT and SWING components;
- design a user friendly Interface;
- applying color to interfaces;
- using Font class in programs;
- use the Layout Manager, and
- use Container Classes.

2.2 GRAPHICS CONTEXTS AND GRAPHICS OBJECTS

How can you build your own animation using Graphics Class? A Java graphics context enables drawing on the screen. A Graphics object manages a graphics context by controlling how objects are drawn. Graphics objects contain methods for drawing, font manipulation, color manipulation and the other related operations. It has been developed using the Graphics object g (the argument to the applet's paint method) to manage the applet's graphics context. In other words you can say that Java's Graphics is capable of:

- Drawing 2D shapes
- Controlling colors
- Controlling fonts
- Providing Java 2D API
- Using More sophisticated graphics capabilities
- Drawing custom 2D shapes
- Filling shapes with colors and patterns.

Before we begin drawing with Graphics, you must understand Java's coordinate system, which is a scheme for identifying every possible point on the screen. Let us start with Graphics Context and Graphics Class.

Graphics Context and Graphics Class

- Enables drawing on screen
- Graphics object manages graphics context
- Controls how objects is drawn
- Class Graphics is abstract
- Cannot be instantiated
- Contributes to Java's portability
- Class Component method paint takes Graphics object.

The *Graphics* class is the abstract base class for all graphics contexts. It allows an application to draw onto components that are realized on various devices, as well as on to off-screen images.

Public Abstract Class Graphics Extends Object

You have seen that every applet performs drawing on the screen.

Graphics Objects

In Java all drawing takes place via a Graphics object. This is an instance of the class *java.awt.Graphics*.

Initially the Graphics object you use will be passed as an argument to an applet's paint() method. The drawing can be done Applet Panels, Frames, Buttons, Canvases etc.

Each Graphics object has its own coordinate system, and methods for drawing strings, lines, rectangles, circles, polygons etc. Drawing in Java starts with particular Graphics object. You get access to the Graphics object through the *paint(Graphics g)* method of your applet.

Each draw method call will look like

g.drawString("Hello World", 0, 50);

Where g is the particular Graphics object with which you're drawing.
For convenience sake in this unit the variable g will always refer to a pre-existing object of the Graphics class. It is not a rule you are free to use some other name for the particular Graphics context, such as myGraphics or applet-Graphics or anything else.

2.2.1 Color Control

It is known that Color enhances the appearance of a program and helps in conveying meanings. To provide color to your objects use class Color, which defines methods and constants for manipulation colors. Colors are created from **red**, **green** and **blue** components **RGB** values.

All three RGB components can be integers in the range 0 to 255, or floating point values in the range 0.0 to 1.0

The first part defines the amount of *red*, the second defines the amount of *green* and the third defines the amount of *blue*. So, if you want to give dark red color to your graphics you will have to give the first parameter value 255 and two parameter zero.

Some of the most common colors are available by name and their RGB values in *Table 1.*

Table 1: Colors and their RGB values

Color Constant	Color	RGB Values
Public final static Color ORANGE	Orange	255, 200, 0
Public final static Color PINK	Pink	255, 175, 175
Public final static Color CYAN	Cyan	0, 255, 255
Public final static Color MAGENTA	Magenta	255, 0, 255
Public final static Color YELLOW	Yellow	255, 255, 0
Public final static Color BLACK	Black	0, 0, 0
Public final static Color WHITE	White	255, 255, 255
Public final static Color GRAY	Gray	128, 128, 128
Public final static Color LIGHT_GRAY	light gray	192, 192, 192
Public final static Color DARK_GRAY	dark gray	64, 64, 64
Public final static Color RED	Red	255, 0, 0
Public final static Color GREEN	Green	0, 255, 0
Public final static Color BLUE	Blue	0, 0, 255

Color Methods

To apply color in your graphical picture (objects) or text two Color methods get Color and set Color are provided. Method get Color returns a Color object representing the current drawing color and method set Color used to sets the current drawing color.

Color constructors

public Color(int r, int g, int b)**:** Creates a color based on the values of red, green and blue components expressed as integers from 0 to 255.

public Color (float r, float g, float b) **:** Creates a color based the values of on red, green and blue components expressed as floating-point values from 0.0 to 1.0.

Color methods:

public int getRed(): Returns a value between 0 and 255 representing the red content
public int getGreen(): Returns a value between 0 and 255 representing the green content
public int getBlue() . Returns a value between 0 and 255 representing the blue content.

Graphics Methods For Manipulating Colors

public Color getColor (): Returns a Color object representing the current color for the graphics context.
public void setColor (Color c): Sets the current color for drawing with the graphics context.
As you do with any variable you should preferably give your colors descriptive names. For instance
Color medGray = new Color(127, 127, 127);
Color cream = new Color(255, 231, 187);
Color lightGreen = new Color(0, 55, 0);

You should note that Color is not a property of a particular rectangle, string or other object you may draw, rather color is a part of the Graphics object that does the drawing. You change the color of your Graphics object and everything you draw from that point forward will be in the new color, at least until you change it again.

When an applet starts running , its color is set to *black by default*. You can change this to red by calling *g.setColor(Color.red)*. You can change it back to black by calling *g.setColor(Color.black)*.

☞ Check Your Progress 1

1) What are the various color constructors?
……
……

2) Write a program to set the Color of a String to red.
……
……

3) What is the method to retrieve the color of the text? Write a program to retrieve R G B values in a given color.
……
……

Now we will discuss about how different types of fonts can be provided to your strings.

2.2.2 Fonts

You must have noticed that until now all the applets have used the default font. However unlike HTML Java allows you to choose your fonts. Java implementations

are guaranteed to have a *serif font* like *Times* that can be accessed with the name *"Serif"*, a monospaced font like *courier* that can be accessed with the name *"Mono",* and a *sans serif* font like *Helvetica* that can be accessed with the name *"SansSerif"*.

How can you know the available fonts on your system for an applet program? You can list the fonts available on the system by using the *getFontList()* method from *java.awt.Toolkit*. This method returns an array of strings containing the names of the available fonts. These may or may not be the same as the fonts to installed on your system. It is implementation is dependent on whether or not all the fonts in a system are available to the applet.

Choosing a font face is very easy. You just create a new Font object and then call. setFont(Font f).

To instantiate a Font object the constructor

public Font(String name, int style, int size)

can be used. ***name*** is the name of the font family, e.g. "Serif", "SansSerif", or "Mono".
size is the size of the font in points. In computer graphics a point is considered to be equal to one pixel. 12 points is a normal size font.

style is an mnemonic constant from *java.awt.Font* that tells whether the text will be bold, italics or plain. The three constants are Font.PLAIN, Font.BOLD, and Font.ITALIC.

In other words, The Class Font contains methods and constants for font control. Font constructor takes three arguments

Font name	Monospaced, SansSerif, Serif, etc.
Font style	Font.PLAIN, Font.ITALIC and Font.BOLD
Font size	Measured in points (1/72 of inch)

Graphics method for Manipulating Fonts

public Font get Font(): Returns a Font object reference representing the current Font.
public void setFont(Font f): Sets the current font, style and size specified by the Font object reference f.

2.2.2.1 FontMetrics

Sometimes you will need to know how much space a particular string will occupy. You can find this out with a FontMetrics object.

FontMetrics allow you to determine the height, width or other useful characteristics of a particular string, character, or array of characters in a particular font. In *Figure 1* you can see a string with some of its basic characteristics.

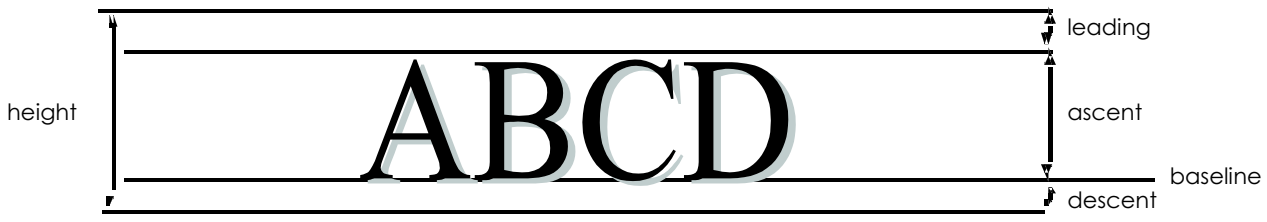


Figure 1: String Characteristics

In order to tell where and whether to wrap a String, you need to measure the string, not its length in characters, which can be variable in its width, and height in pixels. Measurements of this sort on strings clearly depend on the font that is used to draw the string. All other things being equal a 14-point string will be wider than the same string in 12 or 10-point type.

To measure character and string sizes you need to look at the FontMetrics of the current font. To get a FontMetrics object for the current Graphics object you use the *java.awt.Graphics.getFontMetrics()* method.

java.awt.FontMetrics provide method *stringWidth(String s)* to return the width of a string in a particular font, and method *getLeading()* to get the appropriate line spacing for the font. There are many more methods in java.awt.FontMetrics that let you measure the height and width of specific characters as well as ascenders, descenders and more, but these three methods will be sufficient for basic programs.

☞ Check Your Progress 2

1) Write a program to set the font of your String as font name as “Arial”,font size as 12 and font style as FONT.ITALIC.

……

……

2) What is the method to retrieve the font of the text? Write a program for font retrieval.

……

……

3) Write a program that will give you the Fontmetrics parameters of a String.

……

……

Knowledge of co-ordinate system is essential to play with positioning of objects in any drawing. Now you will see how coordinates are used in java drawings.

2.2.3 Coordinate System

By Default the upper left corner of a GUI component (such as applet or window) has the coordinates (0,0). A Coordinate pair is composed of x-coordinate (the horizontal coordinate) and a y-coordinate (the vertical coordinate). The x-coordinate is the horizontal distance moving right from the upper left corner.

The y-coordinate is the vertical distance moving down from the upper left corner. The x-axis describes every horizontal coordinate, and the y-axis describes every vertical coordinate. You must note that different display cards have different resolutions (i.e. the density of pixels varies). *Figure 2* represents coordinate system. This may cause graphics to appear to be different sizes on different monitors.

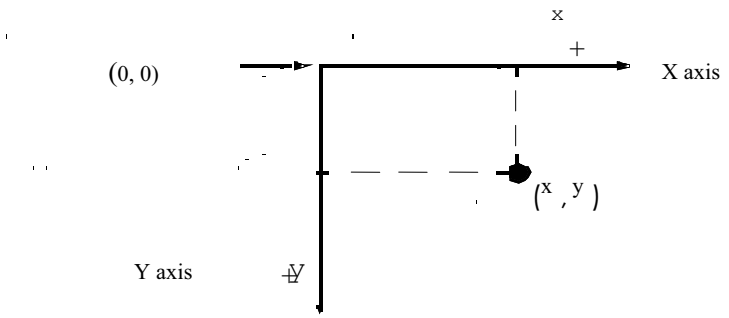


Figure 2: Co-ordinate System

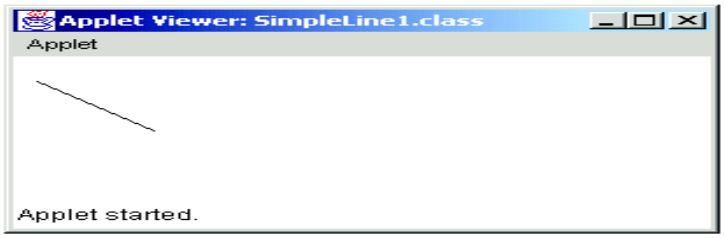
Now we will move towards drawing of different objects. In this section, I will demonstrate drawing in applications.

2.2.3.1 Drawing Lines

Drawing straight lines with Java can be done as follows:
call
g.drawLine(x1,y1,x2,y2)
method, where (x1, y1) and (x2, y2) are the endpoints of your lines and g is the Graphics object you are drawing with. The following program will result in a line on the applet.

```
import java.applet.*;
import java.awt.*;
public class SimpleLine extends Applet
{
 public void paint(Graphics g)
 {
  g.drawLine(10, 20, 30, 40);
} }
```

Output:



2.2.3.2 Drawing Rectangle

Drawing rectangles is simple. Start with a Graphics object g and call its drawRect() method:

public void drawRect(int x, int y, int width, int height)

The first argument int is the left hand side of the rectangle, the second is the top of the rectangle, the third is the width and the fourth is the height. This is in contrast to some APIs where the four sides of the rectangle are given.

Remember that the upper left hand corner of the applet starts at (0, 0),
not at (1, 1). This means that a 100 by 200 pixel applet includes the points with x coordinates between 0 and 99, not between 0 and 100. Similarly the y coordinates are between 0 and 199 inclusive, not 0 and 200.

2.2.3.3 Drawing Ovals and Circles

Java has methods to draw outlined and filled ovals. These methods are called drawOval() and fillOval() respectively. These two methods are:

public void drawOval(int left, int top, int width, int height)
public void fillOval(int left, int top, int width, int height)

Instead of dimensions of the oval itself, the dimensions of the smallest rectangle, which can enclose the oval, are specified. The oval is drawn as large as it can be to touch the rectangle's edges at their centers. *Figure 3* may help you to understand properly.

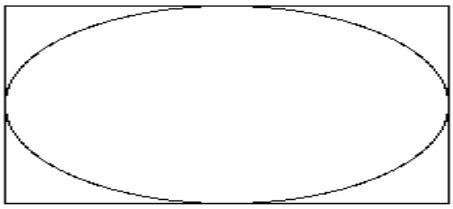


Figure 3: An Oval

The arguments to drawOval() are the same as the arguments to drawRect(). The first int is the left hand side of the enclosing rectangle, the second is the top of the enclosing rectangle, the third is the width and the fourth is the height. There is no special method to draw a circle. Just draw an oval inside a square.

2.2.3.4 Drawing Polygons and Polylines

You have already seen that in Java rectangles are defined by the position of their upper left hand corner, their height, and their width. However it is implicitly assumed that there is in fact an upper left hand corner. What's been assumed so far is that the sides of the rectangle are parallel to the coordinate axes. You can't yet handle a rectangle that has been rotated at an arbitrary angle.

There are some other things you can't handle either, triangles, stars, rhombuses, kites, octagons and more. To take care of this broad class of shapes Java has a **Polygon** class.

Polygons are defined by their corners. No assumptions are made about them except that they lie in a 2-D plane. The basic constructor for the Polygon class is

public Polygon(int[] xpoints, int[] ypoints, int npoints)

xpoints is an array that contains the x coordinates of the polygon. **ypoints** is an array that contains the y coordinates. Both should have the length **npoints**. Thus to construct a right triangle with the right angle on the origin you would type

```
int[] xpoints = {0, 3, 0};
int[] ypoints = {0, 0, 4};
Polygon myTriangle = new Polygon(xpoints, ypoints, 3);
```

To draw the polygon you can use java.awt.Graphics's drawPolygon(Polygon p) method within your paint() method like this:
g.drawPolygon(myTriangle);
You can pass the arrays and number of points directly to the drawPolygon() method if you prefer:
g.drawPolygon(xpoints, ypoints, xpoints.length);
There's also an overloaded fillPolygon() method, you can call this method like
g.fillPolygon(myTriangle);
g.fillPolygon(xpoints, ypoints, xpoints.length());

To simplify user interaction and make data entry easier, Java provides different controls and interfaces. Now let us see some of the basic user interface components of Java.

2.3 USER INTERFACE COMPONENTS

Java provides many controls. Controls are components, such as buttons, labels and text boxes that can be added to containers like frames, panels and applets. The Java.awt package provides an integrated set of classes to manage user interface components.

Components are placed on the user interface by adding them to a container. A container itself is a component. The easiest way to demonstrate interface design is by using the container you have been working with, i.e., the Applet class. The simplest form of Java AWT component is the basic *User Interface Component*. You can create and add these to your applet without any need to know anything about creating containers or panels. In fact, your applet, even before you start painting and drawing and handling events, is an AWT container. Because an applet is a container, you can put any of AWT components, and (or) other containers, in it.

In the next section of this Unit, you will learn about the basic User Interface components (controls) like labels, buttons, check boxes, choice menus, and text fields. You can see in Table 2a, a list of all the Controls in Java AWT and their respective functions. In Table 2b list of classes for these control are given.

2.4 BUILDING USER INTERFACE WITH AWT

In order to add a control to a container, you need to perform the following two steps:

1. Create an object of the control by passing the required arguments to the constructor.
2. Add the component (control) to the container.

Table 2a: Controls in Java

CONTROLS	FUNCTIONS
Textbox	Accepts single line alphanumeric entry.
TextArea	Accepts multiple line alphanumeric entry.
Push button	Triggers a sequence of actions.
Label	Displays Text.
Check box	Accepts data that has a yes/no value. More than one checkbox can be selected.
Radio button	Similar to check box except that it allows the user to select a single option from a group.
Combo box	Displays a drop-down list for single item selection. It allows new value to be entered.
List box	Similar to combo box except that it allows a user to select single or multiple items. New values cannot be entered.

Table 2b: Classes for Controls

CONTROLS	CLASS
Textbox	TextField
TextArea	TextArea
Push button	Button
Check box	CheckBox
Radio button	CheckboxGroup with CheckBox
Combo box	Choice
List box	List

The Button

Let us first start with one of the simplest of UI components: the button. Buttons are used to trigger events in a GUI environment (we have discussed Event Handling in detail in the previous Unit: Unit 1 Block 4 of this course). The Button class is used to create buttons. When you add components to the container, you don't specify a set of coordinates that indicate where the components are to be placed. A layout manager in effect for the container handles the arrangement of components. The default layout for a container is flow layout (for an applet also default layout will be flow layout). More about different layouts you will learn in later section of this unit. Now let us write a simple code to test our button class.

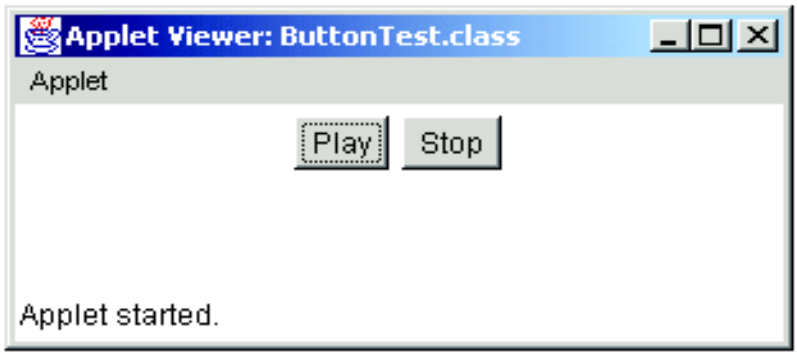
To create a button use, one of the following constructors:

Button() creates a button with no text label.
Button(String) creates a button with the given string as label.

Example Program:

```
/*
<Applet code= "ButtonTest.class"
Width = 500
Height = 100>
</applet>
*/
import java.awt.*;
import java.applet.Applet;
public class ButtonTest extends Applet
{
Button b1 = new Button ("Play");
Button b2 = new Button ("Stop");
public void init(){
add(b1);
add(b2);
}
}
```

Output:



As you can see this program will place two buttons on the Applet with the caption Play and Stop.

The Label

Labels are created using the Label class. Labels are basically used to identify the purpose of other components on a given interface; they cannot be edited directly by the user. Using a label is much easier than using a drawString( ) method because

labels are drawn automatically and don't have to be handled explicitly in the paint( ) method. Labels can be laid out according to the layout manager, instead of using [x, y] coordinates, as in drawString( ).

To create a Label, use any one of the following constructors:
Label( ): creates a label with its string aligned to the left.
Label(String): creates a label initialized with the given string, and aligned left.
Label(String, int): creates a label with specified text and alignment indicated by any one of the three int arguments. Label.Right, Label.Left and Label.Center.
getText( ) method is used to indicate the current label's text setText( ) method to change the label's and text. setFont( ) method is used to change the label's font.

The Checkbox

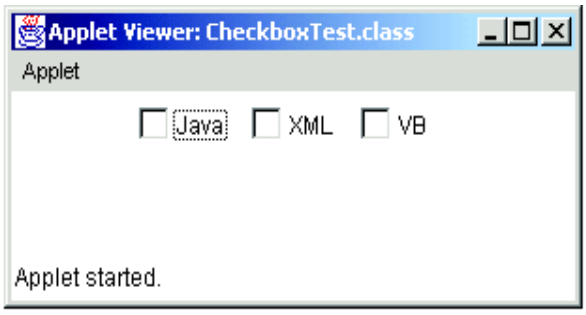
Check Boxes are labeled or unlabeled boxes that can be either "Checked off" or "Empty". Typically, they are used to select or deselect an option in a program.

Sometimes Check are *nonexclusive*, which means that if you have six check boxes in a container, all the six can either be checked or unchecked at the same time. This component can be organized into Check Box Group, which is sometimes called radio buttons. Both kinds of check boxes are created using the Checkbox class. To create a nonexclusive check box you can use one of the following constructors:
Checkbox() creates an unlabeled checkbox that is not checked.
Checkbox(String) creates an unchecked checkbox with the given label as its string.

After you create a checkbox object, you can use the setState(boolean) method with a true value as argument for checked checkboxes, and false to get unchecked. Three checkboxes are created in the example given below, which is an applet to enable you to select up to three courses at a time.

```
import java.awt.*;
public class CheckboxTest extends java.applet.Applet
{
Checkbox c1 = new Checkbox ("Java");
Checkbox c2 = new Checkbox ("XML");
Checkbox c3 = new Checkbox ("VB");
public void init(){
add(c1);
add(c2);
add(c3);
 }
}
```

Output:



The Checkbox group

CheckboxGroup is also called like a radio button or exclusive check boxes. To organize several Checkboxes into a group so that only one can be selected at a time,

you can create CheckboxGroup object as follows:
CheckboxGroup radio = new CheckboxGroup ();

The CheckboxGroup keeps track of all the check boxes in its group. We have to use this object as an extra argument to the *Checkbox* constructor.

Checkbox (String, CheckboxGroup, Boolean) creates a checkbox labeled with the given string that belongs to the CheckboxGroup indicated in the second argument. The last argument equals true if box is checked and false otherwise.

The set Current (checkbox) method can be used to make the set of currently selected check boxes in the group. There is also a get Current () method, which returns the currently selected checkbox.

The Choice List

Choice List is created from the Choice class. List has components that enable a single item to be picked from a pull-down list. We encounter this control very often on the web when filling out forms.
The first step in creating a Choice
You can create a choice object to hold the list, as shown below:

```
Choice cgender = new Choice();
```

Items are added to the Choice List by using addItem(String) method the object. The following code adds two items to the gender choice list.

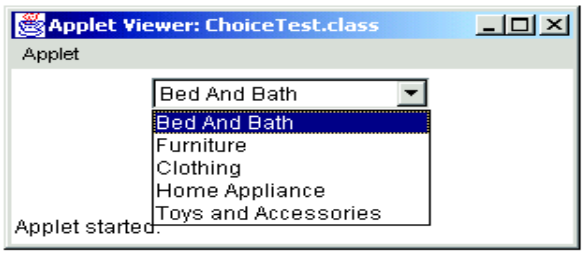
```
cgender.addItem("Female");
cgender.addItem("Male");
```

After you add the Choice List it is added to the container like any other component using the add() method.

The following example shows an Applet that contains a list of shopping items in the store.

```
import java.awt.*;
Public class ChoiceTest extends java.applet.Applet
{
Choice shoplist = new Choice();

Public void init(){
shoplist.addItem("Bed And Bath");
shoplist.addItem("Furniture");
shoplist.addItem("Clothing");
shoplist.addItem("Home Appliance");
shoplist.addItem("Toys and Accessories");
add(shoplist);
}
```

Output:



The choice list class has several methods, which are given in *Table 3*.

Table 3: Choice List Class Methods

Method	Action
getItem()	Returns the string item at the given position (items inside a choice begin at 0, just like arrays)
countItems()	Returns the number of items in the menu
getSelectedIndex()	Returns the index position of the item that's selected
getSelectedItem()	Returns the currently selected item as a string
select(int)	Selects the item at the given position
select(String)	Selects the item with the given string

The Text Field

To accept textual data from user, AWT provided two classes, *TextField and TextArea*. The TextField handles a single line of text and does not have scrollbars, whereas the TextArea class handles multiple lines of text. Both the classes are derived from the TextComponent class. Hence they share many common methods. TextFields provide an area where you can enter and edit a single line of text. To create a text field, use one of the following constructors:

TextField(): creates an empty TextField with no specified width.
TextField(int): creates an empty text field with enough width to display the specified number of characters (this has been depreciated in Java2).
TextField(String): creates a text field initialized with the given string.
TextField(String, int) :creates a text field with specified text and specified width.

For example, the following line creates a text field 25 characters wide with the string "Brewing Java" as its initial contents:
TextField txtfld = new TextField ("Brewing Java", 25); add(txtfld);

TextField, can use methods like:
setText(): Used to set the text in text field.
getText(): Used to get the text currently contained by text field.
setEditable(): Used to provide control whether the content of text field may be modified by user or not.
isEditable(): It return **true** if the text in text filed may be changed and **false** otherwise.

Text Area

The TextArea is an editable text field that can handle more than one line of input. Text areas have horizontal and vertical scrollbars to scroll through the text. Adding a text area to a container is similar to adding a text field. To create a text area you can use one of the following constructors:

TextArea(): creates an empty text area with unspecified width and height.
TextArea(int, int): creates an empty text area with indicated number of lines and specified width in characters.
TextArea(String): creates a text area initialized with the given string.
TextField(String, int, int): creates a text area containing the indicated text and specified number of lines and width in the characters.
The TextArea, similar to TextField, can use methods like setText(), getText(), setEditable(), and isEditable().
In addition, there are two more methods like these. The first is the insertText(String, int) method, used to insert indicated strings at the character index specified by the

second argument. The next one is replaceText(String, int, int) method, used to replace text between given integer position specified by second and third argument with the indicated string.

The basic idea behind the AWT is that a graphical Java program is a set of nested components, starting from the outermost window all the way down to the smallest UI component. Components can include things you can actually see on the screen, such as windows, menu bars, buttons, and text fields, and they can also include containers, which in turn can contain other components.

Hope you have got a clear picture of Java AWT and its some basic UI components, In the next section of the Unit we will deal with more advance user interface components.

2.5 SWING - BASED GUI

You must be thinking that when you can make GUI interface with AWT package then what is the purpose of learning Swing-based GUI? Actually Swing has *lightweight* components and does not write itself to the screen, but redirects it to the component it builds on. On the other hand AWT are heavyweight and have their own view port, which sends the output to the screen. Heavyweight components also have their own z-ordering (look and feel) dependent on the machine on which the program is running. This is the reason why you can't combine AWT and Swing in the same container. If you do, AWT will always be drawn on top of the Swing components.

Another difference is that Swing is pure Java, and therefore platform independent. Swing looks identically on all platforms, while AWT looks different on different platforms.

See, basically Swing provides a rich set of GUI components; features include model-UI separation and a plug able look and feel. Actually you can make your GUI also with AWT but with Swing you can make it more user-friendly and interactive. Swing components make programs efficient.

Swing GUI components are packaged into Package javax.swing. In the Java class hierarchy there is a class
Class Component which contains method paint for drawing Component onscreen
Class Container which is a collection of related components and contains method add for adding components and Class JComponent which has
Pluggable look and feel for customizing look and feel
Shortcut keys (*mnemonics*)
Common event-handling capabilities

The Hierarchy is as follows:

Object-----→ Component--→Container---→JComponent

In Swings we have classes prefixed with the letter 'J' like
JLabel -> Displays single line of read only text

JTextField -> Displays or accepts input in a single line

JTextArea -> Displays or accepts input in multiple lines

JCheckBox ->Gives choices for multiple options
JButton -> Accepts command and does the action

JList - > Gives multiple choices and display for selection
JRadioButton - > Gives choices for multiple option, but can select one at a time.

☞ Check Your Progress 3

1) Write a program which draws a line, a rectangle, and an oval on the applet.
……

……

2) Write a program that draws a color-filled line, a color-filled rectangle, and a color filled oval on the applet.
……

……

3) Write a program to add various checkboxes under the CheckboxGroup
……

……

4) Write a program in which the Applet displays a text area that is filled with a string, when the programs begin running.
……

……

5) Describe the features of the Swing components that subclass J Component. What are the difference between Swing and AWT?
……

……

Now we will discuss about different layouts to represent components in a container.

2.6 LAYOUTS AND LAYOUT MANAGER

When you add a component to an applet or a container, the container uses its *layout manager* to decide where to put the component. Different LayoutManager classes use different rules to place components.

java.awt.LayoutManager is an interface. Five classes in the java packages implement it:

- FlowLayout
- BorderLayout
- CardLayout
- GridLayout
- GridBagLayout
- plus javax.swing.BoxLayout

FlowLayout

A FlowLayout arranges widgets from left to right until there's no more space left. Then it begins a row lower and moves from left to right again. Each component in a FlowLayout gets as much space as it needs and no more.
This is the *default LayoutManager* for applets and panels. FlowLayout is the default layout for java.awt.Panel of which java.applet.Applet is a subclasses.
Therefore you don't need to do anything special to create a FlowLayout in an applet. However you do need to use the following constructors if you want to use a FlowLayout in a Window.LayoutManagers have constructors like any other class.

The constructor for a FlowLayout is
public FlowLayout()
Thus to create a new FlowLayout object you write
FlowLayout fl;
fl = new FlowLayout();
As usual this can be shortened to
FlowLayout fl = new FlowLayout();

You tell an applet to use a particular LayoutManager instance by passing the object to the applet's setLayout() method like this: this.setLayout(fl);

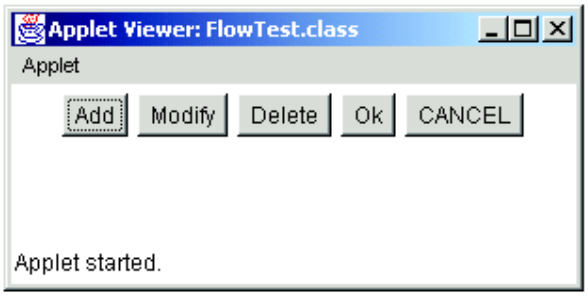
Most of the time setLayout() is called in the init() method. You normally just create the LayoutManager right inside the call to setLayout() like this

this.setLayout(new FlowLayout());

For example the following applet uses a FlowLayout to position a series of buttons that mimic the buttons on a tape deck.

```
import java.applet.*;
import java.awt.*;
public class FlowTest extends Applet {
  public void init() {
    this.setLayout(new FlowLayout());
    this.add( new Button("Add"));
    this.add( new Button("Modify"));
    this.add( new Button("Delete"));
    this.add( new Button("Ok"));
    this.add( new Button("CANCEL"));
  }
}
```

Output:



You can change the alignment of a FlowLayout in the constructor. Components are normally centered in an applet. You can make them left or right justified. To do this just passes one of the defined constants FlowLayout.LEFT, FlowLayout.RIGHT or FlowLayout.CENTER to the constructor, e.g.
this.setLayout(new FlowLayout(FlowLayout.LEFT));

Another constructor allows you spacing option in FlowLayout:
public FlowLayout(int alignment, int horizontalSpace, int verticalSpace);

For instance to set up a FlowLayout with a ten pixel horizontal gap and a twenty pixel vertical gap, aligned with the left edge of the panel, you would use the constructor
FlowLayout fl = new FlowLayout(FlowLayout.LEFT, 20, 10);
Buttons arranged according to a center-aligned FlowLayout with a 20 pixel horizontal spacing and a 10 pixel vertical spacing

BorderLayout

A BorderLayout organizes an applet into North, South, East, West and Center sections. North, South, East and West are the rectangular edges of the applet. They're continually resized to fit the sizes of the widgets included in them. Center is whatever is left over in the middle.

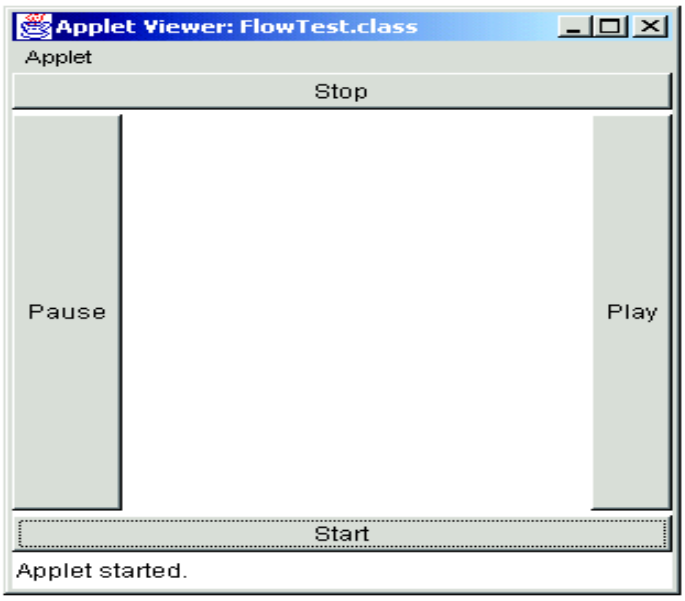
A BorderLayout places objects in the North, South, East, West and center of an applet. You create a new BorderLayout object much like a FlowLayout object, in the init() method call to setLayout like this:
this.setLayout(new BorderLayout());

There's no centering, left alignment, or right alignment in a BorderLayout. However, you can add horizontal and vertical gaps between the areas. Here is how you would add a two pixel horizontal gap and a three pixel vertical gap to a BorderLayout:
this.setLayout(new BorderLayout(2, 3));

To add components to a BorderLayout include the name of the section you wish to add them to do like done in the program given below.

this.add("South", new Button("Start"));
import java.applet.*;
import java.awt.*;
public class BorderLayouttest extends Applet
{
public void init() {
this.setLayout(new BorderLayout(2, 3));
this.add("South", new Button("Start"));
this.add("North", new Button("Stop"));
this.add("East", new Button("Play"));
this.add("West", new Button("Pause"));
}

}



Card Layout

A CardLayout breaks the applet into a deck of cards, each of which has its own Layout Manager. Only one card appears on the screen at a time. The user flips between cards, each of which shows a different set of components. The common analogy is with HyperCard on the Mac and Tool book on Windows. In Java this might

be used for a series of data input screens, where more input is needed than will comfortably fit on a single screen.

Grid Layout

A GridLayout divides an applet into a specified number of rows and columns, which form a grid of cells, *each equally sized and spaced.* It is important to note that each is equally sized and spaced as there is another similar named Layout known as GridBagLayout .As Components are added to the layout they are placed in the cells, starting at the upper left hand corner and moving to the right and down the page. Each component is sized to fit into its cell. This tends to squeeze and stretch components unnecessarily.

You will find the GridLayout is great for arranging Panels. A GridLayout specifies the number of rows and columns into which components will be placed. The applet is broken up into a table of equal sized cells.

GridLayout is useful when you want to place a number of similarly sized objects. It is great for putting together lists of checkboxes and radio buttons as you did in the Ingredients applet. GridLayout looks like *Figure 3*.

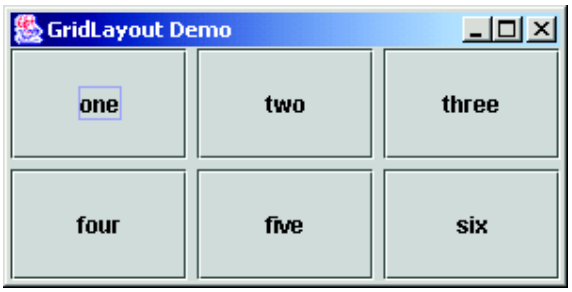


Figure 3: GridLayout Demo

Grid Bag Layout

GridBagLayout is the most precise of the five AWT Layout Managers. It is similar to the GridLayout, but components do not need to be of the same size. Each component can occupy one or more cells of the layout. Furthermore, components are not necessarily placed in the cells beginning at the upper left-hand corner and moving to the right and down.

In simple applets with just a few components you often need only one layout manager. In more complicated applets, however, you will often split your applet into panels, lay out the panels according to a layout manager, and give each panel its own layout manager that arranges the components inside it.
The GridBagLayout constructor is trivial, GridBagLayout() with no arguments.

GridBagLayout gbl = new GridBagLayout();

Unlike the GridLayout() constructor, this does not say how many rows or columns there will be. The cells your program refers to determine this. If you put a component in row 8 and column 2, then Java will make sure there are at least nine rows and three columns. (Rows and columns start counting at zero.) If you later put a component in row 10 and column 4, Java will add the necessary extra rows and columns. You may have a picture in your mind of the finished grid, but Java does not need to know this when you create a GridBagLayout.

Unlike most other LayoutManagers you should not create a GridBagLayout inside a cell to setLayout(). You will need access to the GridBagLayout object later in the applet when you use a GridBagConstraints.

A **GridBagConstraints** object specifies the location and area of the component's display area within the container (normally the applet panel) and how the component is laid out inside its display area. The GridBagConstraints, in conjunction with the component's minimum size and the preferred size of the component's container, determines where the display area is placed within the applet.

The GridBagConstraints() constructor is trivial

GridBagConstraints gbc = new GridBagConstraints();

Your interaction with a GridBagConstraints object takes place through its eleven fields and fifteen mnemonic constants.

gridx and gridy

The gridx and gridy fields specify the x and y coordinates of the cell at the upper left of the Component's display area. The upper-left-most cell has coordinates (0, 0). The mnemonic constant GridBagConstraints.RELATIVE specifies that the Component is placed immediately to the right of (gridx) or immediately below (gridy) the previous Component added to this container.

Gridwidth and Gridheight

The gridwidth and gridheight fields specify the number of cells in a row (gridwidth) or column (gridheight) in the Component's display area. The mnemonic constant GridBagConstraints.REMAINDER specifies that the Component should use all remaining cells in its row (for gridwidth) or column (for gridheight). The mnemonic constant GridBagConstraints.RELATIVE specifies that the Component should fill all but the last cell in its row (gridwidth) or column (gridheight).

Fill

The GridBagConstraints fill field determines whether and how a component is resized if the component's display area is larger than the component itself. The mnemonic constants you use to set this variable are
GridBagConstraints.NONE :Don't resize the component
GridBagConstraints.HORIZONTAL: Make the component wide enough to fill the display area, but don't change its height.
GridBagConstraints.VERTICAL: Make the component tall enough to fill its display area, but don't change its width.
GridBagConstraints.BOTH: Resize the component enough to completely fill its display area both vertically and horizontally.

Ipadx and Ipady

Each component has a minimum width and a minimum height, smaller than which it will not be. If the component's minimum size is smaller than the component's display area, then only part of the component will be shown.

The ipadx and ipady fields let you increase this minimum size by padding the edges of the component with extra pixels. For instance setting ipadx to two will guarantee that the component is at least four pixels wider than its normal minimum. (ipadx adds two pixels to each side.)

Insets

The insets field is an instance of the java.awt.Insets class. It specifies the padding between the component and the edges of its display area.

Anchor

When a component is smaller than its display area, the anchor field specifies where to place it in the grid cell. The mnemonic constants you use for this purpose are similar to those used in a BorderLayout but a little more specific.

They are

GridBagConstraints.CENTER
GridBagConstraints.NORTH
GridBagConstraints.NORTHEAST
GridBagConstraints.EAST
GridBagConstraints.SOUTHEAST
GridBagConstraints.SOUTH
GridBagConstraints.SOUTHWEST
GridBagConstraints.WEST
GridBagConstraints.NORTHWEST
The default is GridBagConstraints.CENTER.

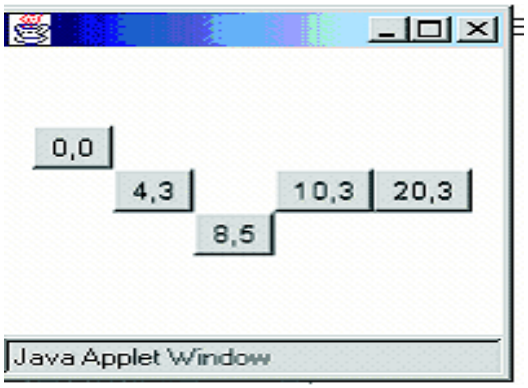
weightx and weighty

The weightx and weighty fields determine how the cells are distributed in the container when the total size of the cells is less than the size of the container. With weights of zero (the default) the cells all have the minimum size they need, and everything clumps together in the center. All the extra space is pushed to the edges of the container. It doesn't matter where they go and the default of center is fine.
See the program given bellow for visualizing GridbagLayout.

```
import java.awt.*;
public class GridbagLayouttest extends Frame
{
Button b1,b2,b3,b4,b5;
GridBagLayout gbl=new GridBagLayout();
GridBagConstraints gbc=new GridBagConstraints();
public GridbagLayouttest()
{
setLayout(gbl);
gbc.gridx=0;
gbc.gridy=0;
gbl.setConstraints(b1=new Button("0,0"),gbc);
gbc.gridx=4;   //4th column
gbc.gridy=3;   //3rd row
gbl.setConstraints(b2=new Button("4,3"),gbc);
gbc.gridx=8;   //8th column
gbc.gridy=5;   //5rd row
gbl.setConstraints(b3=new Button("8,5"),gbc);
gbc.gridx=10;   //10th column
gbc.gridy=3;   //3rd row
gbl.setConstraints(b4=new Button("10,3"),gbc);
gbc.gridx=20;   //20th column
gbc.gridy=3;   //3rd row
gbl.setConstraints(b5=new Button("20,3"),gbc);
add(b1);
```

```
add(b2);
add(b3);
add(b4);
add(b5);
setSize(200,200);
setVisible(true);
}
public static void main(String a[])
{
GridbagLayouttest gb=  new GridbagLayouttest();
}
}
```

Output:



Now we will discuss about different containers.

2.7 CONTAINER

You must be thinking that container will be a thing that contains something, like a bowl. Then you are right!! Actually container object is derived from the java.awt.Container class and is one of (or inherited from) three primary classes: java.awt.Window, java.awt.Panel, java.awt.ScrollPane.

The Window class represents a standalone window (either an application window in the form of a java.awt.Frame, or a dialog box in the form of a java.awt.Dialog). The java.awt.Panel class is not a standalone window by itself; instead, it acts as a background container for all other components on a form. For instance, the java.awt.Applet class is a direct descendant of java.awt.Panel.

The three steps common for all Java GUI applications are:

1. Creation of a container
2. Layout of GUI components.
3. Handling of events.

The Container class contains the setLayout() method so that you can set the default LayoutManager to be used by your GUI. To actually add components to the container, you can use the container's add() method:

```
Panel p = new java.awt.Panel();
Button b = new java.awt.Button("OK");
p.add(b);
```

A JPanel is a Container, which means that it can contain other components. GUI design in Java relies on a layered approach where each layer uses an appropriate layout manager.

FlowLayout is the default for JPanel objects. To use a different manager use either of the following:

```
JPanel pane2 = new JPanel()          // make the panel first
pane2.setLayout(new BorderLayout()); // then reset its manager
JPanel pane3 = new JPanel(new BorderLayout()); // all in one!
```

☞ Check Your Progress 4

1) Why do you think Layout Manager is important?
……
……
……
……

2) How does repaint() method work with Applet?
……
……
……

3) Each type of container comes with a default layout manager. Default for a Frame,Window or Dialog is _____

4) Give examples of stretchable components and non-stretchable components
……
……
……
……

5) The Listener interfaces inherit directly from which package.
……
……
……
……
……

6) How many Listeners are there for trapping mouse movements.
……
……
……
……
……

2.8 SUMMARY

This unit is designed to know about the graphical interfaces in Java. In this unit you become familiar with AWT and SWING packages, which are used to add components in the container or a kind of tray. Swings are lightweight components and more flexible as compared to AWT. You learned to beautify your text by using Font class and Color class. For Geometric figures various in built classes of Java like for drawing line, circle, polygons etc are used. You learned to place the components in various

layouts. Java provides various layouts some of which are: FlowLayout, GridBagLayout, GridLayout, CardLayout, BorderLayout.

2.9 SOLUTONS/ANSWERS

Check Your Progress 1

1) Java defines two constructors for the Color Class of which one constructor takes three integer arguments and another takes three float arguments.

public Color (int r, int g, int b)

Creates a color based on red, green and blue components expressed as integers from 0 to 255. Here, the r, g, b means red, green and blue contents respectively.

public Color(float r, float g, float b) Creates a color based on red, green and blue components expressed as floating-point values from 0.0 to 1.0. Here, the r, g, b means red, green and blue contents respectively.

2)

```
import java.awt.*;
import java.applet.Applet;
public class ColorText extends Applet
{
 public void paint(Graphics g)
{
     g.setColor(Color.red);
    g.drawString("My Red color String",40,50);

  }
}
```

Output:

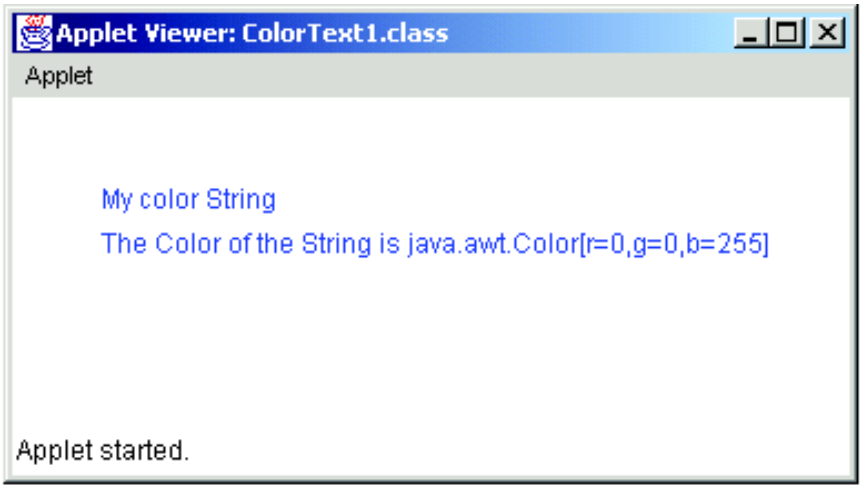
3) If you want to retrieve the Color of a text or a String you can use the method public int getColor().
It returns a value between 0 and 255 representing the color content.

The Program given bellow is written for getting RGB components of the color
import java.awt. *

```
import java.applet.Applet;
public class ColorText extends Applet
{
 public void paint(Graphics g)
{
int color;
g.drawString("My Pink color String",40,50);
```

```
color = getColor(g);
g.drawString("The Color of the String is "+color , 40,35);
 }
}
```

Output:

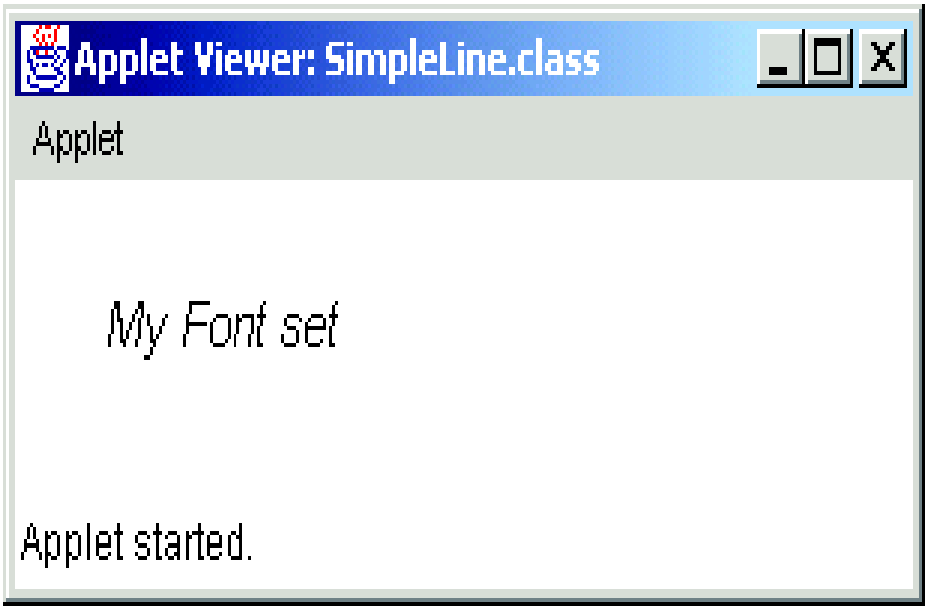


Check Your Progress 2

1)

```
import java.applet.*;
import java.awt.*;
public class SimpleLine extends Applet
{
 public void paint(Graphics g)
 {
  g.drawString("My Font set" , 30, 40);
  g.setFont("Arial",Font.ITALIC,15);
  }
}
```

Output:

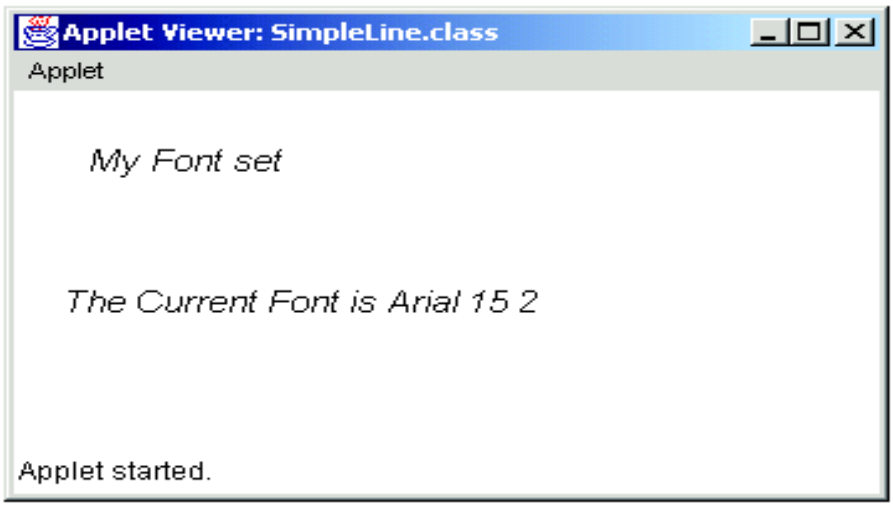


2) To retrieve the font of the string the method **getFont()** is used.

Program for font retrieval:

```
import java.applet.*;
import java.awt.*;
public class SimpleLine extends Applet
{
public void paint(Graphics g)
{
g.drawString("My Font set" , 30, 40);
g.setFont("Arial",Font.ITALIC,15);
g.drawString("current Font is " + g.getFont(),40,50);
g.drawString( g.getFont().getName() + " " + g.getFont().getSize()+" "
+g.getFont().getStyle(), 20, 110 );
 }
}
```

Output:



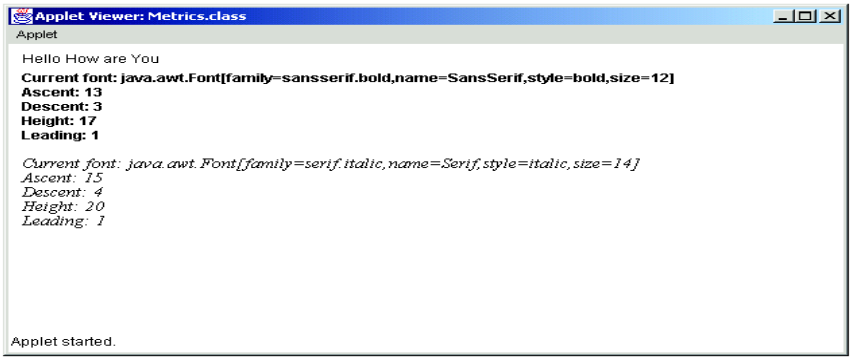
Note: The getStyle() methodreturns the integer value e.g above value 2 represent ITALIC.

3)

```
import java.awt.*;
import java.applet.Applet;
public class Metrics extends Applet {
public void paint(Graphics g)
{
        g.drawString("Hello How are You",50,60);
        g.setFont( new Font( "SansSerif", Font.BOLD, 12 ) );
        FontMetrics metrics = g.getFontMetrics();
        g.drawString( "Current font: " + g.getFont(), 10, 40 );
        g.drawString( "Ascent: " + metrics.getAscent(), 10, 55 );
        g.drawString( "Descent: " + metrics.getDescent(), 10, 70 );
        g.drawString( "Height: " + metrics.getHeight(), 10, 85 );
        g.drawString( "Leading: " + metrics.getLeading(), 10, 100 );
        Font font = new Font( "Serif", Font.ITALIC, 14 );
        metrics = g.getFontMetrics( font );
        g.setFont( font );
        g.drawString( "Current font: " + font, 10, 130 );
        g.drawString( "Ascent: " + metrics.getAscent(), 10, 145 );
        g.drawString( "Descent: " + metrics.getDescent(), 10, 160 );
```

```
        g.drawString( "Height: " + metrics.getHeight(), 10, 175 );
        g.drawString( "Leading: " + metrics.getLeading(), 10, 190 );
      } // end method paint
}
```

Output:

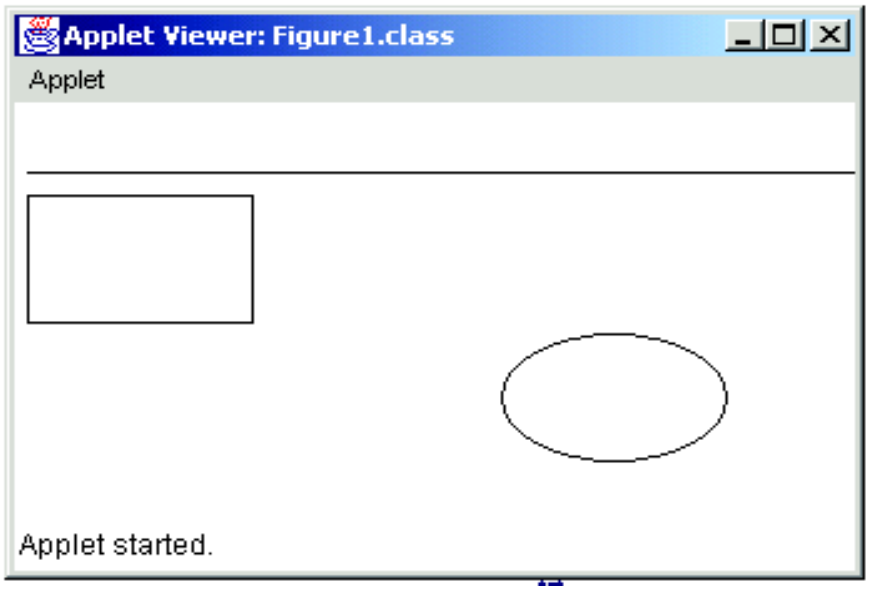


Check Your Progress 3

1)

```
// Drawing lines, rectangles and ovals.
 import java.awt.*;
 import java.applet.Applet;
     public class LinesRectsOvals extends Applet {
      // display various lines, rectangles and ovals
   public void paint( Graphics g )
   {
        g.drawLine( 5, 30, 350, 30 );
        g.drawRect( 5, 40, 90, 55 );
        g.drawOval( 195, 100, 90, 55 );
      } // end method paint
}
```

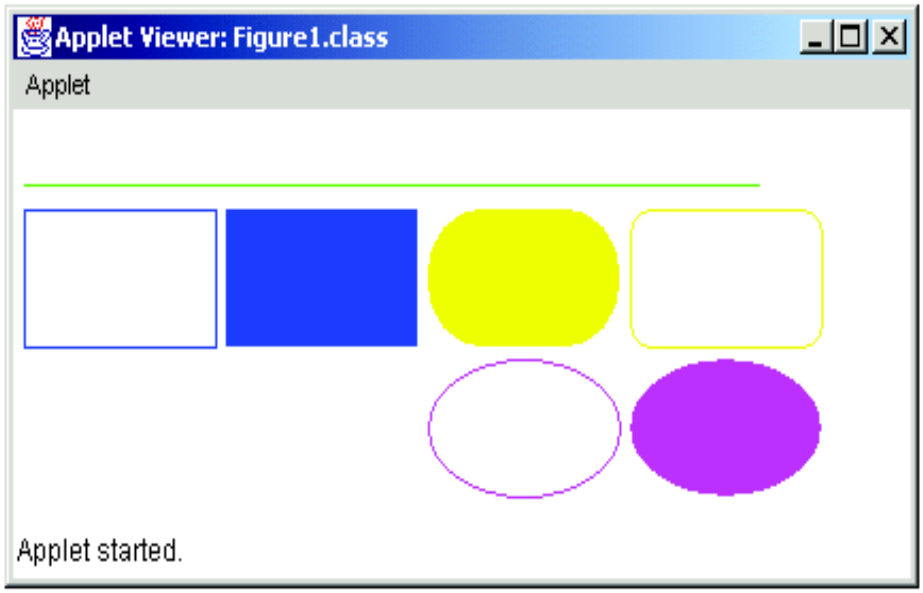
Output:



2)

```
import java.awt.*;
import javax.swing.*;
          public class LinesRectsOvals extends JFrame
          {
          public void paint( Graphics g )
          {
         g.setColor( Color.GREEN );
         g.drawLine( 5, 30, 350, 30 );
         g.setColor( Color.BLUE );
         g.drawRect( 5, 40, 90, 55 );
         g.fillRect( 100, 40, 90, 55 );
//below two lines will draw a filled rounded rectangle with color yellow
         g.setColor( Color.YELLOW );
         g.fillRoundRect( 195, 40, 90, 55, 50, 50 );
//below two lines will draw a rounded figure of rectangle with color Pink
         g.drawRoundRect( 290, 40, 90, 55, 20, 20 );
         g.setColor( Color.PINK);
//below lines will draw an Oval figure with color Magenta
         g.setColor(Color.MAGENTA);
         g.drawOval( 195, 100, 90, 55 );
         g.fillOval( 290, 100, 90, 55 );
}
}
```

Output:



3)

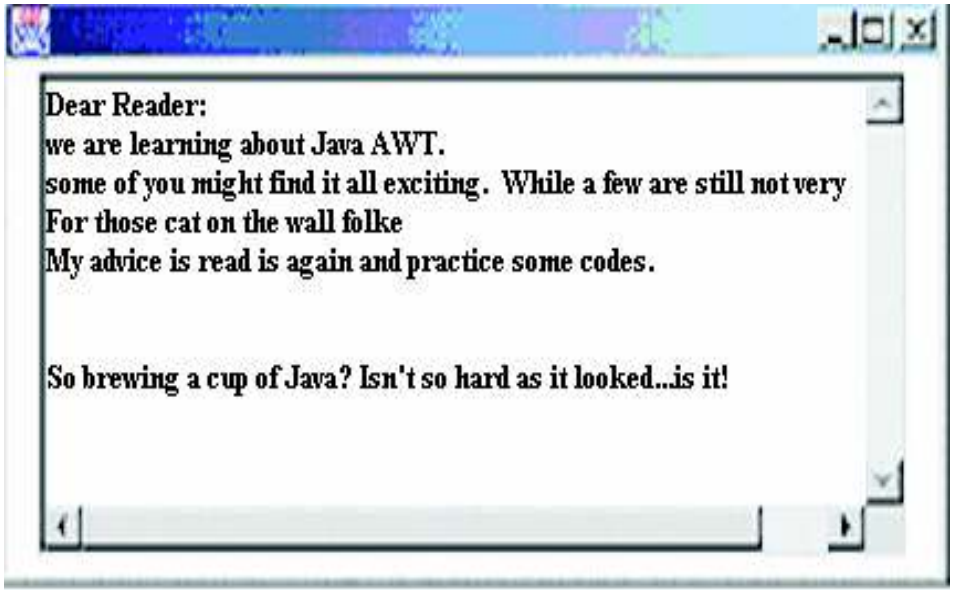
```
import java.awt.*;
class ChkGroup extends java.applet.Applet
{
CheckboxGroup cbgr = new CheckboxGroup();
Panel panel=new Panel();
Frame fm=new Frame();
Checkbox c1 = new Checkbox ("America Online");
Checkbox c2 = new Checkbox ("MSN");
Checkbox c3 = new Checkbox ("NetZero", cbgr, false);
```

```
Checkbox c4 = new Checkbox ("EarthLink", cbgr, false);
Checkbox c5 = new Checkbox ("Bellsouth DSL", cbgr, false);
public void init()
{
fm.setVisible(true);
fm.setSize(300,400);
fm.add(panel);
panel.add(c1);
panel.add(c2);
panel.add(c3);
panel.add(c4);
panel.add(c5);
}
public static void main(String args[])
{
        ChkGroup ch=new ChkGroup();
        ch.init();
}
}
```

In the above example you will notice that the control which is not in the group has become a checkbox and the control which is in the group has become a radio button because checkbox allows you multiple selection but a radio button allows you a single selection at a time.

4)

```
import java.awt.*;
class TextFieldTest extends java.applet.Applet
{
Frame fm=new Frame();
Panel panel=new Panel();
String letter = "Dear Readers: \n" +
"We are learning about Java AWT. \n" +
"Some of you might find it all exciting, while a few are still not very sure \n" +
"For those cat on the wall folks \n" +
"My advice is read it again and practice some codes. \n \n" +
"So brewing a cup of Java? Isn't so hard as it looked…is it! " ;
TextArea ltArea;
public void init(){
ltArea = new TextArea(letter, 10, 50);
fm.setVisible(true);
fm.setSize(300,400);
fm.add(panel);
panel.add(ltArea);

}
public static void main(String args[])
{
  TextFieldTest tt=new TextFieldTest();
        tt.init();
}
}
```



Basically you will notice that along with the textarea control you will see the scrollbars.

5) Swing components that subclass JComponent has many features, including:
A pluggable look and feel that can be used to customize the look and feel when the program executes on different platforms.

We can have Shortcut keys (called) mnemonics) for direct access to GUI components through the keyboard. It has very common event handling capabilities for cases where several GUI components initiate the same actions in the program. It gives the brief description of a GUI component's means (tool tips) that are displayed when the mouse cursor is positioned over the component for a short time. It has a support for technologies such as Braille screen for blind people. It has support for user interface localization-customizing the user interface for display in different languages and cultural conventions.

Check Your Progress 4

1) A LayoutManager rearranges the components in the container based on their size relative to the size of the container.

Consider the window that just popped up. It has got five buttons of varying sizes. Resize the window and watch how the buttons move. In particular try making it just wide enough so that all the buttons fit on one line. Then try making it narrow and tall so that there is only one button on line. See if you can manage to cover up some of the buttons. Then uncover them. Note that whatever you try to do, the order of the buttons is maintained in a logical way. Button 1 is always before button 2, which is always before button 3 and so on.

It is harder to show, but imagine if the components changed sizes, as they might if you viewed this page in different browsers or on different platforms with different fonts.

The layout manager handles all these different cases for you to the greatest extent possible. If you had used absolute positioning and the window were smaller than expected or the components larger than you expected, some components would likely be truncated or completely hidden. In essence a layout manager defers decisions about positioning until runtime.

2) The repaint() method will cause AWT to invoke a component's update() method. AWT passes a Graphics object to update() –the same one that it passes to paint(). So, repaint() calls update() which calls paint().

3) BorderLayout

4) Stretchable: Button, Label and TextField
Non-Stretchable: CheckBox

5) java.awt.event

6) MouseListener and MouseMotionListener

UNIT 3 NETWORKING FEATURES

Structure **Page Nos.**

3.0 INTRODUCTION

Client/server applications are need of the time. It is challenging and interesting to develop Client/server applications. Java provides easier way of doing it than other programming such as C. Socket programming in Java is seamless. The java.net package provides a powerful and flexible infrastructure for network programming. Sun.* packages have some classes for networking. In this unit you will learn the java.net package, using socket based communications which enable applications to view networking operations as I/O operation. A program can read from a socket or write to a socket as simply as reading a file or writing to a file.

With datagram and stream sockets you will be developing connection less and connection oriented applications respectively. Going through various classes and interfaces in java. net, will be useful in learning, how to develop networking applications easily. In this unit you will also learn use of stream sockets and the TCP protocol, which is the most desirable for the majority of java programmers for developing networking applications.

3.1 OBJECTIVES

After going though this Unit, you will be able to:

- define socket and elements of java networking;
- describe the stream and datagram sockets, and their usage;
- Explain how to implement clients and servers programs that communicates with each other;
- define reserved sockets and proxy servers;
- implement Java networking applications using TCP/IP Server Sockets, and
- implement Java networking applications using Datagram Server Sockets.

3.2 SOCKET OVERVIEW

Most of the inter process communication uses the *client server model*. The terms client and server refer to as the two processes, which will be communicating with each other. One of the two processes, the *client*, connects to the other process, the *server*, typically to make a request for information. A good analogy of this type of communication can be a person who makes a railway engineering from other person

may be through phone call: person making enquiry is a client and another person providing information is a server.

Notice that the client needs to know about the existence and the address of the server, but the server does not need to know the address of (or even the existence of) the client prior to the connection being established. Once a connection is established, both sides can send and receive information.

The system calls for establishing a connection are different for the client and the server, but both involve the basic construct of a *socket*.

Socket is a data structure that maintains necessary information used for communication between client & server. Therefore both end of communication has its own sockets.

Java introduces socket based communications, which enable applications to view networking as if it were file I/O– a program which can read from socket or write to a socket with the simplicity as reading from a file or writing to a file. Java provides *stream sockets and datagram sockets*.

Stream Sockets

Stream Sockets are used to provide a connection-oriented service (i.e. TCP-Transmission Control Protocol).

With stream sockets a process establishes a connection to another process. Once the connection is in place, data flows between processes in continuous streams.

Datagram Sockets

This socket are used to provide a connection-less service, which does not guarantee that packets reach the destination and they are in the order at the destination. In this, individual packets of information are transmitted. In fact it is observed that packets can be lost, can be duplicated, and can even be out of sequence.

In this section you will get answers of some of the most common problems to be addressed in sockets programming using Java. Then you will see some example programs to learn how to write client and server applications.

The very first problem you have to address is "How will you open a socket?" Before reaching to the answer to this question you will definitely think that what type of socket is required? Now the answer can be given as follows:

1. If you were programming a client, then you would open a socket like this:

```
Socket MyClient;

MyClient = new Socket("Machine_Name", PortNumber);
```

Port is a unique number association with a socket on a machine. In other word's port is a numbered socket on a machine.

Where "Machine name" is the server machine you are trying to open a connection to, and "PortNumber" is the number of the port on server, on which you are trying to connect it. When selecting a port number, you should note that port numbers between 0 and 1,023 are reserved for privileged users or slandered services like e–mail, HTTP etc. For example, port number 21 is for FTP, 23 is for TLNET, and 80 is for HTTP. It is a good idea to handle exceptions while creating a new Socket.

```
Socket MyClient;
try {
        MyClient = new Socket("Machine name", PortNumber);
    }
catch (IOException e)
```

```
   {
      System.out.println(e);
    }
```

2. If you are programming a server, then this is how you open a socket:

```
ServerSocket MyService;
try  {
      MyServerice = new ServerSocket(PortNumber);
      }
catch (IOException e)
     {
      System.out.println(e);
      }
```

While implementing a server you also need to create a socket object from the ServerSocket in order to listen for client and accept connections from clients.

```
Socket clientSocket = null;
try {
    serviceSocket = MyService.accept();
     }
catch (IOException e)
    {
     System.out.println(e);
     }
```

You can notice that for a Client side programming you use the Socket class and for the Server side programming you use the ServerSocket class.

Now you know how to create client socket and server socket. Now you have to create input stream and output stream to receive and send data respectively.

InputStream

On the client side, you can use the DataInputStream class to create an input stream to receive response from the server:

```
DataInputStream input;
try {
      input = new DataInputStream(MyClient.getInputStream());
    }
catch (IOException e)
   {
    System.out.println(e);
    }
```

The class DataInputStream allows you to read lines of text and Java primitive data types in a portable way. It has methods such as read, readChar, readInt, readDouble, and readLine. Use whichever function you think suits your needs depending on the type of data that you receive from the server.

On the server side, you can use DataInputStream to receive input from the client:

```
DataInputStream input;
try {
      input = new DataInputStream(serviceSocket.getInputStream());
      }
catch (IOException e)
```

```
    {
     System.out.println(e);
     }
```

OutputStream

On the client side, you can create an output stream to send information to the server socket using the class PrintStream or DataOutputStream of java.io:

```
PrintStream output;
try {
     output = new PrintStream(MyClient.getOutputStream());
     }
catch (IOException e)
    {
     System.out.println(e);
    }
```

The class PrintStream has methods for displaying textual representation of Java primitive data types. Its write () and println () methods are important here. Also, you can use the DataOutputStream:

```
DataOutputStream output;
try {
     output = new DataOutputStream(MyClient.getOutputStream());
     }
catch (IOException e)
    {
     System.out.println(e);
    }
```

The class DataOutputStream allows you to write Java primitive data types; many of its methods write a single Java primitive type to the output stream. The method writeBytes () is a useful one.

On the server side, you can use the class PrintStream to send information to the client.

```
PrintStream output;
try {
     output = new PrintStream(serviceSocket.getOutputStream());
    }
catch (IOException e)
   {
    System.out.println(e);
   }
```

If you have opened a socket for connection, after performing the desired operations your socket should be closed. You should always close the output and input stream before you close the socket.

On the client side:

```
try
   {
        output.close();
        input.close();
       MyClient.close();
   }
catch (IOException e)
```

```
  {
      System.out.println(e);
  }
```

On the server side:

```
try
  {
      output.close();
      input.close();
      serviceSocket.close();
      MyService.close();
  }
catch (IOException e)
 {
      System.out.println(e);
  }
```

☞ Check Your Progress 1

1) Write a program to show that from the client side you send a string to a server that reverse the string which is displayed on the client side.

……

……

……

2) Describe different types of sockets.

……

……

……

3) What are Datagram and Stream Protocols?

……

……

……

3.3 RESERVED PORTS AND PROXY SERVERS

Now let us see some reserve ports. As we have discussed earlier, there are some port numbers, which are **reserved** for specific purposes on any computer working as a server and connected to the Internet. Most standard applications and protocols use **reserved** port numbers, such as email, FTP, and HTTP.

When you want two programs to talk to each other across the Internet, you have to find a way to initiate the connection. So at least one of the 'partners' in the conversation has to know where to find the other one or in other words the address of other one. This can be done by address (IP number + port number) of the one side to the other.

However, a problem could arise if this address must not be taken over by any other program. In order to avoid this, there are some port numbers, which are reserved for specific purposes on any computer connected to the Internet. Such ports are reserved for programs such as 'Telnet', 'Ftp' and others. For example, Telnet uses port 23, and

FTP uses port 21. Note that for each kind of service, not only a port number is given, but also a protocol name (usually TCP or UDP).

Two services may use the same port number, provided that they use different protocols. This is possible due to the fact that different protocols have different address spaces: port 23 of a one machine in the TCP protocol address space is not equivalent to port 23 on the same machine, in the UDP protocol address space.

Proxy Servers

A proxy server is a kind of buffer between your computer and the Internet resources you are accessing. They accumulate and save files that are most often requested by thousands of Internet users in a special database, called "cache". Therefore, proxy servers are able to increase the speed of your connection to the Internet. The cache of a proxy server may already contain information you need by the time of your request, making it possible for the proxy to deliver it immediately. The overall increase in performance may be very high.

Proxy servers can help in cases when some owners of the Internet resources impose some restrictions on users from certain countries or geographical regions. In addition to that, a type of proxy server called anonymous proxy servers can hide your IP address thereby saving you from vulnerabilities concerned with it.

Anonymous Proxy Servers

Anonymous proxy servers hide your IP address and thereby prevent your data from unauthorized access to your computer through the Internet. They do not provide anyone with your IP address and effectively hide any information about you. Besides that, they don't even let anyone know that you are surfing through a proxy server. Anonymous proxy servers can be used for all kinds of Web-services, such as Web-Mail (MSN Hot Mail, Yahoo mail), web-chat rooms, FTP archives, etc. *ProxySite.com* will provide a huge list of public proxies.

Any web resource you access can gather personal information about you through your unique IP address – your ID in the Internet. They can monitor your reading interests, spy upon you, and according to some policies of the Internet resources, deny accessing any information you might need. You might become a target for many marketers and advertising agencies that, having information about your interests and knowing your IP address as well as your e-mail. They will be able to send you regularly their spam and junk e-mails.

3.4 INTERNET ADDRESSING: DOMAIN NAMING SERVICES (DNS)

You know there are thousands of computers in a network; it is not possible to remember the IP address of each system. Domain name system provides a convenient way of finding computer systems in network based on their name and IP address. Domain name services resolves names to the IP addresses of a machine and vice-versa. Domain name system is a hierarchical system where you have a top-level domain name server sub domain and clients with names & IP address.

When you use a desktop client application, such as e-mail or a Web browser, to connect to a remote computer, your computer needs to resolve the addresses you have entered, into the IP addresses it needs to connect to the remote server. DNS is a way to resolve domain name to IP addresses on a TCP/IP network.

The major components of DNS are:
Domain Name Space,
Domain Name Servers and Resource Records (RR),
Domain Name Resolvers (DNRs).

The Domain Name Space: It is a tree-structured name space that contains the domain names and data associated with the names. For example, *astrospeak.indiatimes.com* is a node within the *indiatimes.com* domain, which is a node in the com domain. Data associated with *astrospeak.indiatimes.com* includes its IP address. When you use DNS to find a host address, you are querying the Domain Name Space to extract information.

A Domain Name Server provides information about a subset of the Domain Name Space.

The Domain Name Space for an entity is the name by which the entity is known on the Internet. For example, in the organization shown in, you have two entities with two address spaces. The *Times of India* organization is known on the Internet as the *timesofindia.com* domain, and our sample Internet organization is known as the *indiatimes.com domain*. Hosts at these organizations are known as a host name plus the domain name; for example, *money.timesofindia.com*. Similarly, users in these domains can be found by their e-mail aliases; for example, *abc@indiatimes.com.*

The Domain Name Server points to other Domain Name Servers that have information about other subsets of the Domain Name Space. When you query a Domain Name Server, it returns information if it is an authoritative server for that domain. If the Domain Name Server doesn't have the information, it refers you to a higher level Domain Name Server, which in turn can refer you to another Domain Name Server, until it locates the one with the requested information. In this way, no single server needs to have all the information for every host you might need to contact.

A Domain Name Resolver extracts information from Domain Name Servers so you can use host addresses instead of IP addresses in clients such as a Web browser or a File Transfer Protocol (FTP) client, or with utilities such as ping, tracer, or finger. The DNR is typically built into the TCP/IP implementation on the desktop and needs to know only the IP address of the Domain Name Server. Configuring the DNR on the desktop is usually a matter of filling in the TCP/IP configuration data.

3.5 JAVA AND THE NET: URL

You know each package defines a number of classes, interfaces, exceptions, and errors. The java.net package contains these, interfaces, classes, and exceptions: This package is used in programming where you need to know some information regarding the internet or if you want to communicate between two host computers.

Interfaces in java.net

ContentHandlerFactory
FileNameMap
SocketImplFactory
URLStreamHandlerFactory

Classes in java.net

ContentHandler
DatagramSocket
DatagramPacket
DatagramSocketImpl
HttpURLConnection
InetAddress

Exceptions in java.net

BindException
ConnectException
MalformedURLException
NoRouteToHostException

ProtocolException
SocketException
UnknownHostException
UnknownServiceException

MulticastSocket
ServerSocket
Socket
SocketImpl
URL
URLConnection
URLEncoder
URLStreamHandler

URL

URL is the acronym for Uniform Resource Locator. It represent the addresses of resources on the Internet. You need to provide URLs to your favorite Web browser so that it can locate files on the Internet. In other words you can see URL as addresses on letters so that the post office can locate for correspondents URL class is provided in the `java.net` package to represent a URL address.

URL object represents a URL address. The URL object always refers to an absolute URL. You can construct from an absolute URL, a relative URL. URL class provides accessor methods to get all of the information from the URL without doing any string parsing.

You can connect to a URL by calling `openConnection()` on the URL. The `openConnection()` method returns a URLConnection object. URLConnection object is used for general communications with the URL, such as reading from it, writing to it, or querying it for content and other information.

Reading from and Writing to a URL Connection

Some URLs, such as many that are connected to cgi-bin scripts, allow you to write information to the URL. For example, if you have to search something, then search script may require detailed query data to be written to the URL before the search can be performed.

Before interacting with the URL you have to first establish a connection with the Web server that is responsible for the document identified by the URL.

Then you can use TCP socket for the connection is constructed by invoking the oprnConnection method on the URL object. This method also performs the name resolution necessary to determine the IP address of the Web server.

The openConnection method returns an object of type URLConnectionn to the Web server which is requested by calling connection on the URLConnection object. For input and output handling for the document identified by the URL InputStream and OutputStream are used respectively.

Below are the various constructors and methods of URL class of java.net package.

public URL(String protocol, String host, int port, String file)
throws MalformedURLException

public URL(String protocol, String host, String file)
throws MalformedURLException

public URL(String spec) throws MalformedURLException
public URL(URL context, String spec) throws MalformedURLException
public int getPort()

```
public String getFile()
public String getProtocol()
public String getHost()
public String getRef()
public boolean equals(Object obj)
public int hashCode()
public boolean sameFile(URL other)
public String toString()
public URLConnection openConnection() throws IOException
public final InputStream openStream() throws IOException
public static synchronized void setURLStreamHandlerFactory(
URLStreamHandlerFactory factory)
```

In the example program given below, URL of homepage of "rediff.com" is created.

```
import java.net.*;
class URL_Test
{
        public static void main(String[] args) throws MalformedURLException
          {
                URL redURL = new URL("http://in.rediff.com/index.html");
                System.out.println("URl Prtocol:"+redURL.getProtocol());
                System.out.println("URl Port:"+redURL.getPort());
                System.out.println("URl Host:"+redURL.getHost());
                System.out.println("URl File"+redURL.getFile());
          }
}
Output:
URl Prtocol:http
URl Port:-1
URl Host:in.rediff.com
URl File/index.html
```

☞ Check Your Progress 2

1) When should you use Anonymous Proxy Servers?

……

……

……

……

……

2) Explain various kinds of domain name servers.

……

……

……

……

3) Write a program in Java to know the Protocol, Host, Port, File and Ref of a particular URL using *java.net.URL* package.

……

……

……

3.6 TCP/IP SOCKETS

TCP/IP is a set of protocols used for communication between different types of computers and networks.

TCP/IP refers to two of the protocols in the suite: the *Transmission Control Protocol* and the *Internet Protocol*.

These protocols utilize sockets to exchange data between machines. The TCP protocol requires that the machines communicate with one another in a reliable, ordered stream. Therefore, all data sent from one side must be received in correct order and acknowledged by the other side. This takes care of lost and dropped data by means of acknowledgement and re-transmission. UDP, however, simply sends out the data without requiring knowing that the data be received.

In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

UDP is an unreliable protocol; there is no guarantee that the datagrams you have sent will be put in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you send will be put in the order in which they were sent.

In short, TCP is useful for implementing network services: such as remote login (rlogin, telnet) and file transfer (FTP). These services require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. UDP is often used in implementing client/server applications in distributed systems, which are built over local area networks.

We have already discussed about client and server Sockets in section 3.2 of this Unit. Recall the discussions in section 3.2, two packages java.net.ServerSocket and java.net.Socket were used to create sockets.

Before we discuss about Socket and ServerSocket class, it is important to know about InetAddress class. Let us see what is InetAddress claas.

InetAddress

This class is used for encapsulating numerical IP address and domain name for that address.

Because InetAddress is not having any constructor, its objects are created by using any of the following three methods

static InetAddress getLocalHost() throws UnknownHostException:
returns the InetAddress object representing local host
static InetAddress getByName() throws UnknownHostException:
returns the InetAddress object for the host name passed to it.

static InetAddress getAllByName() throws UnknownHostException: returns an array of InetAddresses representing all the addresses that a particular name is resolves to.

Java.net.Socket

Constructors
public Socket(InetAddress addr, int port): creates a stream socket and connects it to the specified port number at the specified IP address

public Socket (String host, int port): creates a stream socket and connects it to the specified port number at the specified host

Methods:
InetAddress getInetAddress() : Return Inet Address of object associated with Socket

int getPort() : Return remote port to which socket is connected
int getLocalPort() Return local port to which socket object is connected.
public InputStream getInputStream(): Get InputStream associated with Socket
public OutputStream getOutputStream():Return OutputStream associated with socket
public synchronized void close() :closes the Socket.

Java.net.ServerSocket

Constructors:
public ServerSocket(int port): creates a server socket on a specified port with a queue length of 50. A port number 0 creates a socket on any free port.

public ServerSocket(int port, int QueLen): creates a server socket on a specified port with a queue length of QueLen.

public ServerSocket(int port, int QueLen, InetAddress localAdd): creates a server socket on a specified port with a queue length specified by QueLet.On a multihomed host, locaAdd specifies the IP address to which this socket binds.

Methods:

public Socket accept(): listens for a connection to be made to this socket and accepts it. public void close():closes the socket.

Java TCP Socket Example

A Server (web server) at ohm.uwaterloo.ca
- listens to port 80 for Client Connection Requests
- Establish InputStream for sending data to client
- Establish OutputStream for receiving data from client

TCP connection example: (Server)

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class myserver {
        public static void main( String [] s) {
                try {
                  ServerSocket s = new ServerSocket( 80 );
                  While (true) {
                    // wait for a connection request from client
                    Socket clientConn = s.accept();
                    InputStream in = clientConn.getInputStream();
                    OutputStream out = clientConn.getOutputStream();
                    // communicate with client
                    // ..
       clientConn.close(); // close client connection
      }
                }catch (Exception e) {
                    System.out.println("Exception!");
                    // do something about the exception
                }
       }
}
```

TCP connection example: (Client)

```
import java.io.*;
import java.net.ServerSocket;
import java.net.Socket;
public class myclient {
        public static void main( String [] s) {
                try {
                  InetAddress addr = InetAddress.getByName(
                                        “ohm.uwaterloo.ca”);
                  Socket s = new Socket(addr, 80);
                  InputStream in = s.getInputStream();
                  OutputStream out = s.getOutputStream();
                  // communicate with remote process
                  // e.g. GET document /~ece454/index.html
                  s.close();
                } catch(Exception e) {
                  System.out.println(“Exception”);
                  // do something about the Exception
                }
        }
}
```

3.7 DATAGRAMS

A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. Datagrams runs over UDP protocol.

The UDP protocol provides a mode of network communication where packets sent by applications are called datagrams. A datagram is an independent, self-contained message sent over the network whose arrival, arrival time, and content are not guaranteed. The datagram Packet and Datagram Socket classes in the java.net package implement system independent datagram communication using UDP.

Actually the DatagramPacket class is a wrapper for an array of bytes from which data will be sent or into which data will be received. It also contains the address and port to which the packet will be sent.

DatagramPacket constructors:

```
public DatagramPacket(byte[] data, int length)
public DatagramPacket(byte[] data, int length, InetAddress host, int port)
```

You can construct a DatagramPacket object by passing an array of bytes and the number of those bytes to the DatagramPacket() constructor:

```
String s = "My first UDP Packet"
byte[] b = s.getBytes();
DatagramPacket dp = new DatagramPacket(b, b.length());
```

Normally the object of DatagramPacket is created by passing in the host and port to which you want to send the packet with data and its length. For example, object m in the code given below:

```
try
{
InetAddress m = new InetAddress("http://mail.yahoo.com");
int chargen = 19;
```

```
String s = "My second UDP Packet"
byte[] b = s.getBytes();
DatagramPacket dp = new DatagramPacket(b, b.length, m, chargen);
 }
catch (UnknownHostException ex)
{
System.err.println(ex);
 }
```

The byte array that's passed to the constructor is stored by reference, not by value. If you change its contents elsewhere, the contents of the DatagramPacket change as well.

DatagramPackets themselves are not immutable. You can change the data, the length of the data, the port, or the address at any time using the following four methods:

```
public void setAddress(InetAddress host)
public void setPort(int port)
public void setData(byte buffer[])
public void setLength(int length)
```

You can retrieve address, port, data , and length of data using the following four get methods:

```
public InetAddress getAddress()
public int getPort()
public byte[] getData()
public int getLength()
```

DatagramSocket constructors: The java.net.DatagramSocket class has three constructors:

```
public DatagramSocket() throws SocketException
public DatagramSocket(int port) throws SocketException
public DatagramSocket(int port, InetAddress laddr) throws SocketException
```

The first is used for datagram sockets that are primarily intended to act as clients, i.e., a sockets that will send datagrams before receiving anything from anywhere.
The second constructors that specify the port and optionally the IP address of the socket, are primarily intended for servers that must run on a well-known port.

Sending UDP Datagrams

To send data to a particular server, you first must convert the data into byte array. Next you pass this byte array, the length of the data in the array (most of the time this will be the length of the array) the InetAddress and port to DatagramPacket() constructor.

For example, first you create atagramPacket object

```
try
{
  InetAddress m = new InetAddress("http://mail.yahoo.com");
  int chargen = 19;
  String s = "My second UDP Packet";
  byte[] b = s.getBytes();
  DatagramPacket dp = new DatagramPacket(b, b.length, m, chargen);
 }
catch (UnknownHostException ex)
{
  System.err.println(ex);
```

```
 }
```

Now create a DatagramSocket object and pass the packet to its send() method as try

```
 {
  DatagramSocket sender = new DatagramSocket();
  sender.send(dp);
  }
catch (IOException ex)
{
  System.err.println(ex);
 }
```

Receiving UDP Datagrams

To receive data sent to you, construct a DatagramSocket object bound to the port on which you want to receive the data. Then you pass an empty DatagramPacket object to the DatagramSocket's receive() method.

public void receive(DatagramPacket dp) throws IOException.

The calling threads blocks until the datagram is received. Then dp is filled with the data from that datagram. You can then use getPort() and getAddress() to tell where the packet came from, getData() to retrieve the data, and getLength() to see how many bytes were in the data. If the received packet is too long for the buffer, then it is truncated to the length of the buffer. You can write program with the help of code written below:

```
 try
 {
  byte buffer = new byte[65536]; // maximum size of an IP packet
  DatagramPacket incoming = new DatagramPacket(buffer, buffer.length);
  DatagramSocket ds = new DatagramSocket(2134);
  ds.receive(dp);
  byte[] data = dp.getData();
  String s = new String(data, 0, data.getLength());
  System.out.println("Port " + dp.getPort() + " on " + dp.getAddress()
   + " sent this message:");
  System.out.println(s);
  }
catch (IOException ex)
 {
  System.err.println(ex);
 }
```

☞ Check Your Progress 3

1) Write a program to print the address of your local machine and Internet web site rediff.com and webduniya.com.

……

……

……

……

2) Give a brief explanation of TCP Client/Server Interaction.

……

……

……

3) Differentiate between TCP or UDP Protocols.

……

……

……

4) Write a program, which does UDP Port scanner by checking out various port numbers status (i.e.) Are they occupied or free?

……

……

……

……

……

3.8 SUMMARY

You have learnt that Java provides stream sockets and datagram sockets. With stream sockets a process establishes a connection to another process. While the connection is in place, data flows between the processes in continuous streams.

Stream sockets provide connection-oriented service. The TCP protocol is used for this purpose. With datagram sockets individual packets of information are transmitted. UDP protocol is used for this kind of communication. Stream based connections are managed with Sockets objects.

Datagram packets are used to create the packets to send and receive information using Datagram Sockets. Connection oriented services can be seen as your telephone service and connection –less services can be seen as Radio Broadcast.

Domain naming services solve the problem of remembering the long IP address of various web sites and computers. You also have learn that there are some dedicated Port numbers for various protocols which are known as reserved port like for FTP you have port no. 21.

With Proxy servers you can prevent your computer not accessible to anyone you don't want. Your computer data cannot be traced easily if you are using proxy servers, as the IP addresses will not be known to the second person directly.

3.9 SOLUTIONS/ANSWERS

Check Your Progress 1

1)
Client side Programming

```
import java.io.*;
import java.net.*;

public class Client
{
   public static final int DEFAULT_PORT = 8000;
   public static void usage()
   {
      System.out.println("Usage: java Client <hostname> [<port>]");
      System.exit(0);
```

```
 }
public static void main(String[] args)
{
   int port = DEFAULT_PORT;
   Socket s = null;

   // Parse the port specification
   if ((args.length != 1) && (args.length != 2)) usage();
   if (args.length == 1) port = DEFAULT_PORT;
   else
   {
       try
      {
        port = Integer.parseInt(args[1]);
       }
      catch (NumberFormatException e)
      { usage();
       }
  }
   try
     {
      // Here is  a socket to communicate to the specified host and port
      s = new Socket(args[0], port);

      BufferedReader sin = new BufferedReader(new
      InputStreamReader(s.getInputStream()));//stream for reading
      PrintStream sout = new PrintStream(s.getOutputStream());
                        // stream for writing lines of text
      // Here ise a stream for reading lines of text from the console
        BufferedReader in = new BufferedReader(new
       InputStreamReader(System.in));

      System.out.println("Connected to " + s.getInetAddress()
              + ":"+ s.getPort());
      String line;
      while(true)
       {
         // print a prompt
         System.out.print("> ");
         System.out.flush();
         // read a line from the console; check for EOF
         line = in.readLine();
         if (line == null) break;
         // Send it to the server
         sout.println(line);
         // Read a line from the server.
         line = sin.readLine();
         // Check if connection is closed (i.e. for EOF)
         if (line == null)
         {
            System.out.println("Connection closed by server.");
            break;
         }
         // And write the line to the console.
         System.out.println(line);
      }
   }
   catch (IOException e)
```

```
    {
     System.err.println(e);
    }
    finally
    {
        try
          {
            if (s != null)
            s.close();
            }
        catch (IOException e2)  { ; }
     }
  }
}
```

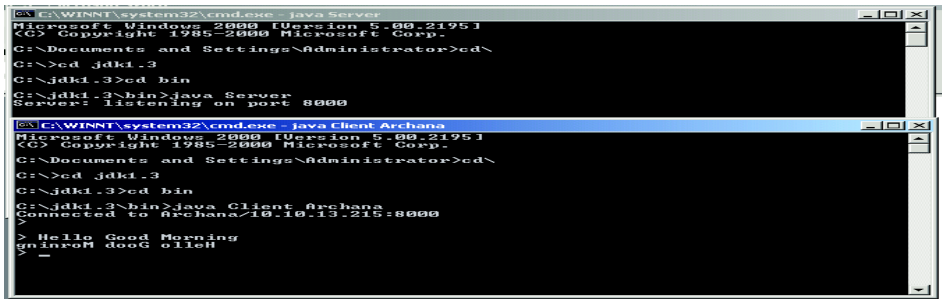
Server Side Programming

```
import java.io.*;
import java.net.*;
public class Server extends Thread
{
   public final static int DEFAULT_PORT = 8000;
   protected int port;
   protected ServerSocket listen_socket;
   public static void fail(Exception e, String msg)
   {
      System.err.println(msg + ": " +  e);
      System.exit(1);
    }
      // Creating  a ServerSocket to listen for connections on;
   public Server(int port)
    {
      if (port == 0) port = DEFAULT_PORT;
      this.port = port;
      try { listen_socket = new ServerSocket(port);
    }
    catch (IOException e)
    {
     fail(e, "Exception creating server socket");
    }
      System.out.println("Server: listening on port " + port);
      this.start();
   }
    // create a Connection object to handle communication through the new Socket.
   public void run()
   {
      try
        {
          while(true)
          {
            Socket client_socket = listen_socket.accept();
            Connection c = new Connection(client_socket);
          }
          }
      catch (IOException e)
        {

          fail(e, "Exception while listening for connections");
```

```
        }
    }
   public static void main(String[] args)
     {
      int port = 0;
      if (args.length == 1)
      {
         try
           {
            port = Integer.parseInt(args[0]);
           }
         catch (NumberFormatException e)
           {
            port = 0;
           }
      }
      new Server(port);
   }
}

// A thread  class that handles all communication with a client
class Connection extends Thread
   {
    protected Socket client;
    protected BufferedReader in;
    protected PrintStream out;
    // Initialize the streams and start the thread
    public Connection(Socket client_socket)
    {
        client = client_socket;
        try
          {
            in = new BufferedReader(new InputStreamReader(client.getInputStream()));
            out = new PrintStream(client.getOutputStream());
          }
        catch (IOException e)
         {
           try
           {
            client.close();
           }
           catch (IOException e2) { ; }
           System.err.println("Exception while getting socket streams: " + e);
           return;
        }
        this.start();
    }
    public void run()
      {
        String line;
        StringBuffer revline;
        int len;
        try
         {
           for(;;)
           {
               // read in a line
               line = in.readLine();
```

```
            if (line == null) break;
            // reverse it
            len = line.length();
            revline = new StringBuffer(len);
            for(int i = len-1; i >= 0; i--)
               revline.insert(len-1-i, line.charAt(i));
            // and write out the reversed line
            out.println(revline);
          }
        }
        catch (IOException e) { ; }
        finally
          {
            try
              {
              client.close();
              }
               catch (IOException e2) {;}
          }
    }
}
```

Output :



In the output screen you can see that that server is running and listening to on port number 8000.

At the client side you can see that whenever you will write a string as “Hello Good Morning”
Then the string goes to server side and the server reverses it as the reverse function is written on the server.

2) In Table given below Socket type protocols and their description is given:

Table 1:Types of Sockets

Socket type	Protocol	Description
SOCK_STREAM	Transmission Control Protocol (TCP)	The stream socket (SOCK_STREAM) interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent.

SOCK_DGRAM	User Datagram Protocol (UDP)	The datagram socket (SOCK_DGRAM) interface defines a connectionless service for datagrams, or messages. Datagrams are sent as independent packets. The reliability is not guaranteed, data can be lost or duplicated, and datagrams can arrive out of order. However, datagram sockets have improved performance capability over stream sockets and are easier to use.
SOCK_RAW	IP, ICMP, RAW	The raw socket (SOCK_RAW) interface allows direct access to lower-layer protocols such as Internet Protocol (IP).

Note:

The type of socket you use is determined by the data you are transmitting:

- When you are transmitting data where the integrity of the data is high priority, you ***must*** use stream sockets.
- When the data integrity is not of high priority (for example, for terminal inquiries), use datagram sockets because of their ease of use and higher performance capability.

3) There are two communication protocols that one can use for socket programming: datagram communication and stream communication.

Datagram communication:

The datagram communication protocol, known as UDP (user datagram protocol), is a connectionless protocol, meaning that each time you send datagrams, you also need to send the local socket descriptor and the receiving socket's address. As you can tell, socket address is required each time a communication is made.

Stream communication:

The stream communication protocol is known as TCP (Transfer Control Protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client). Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

Check Your Progress 2

1. Using an anonymous proxy server you don't give anybody a chance to find out your IP address to use it in their own interests. We can offer you two ways to solve your IP problem:

i) Secure Tunnel - a pay proxy server with plenty of features. Effective for personal use, when your Internet activities are not involved in very active surfing, web site development, mass form submitting, etc. In short, Secure Tunnel is the best solution for most of Internet users. Ultimate protection of privacy - nobody can find out where you are engaged in surfing. Blocks all methods of tracking. Provides an encrypted connection for all forms of web browsing, including http, news, mail, and the especially vulnerable IRC and ICQ. Comes with special totally preconfigured software.

ii) ProxyWay - a proxy server agent which you use together with your web browser to ensure your anonymity when you surf the Internet. It contains a database of anonymous proxy servers and allows you to easily test their anonymity. Using a network of publicly accessible servers ProxyWay shields your current connection when you visit websites, download files, or use web-based e-mail accounts.

2) Our own small proxy list is also a good place to start with if you are a novice.

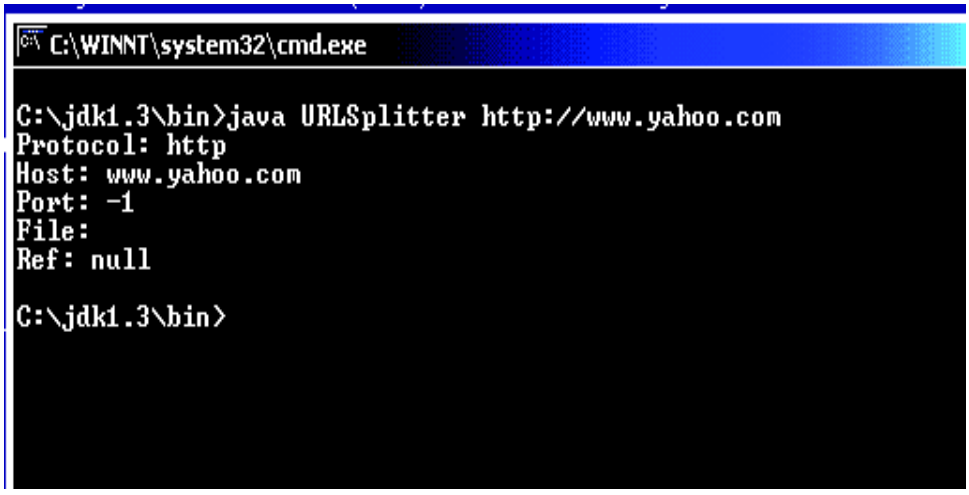
There are two types of Domain Name Servers: primary and secondary.

A primary server maintains a set of configuration files that contain information for the subset of the name space for which the server is authoritative. For example, the primary server for *indiatimes.com* contains IP addresses for all hosts in the *indiatimes.com* domain in configuration files. Resource Records are the entries in the configuration files that contain the actual data. A secondary server does not maintain any configuration files, but it copies the configuration files from the primary server in a process called a zone transfer. A secondary name server can respond to requests for name resolution, and it looks just like a primary name server from a user's perspective. Primary and secondary Domain Name Servers provide both performance and fault-tolerance benefits because you can split the workload between the servers, and if one goes down, the other can take over.

3)

```
import java.net.URL;
public class URLSplitter {
  public static void main(String[] args) {
    for (int i = 0; i < args.length; i++) {
      try {
       java.net.URL u = new java.net.URL(args[i]);
       System.out.println("Protocol: " + u.getProtocol());
       System.out.println("Host: "     + u.getHost());
       System.out.println("Port: "     + u.getPort());
       System.out.println("File: "     + u.getFile());
       System.out.println("Ref: "      + u.getRef());
       }
      catch (java.net.MalformedURLException e) {
       System.err.println(args[i] + " is not a valid URL");
      }}
  }
}
```

Here's the output:



Check Your Progress 3

Program to print the addresses:

1)

```
import java.net.*;
class I_Address
{
        public static void main(String[] args) throws UnknownHostException
        {
                InetAddress  Addr = InetAddress.getLocalHost() ;
                System.out.println(Addr);
    Addr = InetAddress.getByName("yahoo.com");
    System.out.println(Addr);
    InetAddress  Addr1[]  = InetAddress.getAllByName("indiatimes.com") ;
    for( int i = 0; i < Addr1.length; i++)
    System.out.println(Addr1[i]);
        }
}
```

Output:

Run this program on your machine while connected to Internet to get proper output otherwise UnkwonHostException will occur.

Output on the machine is:

shashibhushan/190.10.19.205
rediff.com/208.184.138.70
webduniya.com/65.182.162.66

2) **TCP Client/Server Inetraction:**

The Server starts by getting ready to receive client connections…

Client	**Server**
1. Create a TCP socket	1. Create a TCP socket
2. Establish connection	2. Assign a port to socket
3. Communicate	3. Set socket to listen
4. Close the connection	4. Repeatedly:
	a. Accept new connection
	b. Communicate
	c. Close the connection

3) In UDP, as you have read above, every time you send a datagram, you have to send the local descriptor and the socket address of the receiving socket along with it. Since TCP is a connection-oriented protocol, on the other hand, a connection must be established before communications between the pair of sockets start. So there is a connection setup time in TCP.

Once a connection is established, the pair of sockets behaves like streams: All available data are read immediately in the same order in which they are received.

UDP is an unreliable protocol- there is no guarantee that the datagrams you have sent will be received in the same order by the receiving socket. On the other hand, TCP is a reliable protocol; it is guaranteed that the packets you receive are put in the order in which they were sent.

In short, TCP is useful for implementing network services-such as remote login (rlogin, telnet) and file transfer (FTP)- which require data of indefinite length to be transferred. UDP is less complex and incurs fewer overheads. It is often used in

implementing client/server applications in distributed systems built over local area networks.

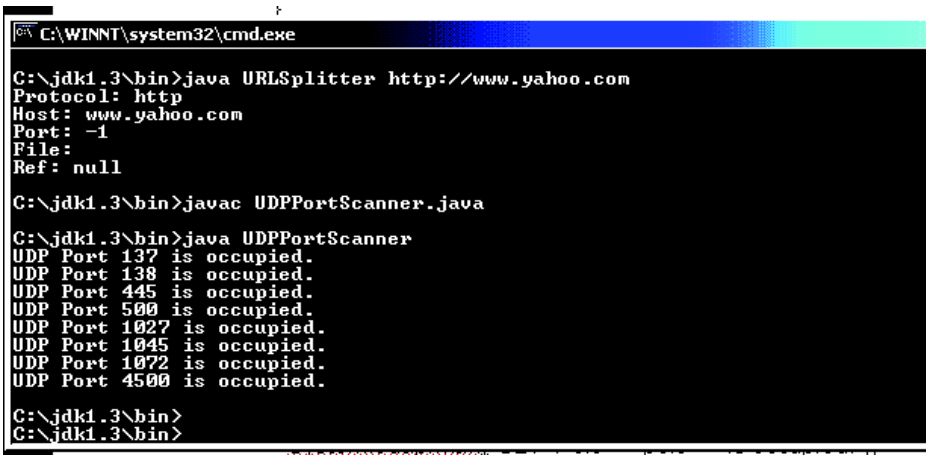
In client-server applications, the server provides some service, such as processing database queries or sending out current stock prices. The client uses the service provided by the server, either displaying database query results to the user or making stock purchase recommendations to an investor. The communication that occurs between the client and the server must be reliable. That is, no data can be dropped and it must arrive on the client side in the same order in which the server sent it.

TCP provides a reliable, point-to-point communication channel those client-server applications use on the Internet to communicate with each other. To communicate over TCP, a client program and a server program establish a connection between them. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

4) The LocalPortScanner developed earlier only found TCP ports. The following program detects UDP ports in use. As with TCP ports, you must be root on Unix systems to bind to ports below 1024.

```
import java.net.*;
import java.io.IOException;
public class UDPPortScanner {
  public static void main(String[] args) {
    // first test to see whether or not we can bind to ports
    // below 1024
    boolean rootaccess = false;
    for (int port = 1; port < 1024; port += 50) {
      try {
        ServerSocket ss = new ServerSocket(port);
        // if successful
        rootaccess = true;
        ss.close();
        break;
      }
      catch (IOException ex) {
      }
    }
    int startport = 1;

    if (!rootaccess) startport = 1024;
    int stopport = 65535;

    for (int port = startport; port <= stopport; port++)
    {
      try {
        DatagramSocket ds = new DatagramSocket(port);
        ds.close();
      }
      catch (IOException ex) {
        System.out.println("UDP Port " + port + " is occupied.");
      }
      }
 } }
```

Output



Since UDP is connectionless it is not possible to write a remote UDP port scanner. The only way you know whether or not a UDP server is listening on a remote port is if it sends something back to you.

UNIT 4 ADVANCE JAVA

Structure **Page Nos.**

4.0 INTRODUCTION

This unit will introduce you to the advanced features of Java. To save data, you have used the file system, which gives you functionality of accessing the data but it does not offer any capability for querying on data conveniently.

You are familiar with various databases like Oracle, Sybase, SQL Server etc. They do not only provide the file-processing capabilities, but also organize data in a manner that facilitates applying queries.

Structured Query Language (SQL) is almost universally used in relational database systems to make queries based on certain criteria. In this unit you will learn how you can interact to a database using ***Java Database Connectivity (JDBC)*** feature of Java.

You will also learn about RMI (Remote Method Invocation) feature of Java. This will give the notion of client/server distributed computing. ***RMI*** allows Java objects running on the same or separate computers to communicate with one another via remote method calls.

A request-response model of communication is essential for the highest level of networking. **The Servlets** feature of Java provides functionality to extend the capabilities of servers that host applications accessed via a request-response programming model. In this unit we will learn the basics of Servlet programming.

Java beans are nothing but small reusable pieces of components that you can add to a program without disturbing the existing program code, are also introduced in this unit.

4.1 OBJECTIVES

After going through of this unit you will be able to:

- interact with databases through java programs;
- use the classes and interfaces of the *java.sql* package;

- use basic database queries using Structured Query Language (SQL) in your programs;
- explain Servlet Life Cycle;
- write simple servlets programs;
- explain the model of client/server distributed computing;
- explain architecture of RMI, and
- describe Java Beans and how they facilitate component-oriented software construction.

4.2 JAVA DATABASE CONNECTIVITY

During programming you may need to interact with database to solve your problem. Java provides JDBC to connect to databases and work with it. Using standard library routines, you can open a connection to the database. Basically JDBC allows the integration of SQL calls into a general programming environment by providing library routines, which interface with the database. In particular, Java's JDBC has a rich collection of routines which makes such an interface extremely simple and intuitive.

4.2.1 Establishing A Connection

The first thing to do, of course, is to install Java, JDBC and the DBMS on the working machines. Since you want to interface with a database, you would need a driver for this specific database.

Load the vendor specific driver

This is very important because you have to ensure portability and code reuse. The API should be designed as independent of the version or the vendor of a database as possible. Since different DBMS's have different behaviour, you need to tell the driver manager which DBMS you wish to use, so that it can invoke the correct driver.

For example, an Oracle driver is loaded using the following code snippet:

Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")

Make the connection

Once the driver is loaded and ready for a connection to be made, you may create an instance of a Connection object using:

Connection con = DriverManager.getConnection(url, username, password);

Let us see what are these parameters passed to get Connection method of DriverManager class. The first string is the URL for the database including the protocol, the vendor, the driver, and the server the port number. The username and password are the name of the user of database and password is user password. The connection *con* returned in the last step is an open connection, which will be used to pass SQL statements to the database.

Creating JDBC Statements

A JDBC Statement object is used to send the SQL statements to the DBMS. It is entirely different from the SQL statement. A JDBC Statement object is an open connection, and not any single SQL Statement. You can think of a JDBC Statement object as a channel sitting on a connection, and passing one or more of the SQL statements to the DBMS.

An active connection is needed to create a Statement object. The following code is a snippet, using our Connection object *con*

Statement statmnt = con.createStatement();

At this point, you will notice that a Statement object exists, but it does not have any SQL statement to pass on to the DBMS.

Creating JDBC PreparedStatement

PreparedStatement object is more convenient and efficient for sending SQL statements to the DBMS. The main feature, which distinguishes PreparedStatement object from objects of Statement class, is that it gives an SQL statement right when it is created. This SQL statement is then sent to the DBMS right away, where it is compiled. Thus, in effect, a PreparedStatement is associated as a channel with a connection and a compiled SQL statement.

Another advantage offered by PreparedStatement object is that if you need to use the same or similar query with different parameters multiple times, the statement can be compiled and optimized by the DBMS just once. While with a normal Statement, each use of the same SQL statement requires a compilation all over again.

PreparedStatements are also created with a Connection method. The following code shows how to create a parameterized SQL statement with three input parameters:

```
  PreparedStatement prepareUpdatePrice
= con.prepareStatement( "UPDATE Employee SET emp_address =? WHERE
   emp_code =“1001” AND emp_name =?");
```

You can see two? symbol in the above PreparedStatement *prepareUpdatePrice*. This means that you have to provide values for two variables emp_address and emp_name in PreparedStatement before you execute it. Calling one of the setXXX methods defined in the class PreparedStatement can provide values. Most often used methods are setInt, setFloat, setDouble, setString, etc. You can set these values before each execution of the prepared statement.

You can write something like:

```
prepareUpdatePrice.setInt(1, 3);
prepareUpdatePrice.setString(2, "Renuka");
prepareUpdatePrice.setString(3, "101, Sector-8,Vasundhara, M.P");
```

Executing CREATE/INSERT/UPDATE Statements of SQL

Executing SQL statements in JDBC varies depending on the intention of the SQL statement. DDL (Data Definition Language) statements such as table creation and table alteration statements, as well as statements to update the table contents, all are executed using the *executeUpdate* method. The following snippet has examples of executeUpdate statements.

```
Statement stmt = con.createStatement();
stmt.executeUpdate("CREATE TABLE Employee " +
"(emp_name VARCHAR2(40), emp_address VARCHAR2(40), emp_sal REAL)" );
stmt.executeUpdate("INSERT INTO Employee " +
   "VALUES ('Archana', '10,Down California', 30000" );
String sqlString = "CREATE TABLE Employee " +
   "(name VARCHAR2(40), address VARCHAR2(80), license INT)" ;
stmt.executeUpdate(sqlString);
```

Since the SQL statement will not quite fit on one line on the page, you can split it into two or more strings concatenated by a plus sign(+).
"INSERT INTO Employee" to separate it in the resulting string from "VALUES".

The point to note here is that the same Statement object is reused rather than to create a new one each time.

When executeUpdate is used to call DDL statements, the return value is always zero, while data modification statement executions will return a value greater than or equal to zero, which is the number of tuples affected in the relation by execution of modification statement.

While working with a PreparedStatement, you should execute such a statement by first plugging in the values of the parameters (as you can see above), and then invoking the executeUpdate on it. For example:

```
int n = prepareUpdateEmployee.executeUpdate() ;
```

Executing SELECT Statements

A query is expected to return a set of tuples as the result, and not change the state of the database. Not surprisingly, there is a corresponding method called execute Query, which returns its results as a ResultSet object. It is a table of data representing a database result set, which is usually generated by executing a statement that queries the database.

A ResultSet object maintains a cursor pointing to its current row of data. Initially the cursor is positioned before the first row. The *next method* moves the cursor to the next row, and because it returns false when there are no more rows in the ResultSet object, it can be used in a while loop to iterate through the result set. A default ResultSet object is not updatable and has a cursor that moves forward only

In the program code given below:

```
 String ename,eaddress;
  float esal;

  ResultSet rs = stmt.executeQuery("SELECT * FROM Employee");
  while ( rs.next() ) {
    ename = rs.getString("emp_name");
    eaddress = rs.getString("emp_address");
    esal = rs.getFloat("emp_salary");
    System.out.println(ename + " address is" + eaddress + " draws salary  " + esal + "
in dollars");
  }
```

The tuples resulting from the query are contained in the variable rs which is an instance of ResultSet. A set is of not much use to you unless you can access each row and the attributes in each row. The?

Now you should note that each invocation of the *next method* causes it to move to the next row, if one exists and returns true, or returns false if there is no remaining row.

You can use the getXXX method of the appropriate type to retrieve the attributes of a row. In the above program code getString and getFloat methods are used to access the column values. One more thing you can observe that the name of the column whose value is desired is provided as a parameter to the method.

Similarly,while working with a PreparedStatement, you can execute a query by first plugging in the values of the parameters, and then invoking the executeQuery on it.

1.
```
ename = rs.getString(1);
eaddress = rs.getFloat(3);
esal = rs.getString(2);
```

2. ResultSet rs = prepareUpdateEmployee.executeQuery() ;

Accessing ResultSet

Now to reach each record of the database, JDBC provides methods like getRow, isFirst, isBeforeFirst, isLast, isAfterLas to access ResultSet.Also there are means to make scroll-able cursors to allow free access of any row in the ResultSet. By default, cursors scroll forward only and are read only. When creating a Statement for a Connection, we can change the type of ResultSet to a more flexible scrolling or updatable model:

```
Statement stmt = con.createStatement
ResultSet.TYPE_FORWARD_ONLY, ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery("SELECT * FROM Sells");
```

The different options for types are TYPE_FORWARD_ONLY, TYPE_SCROLL_INSENSITIVE, and TYPE_SCROLL_SENSITIVE. We can choose whether the cursor is read-only or updatable using the options CONCUR_READ_ONLY, and CONCUR_UPDATABLE.

With the default cursor, we can scroll forward using rs.next(). With scroll-able cursors we have more options:

```
rs.absolute(3);         // moves to the third tuple or row
rs.previous();          // moves back one tuple (tuple 2)
rs.relative(2);         // moves forward two tuples (tuple 4)
rs.relative(-3);        // moves back three tuples (tuple 1)
```

4.2.2 Transactions with Database

When you go to some bank for deposit or withdrawal of money, you get your bank account updated, or in other words you can say some transaction takes place.

JDBC allows SQL statements to be grouped together into a single transaction. Thus, you can ensure the ACID (Atomicity, Consistency, Isolation, Durability) properties using JDBC transactional features.

The Connection object performs transaction control. When a connection is created, by default it is in the *auto-commit* mode. This means that each individual SQL statement is treated as a transaction by itself, and will be committed as soon as its execution is finished.

You can turn off *auto-commit* mode for an active connection with:
con.setAutoCommit(false) ;
And turn it on again if needed with:
con.setAutoCommit(true) ;
Once *auto-commit* is off, no SQL statements will be committed (that is, the database will not be permanently updated) until you have explicitly told it to commit by invoking the commit () method:
con.commit() ;

At any point before commit, you may invoke rollback () to rollback the transaction, and restore values to the last commit point (before the attempted updates).

☞ Check Your Progress 1

1) How is a program written in java to access database?

……

……

……

2) What are the different kinds of drivers for JDBC?

……

……

3) Write a program code to show how you will perform commit() and rollback().

……

……

4) Read the following program assuming mytable already exists in database and answer the questions given below:

i. What is the use of rs.next()?
ii. Value of which attribute will be obtained by rs.getString(3).
iii. If the statement "Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")" is removed from the program what will happen.

```
import java.sql.*;
import java.io.*;
public class TestJDBC
   {
   public static void main(String[] args)
      {
      String dataSourceName = "mp";
       String dbURL = "jdbc:odbc:" + dataSourceName;
       try
       {
       Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
       Connection con = DriverManager.getConnection(dbURL, "","");
          Statement s = con.createStatement();
          s.execute("insert into mytable values('AKM',20,'azn')");
          s.executeQuery("select * from mytable ");
          ResultSet rs = s.getResultSet();
             rs.next();
             String n = rs.getString(1);
             System.out.println("Name:"+ n);
       if (rs != null)
       while ( rs.next() )
          {
           System.out.println("Data from column_name: " + rs.getString(1) );
     System.out.println("Data from column_age: " + rs.getInt(2) );
     System.out.println("Data from column_address: " + rs.getString(3) );
       }
   }
   catch (Exception err)
     {
      System.out.println( "Error: " + err );
     }
}
```

4.3 AN OVERVIEW OF RMI APPLICATIONS

Many times you want to communicate between two computers. One of the examples of this type of communication is a chatting program. How do chatting happens or two computers communicate each other? RPC (Remote Procedure Call) is one of the ways to perform this type of communication. In this section you will learn about RMI (Remote Method Invocation).

Java provides RMI (Remote Method Invocation), which is *"a mechanism that allows one to invoke a method on an object that exists in another address space. The other address space could be on the same machine or a different one. The RMI mechanism is basically an object-oriented RPC mechanism."*

RPC (Remote Procedure Call) organizes the types of messages which an application can receive in the form of functions. Basically it is a management of streams of data transmission.

RMI applications often comprised two separate programs: *a server and a client*.
A typical server application creates some remote objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects.
A typical client application gets a remote reference to one or more remote objects in the server and then invokes methods on them.

RMI provides the mechanism by which the server and the client communicate and pass information back and forth. Such an application is sometimes referred to as a *distributed object application*.

There are three processes that participate in developing applications based on remote method invocation.

1. The *Client* is the process that is invoking a method on a remote object.
2. The *Server* is the process that owns the remote object. The remote object is an ordinary object in the address space of the server process.
3. The *Object Registry* is a name server that relates objects with names. Objects are *registered* with the Object Registry. Once an object has been registered, one can use the Object Registry to obtain access to a remote object using the name of the object.

4.3.1 Remote Classes and Interfaces

A *Remote* class is one whose instances can be used remotely. An object of such a class can be referenced in two different ways:

1. Within the address space where the object was constructed, the object is an ordinary object, which can be used like any other object.
2. Within other address spaces, the object can be referenced using an object handle While there are limitations on how one can use an object handle compared to an object, for the most part one can use object handles in the same way as an ordinary object.

For simplicity, an instance of a Remote class is called a *remote object*.

A Remote class has two parts: the interface and the class itself.

The Remote interface must have the following properties:

Interface must be public.
Interface must extend the *java.rmi.Remote* interface. Every method in the interface must declare that it throws java.rmi.RemoteException. Maybe other exceptions also ought to be thrown.

The Remote class itself has the following properties:

It must implement a Remote interface.

It should extend the *java.rmi.server.UnicastRemoteObject* class. Objects of such a class exist in the address space of the server and can be invoked remotely. While there are other ways to define a Remote class, this is the simplest way to ensure that objects of a class can be used as remote objects.

It can have methods that are not in its Remote interface. These can only be invoked locally. It is not necessary for both the Client and the Server to have access to the definition of the Remote class.

The Server requires the definition of both the Remote class and the Remote interface, but the client only uses the Remote interface.

All of the Remote interfaces and classes should be compiled using *javac*. Once this has been completed, the stubs and skeletons for the Remote interfaces should be compiled by using the *rmic stub* compiler. The stub and skeleton of the example Remote interface are compiled with the command:
rmic <filename.class>

4.3.2 RMI Architecture

It consists of three layers as given in Figure 1

1. Stub/Skeleton layer – client-side stubs and server-side skeletons.
2. Remote reference layer-invocation to single or replicated object
3. Transport layer-connection set up and management, also remote object tracking.

JAVA RMI Architecture

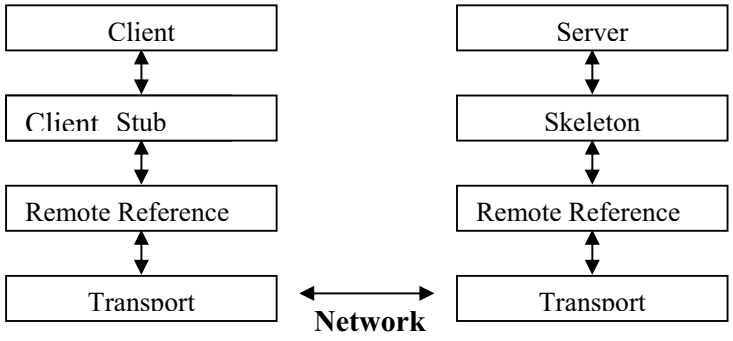


Figure 1: Java RMI Architecture

4.3.3 RMI Object Hierarchy

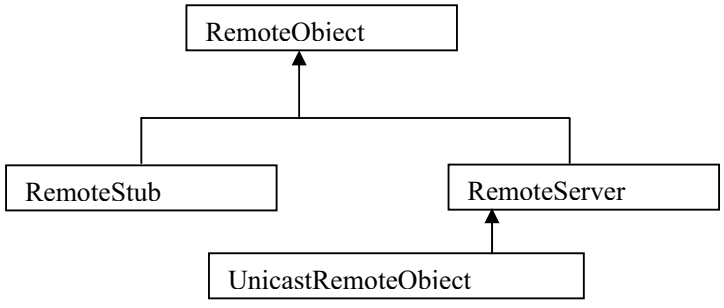


Figure 2:RemoteObject Hierarchy

In *Figure 2* above, the RMI Object Hierarchy is shown. The most general feature set associated with RMI is found in the java.rmi.Remote interface. Abstract class java.rmi.server.RemoteObject supports the needed modifications to the Java object model to cope with the indirect references.

Remote Server is a base class, which encapsulates transport semantics for RemoteObjects. Currently RMI ships with a UnicastRemoteObject(single object)

A server in RMI is a named service which is registered with the RMI registry, and listens for remote requests. For security reasons, an application can bind or unbind only in the registry running on the same host.

4.3.4 Security

One of the most common problems with RMI is a failure due to security constraints. Let us see Java the security model related to RMI. A Java program may specify a security manager that determines its security policy. A program will not have any security manager unless one is specified. You can set the security policy by constructing a *SecurityManager object* and calling the *setSecurityManager* method of the *System* class. Certain operations require that there be a security manager. For example, RMI will download a Serializable class from another machine only if there is a security manager and the security manager permits the downloading of the class from that machine. The RMISecurityManager class defines an example of a security manager that normally permits such download. However, many Java installations have instituted security policies that are more restrictive than the default. There are good reasons for instituting such policies, and you should not override them carelessly.

Creating Distributed Applications Using RMI

The following are the basic steps be followed to develop a distributed application using RMI:

- Design and implement the components of your distributed application.
- Compile sources and generate stubs.
- Make classes network accessible.
- Start the application.

Compile Sources and Generate Stubs

This is a two-step process. In the first step you use the javac compiler to compile the source files, which contain the implementation of the remote interfaces and implementations, of the server classes and the client classes. In the second step you use the rmic compiler to create stubs for the remote objects. RMI uses a remote object's stub class as a proxy in clients so that clients can communicate with a particular remote object.

Make Classes Network Accessible

In this step you have to make everything: the class files associated with the remote interfaces, stubs, and other classes that need to be downloaded to clients, accessible via a Web server.

Start the Application

Starting the application includes running the RMI remote object registry, the server, and the client.

☞ Check Your Progress 2

1) What is Stub in RMI?
……
……

2) What are the basic actions performed by receiver object on server side?
……
……

3) What is the need of and Registry Service of RMI?
……
……

4.4 JAVA SERVLETS

In this section you will be introduced to server side-programming. Java has utility known as servlets for server side-programming.

A *servlet* is a class of Java programming language used to extend the capabilities of servers that host applications accessed via a request-response programming model.

Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. Java Servlet technology also defines HTTP-specific servlet classes. The javax.servlet and java.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.

When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. The HttpServlet class provides methods, such as do get and do Post, for handling HTTP-specific services.

In this section we will focus on writing servlets that generate responses to HTTP requests. Here it is assumed that you are familiar with HTTP protocol.

4.4.1 Servlet Life Cycle

The container in which the servlet has been deployed controls the life cycle of a servlet. When a request is mapped to a servlet, the container performs the following steps.

Loads the servlet class.
Creates an instance of the servlet class.
Initializes the servlet instance by calling the init() method.
When servlet is executed it invokes the service method, passing a request and response object.

If the container needs to remove the servlet, it finalizes the servlet by calling the servlet's destroy method.

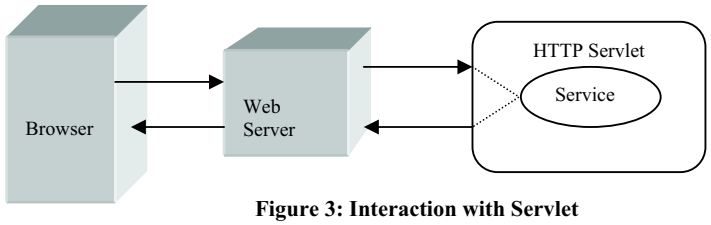


Figure 3: Interaction with Servlet

Servlets are programs that run on servers, such as a web server. You all do net surfing and well known the data on which the web is submitted and you get the respond accordingly. On web pages the data is retrieved from the corporate databases, which should be secure. For these kinds of operations you can use servlets.

4.4.2 GET and POST Methods

The GET methods is a request made by browsers when the user types in a URL on the address line, follows a link from a Web page, or makes an HTML form that does not specify a METHOD. Servlets can also very easily handle POST requests, which are generated when someone creates an HTML form that specifies METHOD="POST".

The program code given below will give you some idea to write a servlet program:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SomeServlet extends HttpServlet
     {
     public void doGet(HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException
    {
    // Use "request" to read incoming HTTP headers (e.g. cookies)
    // and HTML form data (e.g. data the user entered and submitted)

    // Use "response" to specify the HTTP response line and headers
    // (e.g. specifying the content type, setting cookies).
   PrintWriter out = response.getWriter();
    // Use "out" to send content to browser
  }
}
```

To act as a servlet, a class should extend HttpServlet and override doGet or doPost (or both), depending on whether the data is being sent by GET or by POST. These methods take two arguments: an HttpServletRequest and an HttpServletResponse objects.
The HttpServletRequest has methods for information about incoming information such as FORM data, HTTP request headers etc.
The httpServletResponse has methods that let you specify the HTTP response line (200, 404, etc.), response headers (Content-Type, Set-Cookie, etc.), and, most importantly, a PrintWriter used to send output back to the client.

A Simple Servlet: Generating Plain Text

Here is a simple servlet that just generates plain text:

```
//Program file name: HelloWorld.java
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet
 {
  public void doGet (HttpServletRequest request, HttpServletResponse response)
     throws ServletException, IOException
     {
     PrintWriter out = response.getWriter();
     out.println("Hello World");
     }
 }
```

Compiling and Installing the Servlet

Note that the specific details for installing servlets vary from Web server to Web server. Please refer to the Web server documentation for definitive directions. The on-line examples are running on Java Web Server (JWS) 2.0, where servlets are expected to be in a directory called servlets in the JWS installation hierarchy.

You have to set the CLASSPATH to point to the directory above the one actually containing the servlets. You can then compile normally from within the directory.

```
DOS> set CLASSPATH=C:\JavaWebServer\servlets;%CLASSPATH%
DOS> cd C:\JavaWebServer\servlets\
DOS> javac HelloWorld.java
```

Running the Servlet

With the Java Web Server, servlets are placed in the servlets directory within the main JWS installation directory, and are invoked via *http://host/servlet/ServletName*. Note that the directory is servlets, plural, while the URL refers to servlet, singular. Other Web servers may have slightly different conventions on where to install servlets and how to invoke them. Most servers also let you define aliases for servlets, so that a servlet can be invoked via *http://host/any-path/any-file.html*.

The Url that you will give on the explorer will be:
http://localhost:8080/servlet/HelloWorld then you will get the output as follows:

A Servlet that Generates HTML

Most servlets generate HTML, not plain text as in the previous example. To do that, you need two additional steps: tell the browser that you're sending back HTML, and modify the println statements to build a legal Web page. First set the Content-Type response header. In general, headers can be set via the setHeader method of HttpServletResponse, but setting the content type is such a common task that there is also a special setContentType method just for this purpose. You need to set response headers *before* actually returning any of the content via the PrintWriter.
Here is an example for the same:

```
//HelloWWW.java

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class HelloWWW extends HttpServlet
{
 public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
 {
  response.setContentType("text/html");
  PrintWriter out = response.getWriter();
  out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 " +
                          "Transitional//EN\">\n" +
          "<HTML>\n" +
```

```
            "<HEAD><TITLE>Hello WWW</TITLE></HEAD>\n" +
            "<BODY>\n" +
            "<H1>Hello WWW</H1>\n" +
             "</BODY></HTML>");
  }
}
```

URL: http://localhost:8080/servlet/HelloWWW

HelloWWW Output:

4.4.3 Session Handling

It is essential to track client's requests. To perform this task, Java servlets offers two different ways:

1. It is possible to save information about client state on the server using a *Session* object
2. It is possible to save information on the client system using cookies.

*A **session** is sequence of HTTP requests, from the same client, over a period of time.*

HTTP is a **stateless protocol.** If a client makes a **series** of requests on a server, HTTP provides no help whatsoever to determine if those requests originated from the **same** client. There is no way in HTTP to link two separate requests to the same client. Hence, there is no way to maintain state between client requests in HTTP.

You need to maintain the state on the web for e-commerce type of applications. Just like other software systems, web applications want and need state. The classic web application example is the shopping cart that maintains a list of items you wish to purchase at a web site. The shopping cart's state is the items in the shopping basket at any given time. This state, or shopping items, needs to be maintained over a series of client requests. HTTP alone cannot do this; it needs help.

Now the question arises, for how long can you maintain the state of the same client? Of course, this figure is application-dependent and brings into play the concept of a web session. If a session is configured to last for 30 minutes, once it has expired the client will need to start a new session. Each session requires a unique identifier that can be used by the client.

There are various ways through which you maintain the state.

- Hidden Form Fields
- URL Rewriting
- Session Handling
- Cookies.

Hidden Form Fields

Hidden form fields are HTTP tags that are used to store information that is invisible to the user. In terms of session tracking, the hidden form field would be used to hold a client's unique session id that is passed from the client to the server on each HTTP request. This way the server can extract the session id from the submitted form, like it does for any of form field, and use it to identify which client has made the request and act accordingly.

For example, using servlets you could submit the following search form:

```
<form method="post" action="/servlet/search">
  <input type="text" name="searchtext">
  <input type="hidden" name="sessionid" value="1211xyz">
  ...
</form>
```

When it is submitted to the servlet registered with the name ***search***, it pulls out the sessionid from the form as follows:

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
{
  ...
  String theSessionId = request.getParameterValue("sessionid");
  if( isAllowToPerformSearch(theSessionId) )
  {

  }
  ...

 }
```

In this approach' the search servlet gets the session id from the hidden form field and uses it to determine whether it allows performing any more searches.

Hidden form fields implement the required ***anonymous*** session tracking features the client needs but not without cost. For hidden fields to work the client must send a hidden form field to the server and the server must always return that same hidden form field. This tightly coupled dependency between client requests and server responses requires sessions involving hidden form fields to be an unbreakable chain of dynamically generated web pages. If at any point during the session the client accesses a static page that is not point of the chain, the hidden form field is lost, and with it the session is also lost.

URL Rewriting

URL rewriting stores session details as part of the URL itself. You can see below how we look at request information for our search servlet:

i) http://www.archana.com/servlet/search
ii) http://www.archana.com/servlet/search/23434abc
iii) http://www.archana.com/servlet/search?sessionid=23434abc

For the original servlet [i] the URL is clean. In [ii] we have URL re-written at the server to add extra path information as embedded links in the pages we send back to the client. When the client clicks on one of these links, the search servlet will do the following:

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
{
  ...
  String sessionid = request.getPathInfo(); // return 2343abc from [ii]
  ...
}
```

Extra path information work for both GET and POST methods involved from inside as well as outside of forms with static links.

Technique [iii] simply re-writes the URL with parameter information that can be accessed as follows:
request.getParameterValue("sessionid");

URL re-writing, like, hidden forms provide a means to implement anonymous session tracking. However, with URL rewriting you are not limited to forms and you can re-write URLs in static documents to contain the required session information. But URL re-writing suffers from the same major disadvantage that hidden form fields do, in that they must be dynamically generated and the chain of HTML page generation cannot be broken.

Session object

A *HttpSession* object (derived from *Session* object) allows the servlet to solve part of the HTTP stateless protocol problems. After its creation a *Session* is available until an explicit invalidating command is called on it (or when a default timeout occurs). The same *Session* object can be shared by two or more cooperating servlets. This means each servlet can track client's service request history.

A servlet accesses a *Session* using the *getSession()* method implemented in the *HttpServletRequest* interface.
getSession() method returns the *Session* related to the current *HttpServletRequest*.

Note:

1. It is important to remember that each instance of *HttpServletRequest* has its own *Session*. If a *Session* object has not been created before, *getSession()* creates a new one.
2. *HttpSession* interface implements necessary methods to manage with a session.

Following are the various methods related to session tracking:

public abstract String getId( ): Returns a string containing session's name. This name is unique and is set by *HttpSessionContext()*.

public abstract void putValue (String Name, Object Value): Connect the object *Value* to the Session object identified by *Name* parameter. If another object has been connected to the same session before, it is automatically replaced.

public abstract Object getValue (String name): Returns the object currently held by current session. It returns a *null* value if no object has been connected before to the session.

public abstract void removeValue (String name): Remove, if existing, the object connected to the session identified by the *Name* parameter.

public abstract void invalidate( ): Invalidate the session.

Cookies

A cookie is a text file with a name, a value and a set of attributes.

Using JSDK (Java Servlet Development Kit) it is possible to save client's state sending cookies. Cookies are sent from the server and saved on client's system. On client's system cookies are collected and managed by the web browser. When a cookie is sent, the server can retrieve it in a successive client's connection. Using this strategy it is possible to track client's connections history.

Cookie class of Java is derives directly from the *Object* class. Each Cookie object instance has some attributes like max age, version, server identification, comment.

Below are some cookie methods:

public Cookie (String Name, String Value): Cookie class' constructor. It has two parameters. The first one is the name that will identity the cookie in the future; the second one, *Value*, is a text representing the cookie value. Notice that *Name* parameter must be a "token" according to the standard defined in RFC2068 and RFC2109.

Public String getName( ): Returns cookie's name. A cookie name is set when the cookie is created and can't be changed.

public void setValue(String NewValue): This method can be used to set or change cookie's value.

public String getValue( ): Returns a string containing cookie's value.

public void setComment(Sting Comment): It is used to set cookie's comment attribute.

public String getComment( ): Returns cookie's comment attribute as a string.

public void setMaxAge (int MaxAge): Sets cookie's max age in seconds. This means client's browser will delete the cookie in *MaxAge* seconds. A negative value indicate the cookie has to be deleted when client's web browser exits.

public int getMaxAge( ): Returns cookie's max age.

☞ Check Your Progress 3

1) What are the advantages of Servlets?
……
……

2) What is session tracking?
……
……

3) What is the difference between doGet() and doPost()?
……
……

4) How does HTTP Servlet handle client requests?
……
……

4.5 JAVA BEANS

Java Beans are reusable software component model which allow a great flexibility and addition of features in the existing piece of software. You will find it very interesting and useful to use them by linking together the components to create applets or even new beans for reuse by others. Graphical programming and design environments often called builder tools give a good visual support to bean programmers. The builder tool does all the work of associating of various components together.

A "JavaBeans-enabled" builder tool examines the Bean's patterns, discern its features, and exposes those features for visual manipulation. A builder tool maintains Beans in a palette or toolbox. You can select a Bean from the toolbox, drop it into a form, modify its appearance and behaviour, define its interaction with other Beans, and compose it into applets, application, or new Bean. All this can be done without writing a line of code.

Definition: *A Java Bean is a reusable software component that can be visually manipulated in builder tools*

To understand the precise meaning of this definition of a Bean, you must understand the following terms:

- Software component
- Builder tool
- Visual manipulation.

Reusable software components are designed to apply the power and benefit of reusable, interchangeable parts from other industries to the field of software construction. Other industries have long profited from reusable components. Reusable electronic components are found on circuit boards. A typical part in your car can be replaced by a component made from one of many different competing manufacturers. Lucrative industries are built around parts construction and supply in most competitive fields. The idea is that standard interfaces allow for interchangeable, reusable components.

Reusable software components can be simple like familiar push buttons, text fields list boxes, scrollbars, dialogs boxes etc.

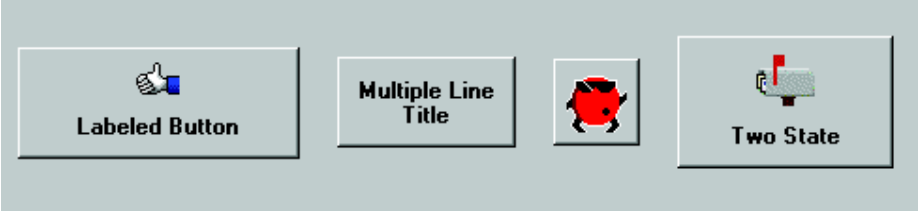


Figure 4: Button Beans

Features of JavaBeans

- Individual Java Beans will vary in functionality, but most share certain common defining features.
- Support for **introspection** allowing a builder tool to analyze how a bean works.
- Support for **customization** allowing a user to alter the appearance and behaviour of a bean.
- Support for **events** allowing beans to fire events, and informing builder tools about both the events they can fire and the events they can handle.
- Support for **properties** allowing beans to be manipulated programmatically, as well as to support the customization mentioned above.
- Support for **persistence** allowing beans that have been customized in an application builder to have their state saved and restored. Typically persistence is used with an application builder's save and load menu commands to restore any work that has gone into constructing an application.

It is not essential that Beans can only be primarily with builder tools. Beans can also be manually manipulated by programmatic interfaces of Text tools. All key APIs,

including support for events, properties, and persistence, have been designed to be easily read and understood by human programmers as well as by builder tools.

BeanBox

BeanBox is a utility from the JavaBeans Development Kit (*BDK).* Basically the BeanBox is a test container for your JavaBeans. It is designed to allow programmers to preview how a Bean created by user will be displayed and manipulated in a builder tool. The BeanBox is not a builder tool.It allows programmers to preview how a bean will be displayed and used by a builder tool.

Example of Java Bean Class

Create a new SimpleBean.java program containing the code given below:

/WEB-INF/classes/com/myBean/bean/test/ folder

```
package com.mybean.bean.test;
public class SimpleBean implements java.io.Serializable
{
        /* Properties */
        private String ename = null;
        private int eage = 0;
        /* Empty Constructor */
        public SimpleBean() {}
        /* Getter and Setter Methods */
        public String getEname()
       {
                return ename;
        }
        public void setEname(String s)
       {
                ename = s;
        }
        public int getAge()
       {
                return eage;
        }

        public void setAge(int i)
        {
                eage = i;
        }
}
```

The class SimpleBean implements java.io.Serializable interface.

There are two variables which hold the name and age of a employee. These variables inside a JavaBean are called properties. These properties are private and are thus not directly accessible by other classes. To make them accessible, methods are defined.

Compiling JavaBean

You can compile JavaBean like you compile any other Java Class file. After compilation, a SimpleBean.class file is created and is ready for use.
Finally you can say, JavaBeans are Java classes which adhere to an extremely simple coding convention. All you have to do is to implement java.io.Serializable interface,

use a public empty argument constructor and provide public methods to get and set the values of private variables (properties).

☞ Check Your Progress 4

1) What are JavaBeans?
……

……

2) What do you understand by Introspection?
……

……

3) What is the difference between a JavaBean and an instance of a normal Java class?
……

……

4) In a single line answer what is the main responsibility of a Bean Developer.
……

……

4.6 SUMMARY

In this unit you have learn Java JDBC are used to connect databases and work with it. JDBC allows the integration of SQL call into a general programming environment. Vender specific drivers are needed in JDBC programming to make a code portable. getConnection() method of DriverManager class is used to create connection object. By PreparedStatement similar queries can be performed in efficient way. Tuples in ResultSet are accessed by using next () method.

Distributed programming can be done using Java RMI (Remote Methods Invocation). Every RMI program has two sets one for client side and other for server side. Remote interface is essentially implemented in RMI programs.

Servlets are used with web servers. The HttpServlet class is an extension of GenericServlet that include methods for handling HTTP. HTTP request for specific data are handled by using doGet () and doPost () methods of HttpServlet. HttpSession objects are used to solve the problems of HTTP caused due to the stateless nature of HTTP.

Java Beans are a new dimension in software component model. Beans provide introspection and persistency.

4.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Programs are written according to the JDBC driver API would talk to the JDBC driver Manager. JDBC driver Manager would use the drivers that were plugged into it at that moment to access the actual database.

2) Four types of drivers are there for JDBC. They are:
   a) JDBC-ODBC Bridge Driver: Talks to an ODBC connection using largely non-Java code.

b) Native API,(Partly Java) : Uses foreign functions to talk to a non-Java API; the non-Java component talks to the database any way it likes.(Written partly in Java and partly in native code, that communicate with the native API of a database.

c) Net Protocol (pure Java): Talks to a middleware layer over a network connection using the middleware's own protocol (Client library is independent of the actual database).

d) Native Protocol (pure Java): Talks directly to the RDBMS over a network connection using an RDBMS-specific protocol. (pure Java library that translates JDBC requests directly to a database-specific protocol.

3) //It is assumed that Employee database is already existing.

```
con.setAutoCommit(false);
The Statement stmt = con.createStatement();
stmt.executeUpdate("INSERT INTO Employee VALUES('Archana', 'xyz',
30000)" );
con.rollback();
stmt.executeUpdate("INSERT INTO Sells Employee('Archie', 'ABC', 40000)" );
con.commit();
con.setAutoCommit(true);
```

4)

i. rs.next() is used to move to next row of the table.
ii. rs.getString(3) will give the value of the third attribute in the current row. In this program the third attribute is Address.
iii The statement "Class.forName("sun.jdbc.odbc.JdbcOdbcDriver")" is essential for the program execution. Database driver cannot be included in JDBC runtime method. If you want to remove this statement from program you must have to provide jdbc.divers property using command line parameter.

Check Your Progress 2

1) When you invoke a remote method on a remote object' the remote method calls a method of java programming language that is encapsulated in a surrogate object called Stub. The Following information is built by Stub:

i. An identifier of the remote object to be used.
ii. A description of the method to be called.
iii. The marshalled parameters.

2) The basic actions performed by receiver object on server side are:

i. Unmarshaling of the parameters.
ii. Locating the object to be called.
iii. Calling the desired method
iv. Capturing the marshals and returning the value or exception of the call.
v. Sending a package consisting of the marshalled return data back to the stub on the client.

3) RMI Registry is required to provide RMI Naming Service which is used to simplify the location of remote objects. The naming service is a JDK utility called rmiregistry that runs at a well-known address.

Check Your Progress 3

1) Java servlets are more efficient, easier to use, more powerful, more portable, and cheaper than traditional CGI than many alternative CGI-like technologies.

Following are the advantages of Services

Efficient. With servlets, the Java Virtual Machine stays up, and each request is handled by a lightweight Java thread, not a heavyweight operating system process. Similarly, in traditional CGI, if there are *N* simultaneous requests to the same CGI program, then the code for the CGI program is loaded into memory N times. With servlets, however, there are *N* threads but only a single copy of the servlet class.

Convenient. Servlets have an extensive infrastructure for automatically parsing and decoding HTML form data, reading and setting HTTP headers, handling cookies, tracking sessions, and many other such utilities.

Powerful. Java servlets let us easily do several things that are difficult or impossible with regular CGI. For example servlets can talk directly to the Web server (regular CGI programs can't). This simplifies operations that need to look up images and other data stored in standard places. Servlets can also share data among each other, making useful things like database connection pools easy to implement.

Portable. Servlets are written in Java and follow a well-standardized API. Consequently, servlets written for, say I-Planet Enterprise Server can run virtually unchanged on Apache, Microsoft IIS, or WebStar. Servlets are supported directly or via a plugin on almost every major Web server.

2) Session tracking is a concept which allows you to maintain a relationship between two successive requests made to a server on the Internet by the same client. Servlet's Provide an API named HttpSession is used in session tracking programming.

3) i. The doGet() method is limited with 2k of data only to be sent, but this limitation is not with doPost() method.
ii. A request string for doGet() looks like the following:

http://www.abc.com/svt1?p1=v1&p2=v2&...&pN=vN
But doPost() method does not need a long text tail after a servlet name in a request.

4) An HTTP Servlet handles client requests through its service method, which supports standard HTTP client requests. The service method dispatches each request to a method designed to handle that request.

Check Your Progress 4

1) Java Beans are components that can be used to assemble a larger Java application. Beans are basically classes that have properties, and can trigger events. To define a property, a bean writer provides accessor methods which are used to get and set the value of a property.

2) Introspection is the process of implicitly or explicitly interrogating Bean.
Implicit Introspection: Bean runtime supplies the default introspection mechanism which uses the Reflection API and a well established set of Naming Conventions.

Explicit Introspection: A bean designer can provide additional information through an object which implements the Bean Info interface.

In a nutshell, Introspection is a how a builder or designer can get information about how to connect a Bean with an Application.

3) The difference in Beans from typical Java classes is ***introspection***. Tools that recognize predefined patterns in method signatures and class definitions can "look inside" a Bean to determine its properties and behavior. A Bean's state can be manipulated at the time it is being assembled as a part within a larger application. The application assembly is referred to as ***design time*** in contrast to ***run time***. In order for this scheme to work, method signatures within Beans must follow a certain pattern for introspection tools to recognize how Beans can be manipulated, both at design time, and run time.

4) To minimize the effort in turning a component into a Bean.