100004/IT400A
PRINCIPLES OF OBJECT-ORIENTED TECHNIQUES

Modules

1. **Object-Oriented Concepts**
2. **Method overloading**
3. **Inheritance**
4. **Exception handling and Multithreaded Programming**
5. **Event Handling, AWT and UML**

(Hours-45)

TEXT BOOK

1. Herbert Schildt, Java: The Complete Reference, 8/e, Tata McGraw Hill, 2011.

2. Bahrami A., Object Oriented Systems Development using the Unified ModelingLanguage, McGraw Hill, 1999.

REFERENCES

1. **Flanagan D., Java in A Nutshell, 5/e, O'Reilly, 2005.**
2. **Sierra K., Head First Java, 2/e, O'Reilly, 2005.**
3. **Balagurusamy E., Programming JAVA a Primer, 5/e, McGraw Hill, 2014.**
4. **Barclay K., J. Savage, Object Oriented Design with UML and Java, Elsevier, 2004.**
5. **James Rumbaugh.,Unified Modeling Language Reference Manual,Addison-WesleyProfessional,2005**

Module-1 (Hours-10)

Object-Oriented Concepts:

- **Object-Oriented Programming vs Procedure-Oriented Programming,**
- **Procedural Languages**
- **The Object-Oriented Approach**
- **Characteristics of Object-Oriented Languages**
- **Objects**
- **Classes**
- **Inheritance**
- **Reusability**
- **Creating New Data Types**
- **Polymorphism and Overloading ,**
- **Object oriented concepts in Java**

Java Overview:

- **Java virtual machine,**
- **data types,**
- **operators,**
- **control statements,**
- **Classes fundamentals,**
- **objects,**
- **methods,**
- **constructors,**
- **this keyword,**
- **Garbage collection**

Module-2 (Hours-8)

- **Method overloading:**
- **Overloading Methods,**
- **Overloading Constructors,**
- **Using Objects as Parameters,**
- **Call by value and Call by reference,**
- **Access control,**
- **use of static and final keywords,**
- **Nested and Inner classes**

Module-3 (Hours-8)

Inheritance:

- **Derived Class and Base Class,**
- **Usage of super keyword,**
- **Creating a Multilevel Hierarchy,**
- **Method Overriding,**
- **Dynamic Method Dispatch,**
- **Using Abstract Classes,**
- **Using final with Inheritance,**
- **Definition and application of Packages and Interfaces**

Module-4 (Hours-9)

Exception handling and Multithreaded Programming:

- **Fundamentals of exception handling,**
- **Exception Types,**
- **Using try and catch, throw, throws, finally,**
- **Java's Built-in Exceptions,**
- **Creating Exception subclasses,**
- **the Java Thread Model,**
- **Creating a Thread,**
- **Creating Multiple Threads,**
- **Thread Priorities, Synchronization, Interthread Communication**

Module-5 (Hours-10)

Event Handling, AWT and UML:

- **Event Handling-delegation event model,**
- **event classes, sources, listeners.**
- **String class - basics.**
- **Applet basics and methods,**
- **AWT- working with frames, graphics, color, font.**
- **AWT Control fundamentals.**

- **Swing overview,**
- **Introduction to Object Oriented Modelling ,**
- **Unified Modeling Language,**
- **UML class diagram,**
- **Use-case diagram,**
- **Familiarisation of UMLtools,**
- **Case study**

QUESTION PAPER PATTERN

Maximum Marks: 100 **Exam Duration: 3 hours**

The question paper shall consist of two parts: Part A and Part B

Part A contain **10 questions with 2 questions from each module, having 3 marks for each question.**

Students should answer all questions. (10×3=30 marks).

Part B contains 2 questions from each module of which student should answer any one. Each question can have maximum 2 sub-divisions and carry 14 marks. (5×14=70 marks).

Course Outcomes:

Students will be able to:

1. Describe the specific capabilities of Object-Oriented paradigm compared to procedure oriented paradigm.

2. Explain the use of object-oriented concepts to analyse the given problem.

.

.

3. Describe the different inheritance features available in Java.

4. Construct robust programs using Exception Handling.

6. Construct applets utilising multithreading, event handlingand Graphical User Interface, also model the problem scenarios using UML diagrams

Module-1

Object-Oriented Concepts

Differences between Procedural and Object-Oriented Programming

Two Programming Paradigms

- Each statement in the program tells the computer to do something.

<u>Division into Functions</u>

- Large size programs are divided into smaller programs known as **functions**. These functions can call one another.

- Objects can communicate **with each other** through functions.
- An ***object*** is anything that can be represented by data in a computer's memory and manipulated by a computer program.

Procedural Programming

- **In the procedural programming languages such as FORTRAN and COBOL, programs are divided into a number of segments known as subprograms.**
- **Figure 1.6 describes a program of procedural type.**
- **It shows different subprograms accessing the same global data. Here also, the data is not fully protected.**
- **The control of program is transferred using unsafe goto statement.**
- **Data is global and all the subprograms share the same data.**
- **These languages are used for developing medium sized applications.**

Fig. 1.6 Program in procedural programming

Procedural Programming:

- **Procedural Programming can be defined as a programming model which is derived from structured programming, based upon the concept of calling procedure.**
- **Procedures, also known as routines, subroutines or functions, simply consist of a series of computational steps to be carried out.**
- **During a program's execution, any given procedure might be called at any point, including by other procedures or itself.**

Languages used in Procedural Programming:

1. **FORTRAN,**
2. **ALGOL,**
3. **COBOL,**
4. **BASIC,**
5. **Pascal and**
6. **C.**

Object Oriented Programming:

- **Object oriented programming can be defined as a programming model which is based upon the concept of objects.**
- **Objects contain data in the form of attributes and code in the form of methods.**
- **In object oriented programming, computer programs are designed using the concept of objects that interact with real world.**
- **Object oriented programming languages are various but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types.**

Languages used in Object Oriented Programming:

- **Java,**
- **C++,**
- **C#,**
- **Python,**
- **PHP,**
- **JavaScript,**
- **Ruby,**
- **Perl,**

- **Objective-C,**
- **Dart,**
- **Swift,**
- **Scala.**

Difference between Procedural Programming and Object Oriented Programming:

Procedure Oriented Programming (POP)	Object Oriented Programming (OOP)
Program is divided into small parts called **functions**.	Program is divided into parts called **objects**.
Importance is not given to data but to functions as well as sequence of actions to be done.	Importance is given to the data rather than procedures or functions because it works as a real world.
It follows Top Down approach.	OOP follows Bottom Up approach.
Data can move freely from function to function in the system.	Objects can move and communicate with each other through member functions.
It does not have any access specifier.	OOP has access specifiers named Public, Private, Protected, etc.
To add new data and function in POP is not so easy.	OOP provides an easy way to add new data and function.
Most function uses Global data for sharing that can be accessed freely from function to function in the system.	In OOP, data cannot move easily from function to function, it can be kept public or private so we can control the access of data.
Overloading is not possible.	Overloading is possible in the form of function overloading and operator overloading.
It doesn't have any proper way for hiding data so it is less secure.	It provides data hiding so provides more security.
Examples: C. FORTRAN, Pascal etc.	Examples: C++, JAVA, .NET etc.

top-down approach Vs bottom-up approach

- **While the <u>top-down approach</u> focuses on breaking down a big problem into smaller and understandable chunks.**
- **Eg: Mainly used by structured programming language such as COBOL, Fortran, C, etc.**
- **the <u>bottom-up approach</u> first focuses on solving the smaller problems at the fundamental level and then integrating them into a whole and complete solution.**
- **Eg: Mainly used by object oriented programming language such as C++, C#, Python.**

Procedural Languages

- **In procedural languages, the program code is written as a sequence of instructions.**
- **User has to specify "what to do" and also "how to do" (step by step procedure).**
- **These instructions are executed in the sequential order.**
- **These instructions are written to solve specific problems.**
- **Examples of Procedural languages:**
- **FORTRAN, COBOL, ALGOL, BASIC, C and Pascal.**

The Object-Oriented Approach

- **The OOP concept was basically designed to overcome the drawback of the procedural programming methodologies, <u>which were not so close to real-world applications</u>.**
- **We can say that Object-oriented approach <u>is very close to the real-world</u> and <u>its applications</u> because the state and behaviour of these classes and objects are almost the same as real-world objects.**

- **The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural approach.**
- **OOP treats data as a critical element in the program development and does not allow it to flow freely around the system.**
- **It ties data more closely to the functions that operate on it, and protects it from accidental modification from outside functions.**
- **OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.**
- **The organization of data and functions in object-oriented program is shown in figure.**

Fig: Organization of data and function in OOP

- **The data of an object can be accessed only by the functions associated with the object.**
- **However, functions of one object can access the functions of other objects.**

General concepts of OOP

1. Class & Object

Objects

- **Objects are primary run-time entities in an object-oriented programming.**
- **An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.**
- **Figure describes various objects.**
- **Objects occupy space in memory.**
- **Every object has its own properties or features.**

Given below are some objects that we use in our daily life

Fig. 1.10 Commonly available objects

What is an object?

OBJECTS: STUDENT

DATA
- Name
- Date-of-birth
- Marks

FUNCTIONS
- Total
- Average
- Display
- .....

representing an object

Classes

- **The mechanism that allows you to combine data and the function in a single unit is called a class.**
- **Classes are user-defined data types**
- **Once a class is defined, you can declare variables of that type. A class variable is called object or instance.**
- **A class is grouping of objects having identical properties, common behavior, and shared relationship. i.e a class is a collection of similar objects.**
- **Classes are generally declared using the keyword class, with the following format:**

The general form of a class definition

```
class class_name
 {
        variable declaration;
        function declaration;
}
```

- **Classes define types of data structures and the functions that operate on those data structures.**
- **A class is a definition of an object.**
- **Remember : A class is a type, and an object of this class is just a variable.**

Class Interface Diagram

- **Once a class has been declared, the programmer can create a number of objects associated with that class.**
- **In Figure shows some common objects, their properties, and the actions they perform have been described.**
- **A car can be distinguished from another car on the basis of its properties, for example, company, model, color, capacity etc. and also on the basis of its actions for example, speed, average, break etc.**

Fig. 1.11 Objects and their properties

- **Figure 1.12 describes different classes and related data that come under them**

Fig. 1.12 Classes and their members

(2) Data Abstraction

- **Abstraction directs to the procedure of representing essential features without including the background details or explanation about that feature..**
- **Classes use the theory of abstraction and are defined as a list of abstract properties such as size, cost, height, and few functions to operate on these properties.**

Example :1

- **Let's take one real life example of a TV,**
  - **which you can turn on and off,**
  - **change the channel,**
  - **adjust the volume, and**
  - **add external components such as speakers, VCRs, and DVD players,**
- **BUT you do not know its internal details,**
- **that is, you do not know how it receives signals over the air or through a cable,**
- **how it translates them, and finally displays them on the screen.**

- **Thus, we can say a television clearly separates its internal implementation from its external interface and**
- **you can play with its interfaces like the power button, channel changer, and volume control without having zero knowledge of its internals.**

Example: 2

- **For example, a computer as shown in Figure 1.14 is made of various parts such as cpu, keyboard, and so on.**
- **We think it as a single unit, but it has several sub units.**
- **These units when combined together perform the required task.**
- **By assembling subparts we can make a single system.**

(a) Computer as a single unit (b) Different components of a computer

Fig. 1.14 Computer and its parts

Example :3

- **ATM is a good example of abstraction,**

we doesn't know what are all the internal processes that are going when we access it..

(3) Encapsulation

- **The packing of data and functions into a single component is known as encapsulation.**
- **Java supports the features of encapsulation using classes.**
- **The data is not accessible by outside functions. Only those functions that are able to access the data are defined within the class.**
- **These functions prepare the interface between the object's data and the program.**
- **With encapsulation data hiding can be accomplished. Data hiding is an important feature.**

(4) Inheritance

- **Inheritance is the method by which objects of one class get the properties of objects of another class.**
- **In object-oriented programming, inheritance provides the thought of reusability and reduce its redundancy.**

example

- **Acquiring some of the common qualities from parents (for instance eyes like mother, nose like father etc..) is called inheritance.**
- **It can also be defined as acquiring some property from others is called inheritance.**
- **based on the sources its different.**
- **The programmer can add new properties to the existing class without changing it. This can be achieved by deriving a new class from the existing one. The new class will possess features of both the classes.**
- **The actual power of inheritance is that it permits the programmer to reuse a class .**
- **i.e , inheritance is the feature that permits the reuse of an existing class to make a new class.**
- **The Figure 1.16 shows an example of inheritance.**
- **In Figure 1.16 red, yellow and blue are the main colours.**
- **Orange colour is created from the combination of red and yellow,**
- **green is created from yellow and blue, and violet is created from red and blue.**
- **Orange colour has attributes of both red and yellow, which produces a new effect. Thus, many combinations are possible.**

Fig. 1.16 Inheritance

(5) Polymorphism

- **Polymorphism allows the same function to act differently in different classes.**
- **Polymorphism makes possible the same functions to act differently on different classes as shown in Figure 1.17. It is an important feature of OOP concept and has the ability to take more than one form.**

Fig. 1.17 Polymorphism in OOP

Some important points to know about OOP:

1. **OOP treats data as a critical element.**
2. **Importance is on data rather than procedure.**
3. **Decomposition of the problem into simpler modules.**
4. **Doesn't allow data to freely flow in the entire system, ie localized control flow.**
5. **Data is protected from external functions.**

Advantages of OOPs –

- **It models the real world very well.**
- **With OOP, programs are easy to understand and maintain.**
- **OOP offers code reusability. Already created classes can be reused without having to write them again.**
- **OOP facilitates the quick development of programs where parallel development of classes is possible.**
- **With OOP, programs are easier to test, manage and debug.**

Benefits of object oriented programming (OOPs)

1. Reusability: In OOP's programs functions and modules that are written by a user can be reused by other users without any modification.

2. Inheritance: Through this we can eliminate redundant code and extend the use of existing classes.

3. Data Hiding: The programmer can hide the data and functions in a class from other classes. It helps the programmer to build the secure programs.

4. Reduced complexity of a problem: The given problem can be viewed as a collection of different objects. Each object is responsible for a specific task. The problem is solved by interfacing the objects. This technique reduces the complexity of the program design.

5. Easy to Maintain and Upgrade: OOP makes it easy to maintain and modify existing code as new objects can be created with small differences to existing ones. Software complexity can be easily managed.

6. Message Passing: The technique of message communication between objects makes the interface with external systems easier.

7. Modifiability: it is easy to make minor changes in the data representation or the procedures in an OO program. Changes inside a class do not affect any other part of a program, since the only public interface that the external world has to a class is through the use of methods.

Characteristics of Object – Oriented Languages

KEY CONCEPTS OF OBJECT-ORIENTED PROGRAMMING

- **There are several fundamental or key concepts in object-oriented programming.**
- **Some of these are shown in Figure and are discussed below:**

(1) Objects

(2) Classes

(3) Method

(4) Data Abstraction

(5) Encapsulation

(6) Inheritance

(7) Polymorphism

(8) Message Passing

(9) Reusability

1. Objects

- **Objects are primary run-time entities in an object-oriented programming.**
- **An *object* is anything that can be represented by data in a computer's memory and manipulated by a computer program.**
- **Figure describes various objects.**

- **Objects occupy space in memory.**
- **Every object has its own properties or features.**

Given below are some objects that we use in our daily life

Fig. 1.10 Commonly available objects

What is an object?

OBJECTS: STUDENT

DATA
- Name
- Date-of-birth
- Marks

FUNCTIONS
- Total
- Average
- Display
- .....

representing an object

(2) Classes

- **The mechanism that allows you to combine data and the function in a single unit is called a class.**
- **Classes are user-defined data types**
- **Once a class is defined, you can declare variables of that type. A class variable is called object or instance.**
- **A class is grouping of objects having identical properties, common behavior, and shared relationship. i.e a class is a collection of similar objects.**
- **Classes are generally declared using the keyword class, with the following format:**
- **The general form of a class definition**

```
class class_name

 {

        variable declaration;

        function declaration;

}
```

- **Classes define types of data structures and the functions that operate on those data structures.**
- **A class is a definition of an object.**
- **Remember : A class is a type, and an object of this class is just a variable.**

Class Interface Diagram

- **Once a class has been declared, the programmer can create a number of objects associated with that class.**
- **In Figure shows some common objects, their properties, and the actions they perform have been described.**
- **A car can be distinguished from another car on the basis of its properties, for example, company, model, color, capacity etc. and also on the basis of its actions for example, speed, average, break etc.**

Fig. 1.11 Objects and their properties

- **Figure 1.12 describes different classes and related data that come under them**

Fig. 1.12 Classes and their members

(3) Method

- **The operations that are required for an object are to be defined in a class.**
- **All objects in a class perform certain common actions or operations.**
- **The action of the object depends upon the member function defined within its class and it is also referred to as a method.**

(4) Data Abstraction

- **Abstraction directs to the procedure of representing essential features without including the background details.**
- **Classes use the theory of abstraction and are defined as a list of abstract properties such as size, cost, height, and few functions to operate on these properties.**

Example :1

- **Let's take one real life example of a TV,**
- **which you can turn on and off,**
- **change the channel,**
- **adjust the volume, and**

- **add external components such as speakers, VCRs, and DVD players,**
- **BUT <u>you do not know</u> its internal details,**
- **that is, <u>you do not know</u> how it receives signals over the air or through a cable,**
- **how it translates them, and finally displays them on the screen.**
- **Thus, we can say a television <u>clearly separates its internal implementation</u> from its external interface and**
- **you can play <u>with its interfaces</u> like the power button, channel changer, and volume control <u>without</u> <u>having zero knowledge</u> of its internals.**

Example: 2

- **For example, a computer as shown in Figure 1.14 is made of various parts such as cpu, keyboard, and so on.**
- **We think it as a single unit, but it has several sub units.**
- **These units when combined together perform the required task.**
- **By assembling subparts we can make a single system.**

(a) Computer as a single unit (b) Different components of a computer

Fig. 1.14 Computer and its parts

Example :3

- **ATM is a good example of abstraction,**

<u>we doesn't know</u> what are all the internal processes that are going when we access it..

(5) Encapsulation

- **The packing of data and functions into a single component is known as encapsulation.**
- **Java supports the features of encapsulation using classes.**

- **The data is not accessible by outside functions. Only those functions that are able to access the data are defined within the class.**
- **These functions prepare the interface between the object's data and the program.**
- **With encapsulation data hiding can be accomplished. Data hiding is an important feature.**

(6) Inheritance

- **Inheritance is the method by which objects of one class get the properties of objects of another class.**
- **In object-oriented programming, inheritance provides the thought of reusability.**

example

- **Acquiring some of the common qualities from parents (for instance eyes like mother, nose like father etc..) is called inheritance.**
- **It can also be defined as acquiring some property from others is called inheritance.**
- **based on the sources its different.**
- **The programmer can add new properties to the existing class without changing it. This can be achieved by deriving a new class from the existing one. The new class will possess features of both the classes.**
- **The actual power of inheritance is that it permits the programmer to reuse a class .**
- **i.e , inheritance is the feature that permits the reuse of an existing class to make a new class.**
- **The Figure 1.16 shows an example of inheritance.**
- **In Figure 1.16 red, yellow and blue are the main colours.**
- **Orange colour is created from the combination of red and yellow,**
- **green is created from yellow and blue, and violet is created from red and blue.**
- **Orange colour has attributes of both red and yellow, which produces a new effect. Thus, many combinations are possible.**
-

Fig. 1.16 Inheritance

(7) Polymorphism

- **Polymorphism allows the same function to act differently in different classes.**
- **Polymorphism makes possible the same functions to act differently on different classes as shown in Figure 1.17. It is an important feature of OOP concept and has the ability to take more than one form.**
- **Polymorphism accomplishes an important part in allowing objects of different classes to share the same external interface. It is possible to code a non-specific (generic) interface to a set of associated actions.**
-

Fig. 1.17 Polymorphism in OOP

(8) Message Passing

- **Object-oriented programming includes objects which communicate with each other.**
- **Programming with these objects should be followed in steps shown below:**

(1) Declaring classes that define objects and their actions.

(2) Declaring objects from classes.

(3) Implementing relation between objects.

- **Data is transferred from one object to another.**
- **Message passing as shown in Figure 1.18 contains indicating the name of the object, function, and required data elements.**
- **Objects can be created, released, and interacted with each other.**
- **An object is workable, as long as it is active.**

Fig. 1.18 Message passing

(9) Reusability

- **Object-oriented technology allows reusability of the classes by extending them to other classes using inheritance.**
- **Once a class is defined, the other programmer can also use it in their programs and add new features to the derived classes.**

- **Once verified the qualities of base classes need not be redefined. Thus, reusability saves time.**
- **In Figure 1.19**

class A is reused and class B is created. Again class B is reused and class C is created.

Fig. 1.19 Reusability

Object Oriented Programming (OOPs) Concept in Java

- **Object-Oriented Programming or OOPs refers to languages that uses objects in programming.**
- **Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism etc in programming.**
- **The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.**

OOPs Concepts:

1. **Object**
2. **Class**
3. **Method**
4. **Abstraction**
5. **Encapsulation**
6. **Inheritance**
7. **Polymorphism**
8. **Message Passing**

1. Object:

- **It is a basic unit of Object Oriented Programming and represents the real life entities.**
- **A typical Java program creates many objects, which as you know, interact by invoking methods.**
- **An object consists of:**

1. **State : It is represented by attributes of an object. It also reflects the properties of an object.**
2. **Behavior : It is represented by methods of an object. It also reflects the response of an object with other objects.**
3. **Identity : It gives a unique name to an object and enables one object to interact with other objects.**

Example of an object: dog

2. Class:

A class is a user defined blueprint or prototype from which objects are created.

It represents the set of properties or methods that are common to all objects of one type.

In general, class declarations can include these components, in order:

1. **Modifiers: A class can be public or has default access**
2. **Class name: The name should begin with a initial letter (capitalized by convention).**

3. **Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.**

4. **Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.**

5. **Body: The class body surrounded by braces, { }.**

3. Method/function:

- **A method is a collection of statements that perform some specific task and return result to the caller.**
- **A method can perform some specific task without returning anything.**
- **Methods allow us to reuse the code without retyping the code.**
- **In Java, every method must be part of some class which is different from languages like C, C++ and Python.**
- **Methods are time savers and help us to reuse the code without retyping the code.**

Method Declaration

- **In general, method declarations has six components:**

1. **Access Modifier: Defines access type of the method i.e. from where it can be accessed in your application.**

- **In Java, there 4 type of the access specifiers.**
    - **public: accessible in all class in your application.**
    - **protected: accessible within the package in which it is defined and in its subclass(es)(including subclasses declared outside the package)**
    - **private: accessible only within the class in which it is defined.**
    - **default (declared/defined without using any modifier): accessible within same class and package within which its class is defined.**

2.The return type: The data type of the value returned by the method or void if does not return a value.

3. Method Name: the rules for field names apply to method names as well, but the convention is a little different.

4. Parameter list: Comma separated list of the input parameters are defined, preceded with their data type, within the enclosed parenthesis. If there are no parameters, you must use empty parentheses ().

5. Exception list: The exceptions you expect by the method can throw, you can specify these exception(s).

6. Method body: it is enclosed between braces. The code you need to be executed to perform your intended operations.

(4) Data Abstraction

- **Abstraction directs to the procedure of representing essential features without including the background details or explanation about that feature..**
- **Classes use the theory of abstraction and are defined as <u>a list of abstract properties</u> such as size, cost, height, and <u>few functions to operate</u> on these properties.**

Example :1

- **Let's take one real life example of a TV,**
- **which you can turn on and off,**
- **change the channel,**
- **adjust the volume, and**
- **add external components such as speakers, VCRs, and DVD players,**
- **BUT <u>you do not know</u> its internal details,**
- **that is, <u>you do not know</u> how it receives signals over the air or through a cable,**
- **how it translates them, and finally displays them on the screen.**
- **Thus, we can say a television <u>clearly separates its internal implementation</u> from its external interface and**
- **you can play <u>with its interfaces</u> like the power button, channel changer, and volume control <u>without</u> <u>having zero knowledge</u> of its internals.**

Example: 2

- **For example, a computer as shown in Figure 1.14 is made of various parts such as cpu, keyboard, and so on.**
- **We think it as a single unit, but it has several sub units.**
- **These units when combined together perform the required task.**
- **By assembling subparts we can make a single system.**

(a) Computer as a single unit (b) Different components of a computer

Fig. 1.14 Computer and its parts

Example :3

- **ATM is a good example of abstraction,**

we doesn't know what are all the internal processes that are going when we access it..

(5) Encapsulation

- **The packing of data and functions into a single component is known as encapsulation.**
- **Java supports the features of encapsulation using classes.**
- **The data is not accessible by outside functions. Only those functions that are able to access the data are defined within the class.**
- **These functions prepare the interface between the object's data and the program.**
- **With encapsulation data hiding can be accomplished. Data hiding is an important feature.**

(6) Inheritance

- **Inheritance is the method by which objects of one class get the properties of objects of another class.**
- **In object-oriented programming, inheritance provides the thought of reusability and reduce its redundancy.**

example

- **Acquiring some of the common qualities from parents (for instance eyes like mother, nose like father etc..) is called inheritance.**
- **It can also be defined as acquiring some property from others is called inheritance.**
- **based on the sources its different.**
- **The programmer can add new properties to the existing class without changing it. This can be achieved by deriving a new class from the existing one. The new class will possess features of both the classes.**
- **The actual power of inheritance is that it permits the programmer to reuse a class .**

- **i.e , inheritance is the feature that permits the reuse of an existing class to make a new class.**
- **The Figure 1.16 shows an example of inheritance.**
- **In Figure 1.16 red, yellow and blue are the main colours.**
- **Orange colour is created from the combination of red and yellow,**
- **green is created from yellow and blue, and violet is created from red and blue.**
- **Orange colour has attributes of both red and yellow, which produces a new effect. Thus, many combinations are possible.**

Fig. 1.16 Inheritance

(7) Polymorphism

- **Polymorphism allows the same function to act differently in different classes.**
- **Polymorphism makes possible the same functions to act differently on different classes as shown in Figure 1.17. It is an important feature of OOP concept and has the ability to take more than one form.**

Fig. 1.17 Polymorphism in OOP

(8) Message Passing

- **Object-oriented programming includes objects which communicate with each other.**
- **Programming with these objects should be followed in steps shown below:**

(1) Declaring classes that define objects and their actions.

(2) Declaring objects <u>from classes</u>.

(3) Implementing <u>relation between objects</u>.

- **Data is transferred <u>from</u> one object to another.**
- **Message passing as shown in Figure 1.18 contains indicating the name of the object, function, and required data elements.**
- **Objects can be created, released, and interacted with each other.**
- **An object is workable, <u>as long as it is active</u>.**

Fig. 1.18 Message passing

Introduction to Java

- **JAVA was developed by James Gosling at Sun Microsystems in the year 1991.**

- **Designed as an object-oriented , machine-independent programming language.**
- **Java is both a compiled and an interpreted language.**
- **It is a simple programming language.**
- **Java makes writing, compiling, and debugging programming easy.**
- **It helps to create reusable code and modular programs.**
- **Java is a class-based, object-oriented programming language and is designed to have as few implementation dependencies as possible.**
- **A general-purpose programming language made for developers to *write once run anywhere* that is compiled Java code can run on all platforms that support Java.**
- **Java applications are compiled to byte code that can run on any Java Virtual Machine.**
- **The syntax of Java is similar to c/c++.**

Primary/Main Features of Java

1.Platform Independent:

2. Object-Oriented Programming Language:

3. Simple:

4. Robust:

5. Secure:

6. Distributed:

7. Multithreading:

8. Portable:

1. Platform Independent: Compiler converts source code to bytecode and then the JVM executes the bytecode generated by the compiler. This bytecode can run on any platform be it Windows, Linux, macOS which means if we compile a program on Windows, then we can run it on Linux and vice versa. Each operating system has a different JVM, but the output produced by all the OS is the same after the execution of bytecode. That is why we call java a platform-independent language.

2. Object-Oriented Programming Language: Organizing the program in the terms of collection of objects is a way of object-oriented programming, each of which represents an instance of the class.

- **The four main concepts of Object-Oriented programming are:**
    - **Abstraction**
    - **Encapsulation**
    - **Inheritance**
    - **Polymorphism**

3. Simple: Java is one of the simple languages as it does not have complex features like pointers, operator overloading, multiple inheritances, Explicit memory allocation.

4. Robust: Java language is robust that means reliable. It is developed in such a way that it puts a lot of effort into checking errors as early as possible, that is why the java compiler is able to detect even those errors that are not easy to detect by another programming language. The main features of java that make it robust are garbage collection, Exception Handling, and memory allocation.

5. Secure: In java, we don't have pointers, and so we cannot access out-of-bound arrays i.e it shows ArrayIndexOutOfBoundsException if we try to do so. That's why several security flaws like stack corruption or buffer overflow is impossible to exploit in Java.

6. Distributed: We can create distributed applications using the java programming language. Remote Method Invocation and Enterprise Java Beans are used for creating distributed applications in java. The java programs can be easily distributed on one or more systems that are connected to each other through an internet connection.

7. Multithreading: Java supports multithreading. It is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.

8. Portable: As we know, java code written on one machine can be run on another machine. The platform-independent feature of java in which its platform-independent bytecode can be taken to any platform for execution makes java portable.

Why java is more popular & powerful language today?

- **Even though there are two popular language like C and C++.**
- **If we consider C language , sometime it is called middle level language.**
- **That means we can write C program in either Low-level language or High-level language.**
- **By using low-level language we can develop System Software and by using High-level language we can develop Application Software.**
- **So by using C language we can develop two powerful softwares that can run on your PC or any other platform. That is why C is a powerful language even today.**
- **But problem with these language like C and C++ is that, if you write a program using C or C++ for windows OS machine.**
- **After compilation you will get executable file .i.e .exe and this .exe file will be used for only Windows OS but not for Linux or any other OS or Platform.**
- **That is why C or C++ are called platform dependent language.**
- **But if you write a program using Java .**
- **After compilation javac compiler generate bytecode and this byte code makes Java in Platform independent language.**
- **Because of this reason Java become more powerful language today.**

Compilation process in "C"
SOURCE CODE
(.C FILE)
FOR
WINDOWS
COMPILER
MACHINE LEVEL
LANGUAGE
(.EXE FILE)

Compiler
100101011
0100101
Source Code
Executable
program
(Object Code)

First.c
First.exe
Compiler
100101011
0100101
Source Code
Executable
program
(Object Code)

- **What is meant by Platform Independent?**

  Platform Independent means writing codes in one operating system 'Windows XP' and executing that code on another platform 'Linux'

- **Bytecode**

  Bytecode is the machine language of the Java Virtual Machine, by using *bytecode* execution Java proves it's a Platform Independent Language

- Different types of JVM available on different Operating Systems

  Sun microsystems – JVM

  Microsoft – JVM

  IBM – JVM

- Same .class files can *run* on various platforms
- This is the way *bytecode* executes Java programs
- Proves Java is a **Platform Independent Language!**

Java Terminology

- **Before learning Java, one must be familiar with these common terms of Java.**

1. Java Virtual Machine(JVM): This is generally referred to as JVM. There are three execution phases of a program. They are written, compile and run the program.

- **Writing a program is done by a java programmer like you and me.**
- **The compilation is done by the JAVAC compiler which is a primary Java compiler included in the Java development kit (JDK). It takes Java program as input and generates bytecode as output.**
- **In the Running phase of a program, JVM executes the bytecode generated by the compiler.**
- **Now, we understood that the function of Java Virtual Machine is to execute the bytecode produced by the compiler. Every Operating System has a different JVM but the output they produce after the execution of bytecode is the same across all the operating systems. This is why Java is known as a platform-independent language.**

2. Bytecode in the Development process: As discussed, the Javac compiler of JDK compiles the java source code into bytecode so that it can be executed by JVM. It is saved as .class file by the compiler. To view the bytecode, a disassembler like javap can be used.

3. Java Development Kit(JDK): While we were using the term JDK, when we learn about bytecode and JVM . So, as the name suggests, it is a complete Java development kit that includes everything including compiler, Java Runtime Environment (JRE), java debuggers, java docs, etc. For the program to execute in java, we need to install JDK on our computer in order to create, compile and run the java program.

4. Java Runtime Environment (JRE): JDK includes JRE. JRE installation on our computers allows the java program to run, however, we cannot compile it. JRE includes a browser, JVM, applet supports, and plugins. For running the java program, a computer needs JRE.

5. Garbage Collector: In Java, programmers can't delete the objects. To delete or recollect that memory JVM has a program called Garbage Collector. Garbage Collectors can recollect the of objects that are not referenced. So Java makes the life of a programmer

easy by handling memory management. However, programmers should be careful about their code whether they are using objects that have been used for a long time. Because Garbage cannot recover the memory of objects being referenced.

6. ClassPath: The classpath is the file path where the java runtime and Java compiler look for .class files to load. By default, JDK provides many libraries. If you want to include external libraries they should be added to the classpath.

➢There are two ways to run and compile a java program .

1)NOTEPAD

2)ANY IDE'S LIKE a) ECLIPSE

b) NETBEANS etc

- **integrated development environment (IDE)**

- **The class doesn't contain main() method , we got Exception (No Such method Error:main)**
- **Compiler is not check, whether the class contain main() method or not.**
- **But at the run time , the JVM its responsible to check whether the class contain main() method or not.**

This is required prototype for the JVM writes:

- **main() method should be public.**
- **main () method should be static.**
- **The return type of main () method should be void.**
- **The name of main() method is main under the argument String args[]**

public static void main(String args[])

1. The order of modifiers is not important that is instead of "public static" we can take "static public" also.

We can declare "String[]" in any acceptable form.

main(String[] args)

main(String []args)

main(String args[])

3. instead of 'args' we can take any valid java identifier.

Q.1)Writing a “HELLO WORLD” Program in Java.

hello - Notepad

File Edit Format View Help

```
import java.io.*;

class hello
  {
   public static void main(String args[])
     {
       System.out.println("Hello World");
     }

  }
```

Ln 8, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac hello.java

D:\java\programs\module-1>java hello
Hello World
```

1. **import java.io.*: This means all the classes of io package can be imported. Java io package <u>provides</u> a set of input and output streams for reading and writing data to files or other input or output sources.**
2. **class: The class contains the data and methods to be used in the program. Methods define the behavior of the class. Class GFG has only one method Main in JAVA.**
3. **static void Main(): static keyword tells us that this method is accessible without instantiating the class.**
4. **void: keywords tell that this method will not return anything. The main() method is the entry point of our application.**
5. **System.in: This is the standard input stream that is used to read characters from the keyboard or any other standard input device.**

6. **System.out: This is the standard output stream that is used to produce the result of a program on an output device like the computer screen.**
7. **println(): This method in Java is also used to display text on the console. It prints the text on the console and the cursor moves to the start of the next line at the console. The next printing takes place from the next line.**

Everything in java , is represented in Class as an object including the main function.

Instructions to compile and run Java code

To compile:

```
javac helloworld.java
```

(This will create a file called *helloworld.class*)

To run:

```
java helloworld
```

Comments

- **Comments: Comments are used for explaining code and are used in a similar manner in Java or C or C++. Compilers ignore the comment entries and do not execute them. Comments can be of a single line or multiple lines.**

Single line Comments:

Syntax:

```
// Single line comment
```

Multi-line **comments:**

Syntax:

```
/* Multi line comments*/
```

Q.2) write a java program to Read & display the student information.

student - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class student
{
public static void main(String args[])
  {
    info o=new info();
    o.read();
  }
}
class info
{
public static void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Student Name");
   String name=io.next();
   System.out.println("Enter the Age");
   int age=io.nextInt();
   System.out.println("Enter the Percentage");
   float per=io.nextFloat();
   display(name,age,per);
  }

 public static void display(String name,int age, float per)
   {
        System.out.println("Name:"+name);
        System.out.println("Age:"+age);
        System.out.println("Per:"+per);
   }
}
```

Ln 13, Col 1 | 100% | Windows (CRLF) | UTF-8

Output

```
D:\java\programs\module-1>javac student.java

D:\java\programs\module-1>java student
Enter the student Name:
Anil
Enter the Register No:
101
Enter the Percentage:
75
Name:Anil
Regno:101
Per:75.0
```

Data types in Java

Data Types

- **Data types are means to identify the type of data.**
- **eg: 101 – this number has no fractional part . Therefore we can easily identify, this is an Integer type.**

Data type determine the following

- **Type of data stored**
- **Number of bytes it occupies in memory**
- **Range of data**
- **Operations that can be performed on the data.**

Data types in Java

- **Data types specify the different sizes and values that can be stored in the variable.**
- **There are two types of data types in Java:**

1. **Primitive data types: The primitive data types include boolean, char, byte, short, int, long, float and double.**
2. **Non-primitive data types: The non-primitive data types include Classes, Interfaces, and Arrays.**

There are 8 primitive data types:

Data Type	Default Value	Default size
boolean	false	1 bit
char	'\u0000'	2 byte
byte	0	1 byte
short	0	2 byte
int	0	4 byte
long	0L	8 byte
float	0.0f	4 byte
double	0.0d	8 byte

1.Boolean Data Type

- **The Boolean data type is used to store only two possible values: true and false.**
- **boolean data type represents only one bit of information either true or false, but the <u>size of the boolean</u> data type is virtual machine-dependent.**
- **Values of type boolean are not converted implicitly or explicitly (with casts) to any other type.**
- **But the programmer can easily write conversion code.**

- **Syntax:**

```
boolean booleanVar;
```

- **Size:**

```
virtual machine dependent
```

- **Values:**

```
true, false
```

- **Default Value:**

```
false
```

```java
// Java program to demonstrate boolean data type

class GeeksforGeeks {
    public static void main(String args[])
    {
        boolean b = true;
        if (b == true)
            System.out.println("Hi Geek");
    }
}
```

Output

```
Hi Geek
```

2. byte

- **The byte data type is an 8-bit signed two's complement integer.**
- **The byte data type is useful for saving memory in large arrays.**

24 in simple binary form is 00011000

- one's complement is 11100111 (inverting all bits)
- two's complement is computed by adding 1 to the one's complement value

--> 11101000 is the two's complement for -24

- **Syntax:**

```
byte byteVar;
```

- **Size:**

```
1 byte ( 8 bits )
```

- **Values:**

```
-128 to 127
```

- **Default Value:**

```
0
```

```java
// Java program to demonstrate byte data type in Java

class GeeksforGeeks {
    public static void main(String args[])
    {
        byte a = 126;

        // byte is 8 bit value
        System.out.println(a);

        a++;
        System.out.println(a);

        // It overflows here because
        // byte can hold values from -128 to 127
        a++;
        System.out.println(a);

        // Looping back within the range
        a++;
        System.out.println(a);
    }
}
```

Output:

```
126
127
-128
-127
```

3. short

- **The short data type is a 16-bit signed two's complement integer.**
- **Similar to byte, use a short to save memory in large arrays, in situations where the memory savings actually matters.**

- **Syntax:**

```
short shortVar;
```

- **Size:**

```
2 byte ( 16 bits )
```

- **Values:**

```
-32, 768 to 32, 767 (inclusive)
```

- **Default Value:**

```
0
```

4. int

- **It is a 32-bit signed two's complement integer.**

- **Syntax:**

```
int intVar;
```

- **Size:**

```
4 byte ( 32 bits )
```

- **Values:**

```
-2, 147, 483, 648 to 2, 147, 483, 647 (inclusive)
```

- **Default Value:**

```
0
```

5. long

- **The long data type is a 64-bit two's complement integer.**

- **Syntax:**

```
long longVar;
```

- **Size:**

```
8 byte ( 64 bits )
```

- **Values:**

```
-9, 223, 372, 036, 854, 775, 808
          to
9, 223, 372, 036, 854, 775, 807
      (inclusive)
```

- **Default Value:**

```
0
```

6. float

- **The float data type is a single-precision 32-bit floating-point.**
- **Use a float (instead of double) if you need to save memory in large arrays of floating-point numbers.**

- **Syntax:**

```
float floatVar;
```

- **Size:**

```
4 byte ( 32 bits )
```

- **Values:**

```
upto 7 decimal digits
```

- **Default Value:**

```
0.0
```

7. double

- **The double data type is a double-precision 64-bit floating-point.**
- **For decimal values, this data type is generally the default choice.**

- **Syntax:**

```
double doubleVar;
```

- **Size:**

```
8 byte ( 64 bits )
```

- **Values:**

```
upto 16 decimal digits
```

- **Default Value:**

```
0.0
```

8. char

- **The char data type is a single 16-bit Unicode character.**

- **Syntax:**

```
char charVar;
```

- **Size:**

```
2 byte ( 16 bits )
```

- **Values:**

```
'\u0000' (0) to '\uffff' (65535)
```

- **Default Value:**

```
'\u0000'
```

Why is the size of char is 2 byte in java..?

- **In other languages like C/C++ uses only ASCII characters and to represent all ASCII characters 8-bits is enough,**
  But java uses the Unicode system not the ASCII code system and to represent Unicode system 8 bit is not enough to represent all characters so java uses 2 bytes for characters.

- ***Unicode defines a fully international character set that can represent most of the world's written languages. It is a unification of dozens of character sets, such as Latin, Greeks, Cyrillic, Katakana, Arabic, and many more.***

Non-Primitive Data Type or Reference Data Types

- **The Reference Data Types will contain a memory address of variable value <u>because</u> the reference types <u>won't store</u> the variable value directly in memory.**

- **They are <u>strings</u>, <u>objects</u>, <u>arrays</u>, etc.**

- **101 memory address**

- **X =5**

1. String

- **Strings are defined as an array of characters.**
- **The difference between a character array and a string in Java is, the string is designed to hold a sequence of characters in a single variable whereas, a character array is a collection of separate char type entities.**
- **Unlike C/C++, Java strings are not terminated with a null character.**
- **Below is the basic syntax for declaring a string in Java programming language.**
  Syntax:

```
<String_Type> <string_variable> = “<sequence_of_string>”;
```

- **Example:**

```
// Declare String without using new operator
String s = "GeeksforGeeks";

// Declare String using new operator
String s1 = new String("GeeksforGeeks");
```

2.Class:

- **A class is a user-defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:**
  - **Modifiers: A class can be public or has default access (Refer this for details).**

- **Class name: The name should begin with a initial letter (capitalized by convention).**
- **Superclass(if any): The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.**
- **Interfaces(if any): A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.**
- **Body: The class body surrounded by braces, { }.**

Object:

- **It is a basic unit of Object-Oriented Programming and represents the real-life entities. A typical Java program creates many objects, which as you know, interact by invoking methods. An object consists of :**
- **State: It is represented by attributes of an object. It also reflects the properties of an object.**
- **Behavior: It is represented by methods of an object. It also reflects the response of an object with other objects.**
- **Identity: It gives a unique name to an object and enables one object to interact with other objects.**

3. Array

- **An array is a group of like-typed variables that are referred to by a common name.**
- **Arrays in Java work differently than they do in C/C++.**
- **The following are some important points about Java arrays.**
- **In Java, all arrays are dynamically allocated. (discussed below)**
- **Since arrays are objects in Java, we can find their length using member length. This is different from C/C++ where we find length using size.**
- **A Java array variable can also be declared like other variables with [] after the data type.**

- **The variables in the array are ordered and each has an index beginning from 0.**
- **Java array can be also be used as a static field, a local variable or a method parameter.**
- **The size of an array must be specified by an int value and not long or short.**
- **The direct superclass of an array type is Object.**

Operators in Java

- **Operator in Java is a symbol which is used to perform operations.**
- **For example: +, -, *, / etc.**
- **There are many types of operators in Java which are given below:**

    1. **Unary Operator,**
    2. **Arithmetic Operator,**
    3. **Shift Operator,**
    4. **Relational Operator,**
    5. **Bitwise Operator,**
    6. **Logical Operator,**
    7. **Ternary Operator and**
    8. **Assignment Operator.**

Java Operator Precedence

Operator Type	Category	Precedence
Unary	postfix	*expr++ expr--*
	prefix	*++expr --expr +expr -expr* ~ !
Arithmetic	multiplicative	* / %
	additive	+ -
Shift	shift	<< >> >>>
Relational	comparison	< > <= >= instanceof
	equality	== !=
Bitwise	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical	logical AND	&&
	logical OR	||
Ternary	ternary	? :
Assignment	assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

1.Java Unary Operator

- **The Java unary operators require only one operand.**
- **Unary operators are used to perform various operations i.e.:**
    - **incrementing/decrementing a value by one**
    - **negating an expression**
    - **inverting the value of a boolean**

Java Unary Operator Example: ++ and --

```
class OperatorExample{
public static void main(String args[]){
int x=10;
System.out.println(x++);//10 (11)
System.out.println(++x);//12
System.out.println(x--);//12 (11)
System.out.println(--x);//10
}}
```

Output:

```
10
12
12
10
```

Java Unary Operator Example: ~ and !

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=-10;
boolean c=true;
boolean d=false;
System.out.println(~a);//-11 (minus of total positive value which starts from 0)
System.out.println(~b);//9 (positive of total minus, positive starts from 0)
System.out.println(!c);//false (opposite of boolean value)
System.out.println(!d);//true
}}
```

```
-11
9
false
true
```

2. Java Arithmetic Operators

- **Java arithmatic operators are used to perform addition, subtraction, multiplication, and division.**
- **They act as basic mathematical operations.**

Operators	Result
+	Addition of two numbers
-	Subtraction of two numbers
*	Multiplication of two numbers
/	Division of two numbers
%	(Modulus Operator)Divides two numbers and returns the remainder
++	Increment Operator
–	Decrement Operator

Java Arithmetic Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

Output:

```
15
5
50
2
0
```

3.Java Left Shift Operator

- **The Java left shift operator << is used to shift all of the bits in a value <u>to the left side</u> of a specified number of times.**

Java Left Shift Operator Example

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10<<2);//10*2^2=10*4=40
System.out.println(10<<3);//10*2^3=10*8=80
System.out.println(20<<2);//20*2^2=20*4=80
System.out.println(15<<4);//15*2^4=15*16=240
}}
```

Output:

```
40
80
80
240
```

Java Right Shift Operator

- **The Java right shift operator >> is used to move left operands value to right by the number of bits specified by the right operand.**

Java Right Shift Operator Example

```
class OperatorExample{
public static void main(String args[]){
System.out.println(10>>2);//10/2^2=10/4=2
System.out.println(20>>2);//20/2^2=20/4=5
System.out.println(20>>3);//20/2^3=20/8=2
}}
```

Output:

```
2
5
2
```

4. Relational Operators

- **The general format of representing relational operator is:**
- **variable1 *relation_operator* variable2**

Operator	Description	Example
==	Is Equal To	3 == 5 returns **false**
!=	Not Equal To	3 != 5 returns **true**
>	Greater Than	3 > 5 returns **false**
<	Less Than	3 < 5 returns **true**
>=	Greater Than or Equal To	3 >= 5 returns **false**
<=	Less Than or Equal To	3 <= 5 returns **true**

1. Equal to' operator (==):

- **This operator is used to check whether the two given operands are equal or not.**
- **The operator returns true if the operand at the left-hand side is equal to the right-hand side, else false.**

Syntax:

```
var1 == var2
```

2. 'Not equal to' Operator(!=):

- **This operator is used to check whether the two given operands are equal or not.**
- **It functions opposite to that of the equal-to operator.**
- **It returns true if the operand at the left-hand side is <u>not equal to</u> the right-hand side, else false.**

Syntax:

```
var1 != var2
```

3. 'Greater than' operator(>):

- **This checks whether the first operand is greater than the second operand or not.**
- **The operator returns true when the operand at the left-hand side is greater than the right-hand side.**

Syntax:

```
var1 > var2
```

4. ‘Less than’ Operator(<):

- **This checks whether the first operand is less than the second operand or not.**
- **The operator returns true when the operand at the left-hand side is less than the right-hand side.**
- **It functions opposite to that of the greater than operator.**

Syntax:

```
var1 < var2
```

5. 'Greater than or equal to' operator(>=):

- **This checks whether the first operand is greater than or equal to the second operand or not.**
- **The operator returns true when the operand at the left-hand side is greater than or equal to the right-hand side.**

Syntax:

```
var1 >= var2
```

6. 'Less than or equal to' Operator(<=):

- **This checks whether the first operand is less than or equal to the second operand or not.**
- **The operator returns true when the operand at the left-hand side is less than or equal to the right-hand side.**

Syntax:

```
var1 <= var2
```

5 and 6. Java AND Operator Example: Logical && and Bitwise &

- **The logical && operator doesn't check second condition if first condition is false.**
- **It checks second condition only if first one is true.**
- **The bitwise & operator always checks both conditions whether first condition is true or false.**

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a<b&&a<c);//false && true = false
System.out.println(a<b&a<c);//false & true = false
}}
```

Output:

```
false
false
```

Java OR Operator Example: Logical || and Bitwise |

- **The logical || operator doesn't check second condition if first condition is true.**
- **It checks second condition only if first one is false.**
- **The bitwise | operator always checks both conditions whether first condition is true or false.**

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int c=20;
System.out.println(a>b||a<c);//true || true = true
System.out.println(a>b|a<c);//true | true = true
//|| vs |
System.out.println(a>b||a++<c);//true || true = true
System.out.println(a);//10 because second condition is not checked
System.out.println(a>b|a++<c);//true | true = true
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
true
true
true
10
true
11
```

7. Java Ternary Operator

- **Java Ternary operator is used as one liner replacement for if-then-else statement and used a lot in Java programming.**
- **it is the only conditional operator which takes three operands.**

Java Ternary Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=2;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
2
```

Another Example:

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
int min=(a<b)?a:b;
System.out.println(min);
}}
```

Output:

```
5
```

8. Java Assignment Operator

- **Java assignment operator is one of the most common operator.**
- **It is used to assign the value on its right to the operand on its left.**

Java Assignment Operator Example

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=20;
a+=4;//a=a+4 (a=10+4)
b-=4;//b=b-4 (b=20-4)
System.out.println(a);
System.out.println(b);
}}
```

Output:

```
14
16
```

Java Assignment Operator Example

```
class OperatorExample{
public static void main(String[] args){
int a=10;
a+=3;//10+3
System.out.println(a);
a-=4;//13-4
System.out.println(a);
a*=2;//9*2
System.out.println(a);
a/=2;//18/2
System.out.println(a);
}}
```

Output:

```
13
9
18
9
```

Control Statement or Control Structures

- **In Java, the program execution starts with the first statement of main() and ends with the last statement of main().**
- **With the help of control statement , we can change the execution order.**
- **Java compiler executes the java code from top to bottom. The statements are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of java code. Such statements are called control flow statements.**
- **Java provides three types of control flow statements.**
- **Java provides three types of control flow statements.**

    1. **Decision Making statements**
    2. **Loop statements**
    3. **Jump statements**

1. Decision-Making statements:

- **Decision-making statements evaluate the Boolean expression and control the program flow depending upon the condition result.**
- **There are two types of decision-making statements in java, I.e., If statement and switch statement.**

If Statement:

- **In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the condition result that is a Boolean value, either true or false. In java, there are four types of if-statements given below.**
    1. **if statement**
    2. **if-else statement**
    3. **if else-if statement**
    4. **Nested if-statement**

1. <u>Simple if</u>

- **In a simple if statement, a condition is tested.**
- **If the condition is true, a set of statements are executed.**
- **If the condition is false, the statements are not executed and the program control goes to the next statement that immediately follows if block.**

The syntax of the if statement is

```
if( condition)
    {
     Statement;
    }
    Next Statement;
```

*test - Notepad

File Edit Format View Help

```
import java.io.*;
class test
{
 public static void main(String args[])
   {
    int num1=10;
    int num2=5;
     if(num1>num2)
       {
        System.out.println(num1+" greater than "+num2);
       }
   }
}
```

Ln 2, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java test
10 greater than 5
```

2. if else Statement

- **In simple if statement, when the condition is true, a set of statements are executed. But when it is false, there is no alternative set of statements.**
- **In if else statement, when the condition is true, a set of statements are executed. When it is false, a set of alternative statements are executed.**

The syntax of if else statement is

```
if( condition)
    {
     Statement set1;
    }
 else
    {
     Statement set2;
    }
```

Q.1) WAP to find the largest of two numbers.

test1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test1
{
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter the First number:");
    int num1=io.nextInt();
    System.out.println("Enter the Second number:");
    int num2=io.nextInt();

     if(num1>num2)
       {
        System.out.println(num1+" greater than "+num2);
       }
     else
       {
        System.out.println(num2+" greater than "+num1);
       }
   }
}
```

Ln 17, Col 10 | 100% | Windows (CRLF) | UTF-8

```
Enter the First number:

5

Enter the Second number:

8

8 greater than 5
```

2. **WAP to check whether given number is even or odd.**
3. **WAP to check whether the two given numbers are equal.**
4. **write a program that determines if a year is a leap year.**

% (Modulus or Remainder Operator)

- **Always find the remainder.**
- **This operator return an integer value.**

Even or Not

- **A number is said to be even if its num % 2 = 0 otherwise the number is said to be odd.**

Leap year

If((year %4==0 && year%100!=0)||year%400==0)

It is a leap year eg:2000

otherwise

not a leap year eg: 1955,1800

```
If the value in cell A1 is this        The formula returns
-------------------------------------------------------------
1992                                   Leap Year
2000                                   Leap Year
1900                                   NOT a Leap Year
```

3. if else if

- **The 'if else if' statement is to check for a sequence of conditions.**
- **When one condition is false, it checks for the next condition and so on.**
- **When all the conditions are false ,the else block is executed.**
- **The statement in that conditional block are executed and the other if statements are skipped.**

The syntax of 'if else if ' statement is

```
if( condition1)
  {
    Statement set1;
  }
else if(condition2)
  {
   Statement set2;
  }
.
.
.
else if(condition n)
  {
   Statement set n;
  }
else
  {
    Statement set x;
  }
Next Statement;
```

Q.1) WAP to checks whether a number given by the user is zero, positive or negative.

test3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test3
{
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter a number:");
    int num=io.nextInt();

     if(num==0)
       {
        System.out.println("num is zero");
       }
     else if(num>0)
       {
        System.out.println("num is positive");
       }
     else
       {
        System.out.println("num is negative");
       }
   }
}
```

Ln 10, Col 6 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java test3
Enter a number:
0
num is zero

D:\java\programs\module-1>java test3
Enter a number:
6
num is positive

D:\java\programs\module-1>java test3
Enter a number:
-7
num is negative
```

4. Nested if-statement

- **An if statement embedded with another if statement is called as nested if.**
- **In nested if-statements, the if statement contains multiple if-else statements as a separate block of code.**

<u>The syntax of nested if else statement is</u>

```
If(condition1)
 {
   if(condition2)
     {
      statement set1;
     }
   else
      {
      statement set2;
      }
 }
else
 {
  statement set3;
 }
 Next Statement;
```

Flowchart of nested if else statement

test4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test4
{
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter a number:");
    int num=io.nextInt();

     if(num>0)
       {

          if(num%2==0)
                {
                 System.out.println(num +" is even");
                 }
           else
                {
                System.out.println(num + " is odd");
                }
       }
     else if(num==0)
       {
        System.out.println("num is Zero");
       }
      else
        {
        System.out.println("num is negative");
        }
   }
}
```

Ln 3, Col 12 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java test4
Enter a number:
0
num is Zero

D:\java\programs\module-1>java test4
Enter a number:
4
4 is even

D:\java\programs\module-1>java test4
Enter a number:
7
7 is odd

D:\java\programs\module-1>java test4
Enter a number:
-3
num is negative
```

Switch

- **'Java 'provides a multiple branch selection statement known as switch.**
- **Mainly used to implement the menu driven program.**
- **The switch statement is a selection control structure that selects a choice from the set of available choice.**

The syntax of switch statement is as follows

```
switch(expression)
    {
    case constant 1: statement sequence 1;
                     break;
    case constant 2: statement sequence 2;
                     break;
    case constant 3: statement sequence 3;
                     break;
    .
    .
    .
    .
     case constant n-1: statement sequence n-1;
                     break;
    default          : statement sequence n;
    }
```

Rules for 'switch' statement

- **The switch expression must be an integer type**
- **Case label must be constant or character constant**
- **Case label must be unique. No two labels can have the same value.**
- **Case label must be end with colon.**
- **The break statement transfers the control out of the switch statement.**
- **The default label is optional. If present, it will be executed when the expression does not find a matching case label.**
- **The default may be placed anywhere but usually placed at the end.**

Q.1) .WAP to input number of week's day (1- 7) and translate to its equivalent name of the day of the week

- **(eg: 1 to Sunday , 2 to Monday ,3. Tuesday,4. Wednesday, 5. Thursday ,6. Friday, 7 Saturday).**

day - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class day
{
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter your choice(1-7):");
    int ch=io.nextInt();

    switch(ch)
       {
        case 1:System.out.println("Sunday");
               break;
        case 2:System.out.println("Monday");
               break;
        case 3:System.out.println("Tuesday");
               break;
        case 4:System.out.println("Wednesday");
               break;
        case 5:System.out.println("Thursday");
               break;
        case 6:System.out.println("Friday");
               break;
        case 7:System.out.println("Saturday");
               break;
        default:System.out.println("Invalid Choice");
        }
   }
}
```

Ln 28, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java day
Enter your choice(1-7):
1
Sunday

D:\java\programs\module-1>java day
Enter your choice(1-7):
2
Monday

D:\java\programs\module-1>java day
Enter your choice(1-7):
7
Saturday

D:\java\programs\module-1>java day
Enter your choice(1-7):
8
Invalid Choice
```

3. **Write a menu driven program to find the area of the circle, rectangle and right angle triangle.**
4. **Write a menu driven program to perform the arithmetic operation. (+, - , * , /).**

2. Iteration or Looping statements

- **The iteration statements allow a set of instruction to be performed repeatedly <u>until</u> a certain condition is satisfied.**
- **The statements are executed as long as the condition is true.**
- **The iteration statements are also called loops or looping statements.**

'Java' provides three kinds of loops

1. **while loop**
2. **do while loop**
3. **for loop**

Java For Loop vs While Loop vs Do While Loop

Comparison	for loop	while loop	do while loop
Introduction	The Java for loop is a control flow statement that iterates a part of the programs multiple times.	The Java while loop is a control flow statement that executes a part of the programs repeatedly on the basis of given boolean condition.	The Java do while loop is a control flow statement that executes a part of the programs at least once and the further execution depends upon the given boolean condition.
When to use	If the number of iteration is fixed, it is recommended to use for loop.	If the number of iteration is not fixed, it is recommended to use while loop.	If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

Syntax	`for(init;condition;incr/decr){`
	`// code to be executed`
	`}`
		`while(condition){`
		`//code to be executed`
		`}`
			`do{`
			`//code to be executed`
			`}while(condition);`

Example	`//for loop`
	`for(int i=1;i<=10;i++){`
	`System.out.println(i);`
	`}`
		`//while loop`
		`int i=1;`
		`while(i<=10){`
		`System.out.println(i);`
		`i++;`
		`}`
			`//do-while loop`
			`int i=1;`
			`do{`
			`System.out.println(i);`
			`i++;`
			`}while(i<=10);`

Syntax for infinitive loop	`for(;;){`
	`//code to be executed`
	`}`
		`while(true){`
		`//code to be executed`
		`}`
			`do{`
			`//code to be executed`
			`}while(true);`

?sa=L&ai=CfBDOgwCLYPH5J5SQmsMP1f2Iw...

1. while loop

- **A while loop is used to repeat certain statements as long as the condition is true.**
- **When the condition becomes false, the while loop is quitted.**
- **This loop control structure is called as an entry-controlled loop because, only when the condition is true, the statements are executed.**

The syntax of the while loop statement is

```
while( condition)
 {
   Set of Statements;
 }
Next Statement;
```

The flowchart of 'while' loop

Execution of while loop statement

1. **The condition is tested**
2. **If condition is initially false, the body of the while statement is never executed and control passes to the next statement in the program.**
3. **If condition is true, the body of the while statement is executed and the process is repeated beginning at step 1.**

1.) WAP to print numbers from 1 to 10.

numbers - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class numbers
{
 public static void main(String args[])
   {

    int i=1;

    System.out.println("Following are the numbers from 1 to 10");

      while(i<=10)
       {
        System.out.println(i);
        i++;
       }

   }
}
```

Ln 10, Col 5 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java numbers
Following are the numbers from 1 to 10
1
2
3
4
5
6
7
8
9
10
```

2. WAP to print first ‘n’ natural numbers and their sum.

numbers1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class numbers1
{
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter the value of n:");
    int n=io.nextInt();
    int i=1,sum=0;
    System.out.println("Following are the numbers from 1 to "+n);
     while(i<=n)
       {
        System.out.println(i);
        sum=sum+i;
        i++;
       }
     System.out.println ("Sum="+sum);
    }
}
```

Ln 20, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java numbers1
Enter the value of n:
10
Following are the numbers from 1 to 10
1
2
3
4
5
6
7
8
9
10
Sum=55
```

3) WAP to print the multiplication table of the given number.

mul - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class mul
{
 public static void main(String args[])
   {
     Scanner io=new Scanner(System.in);
     System.out.println("Enter a number :");
     int num=io.nextInt();
     int i=1;
   while(i<=10)
     {
      System.out.println(i + "*" + num +"=" + i*num);
      i++;
     }

   }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java mul
Enter a number :
5
1*5=5
2*5=10
3*5=15
4*5=20
5*5=25
6*5=30
7*5=35
8*5=40
9*5=45
10*5=50
```

4) **WAP to calculate the factorial of given number.**

5) **WAP to find the sum of digits of a given number.**

6) **WAP to print first 'n' terms in the Fibonacci series**

2. do-while loop

- **The do while loop is very similar to while loop. In do-while loop , the condition is tested at the end of the loop.**
- **Because of this , even when the condition is false, the body of the loop is executed at least once.**
- **This is an exit-controlled loop.**

The syntax of the do while statement is

```
do
{

    Set of statements;

} while (condition);

Next Statement;
```

- **The do while loop repeat a set of statements until a specified condition becomes false.**
- **The condition is evaluated after body of the loop is executed. Hence body of the loop executes at least once if the condition is true / false**

Execution proceeds as follows

- **First the loop is executed , next the condition is evaluated , if the condition is true the loop continues execution else control passes to the next statement.**

Difference between while and do while loop

while loop	do while loop
1. The condition is tested before entering into the loop. 2. When the condition is false at the initial stage, the loop statements are not executed at all. 3. It is an entry-controlled loop.	1. The condition is tested at the end of the loop. 2. The statements are executed at least once even when the condition is false. 3. It is an exit controlled loop.

Q.1) WAP to print first ‘n’ natural numbers and their sum.

numbers1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class numbers1
{
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter the value of n:");
    int n=io.nextInt();
    int i=1,sum=0;
    System.out.println("Following are the numbers from 1 to "+n);
      do
       {
        System.out.println(i);
        sum=sum+i;
        i++;
       }while(i<=n);
     System.out.println ("Sum="+sum);
    }
}
```

Ln 20, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java numbers1
Enter the value of n:
10
Following are the numbers from 1 to 10
1
2
3
4
5
6
7
8
9
10
Sum=55
```

3. for loop

- **The 'for' loops are generally used when certain statements have to be executed a specific number of times.**

The syntax of the for loop statement is

```
for(initialization expression; condition expression; increment expression)
  {
      Set  of Statements;
  }
  Next Statements;
```

Advantages of for loops

- **All the three parts of a loop (1. initialization, 2. condition and 3. increment) can be given in a single statement.**
- **Because of this , there is no chance of user missing out initialization or increment steps which is the common programming error in while and do while loops.**

The flowchart of 'for' loop

Execution of for loop statement

1) **Initialization of the control variable is done first**

2) **The value of the control variable is tested using the test condition.**

If the condition is true, the body of the loop is executed otherwise the loop is terminated.

3) **When the body of the loop is executed, the control is transferred back to the for statements after evaluating the last statement in the loop. The control variable is incremented using an assignment statement such as i=i+1 and then, the new value of the control variable is again tested. If the condition is true, the body of the loop is again executed. This process continues till the value of the control variable fail to satisfy the test condition.**

Q.1) WAP to print the multiplication table of the given number.

fornumbers - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class fornumbers
{
 public static void main(String args[])
   {

     Scanner io=new Scanner(System.in);
     System.out.println("Enter a Number:");
     int num=io.nextInt();

       int i;
       for(i=1;i<=10;i++)
         {
         System.out.println(i+" * "+num+ " = " + i*num);
         }

    }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java fornumbers
Enter a Number:
5
1 * 5 = 5
2 * 5 = 10
3 * 5 = 15
4 * 5 = 20
5 * 5 = 25
6 * 5 = 30
7 * 5 = 35
8 * 5 = 40
9 * 5 = 45
10 * 5 = 50
```

Q.2) WAP to print first 'n' natural numbers.

numbers - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class numbers
{
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter the value of n:");
    int n=io.nextInt();
    int i;

   System.out.println("Following are the numbers from 1 to "+n);
   for(i=1;i<=n;i++)
     {
     System.out.println(i);
     }

   }
}
```

Ln 9, Col 24 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java fornumbers1
Enter the value of n:
10
Following are the numbers from 1 to 10
1
2
3
4
5
6
7
8
9
10
```

3. Jump statement or Unconditional type statement

- **Without using a condition, we can change the sequence of program execution order.**
- **Unconditional type statements are mainly classified into two types.**
    1. **break**
    2. **continue**

1. break Statement

- **The break statement is used in loop such as for, while, do while and switch statement, to terminate the execution of loop or switch statement.**
- **The form of a break statement is**

Syntax

break;

Figure: Flowchart of break statement

egbreak - Notepad

File Edit Format View Help

```
public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter the value of n:");
    int n=io.nextInt();
    int i=1;
    System.out.println("Following are the numbers from 1 to "+n);
      while(i<=n)
       {
        if(i==3)
          {
          break;
          }

        System.out.println(i);
        i++;
       }
    }
}
```

Ln 5, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java egbreak
Enter the value of n:
10
Following are the numbers from 1 to 10
1
2
```

2. <u>continue Statement</u>

- **The continue statement is used in for, while or do while loop.**
- **Continue statement <u>forces</u> the next iteration of the loop to take place and <u>skips the code between</u> continue statement and the end of the loop.**

egcontinue - Notepad

File Edit Format View Help

```
    {
     Scanner io=new Scanner(System.in);
     System.out.println("Enter the value of n:");
     int n=io.nextInt();
     int i=1;
     System.out.println("Following are the numbers from 1 to "+n);
       while(i<=n)
        {
         if(i==3)
           {
            i++;
            continue;
           }

         System.out.println(i);
         i++;
        }
     }
}
```

Ln 17, Col 12 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java egcontinue
Enter the value of n:
10
Following are the numbers from 1 to 10
1
2
4
5
6
7
8
9
10
```

Difference between break and continue

break	continue
1. Can be used within loops or switch.	1. Can be used only within loops .
2. Used to quit an innermost loop or switch	2. Used to continue the innermost loop

Increment and Decrement operator (++ and --)

- **Operators ++ and – are called as increment and decrement operators.**
- **These operators increment or decrement the variable's value by 1.**
- **They are also called as unary operators because they have only one operand to operate.**
- **If the operator is used before the operand , it is prefix and if the operator is used after the operand , it is postfix.**

Eg: ++ i and - - i is called as prefix

i++ and i- - is called as postfix

Difference between Prefix and Postfix

Prefix

- **The prefix increment or decrement operator follow change-then-use rule.**

i.e first change the value of their operand ,then use the new value in evaluating the expression.

- **Prefix operator first increment/decrement and then makes the assignment.**

prefix - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class prefix
{
 public static void main(String args[])
   {

    int i=5;
    System.out.println(i);
    System.out.println(++i);
    System.out.println(i);

    }
}
```

Ln 7, Col 4 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java prefix
5
6
6
```

Postfix

- **The postfix increment or decrement operator follow use-then-change rule.**

i.e they first use the value of their operand in evaluating the expression and then change the operand's value.

- **Postfix operator makes the assignment and then increments/decrements the value.**

postfix - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class postfix
{
 public static void main(String args[])
   {

    int i=5;
    System.out.println(i);
    System.out.println(i++);
    System.out.println(i);

    }
}
```

Ln 13, Col 8 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java postfix
5
5
6
```

Constructors in Java

- **A constructor is a 'special' member function whose task is to initialize the objects of its class.**
- **It is special because its name is the same as the class name.**
- **The constructor is invoked whenever an object of its associated class is created.**
- **It is called constructor because it constructs the values of data members of the class.**
- **A member function with the same name as its class is called constructor and it is used to initialize the objects of that class type with a legal initial value.**
- **A constructor is declared and defined as follows:**

```
// class with a constructor
class Student
{
  int rollno;
  float marks;
    Student()      //constructor
        {
        rollno=0;
        marks=0.0;
        }
};
```

- **In Java, a constructor is a block of codes similar to the method.**
- **It is called when an instance of the class is created.**
- **At the time of calling constructor, memory for the object is allocated in the memory.**
- **It is a special type of method which is used to initialize the object.**
- **Every time an object is created using the new() keyword, at least one constructor is called.**

default constructor

- **It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.**
- **There are two types of constructors in Java:**

1. no-arg constructor, and

2. parameterized constructor.

- **Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if your class doesn't have any.**

Rules for creating Java constructor

- **There are three rules defined for the constructor.**

1. **Constructor name must be the same as its class name**
2. **A Constructor must have no explicit return type**
3. **A Java constructor cannot be abstract, static, final, and synchronized**

Types of Java constructors

- **There are two types of constructors in Java:**

1. **Default constructor (no-arg constructor)**
2. **Parameterized constructor**

1. Java Default Constructor

- **A constructor is called "Default Constructor" when it doesn't have any parameter.**
- **Syntax of default constructor:**

```
<class_name>()
        {
        }
```

Example of default constructor

eg:1

test - Notepad

File Edit Format View Help

```
import java.io.*;
class test
{

  test()
   {
     System.out.println("Constructor is Invoked");
   }

  public static void main(String args[])
  {

   test o=new test();
  }

}
```

Ln 2, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac test.java

D:\java\programs\module-1>java test
Constructor is Invoked
```

Eg:2

test - Notepad

File Edit Format View Help

```
import java.io.*;
class test
{
   double pi;

  test()
   {
    pi=3.14;
   }
  public void display()
   {
   System.out.println("pi="+pi);
   }

  public static void main(String args[])
  {

   test o=new test();
   o.display();
  }
}
```

Ln 13, Col 5 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac test.java

D:\java\programs\module-1>java test
pi=3.14
```

Eg:3

- **In this example, we are creating the no-arg constructor in the Bike class. It will be invoked at the time of object creation.**

```
class Bike1{
        //creating a default constructor
        Bike1()
        {
        System.out.println("Bike is created");
        }
        public static void main(String args[])
        {
        //calling a default constructor
    Bike1 b=new Bike1();
    }
    }
```

```
Bike is created
```

Q) What is the purpose of a default constructor?

- **The default constructor is used to provide the default values to the object like 0, null, etc., depending on the type.**

Example of default constructor that displays the default values

Student2 - Notepad

File Edit Format View Help

```
import java.io.*;

class Student2
  {
   int rno;
   String name;
   Student2()
     {
      System.out.println("Roll No:"+rno);
      System.out.println("Name:"+name);
     }
   public static void main(String args[])
     {
      Student2 o=new Student2();
      }
  }
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac Student2.java

D:\java\programs\module-1>java Student2
Roll No:0
Name:null
```

Eg:2

Student3 - Notepad

File Edit Format View Help

```
import java.io.*;

class Student3
  {
   int rno;
   String name;
   Student3()
     {
      System.out.println("Roll No:"+rno);
      System.out.println("Name:"+name);
     }
   public static void main(String args[])
     {
      Student3 o1=new Student3();
      Student3 o2=new Student3();
      }
  }
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac Student3.java

D:\java\programs\module-1>java Student3
Roll No:0
Name:null
Roll No:0
Name:null
```

- **Explanation:In the above class,you are not creating any constructor so compiler provides you a default constructor. Here 0 and null values are provided by default constructor.**

2.Java Parameterized Constructor

- **A constructor which has a specific number of parameters is called a parameterized constructor.**

Why use the parameterized constructor?

- **The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.**

Example of parameterized constructor

- **In this example, we have created the constructor of Student class that have two parameters. We can have any number of parameters in the constructor.**

Student4 - Notepad

File Edit Format View Help

```
import java.io.*;

class Student4
  {
   int rno;
   String name;
   Student4(int r,String n)
     {
      rno=r;
      name=n;
     }
  public void display()
     {
      System.out.println("Roll No:"+rno);
      System.out.println("Name:"+name);
     }
   public static void main(String args[])
     {
      Student4 o1=new Student4(101,"Arun");
      Student4 o2=new Student4(102,"Sonu");
      o1.display();
      o2.display();
      }
  }
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac Student4.java

D:\java\programs\module-1>java Student4
Roll No:101
Name:Arun
Roll No:102
Name:Sonu
```

Eg:2

Student5 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Student5
  {
   int rno;
   String name;
   Student5(int r,String n)
     {
      rno=r;
      name=n;
     }
  public void display()
     {
      System.out.println("Roll No:"+rno);
      System.out.println("Name:"+name);
     }
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

Student5 - Notepad

File Edit Format View Help

```
    public static void main(String args[])
      {
       int r;
       String s;
       Scanner io=new Scanner(System.in);
       System.out.println("Enter the details of first student");
       System.out.println("Enter the Roll Number:");
       r=io.nextInt();
       System.out.println("Enter the Name:");
       s=io.next();
       Student5 o1=new Student5(r,s);

       System.out.println("Enter the details of Second student");
       System.out.println("Enter the Roll Number:");
       r=io.nextInt();
       System.out.println("Enter the Name:");
       s=io.next();
       Student5 o2=new Student5(r,s);
       o1.display();
       o2.display();
       }
  }
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac Student5.java

D:\java\programs\module-1>java Student5
Enter the details of first student
Enter the Roll Number:
101
Enter the Name:
Ajay
Enter the details of Second student
Enter the Roll Number:
102
Enter the Name:
Vijay
Roll No:101
Name:Ajay
Roll No:102
Name:Vijay
```

Characteristics of Constructor

1. **Constructor has the same name as that of the class it belongs.**
2. **The should be declared as public .**
3. **Constructor are invoked automatically when the objects are created.**
4. **Constructors do not have return types, not even void and therefore, and they cannot return values..**
5. **The main function of constructor is to initialize objects and allocate appropriate memory to objects.**
6. **Though constructors are executed implicitly, they can be invoked explicitly.**
7. **Constructor can have default arguments and can be overloaded.**
8. **The constructor without arguments is called as default constructor.**
9. **Constructor cannot be inherited, though a derived class can call the base class constructor.**
10. **Constructor cannot be virtual.**
11. **We cannot refer to their addresses. i.e it is not possible to take the address of a constructor.**
12. **Member functions may be called from within a constructor.**

Difference between constructor and method in Java

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behavior of an object.
A constructor must not have a return type.	A method must have a return type.
The constructor is invoked implicitly.	The method is invoked explicitly.
The Java compiler provides a default constructor if you don't have any constructor in a class.	The method is not provided by the compiler in any case.
The constructor name must be same as the class name.	The method name may or may not be same as the class name.

Java Arrays

- **Normally, an array is a collection of similar type of elements which has contiguous memory location.**
- **Java array is an object which contains elements of a similar data type.**
- **Additionally, The elements of an array are stored in a contiguous memory location.**
- **It is a data structure where we store similar elements.**
- **We can store only a fixed set of elements in a Java array.**
- **Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.**
- **Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator.**
- **In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java.**
- **Like C/C++, we can also create single dimentional or multidimentional arrays in Java.**
- **Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.**

Advantages

- **Code Optimization: It makes the code optimized, we can retrieve or sort the data efficiently.**
- **Random access: We can get any data located at an index position.**

Disadvantages

Size Limit: We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

Types of Array in java

- **There are two types of array.**
    1. **Single Dimensional Array**
    2. **Multidimensional Array**

1. Single Dimensional Array in Java

- **Syntax to Declare an Array in Java**

```
dataType arr[];    (or )
dataType[] arr;    (or)
dataType []arr;
```

Instantiation of an Array in Java

```
arrayRefVar=new datatype[size];
```

Example of Java Array

- **Let's see the simple example of java array, where we are going to declare, instantiate, initialize and traverse an array.**

```
//Java Program to illustrate how to declare, instantiate, initialize
//and traverse the Java array.
class Testarray{
public static void main(String args[]){
int a[]=new int[5];//declaration and instantiation
a[0]=10;//initialization
a[1]=20;
a[2]=70;
a[3]=40;
a[4]=50;
//traversing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

```
10
20
70
40
50
```

Q.1) Program to print the elements of an array

rdarray - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class rdarray
{
public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    int n,i;
    System.out.println("Enter the value of n:");
    n=io.nextInt();
    int A[]=new int[n];
       System.out.println("Enter the "+n+" numbers into the array:");
       for(i=0;i<n;i++)
         {
          A[i]=io.nextInt();
         }
     System.out.println("Following are the array elements:");
       for(i=0;i<n;i++)
         {
          System.out.println(A[i]);
         }
   }

}
```

Ln 6, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java rdarray
Enter the value of n:
5
Enter the 5 numbers into the array:
10
20
30
40
50
Following are the array elements:
10
20
30
40
50
```

Method :2

rdarray1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class rdarray1
{

public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    int n,i;
    System.out.println("Enter the value of n:");
    n=io.nextInt();
    int A[]=new int[n];
       System.out.println("Enter the "+n+" numbers into the array:");
       for(i=0;i<A.length;i++)
         {
          A[i]=io.nextInt();
         }
     System.out.println("Following are the array elements:");
       for(i=0;i<A.length;i++)
         {
          System.out.println(A[i]);
         }

   }

}
```

Ln 24, Col 5 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java rdarray1
Enter the value of n:
5
Enter the 5 numbers into the array:
10
20
30
40
50
Following are the array elements:
10
20
30
40
50
```

Passing Array to a Method in Java

- **We can pass the java array to method so that we can reuse the same logic on any array.**
- **Let's see the simple example to get the minimum number of an array using a method.**

```
//Java Program to demonstrate the way of passing an array
//to method.
class Testarray2{
//creating a method which receives an array as a parameter
static void min(int arr[]){
int min=arr[0];
for(int i=1;i<arr.length;i++)
 if(min>arr[i])
  min=arr[i];

System.out.println(min);
}

public static void main(String args[]){
int a[]={33,3,4,5};//declaring and initializing an array
min(a);//passing array to method
}}
```

3

Q.2) Program to print the elements of an array in reverse order

rdarrayf - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
 class info
   {

   public static void read()
     {
       Scanner io=new Scanner(System.in);
       int i;
       System.out.println("Enter the size of the array");
       int n=io.nextInt();
       int A[]=new int[n];
         System.out.println("Enter the "+n+" numbers into the array");
          for(i=0;i<n;i++)
             {
               A[i]=io.nextInt();
             }
       display(A,n);
     }

   public static void display(int A[],int n)
     {

       int i;
          System.out.println("Following are the array elements in the reverse order:");
          for(i=n-1;i>=0;i--)
             {
               System.out.println(A[i]);
             }
     }

  }
```

```
class rdarrayf
 {

   public static void main(String args[])
     {

      info o=new info();
      o.read();
     }

 }
```

Ln 20, C

```
D:\java\programs\module-1>java rdarrayf
Enter the size of the array
5
Enter the 5 numbers into the array
10
20
30
40
50
Following are the array elements in the reverse order:
50
40
30
20
10
```

Q.3)Program to print the largest element in an array

arraylarge - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class info
  {
   public static void read()
     {
      Scanner io=new Scanner(System.in);
      int n,i;
      System.out.println("Enter the value of n:");
      n=io.nextInt();
      int A[]=new int[n];
       System.out.println("Enter the "+n+" numbers into the array:");
       for(i=0;i<n;i++)
         {
          A[i]=io.nextInt();
         }
       display(A,n);
     int res=find(A,n);
     System.out.println("The largest element of the array is "+res);

     }

    public static void display(int A[],int n)
     {
     int i;
     System.out.println("Following are the array elements :");
       for(i=0;i<n;i++)
         {
          System.out.println(A[i]);
         }
      }
```

Ln 31, Col 8 | 100% | Windows (CRLF) | UTF-8

arraylarge - Notepad

File Edit Format View Help

```
   public static int find(int A[],int n)
     {
     int i,large;
      large=A[0];
       for(i=1;i<n;i++)
         {
           if(A[i]>large)
            {
             large=A[i];

            }
         }
       return large;
     }
 }

class arraylarge
{

public static void main(String args[])
   {

    info o=new info();
    o.read();

   }

}
```

Ln 33, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java arraylarge
Enter the value of n:
5
Enter the 5 numbers into the array:
10
20
50
40
30
Following are the array elements :
10
20
50
40
30
The largest element of the array is 50
```

2. Multidimensional Array in Java

- **In such case, data is stored in row and column based index (also known as matrix form).**

Syntax to Declare Multidimensional Array in Java

dataType arrayRefVar[][]; (or)

dataType[][] arrayRefVar; (or)

dataType [][]arrayRefVar; (or)

dataType []arrayRefVar[];

Example to instantiate Multidimensional Array in Java

int arr[][] =new int[3][3];//3 row and 3 column

Example of Multidimensional Java Array

- **Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.**

```
//Java Program to illustrate the use of multidimensional array
class Testarray3{
public static void main(String args[]){
//declaring and initializing 2D array
int arr[][]={{1,2,3},{2,4,5},{4,4,5}};
//printing 2D array
for(int i=0;i<3;i++){
 for(int j=0;j<3;j++){
   System.out.print(arr[i][j]+" ");
 }
 System.out.println();
}
}}
```

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

Q.3)Java Program to read and display a matrix

rdmatrix - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class info
  {
   public static void read()
    {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the order of matrix A :");
      int m,n,i,j;
      System.out.println("Enter the number of rows:");
      m=io.nextInt();
      System.out.println("Enter the number of coloms:");
      n=io.nextInt();
       int A[][]=new int[m][n];
       System.out.println("Read matrix A :");
        for(i=0;i<m;i++)
          {
          for(j=0;j<n;j++)
            {
              A[i][j]=io.nextInt();

            }
          }

       display(A,m,n);
     }
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

*rdmatrix - Notepad

File Edit Format View Help

```
public static void display(int A[][],int m,int n)
    {
      int i,j;
       System.out.println("Given matrix A :");
        for(i=0;i<m;i++)
          {
          for(j=0;j<n;j++)
            {
              System.out.print(A[i][j]+" ");
            }

           System.out.println(" ");

          }
     }
}

class rdmatrix
  {
   public static void main(String args[])
     {
       info o=new info();
        o.read();
      }
  }
```

```
D:\java\programs\module-1>java rdmatrix
Enter the order of matrix A :
Enter the number of rows:
3
Enter the number of coloms:
3
Read matrix A :
1
2
3
4
5
6
7
8
9
Given matrix A :
1 2 3
4 5 6
7 8 9
```

Q.3)Java Program to add two matrices

addmatrix - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class info
  {
   public static void read()
     {
      Scanner io=new Scanner(System.in);
      int m,n,p,q,i,j;
      System.out.println("Enter the order of Matrix A");
      System.out.println("Enter the value of rows:");
      m=io.nextInt();
      System.out.println("Enter the value of column:");
      n=io.nextInt();
      int A[][]=new int[m][n];
      System.out.println("Enter the order of Matrix B");
      System.out.println("Enter the value of rows:");
      p=io.nextInt();
      System.out.println("Enter the value of column:");
      q=io.nextInt();
      int B[][]=new int[p][q];
       System.out.println("Read the Matrix A:");
        for(i=0;i<m;i++)
         {
           for(j=0;j<n;j++)
             {
             A[i][j]=io.nextInt();
             }
        }
```

Ln 86, Col 2 | 100% | Windows (CRLF) | UTF-8

addmatrix - Notepad

File Edit Format View Help

```
System.out.println("Read the Matrix B:");
    for(i=0;i<p;i++)
      {
        for(j=0;j<q;j++)
          {
          B[i][j]=io.nextInt();
          }
      }
    display(A,m,n);
    display(B,p,q);
    add(A,m,n,B,p,q);
  }
 public static void display(int A[][],int m,int n)
  {
  int i,j;
  System.out.println("Following are the array elements:");
    for(i=0;i<m;i++)
      {
        for(j=0;j<n;j++)
          {
           System.out.print(A[i][j]+" ");
           }
          System.out.println(" ");
      }
 }
```

Ln 86, Col 2 | 100% | Windows (CRLF) | UTF-8

addmatrix - Notepad

File Edit Format View Help

```
public static void add(int A[][],int m,int n,int B[][],int p,int q)
     {
     int i,j;

     if(m==p&&n==q)
      {
       int C[][]=new int[m][n];
       for(i=0;i<m;i++)
         {
           for(j=0;j<n;j++)
             {
              C[i][j]=A[i][j]+B[i][j];
              }
         }
        display(C,m,n);
      }
     else
      {
       System.out.println("Matrix Addition is not possible");
      }
}

 }
class addmatrix
{
public static void main(String args[])
   {
    info o=new info();
    o.read();
   }
}
```

Ln 86, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>java addmatrix
Enter the order of Matrix A
Enter the value of rows:
3
Enter the value of column:
3
Enter the order of Matrix B
Enter the value of rows:
3
Enter the value of column:
3
Read the Matrix A:
1 2 3
4 5 6
7 8 9
Read the Matrix B:
3 4 5
6 7 8
9 10 11
Following are the array elements:
1 2 3
4 5 6
7 8 9
Following are the array elements:
3 4 5
6 7 8
9 10 11
Following are the array elements:
4 6 8
10 12 14
16 18 20
```

Q.1)Program to print the smallest element in an array

Q.2)Java Program to print the sum of all the items of the array

Q.3)Java Program to sort the elements of an array in ascending order

Q.4)Java Program to subtract the two matrices

Q.6)Java Program to multiply two matrices

Q.7)Java Program to transpose matrix

How to Create Array of Objects in Java

- **In this section, we will learn how to create and initialize an array of objects in Java.**

Array of Objects in Java

- **Java is an object-oriented programming language.**
- **Most of the work done with the help of objects.**
- **We know that an array is a collection of the same data type that dynamically creates objects and can have elements of primitive types.**
- **Eg: int A[]=new int[n];**
- **Java allows us to store objects in an array.**
- **In Java, the class is also a user-defined data type.**
- **An array that contains class type elements are known as an array of objects.**
- **It stores the reference variable of the object.**

```
class test
{
int variable1;
int variable2;
}
O[0],o[1],o[2]
test o=new test();
test new test[3];
test Objo[]=ect[]=new test[3];
```

Creating an Array of Objects

- **Before creating an array of objects, we must create an instance of the class by using the new keyword.**
- **We can use any of the following statements to create an array of objects.**

Syntax:

```
ClassName obj[]=new ClassName[array_length]; //declare and instantiate an array of objects
```

Or

```
ClassName[] objArray;
```

Or

```
ClassName objeArray[];
```

- **Suppose, we have created a class named Employee.**
- **We want to keep records of 20 employees of a company <u>having three departments</u>.**
- **In this case, we will not create 20 separate variables.**
- **Instead of this, we will create an array of objects, as follows.**

```
Employee department1[20];
Employee department2[20];
Employee department3[20];
```

- **The above statements create an array of objects with 20 elements.**

Eg: 1) Write a java program to read the information (name ,regno and percentage) of 'n' students using array of objects

nstudent - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class nstudent
 {
   public static void main(String arg[])
    {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the value of n:");
      int n=io.nextInt();
      info o[]=new info[n];
      int i;
      for(i=0;i<n;i++)
       {
        o[i]=new info();
       }

      for(i=0;i<n;i++)
       {
        o[i].read();
       }
     for(i=0;i<n;i++)
       {
        o[i].display();
       }

    }
 }
```

Ln 22, Col 15 | 100% | Windows (CRLF) | UTF-8

*nstudent - Notepad

File Edit Format View Help

```
class info
{
private String name;
private  int reg;
private  float per;
  public void read()
     {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the student Name:");
      name=io.next();
      System.out.println("Enter the Register No:");
      reg=io.nextInt();
      System.out.println("Enter the Percentage:");
      per=io.nextFloat();


     }
   public void display()
     {
      System.out.println("Name:"+name);
      System.out.println("Regno:"+reg);
      System.out.println("Per:"+per);
     }
}
```

Ln 52, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac nstudent.java

D:\java\programs\module-1>java nstudent
Enter the value of n:
2
Enter the student Name:
anil
Enter the Register No:
101
Enter the Percentage:
67.89
Enter the student Name:
sunil
Enter the Register No:
102
Enter the Percentage:
89.65
Name:anil
Regno:101
Per:67.89
Name:sunil
Regno:102
Per:89.65
```

Q.2) Write a java program to read the information (name ,regno and percentage) of 'n' students using array of objects and print the output in the given format.

```
Name:    Regno:   Per:
anil     101      78.68
binu     102      67.89
sonu     103      57.89
```

nstudent1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class nstudent1
 {
   public static void main(String arg[])
    {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the value of n:");
      int n=io.nextInt();
      info o[]=new info[n];  //creating n objects
      int i;
      for(i=0;i<n;i++)
       {
        o[i]=new info();    // Allocating memory to each object
       }
      for(i=0;i<n;i++)
       {
        o[i].read();
       }
     System.out.println("Name:\tRegno:\tPer:");
     for(i=0;i<n;i++)
       {
        o[i].display();
       }
    }
 }
```

Ln 16, Col 1 | 100% | Windows (CRLF) | UTF-8

nstudent1 - Notepad

File Edit Format View Help

```
class info
{
String name;
int reg;
float per;
  public void read()
     {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the student Name:");
      name=io.next();
      System.out.println("Enter the Register No:");
      reg=io.nextInt();
      System.out.println("Enter the Percentage:");
      per=io.nextFloat();

     }
   public void display()
     {
      System.out.print(name+"\t");
      System.out.print(reg+"\t");
      System.out.print(per+"\t");
      System.out.println(" ");
     }
}
```

Ln 30, Col 1 | 100% | Windows (CRLF) | UTF-8

Command Prompt

```
D:\java\programs\module-1>javac nstudent1.java

D:\java\programs\module-1>java nstudent1
Enter the value of n:
3
Enter the student Name:
anil
Enter the Register No:
101
Enter the Percentage:
78.68
Enter the student Name:
binu
Enter the Register No:
102
Enter the Percentage:
67.89
Enter the student Name:
sonu
Enter the Register No:
103
Enter the Percentage:
57.89
Name:    Regno:   Per:
anil     101      78.68
binu     102      67.89
sonu     103      57.89
```

Lab Cycle Program

Q.3) Construct a Java class to store some employee details and provide methods to set and get values.

employee - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class info
{
int empid;
String name;
String dept;
String desg;
double salary;
    public void set()
     {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the Employee Id");
      empid=io.nextInt();
      System.out.println("Enter the Employee Name");
      name=io.next();
      System.out.println("Enter the Name of the Department");
      dept=io.next();
      System.out.println("Enter the Designation");
      desg=io.next();
      System.out.println("Enter the Employee Salary");
      salary=io.nextDouble();
     }
    public void get()
     {
      System.out.print(empid+"\t");
      System.out.print(name+"\t\t");
      System.out.print(dept+"\t");
      System.out.print(desg+"\t");
      System.out.print(salary+"\t");
      System.out.println(" ");
     }
}
```

*employee - Notepad

File Edit Format View Help

```
class employee
  {
   public static void main(String args[])
     {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the value of n :");
      int n=io.nextInt();
      info o[]=new info[n];
      int i;
     for(i=0;i<n;i++)
       {
         o[i]=new info();
       }
      System.out.println("Enter the "+n+" employees information");
      for(i=0;i<n;i++)
       {
         o[i].set();
       }
       System.out.println("EmpId\tName\tDepartment\tDesignation\tSalary");
      for(i=0;i<n;i++)
       {
         o[i].get();
       }
     }
   }
```

Ln 34, Col 1 100% Windows (CRLF) UTF-8

```
EmpId    Name     Department      Designation     Salary
101      Arun            IT       Professor       75000.0
102      Smitha          IT       Asst.Professor  45000.0
103      Joseph          Cs       Asst.Professor  45000.0
```

Lab Cycle Program

Q.1) Develop a program in Java to implement Stack operations using Class

*nstack - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class operation
{
 int top;
 int A[];
 int size;

    operation(int n)
     {
       top=-1;
       A=new int[n];
       size=n;
     }

  public void push(int num)
    {
      if(top==size-1)
        {

         System.out.println("Stack is Full");

         }
      else
         {
           A[++top]=num;

         }

   }
```

Ln 31, Col 1 | 100% | Windows (CRLF) | UTF-8

nstack - Notepad

File Edit Format View Help

```
public void pop()
  {
   int re;
    if(top==-1)
     {
      System.out.println("Stack is empty");
     }
    else
     {
      re=A[top];
      System.out.println("removed element from the stack is :"+re);
      top--;

     }
  }
public void display()
  {

  int i;
  if(top==-1)
    {
    System.out.println("Stack is empty");
    }
  else
    {
     for(i=top; i>=0;i-- )
     {
       System.out.println(A[i]);
     }

    }
  }

}
```

Ln 56, Col 7 | 100% | Windows (CRLF) | UTF-8

nstack - Notepad

File Edit Format View Help

```
class nstack
{
public static void main(String args[])
   {
     Scanner io=new Scanner(System.in);
     System.out.println("Enter the value of n:");
     int n=io.nextInt();
     int ch,num;
     char c;
     operation o=new operation(n);
       do
        {
          System.out.println("\n1.push\n2.pop\n3.display");
          System.out.println("Enter your choice(1-3):");
          ch=io.nextInt();
            switch(ch)
               {
                case 1:System.out.println("Enter the number:");
                        num=io.nextInt();
                        o.push(num);
                       break;
                 case 2:o.pop();
                       break;
                 case 3:o.display();
                       break;
                 default:System.out.println("Invalid Choice");
               }
            System.out.println("Do you want to continue(y/n):");
            c=io.next().charAt(0);
       }while(c=='Y'||c=='y');

   }
}
```

Ln 75, Col 1 100% Windows (CRLF) UTF-8

Lab Cycle Program

Q.2) Develop a program in Java to display the details of bank account using Class.

bank1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class bank1
 {
   public static void main(String arg[])
    {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the value of n:");
      int n=io.nextInt();
      account1 o[]=new account1[n];  //creating n objects
      int i,amt,flag=0,pos=0,ac;
      char c;
      for(i=0;i<n;i++)
       {
        o[i]=new account1();    // Allocating memory to each object
       }

      for(i=0;i<n;i++)
       {
        o[i].read();
       }
     System.out.println("Name:\tA/C No:\tAmount:");
     for(i=0;i<n;i++)
       {
        o[i].display();
       }
```

Ln 163, Col 1 | 100% | Windows (CRLF) | UTF-8

bank1 - Notepad

File Edit Format View Help

```
do
 {
  System.out.println("1.Balance Check\n2.Deposit\n3.Withdrow\n4.Exit");
  System.out.println("Enter your choice(1-4):");
  int ch=io.nextInt();
   switch(ch)
      {
       case 1:  System.out.println("Enter the A/C no:");
                ac=io.nextInt();
                 for(i=0;i<n;i++)
                     {
                        if(o[i].acno==ac)
                          {
                           flag=1;
                           pos=i;
                           break;
                           }
                      }
                if(flag==0)
                  {
                   System.out.println("Invalid A/C no");
                   }
                else
                    {
                     o[pos].display();
                    }
                break;
```

Ln 36, Col 23 | 100% | Windows (CRLF) | UTF-8

bank1 - Notepad

File Edit Format View Help

```
            case 2: System.out.println("Enter the A/C no:");
                    ac=io.nextInt();
                    System.out.println("Enter the Amount:");
                    amt=io.nextInt();
                       for(i=0;i<n;i++)
                          {
                             if(o[i].acno==ac)
                               {
                                flag=1;
                                pos=i;
                                break;
                                }
                           }

                    if(flag==0)
                       {
                        System.out.println("Invalid A/C no");
                        }
                     else
                         {
                          o[pos].deposit(amt);
                          o[pos].display();
                         }
                      break;
        case 3: System.out.println("Enter the A/C no:");
                    ac=io.nextInt();
                    System.out.println("Enter the Amount:");
```

Ln 36, Col 23 | 100% | Windows (CRLF) | UTF-8

bank1 - Notepad

File Edit Format View Help

```
                        amt=io.nextInt();
                           for(i=0;i<n;i++)
                              {
                                 if(o[i].acno==ac)
                                   {
                                    flag=1;
                                    pos=i;
                                    break;
                                    }
                                 }

                        if(flag==0)
                           {
                            System.out.println("Invalid A/C no");
                            }
                         else
                             {
                              o[pos].withdrow(amt);
                              o[pos].display();
                             }
                          break;
                 default:System.out.println("Invalid choice");
                }
                 System.out.println("Do you want to continue(y/n):");
                  c=io.next().charAt(0);
               }while(c=='y'||c=='Y');
    }
 }
```

Ln 104, Col 1 | 100% | Windows (CRLF) | UTF-8

bank1 - Notepad

File Edit Format View Help

```
class account1
{
String name;
int acno;
double amount;
  public account1()
   {
    amount=1000.0;
   }
  public void read()
     {
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the Customer Name:");
      name=io.next();
      System.out.println("Enter the A/C No:");
      acno=io.nextInt();
     }
   public void display()
     {
      System.out.print(name+"\t");
      System.out.print(acno+"\t");
      System.out.print(amount+"\t");
      System.out.println(" ");
     }
  public void deposit(int amt)
     {
      amount=amount+amt;
     }
```

Ln 104, Col 1 | 100% | Windows (CRLF) | UTF-8

bank1 - Notepad

File Edit Format View Help

```
public void withdrow(int amt)
     {
        if(amount-amt>=1000)
          {
          amount=amount-amt;
          }
        else
          {
         System.out.println("Insufficient amount in the A/c");
          }
     }
}
```

Ln 139, Col 7 100% Windows (CRLF) UTF-8

```
D:\java\programs\module-1>javac bank1.java

D:\java\programs\module-1>java bank1
Enter the value of n:
3
Enter the Customer Name:
Anil
Enter the A/C No:
101
Enter the Customer Name:
Sunil
Enter the A/C No:
102
Enter the Customer Name:
Rajesh
Enter the A/C No:
103
Name:    A/C No: Amount:
Anil     101     1000.0
Sunil    102     1000.0
Rajesh   103     1000.0
1.Balance Check
2.Deposit
3.Withdrow
4.Exit
Enter your choice(1-4):
1
Enter the A/C no:
102
Sunil    102     1000.0
Do you want to continue(y/n):
```

```
1.Balance Check
2.Deposit
3.Withdrow
4.Exit
Enter your choice(1-4):
1
Enter the A/C no:
102
Sunil    102     1000.0
Do you want to continue(y/n):
y
1.Balance Check
2.Deposit
3.Withdrow
4.Exit
Enter your choice(1-4):
2
Enter the A/C no:
102
Enter the Amount:
4000
Sunil    102     5000.0
Do you want to continue(y/n):
y
1.Balance Check
2.Deposit
3.Withdrow
4.Exit
Enter your choice(1-4):
3
Enter the A/C no:
102
Enter the Amount:
2000
Sunil    102     3000.0
Do you want to continue(y/n):
```

this keyword in java

- **There can be a lot of usage of java this keyword.**
- **In java, this is a reference variable <u>that refers to the current object.</u>**

Usage of java this keyword

Here is given the 6 usage of java this keyword.

1. **this can be used to refer current class instance variable.**
2. **this can be used to invoke current class method (implicitly)**
3. **this() can be used to invoke current class constructor.**
4. **this can be passed as an argument in the method call.**
5. **this can be passed as argument in the constructor call.**
6. **this can be used to return the current class instance from the method.**

1) this: to refer current class instance variable

- **The this keyword can be used to refer current class instance variable.**
- **If there is ambiguity between the instance variables and parameters, this keyword <u>resolves the problem of ambiguity.</u>**

Understanding the problem without this keyword

Student6 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Student6
  {
   int rno;
   String name;
   Student6(int rno,String name)
     {
      rno=rno;
      name=name;
     }
  public void display()
     {
      System.out.println("Roll No:"+rno);
      System.out.println("Name:"+name);
     }
   public static void main(String args[])
     {
      int r;
      String s;
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the details of first student");
      System.out.println("Enter the Roll Number:");
      r=io.nextInt();
      System.out.println("Enter the Name:");
      s=io.next();
      Student6 o1=new Student6(r,s);
```

Ln 7, Col 18 | 100% | Windows (CRLF) | UTF-8

Student6 - Notepad

File Edit Format View Help

```
        System.out.println("Enter the details of Second student");
        System.out.println("Enter the Roll Number:");
        r=io.nextInt();
        System.out.println("Enter the Name:");
        s=io.next();
        Student6 o2=new Student6(r,s);
        o1.display();
        o2.display();
        }
    }
```

Ln 7, Col 18 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac Student6.java

D:\java\programs\module-1>java Student6
Enter the details of first student
Enter the Roll Number:
101
Enter the Name:
Anil
Enter the details of Second student
Enter the Roll Number:
102
Enter the Name:
Sunil
Roll No:0
Name:null
Roll No:0
Name:null
```

- **In the above example, parameters (formal arguments) and instance variables <u>are same</u>.**
- **So, we are using this keyword to distinguish formal arguments and instance variable.**
- **Solution of the above problem by this keyword**

Student6 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Student6
  {
   int rno;
   String name;
   Student6(int rno,String name)
     {
      this.rno=rno;
      this.name=name;
     }
  public void display()
     {
      System.out.println("Roll No:"+rno);
      System.out.println("Name:"+name);
     }
   public static void main(String args[])
     {
      int r;
      String s;
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the details of first student");
      System.out.println("Enter the Roll Number:");
      r=io.nextInt();
      System.out.println("Enter the Name:");
      s=io.next();
      Student6 o1=new Student6(r,s);
```

Ln 10, Col 12 | 100% | Windows (CRLF) | UTF-8

Student6 - Notepad

File Edit Format View Help

```
        System.out.println("Enter the details of Second student");
        System.out.println("Enter the Roll Number:");
        r=io.nextInt();
        System.out.println("Enter the Name:");
        s=io.next();
        Student6 o2=new Student6(r,s);
        o1.display();
        o2.display();
        }
    }
```

Ln 10, Col 12 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac Student6.java

D:\java\programs\module-1>java Student6
Enter the details of first student
Enter the Roll Number:
101
Enter the Name:
Anil
Enter the details of Second student
Enter the Roll Number:
102
Enter the Name:
Sunil
Roll No:101
Name:Anil
Roll No:102
Name:Sunil
```

- **If formal arguments and instance variables are different, there is no need to use this keyword like in the following program:**
- **Program where this keyword is not required**

Student6 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Student6
  {
   int rno;
   String name;
   Student6(int r,String n)
     {
      rno=r;
      name=n;
     }
  public void display()
     {
      System.out.println("Roll No:"+rno);
      System.out.println("Name:"+name);
     }
   public static void main(String args[])
     {
      int r;
      String s;
      Scanner io=new Scanner(System.in);
      System.out.println("Enter the details of first student");
      System.out.println("Enter the Roll Number:");
      r=io.nextInt();
      System.out.println("Enter the Name:");
      s=io.next();
      Student6 o1=new Student6(r,s);
```

Ln 10, Col 7 | 100% | Windows (CRLF) | UTF-8

Student6 - Notepad

File Edit Format View Help

```
        System.out.println("Enter the details of Second student");
        System.out.println("Enter the Roll Number:");
        r=io.nextInt();
        System.out.println("Enter the Name:");
        s=io.next();
        Student6 o2=new Student6(r,s);
        o1.display();
        o2.display();
        }
    }
```

Ln 10, Col 7 | 100% | Windows (CRLF) | UTF-8

Command Prompt

```
D:\java\programs\module-1>javac Student6.java

D:\java\programs\module-1>java Student6
Enter the details of first student
Enter the Roll Number:
101
Enter the Name:
Anil
Enter the details of Second student
Enter the Roll Number:
102
Enter the Name:
Sunil
Roll No:101
Name:Anil
Roll No:102
Name:Sunil
```

2) this: to invoke current class method

- **You may invoke the method of the current class by using the this keyword.**
- **If you don't use the this keyword, <u>compiler automatically adds</u> this keyword while invoking the method.**
- **Let's see the example**

TestThis - Notepad

File Edit Format View Help

```
import java.io.*;
class A
{
 public void n()
   {
    System.out.println("Hello from n");
    m(); //same as this.m();
   }
  public void m()
   {
   System.out.println("Hello from m");
   }
}
class TestThis
{
public static void main(String args[])
  {
   A o=new A();
   o.n();
  }
}
```

Ln 7, Col 5 | 100% | Windows (CRLF) | UTF-8

OR

TestThis - Notepad

File Edit Format View Help

```
import java.io.*;
class A
{
 public void n()
   {
    System.out.println("Hello from n");
    this.m(); //same as m();
   }
  public void m()
   {
   System.out.println("Hello from m");
   }
}
class TestThis
{
public static void main(String args[])
  {
   A o=new A();
   o.n();
  }
}
```

Command Prompt

```
D:\java\programs\module-1>javac TestThis.java

D:\java\programs\module-1>java TestThis
Hello from n
Hello from m
```

3) this() : to invoke current class constructor

test - Notepad

File Edit Format View Help

```
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        this(10, 20);
        System.out.println("Inside  default constructor \n");
    }

    //Parameterized constructor
    Test(int a, int b)
    {
        this.a = a;
        this.b = b;
        System.out.println("a="+a+"  b="+b);
        System.out.println("Inside parameterized constructor");
    }

    public static void main(String[] args)
    {
        Test object = new Test();
    }
}
```

Ln 18, Col 9 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac Test.java

D:\java\programs\module-1>java Test
a=10  b=20
Inside parameterized constructor
Inside  default constructor
```

4) this: to pass as an argument in the method

- **The this keyword can also be passed as an argument in the method.**
- **It is mainly used in the event handling.**
- **Let's see the example:**

test - Notepad

File Edit Format View Help

```
class Test
{
    int a;
    int b;
    Test()                  // Default constructor
    {
        a = 10;
        b = 20;
    }

    // Method that receives 'this' keyword as parameter
    void display(Test obj)
    {
        System.out.println("a = " +obj.a + "  b = " + obj.b);
    }

    // Method that returns current class instance
    void get()
    {
        display(this);
    }
     public static void main(String[] args)
    {
        Test o= new Test();
        o.get();
    }
}
```

```
D:\java\programs\module-1>javac Test.java

D:\java\programs\module-1>java Test
a = 10  b = 20
```

5) this: to pass as argument in the constructor call

A4 - Notepad

File Edit Format View Help

```
class B
     {
     A4 obj;
      B(A4 obj)
         {
         this.obj=obj;
          }
      void display()
         {
         System.out.println("data="+obj.data);//using data member of A4 cl
         }
     }

class A4
 {
  int data=10;
    A4()
     {
      B b=new B(this);
      b.display();
      }
  public static void main(String args[])
     {
     A4 o=new A4();
     }
}
```

Ln 10, Col 37 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac A4.java

D:\java\programs\module-1>java A4
data=10
```

6) this keyword can be used to return current class instance

- **We can return this keyword as an statement from the method.**
- **In such case, return type of the method must be the class type (non-primitive).**
- **Let's see the example:**

Syntax of this that can be returned as a statement

```
return_type method_name(){
return this;
}
```

test - Notepad

File Edit Format View Help

```
class Test
{
    int a;
    int b;

    //Default constructor
    Test()
    {
        a = 10;
        b = 20;
    }

    //Method that returns current class instance
    Test get()
    {
        return this;
    }

    //Displaying value of variables a and b
    void display()
    {
        System.out.println("a = " + a + "  b = " + b);
    }

    public static void main(String[] args)
    {
        Test object = new Test();
        object.get().display();
    }
}
```

Ln 5, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-1>javac Test.java

D:\java\programs\module-1>java Test
a = 10  b = 20
```

Garbage Collector

- In Java, programmers **can't delete <u>the objects</u>**.
- To **delete** or **recollect that memory JVM has a program** called **<u>Garbage Collector</u>.**
- **Garbage Collectors can recollect the of objects <u>that are not referenced</u>**.
- **So Java makes** the life of a **programmer <u>easy by handling memory management.</u>**
- However, programmers should be **careful about their code** whether they are using **objects that have been used for a long time**. Because **Garbage <u>cannot recover the memory of objects</u> being referenced**.

/* write a java program to read and display a character */

*schar - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
public class schar
{
public static void main(String[] args)
{
Scanner io = new Scanner(System.in);

System.out.print("Input a character: ");

char c = io.next().charAt(0);

System.out.println("You have entered "+c);
}
}
```

Ln 12, Col 1 | 100% | Windows (CRLF) | UTF-8

/* Write a Java program to implement Do you want to continue(y/n): */

*schar1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
public class schar1
{
public static void main(String[] args)
{
Scanner io = new Scanner(System.in);
char c;
do
{
 System.out.println("hello ");
 System.out.println("Do you want to continue(y/n):");
 c=io.next().charAt(0);
}while(c=='y'||c=='Y');
}
}
```

Ln 12, Col 1 | 100% | Windows (CRLF) | UTF-8

----------End of Module-1----------

Module-2
Method overloading

Syllabus

- **Overloading Methods,**
- **Overloading Constructors,**
- **Using Objects as Parameters,**
- **Call by value and Call by reference,**
- **Access control,**
- **use of static and final keywords,**
- **Nested and Inner classes**

Overloading Methods

Example of Polymorphism in Java
Shape
area()
Circle
area (circle)
Triangle
area (triangle)
Rectangle
area (rectangle)
600 × 400

Water changes to solid when it frozen
Water changes to gas when it is heated at its boiling point
Solid
Liquid
Gas
Human Behavior
Computer ON OFF Switch
Real-time Examples of Polymorphism
Scientech Easy
600 × 400

Polymorphism in Java

- Polymorphism is considered one of the important features of Object-Oriented Programming.
- Polymorphism allows us **to perform a single action in different ways**.
- **Real life example of polymorphism:** A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called **polymorphism**.
- **In Java polymorphism is mainly divided into two types:**
    1. **Compile time Polymorphism**
    2. **Runtime Polymorphism**

1. Compile-time polymorphism

- It is also known as **static polymorphism**.
- This type of polymorphism **is achieved by function overloading** or **operator overloading**.
- But **Java doesn't support the Operator Overloading**.

Method Overloading:

- If a class has **multiple methods/functions having same name but different in parameters**, it is known as **Method Overloading**.
- **Functions can be overloaded by change in number of arguments** or/and **change in type of arguments**.

Advantage of method overloading

- Method overloading ***increases the readability of the program*.**

Different ways to overload the method

- There are **two ways to overload the method** in java
    1. **By changing number of arguments**
    2. **By changing the data type**
    3. **The Order of the parameters of methods.**

In c	In java

void area(int r);
void area(int l,int b);
void area(int b,int h)
	void area(int r);
	void area(float l,int b);
	void area(int b,float h)

method overloading not allowed in c	method overloading allowed in java

Different ways of Method Overloading in Java

- Overloaded methods are differentiated based on the number and type of the parameters passed as an argument to the methods.
- You can not define more than one method with the same name, Order and the type of the arguments. It would be a compiler error.
- The **compiler does not consider the return type while differentiating the overloaded method**. But **you cannot declare two methods with the same signature** and **different return type**. **It will throw a compile-time error**.

Example 1: By using different numbers of arguments

```java
// Java program for Method overloading

class MultiplyFun {

    // Method with 2 parameter
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method with the same name but 3 parameter
    static int Multiply(int a, int b, int c)
    {
        return a * b * c;
    }
}

class Main {
    public static void main(String[] args)
    {
        System.out.println(MultiplyFun.Multiply(2, 4));

        System.out.println(MultiplyFun.Multiply(2, 7, 3));
    }
}
```

Output:

```
8
42
```

1) Method Overloading: changing no. of arguments

- In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.
- In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
class TestOverloading1{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(11,11,11));
}}
```

Output:

```
22
33
```

Example 2: By using different types of arguments

```java
// Java program for Method overloading

class MultiplyFun {

    // Method with 2 parameter
    static int Multiply(int a, int b)
    {
        return a * b;
    }

    // Method with the same name but 2 double parameter
    static double Multiply(double a, double b)
    {
        return a * b;
    }
}

class Main {
    public static void main(String[] args)
    {

        System.out.println(MultiplyFun.Multiply(2, 4));

        System.out.println(MultiplyFun.Multiply(5.5, 6.3));
    }
}
```

Output:

```
8
34.65
```

Eg: 2 :- Multiply the two numbers by using function overloading

mul - Notepad

File Edit Format View Help

```java
import java.io.*;
class info
{
 public void multiply(int x,int y)
     {
      int res;
      res=x*y;
      System.out.println("Result="+res);
     }
 public void multiply(int x,int y)
     {
      int res;
      res=x*y;
      System.out.println("Result="+res);
     }
}
class mul
 {
  public static void main(String args[])
    {
     info o=new info();
     o.multiply(2,3);
     o.multiply(4,5);
    }
}
```

Ln 23. Col 22 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\overloading>javac mul.java
mul.java:10: error: method multiply(int,int) is already defined in class info
 public void multiply(int x,int y)
             ^
1 error
```

mul - Notepad

File Edit Format View Help

```
import java.io.*;
class info
{
 public void multiply(int x,int y)
     {
      int res;
      res=x*y;
      System.out.println("Result="+res);
     }
 public int multiply(int x,int y)
     {
      int res;
      res=x*y;
      return res;
     }
}
class mul
 {
  public static void main(String args[])
    {
     info o=new info();
     int res;
     o.multiply(2,3);
     res=o.multiply(4,5);
     System.out.println("Result="+res);
    }
}
```

Ln 14, Col 18 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\overloading>javac mul.java
mul.java:10: error: method multiply(int,int) is already defined in class info
 public int multiply(int x,int y)
            ^
mul.java:24: error: incompatible types: void cannot be converted to int
     res=o.multiply(4,5);
                   ^
2 errors
```

Ans:

mul1 - Notepad

File Edit Format View Help

```
import java.io.*;
class info
{
 public void multiply(int x,int y)
     {
      int res;
      res=x*y;
      System.out.println("Result="+res);
     }
 public void multiply(int x,int y,int z)
     {
      int res;
      res=x*y*z;
      System.out.println("Result="+res);
     }
}
class mul1
 {
  public static void main(String args[])
    {
     info o=new info();
     o.multiply(2,3);
     o.multiply(4,5,6);
    }
}
```

Ln 15, Col 7 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\overloading>java mul1
Result=6
Result=120
```

2) Method Overloading: changing data type of arguments

- In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
static int add(int a, int b){return a+b;}
static double add(double a, double b){return a+b;}
}
class TestOverloading2{
public static void main(String[] args){
System.out.println(Adder.add(11,11));
System.out.println(Adder.add(12.3,12.6));
}}
```

Output:

```
22
24.9
```

Eg: 2

mul2 - Notepad

File Edit Format View Help

```
import java.io.*;
class info
{
 public void multiply(int x,int y)
     {
      int res;
      res=x*y;
      System.out.println("Result="+res);
     }
 public void multiply(int x,float y)
     {
      float res;
      res=x*y;
       System.out.println("Result="+res);
     }
}
class mul2
 {
  public static void main(String args[])
    {
     info o=new info();
     o.multiply(2,3);
     o.multiply(4,5.2f);

    }
}
```

Ln 10, Col 37 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\overloading>java mul2
Result=6
Result=20.8
```

Method 3: By changing the Order of the parameters

```java
class Geek {

    public void geekIdentity(String name, int id)
    {

        System.out.println("geekName :" + name + " "
                            + "Id :" + id);
    }

    public void geekIdentity(int id, String name)
    {

        System.out.println("Id :" + id + " "
                            + "geekName :" + name);
    }
}

class GFG {
    public static void main(String[] args)
    {

        Geek geek = new Geek();

        geek.geekIdentity("Mohit", 1);
        geek.geekIdentity("shubham", 2);
    }
}
```

Output

```
geekName :Mohit Id :1
geekName :shubham Id :2
```

Eg:2

```java
// Java program to demonstrate working of method
// overloading in Java.

public class Sum {

    // Overloaded sum(). This sum takes two int parameters
    public int sum(int x, int y)
    {
        return (x + y);
    }

    // Overloaded sum(). This sum takes three int parameters
    public int sum(int x, int y, int z)
    {
        return (x + y + z);
    }

    // Overloaded sum(). This sum takes two double parameters
    public double sum(double x, double y)
    {
        return (x + y);
    }

    // Driver code
    public static void main(String args[])
    {
        Sum s = new Sum();
        System.out.println(s.sum(10, 20));
        System.out.println(s.sum(10, 20, 30));
        System.out.println(s.sum(10.5, 20.5));
    }
}
```

Output :

```
30
60
31.0
```

Eg: 3

mul3 - Notepad

File Edit Format View Help

```
import java.io.*;
class info
{
 public void multiply(float x,int y)
     {
      float res;
      res=x*y;
      System.out.println("Result="+res);
     }
 public void multiply(int x,float y)
     {
      float res;
      res=x*y;
       System.out.println("Result="+res);
     }
}
class mul3
 {
  public static void main(String args[])
    {
     info o=new info();
     o.multiply(2.5f,3);
     o.multiply(4,5.2f);

    }
}
```

Ln 17, Col 11 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\overloading>java mul3
Result=7.5
Result=20.8
```

2. Runtime polymorphism:

- It is also known as Dynamic Method Dispatch. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.
- Method overriding, on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```java
// Java program for Method overriding

class Parent {

    void Print()
    {
        System.out.println("parent class");
    }
}

class subclass1 extends Parent {

    void Print()
    {
        System.out.println("subclass1");
    }
}

class subclass2 extends Parent {

    void Print()
    {
        System.out.println("subclass2");
    }
}
```

```java
class TestPolymorphism3 {
    public static void main(String[] args)
    {

        Parent a;

        a = new subclass1();
        a.Print();

        a = new subclass2();
        a.Print();
    }
}
```

Output:

```
subclass1
subclass2
```

What is the difference between Overloading and Overriding?

- Overloading is about same function have different signatures. Overriding is about same function, same signature but different classes connected through inheritance.
- Overloading is an example of compiler time polymorphism and overriding is an example of run time polymorphism.

Lab Cycle Program

1. **Develop a Java program to implement functions to display an input integer, string and float values using the concept of method overloading.**

over - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class info
{
 public void display(int num)
   {
    System.out.println(num+" is an Integer number");
   }
 public void display(float num)
   {
    System.out.println(num+" is a Float number");
   }
 public void display(double num)
   {
    System.out.println(num+" is a Double number");
   }
  public void display(String num)
   {
    System.out.println(num+" is a String");
   }
}

class over
{
 public static void main(String args[])
   {
     info o=new info();
     o.display(10);
     o.display(5.78f);
     o.display(7.89d);
     o.display("Information Technology");
   }
}
```

Ln 33, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\overloading>java over
10 is an Integer number
5.78 is a Float number
7.89 is a Double number
Information Technology is a String
```

Lab Cycle Program

2.Write a Java program that displays area of different

Figures(Rectangle,Square,Triangle) using the method overloading.

shaps - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class info
{
double res;
 public void area(int r)
   {
    res=3.14*r*r;
    System.out.println("Area of Circle is="+res);
   }
public void area(int l,int b)
   {
    res=l*b;
    System.out.println("Area of Rectangle is="+res);
   }
public void area(float b,float h)
   {
    res=0.5*b*h;
    System.out.println("Area of Right Angle Triangle is="+res);
   }
}
class shaps
{
 public static void main(String args[])
   {
     Scanner io=new Scanner(System.in);
     info o=new info();
     int r,l,b;
     float b1,h;
     char c;
```

Ln 31, Col 1 | 100% | Windows (CRLF) | UTF-8

shaps - Notepad

File Edit Format View Help

```
    do
    {
    System.out.println("1.Area of Circle\n2.Area of Rectangle\n3.Area of Right Angle Triangle");
    System.out.println("Enter your Choice(1-3):");
    int ch=io.nextInt();

        switch(ch)
               {
               case 1:System.out.println("Enter the value of r:");
                      r=io.nextInt();
                      o.area(r);
                      break;
               case 2:System.out.println("Enter the value of l:");
                      l=io.nextInt();
                      System.out.println("Enter the value of b:");
                      b=io.nextInt();
                      o.area(l,b);
                      break;
               case 3:System.out.println("Enter the value of b:");
                      b1=io.nextFloat();
                      System.out.println("Enter the value of h:");
                      h=io.nextFloat();
                      o.area(b1,h);
                      break;

               default:System.out.println("Invalid Choice");
               }
       System.out.println("Do you wants to continue(y/n):");
       c=io.next().charAt(0);
     }while(c=='y'||c=='Y');
   }
}
```

Ln 31, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\overloading>java shaps
1.Area of Circle
2.Area of Rectangle
3.Area of Right Angle Triangle
Enter your Choice(1-3):
1
Enter the value of r:
2
Area of Circle is=12.56
Do you wants to continue(y/n):
y
1.Area of Circle
2.Area of Rectangle
3.Area of Right Angle Triangle
Enter your Choice(1-3):
2
Enter the value of l:
4
Enter the value of b:
5
Area of Rectangle is=20.0
Do you wants to continue(y/n):
y
1.Area of Circle
2.Area of Rectangle
3.Area of Right Angle Triangle
Enter your Choice(1-3):
3
Enter the value of b:
4
Enter the value of h:
3
Area of Right Angle Triangle is=6.0
Do you wants to continue(y/n):
n
```

Lab Cycle Program

3. Write a program to implement function overloading in order to compute power(m, n) where

i) m is double and n is int

ii) m and n are int.

*power - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class info
{
double res;
public void cal(double a,int e)
   {
    double res=1.0,b=a;
    int i;
    for(i=1;i<=e;i++)
     {
      res=res*b;
      }
     System.out.println("pow("+a+","+e+")="+res);
   }
public void cal(int a,int e)
   {
   int res=1,b=a;
  int i;
    for(i=1;i<=e;i++)
     {
      res=res*b;
      }

    System.out.println("pow("+a+","+e+")="+res);
   }
}

class power
{
 public static void main(String args[])
   {
          info o=new info();
          o.cal(2.5,2);
          o.cal(2,3);
   }
}
```

```
D:\java\programs\module-2\overloading>java power
pow(2.5,2)=6.25
pow(2,3)=8
```

Constructor overloading in Java

Eg:

Constructor overloading in Java

- In Java, we can **overload constructors like methods**.
- The constructor overloading can be defined as the concept of **having more than one constructor with different parameters** so that **every constructor can perform a different task.**

When do we need Constructor Overloading?

- Sometimes there is a **need of initializing an object in different ways**.
- This **can be done using constructor overloading**.
- Consider the following Java program, in which we have **used different constructors in the class**.
- In the example, the Student class constructor is overloaded with **two different constructors**, I.e., **default** and **parameterized**.

info - Notepad

File Edit Format View Help

```
import java.io.*;
class Student
{

 int rno;
 String name;
   Student()
    {
     System.out.println("This is a Default Constructor");
    }
  Student(int r,String n)
   {
   rno=r;
   name=n;
   System.out.println("\nThis is a Parametrized Constructor");
   }
}

class info
 {
  public static void main(String args[])
    {
     Student o1=new Student();
     System.out.println("Default Constructor Values");
     System.out.println("Roll No:"+o1.rno+"  Name:"+o1.name);

     Student o2=new Student(101,"Anil");
     System.out.println("Parametrized Constructor Values");
     System.out.println("Roll No:"+o2.rno+"  Name:"+o2.name);
    }
}
```

Ln 2, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\constructor overloading>java info
This is a Default Constructor
Default Constructor Values
Roll No:0  Name:null

This is a Parametrized Constructor
Parametrized Constructor Values
Roll No:101  Name:Anil
```

- Here, we need to understand the purpose of constructor overloading. Sometimes, **we need to use multiple constructors to initialize the different values** of the class.
- We must also notice that the **java compiler invokes a default constructor** when **<u>we do not use any constructor</u> in the class**. However, **<u>the default constructor is not</u>**

invoked if we have **used any constructor** in the class, whether **it is default** or **parameterized**.

Eg:2

- These types of problems of **different ways of initializing an object can be solved by constructor overloading**. Below is the improved version of class Box with constructor overloading.

cal - Notepad

File Edit Format View Help

```
import java.io.*;
class Box
{
 double width,height,depth;
 String name;
   Box()
    {
     width=height=depth=0.0;
    }
  Box(double len)
    {
     width=height=depth=len;
    }
   Box(double w,double h,double d)
    {
     width=w;
     height=h;
     depth=d;
    }

  public double volume()
   {
    return(width*height*depth);
   }
}

class cal
 {
  public static void main(String args[])
    {
     Box fbox=new Box();
     Box sbox=new Box(2.0);
     Box tbox=new Box(2.0,3.0,4.0);
     System.out.println("Volume of first box :"+fbox.volume());
     System.out.println("Volume of second box :"+sbox.volume());
     System.out.println("Volume of third box :"+tbox.volume());
    }
}
```

Ln 26, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\constructor overloading>java cal
Volume of first box :0.0
Volume of second box :8.0
Volume of third box :24.0
```

Use of this () in constructor overloading

- However, we **can use this keyword inside the constructor**, **which can be used to invoke the other constructor of the same class**.
- Please note, **this() should be the first statement inside a constructor.**
- Consider the following example to understand the use of **this keyword** in constructor overloading.

Studinfo - Notepad

File Edit Format View Help

```
import java.io.*;
class Student
{
 int rno,passoutyear;
 String name,contactno,collegename;

 Student(String cno,String cname,int year)
    {
     contactno=cno;
     collegename=cname;
     passoutyear=year;
    }

 Student(int r,String n)
   {
   this("9746772992","Toc H",2020);
   rno=r;
   name=n;

   }
}

class Studinfo
 {
  public static void main(String args[])
    {
    Student o=new Student(101,"Anil");

     System.out.println("\nStudent Information\n");
     System.out.println("Roll No:"+o.rno);
     System.out.println("Name:"+o.name);
     System.out.println("Contact No:"+o.contactno);
     System.out.println("College Name:"+o.collegename);
     System.out.println("Passout Year:"+o.passoutyear);
    }
}
```

Ln 27, Col 26 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\constructor overloading>java Studinfo

Student Information

Roll No:101
Name:Anil
Contact No:9746772992
College Name:Toc H
Passout Year:2020
```

Eg:2

```java
// Java program to illustrate role of this() in
// Constructor Overloading
class Box
{
    double width, height, depth;
    int boxNo;

    // constructor used when all dimensions and
    // boxNo specified
    Box(double w, double h, double d, int num)
    {
        width = w;
        height = h;
        depth = d;
        boxNo = num;
    }

    // constructor used when no dimensions specified
    Box()
    {
        // an empty box
        width = height = depth = 0;
    }

    // constructor used when only boxNo specified
    Box(int num)
    {
        // this() is used for calling the default
        // constructor from parameterized constructor
        this();

        boxNo = num;
    }
```

```
    public static void main(String[] args)
    {
        // create box using only boxNo
        Box box1 = new Box(1);

        // getting initial width of box1
        System.out.println(box1.width);
    }
}
```

Output:

```
0.0
```

- Please note, **this() should be the first statement inside a constructor.**
- this() reference can be used during constructor overloading **to call default constructor implicitly from parameterized constructor**.
- **Note :** The constructor calling should be first statement in the constructor body. For example, following **fragment is invalid** and **throws compile time error.**

```
Box(int num)
{
    boxNo = num;

    /* Constructor call must be the first
       statement in a constructor */
    this();  /*ERROR*/
}
```

Important points to be taken care while doing Constructor Overloading :

- **Constructor calling <u>must be the first statement of constructor</u> in Java**.
- If **we have defined any parameterized constructor**, **then compiler will not create default constructor**. and **vice versa if we don't define any constructor**, **the compiler creates the default constructor**(also known as no-arg constructor) by default during compilation
- **Recursive constructor calling is invalid in java**.

Constructors overloading vs Method overloading

- **constructor overloading** is somewhat **similar to method overloading**.
- If we want to have **different ways of initializing an object using different number of parameters**, **<u>then we must do</u> constructor overloading** as we do method overloading when we want different definitions of a method based on different parameters.

Lab Cycle Program

Q) Write a Java program that displays that displays the time in different formats in the form of HH,MM,SS using constructor Overloading.

time - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class set
{
int hh,mm,ss;
double h,m,s;
 set(int h,int m,int s)
 {
  hh=h;
  mm=m;
  ss=s;
  System.out.println("HH:MM:SS  "+hh+":"+mm+":"+ss);
  }

set(double hh,double mm,double ss)
 {
  h=hh;
  m=mm;
  s=ss;
  System.out.println("HH:MM:SS "+h+":"+m+":"+s);
  }
}

class time
{
public static void main(String args[])
  {
   Scanner io=new Scanner(System.in);
  char c;
   do
    {
     System.out.println("1.Normal Time format\n2.Railway Time format");
     System.out.println("\nEnter your choice(1-2):");
     int ch=io.nextInt();
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

time - Notepad

File Edit Format View Help

```
     switch(ch)
     {
      case 1:System.out.println("\nEnter the Hour:");
             int h=io.nextInt();
             System.out.println("\nEnter the Minutes:");
             int m=io.nextInt();
             System.out.println("\nEnter the Second:");
             int s=io.nextInt();
              set o1=new set(h,m,s);
              break;
      case 2:System.out.println("\nEnter the Hour:");
             double h1=io.nextDouble();
             System.out.println("\nEnter the Minutes:");
             double m1=io.nextDouble();
             System.out.println("\nEnter the Second:");
             double s1=io.nextDouble();
             set o2=new set(h1,m1,s1);
              break;
    default:System.out.println("Invalid Choice");
   }
    System.out.println("Do you want to continue(y/n):");
    c=io.next().charAt(0);

    }while(c=='y'||c=='Y');
  }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\constructor overloading>java time
1.Normal Time format
2.Railway Time format

Enter your choice(1-2):
1

Enter the Hour:
9

Enter the Minutes:
45

Enter the Second:
30
HH:MM:SS  9:45:30
Do you want to continue(y/n):
y
1.Normal Time format
2.Railway Time format

Enter your choice(1-2):
2

Enter the Hour:
14

Enter the Minutes:
25

Enter the Second:
35
HH:MM:SS 14.0:25.0:35.0
Do you want to continue(y/n):
n
```

Using Objects as Parameters

- **When we pass a primitive type to a method**, it is **passed by value**.
- But **when we pass an object to a method**, the situation changes dramatically, because objects are passed by what is effectively **call-by-reference.**

In Java we can pass objects to methods.
For example, consider the following program :

info - Notepad

File Edit Format View Help

```
import java.io.*;

class Test
{
int a,b;
  Test(int x,int y)
     {
      a=x;
      b=y;
     }
  boolean check(Test o)
   {
    return (o.a==a && o.b==b);
   }
}

class info
{
public static void main(String args[])
  {
   Test ob1=new Test(100,22);
   Test ob2=new Test(100,22);
   Test ob3=new Test(-1,-1);

   System.out.println("ob1==ob2:"+ob1.check(ob2));
   System.out.println("ob1==ob3:"+ob1.check(ob3));
   }
}
```

Ln 13, Col 31 | 100% | Windows (CRLF) | UTF-8

Output:

```
ob1 == ob2: true
ob1 == ob3: false
```

Defining a constructor that takes an object of its class as a parameter

info - Notepad

File Edit Format View Help

```
import java.io.*;

class Box
{
double width,height,depth;
  Box(double w,double h,double d)
     {
      width=w;
      height=h;
      depth=d;
     }
  Box(Box o)
   {
    width=o.width;
    height=o.height;
    depth=o.depth;
   }
  public double volume()
   {
    return(width*height*depth);
   }
}

class info
{
public static void main(String args[])
  {
   Box o1=new Box(10,20,15);

   System.out.println("Volume of First Box="+o1.volume());

   Box o2=new Box(o1);

   System.out.println("Volume of Second Box="+o2.volume());
  }
}
```

Ln 16, Col 18 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\oject as arguments>java info
Volume of First Box=3000.0
Volume of Second Box=3000.0
```

Eg:2

stud - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Student
{
int rno;
String name;
  Student(int r,String n)
     {
      rno=r;
      name=n;
     }
  Student(Student o)
   {
    rno=o.rno;
    name=o.name;
   }
 public void display()
   {
    System.out.println("Roll No:"+rno);
    System.out.println("Name:"+name);
   }
}
class stud
{
public static void main(String args[])
  {
  Scanner io=new Scanner(System.in);
  System.out.println("Enter the Roll Number:");
  int r=io.nextInt();
  System.out.println("Enter the Name:");
  String s=io.next();
  Student o1=new Student(r,s);
  System.out.println("Information about the first Student:");
  o1.display();
  Student o2=new Student(o1);
  System.out.println("Information about the Second Student:");
  o2.display();
  }
}
```

Ln 20, Col 32 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\oject as arguments>java stud
Enter the Roll Number:
101
Enter the Name:
Anil
Information about the first Student:
Roll No:101
Name:Anil
Information about the Second Student:
Roll No:101
Name:Anil
```

Returning Objects

- In java, a **method can return any type of data**, **<u>including objects</u>**.
- For example, in the following program, the **incrByTen( )** method returns an object in which the value of a (an integer variable) is ten greater than it is in the invoking object.

value - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Info
{
int a;

  Info(int x)
     {
      a=x;
     }
  Info incByTen()
   {
    Info temp=new Info(a+10);
    return temp;
   }
 public void display()
   {
    System.out.println("a:"+a);

   }
}
class value
{
public static void main(String args[])
  {
  Scanner io=new Scanner(System.in);
  System.out.println("Enter the Value of a:");
  int a=io.nextInt();
   Info o1=new Info(a);
   o1.display();

   Info o2;
   o2=o1.incByTen();
   o2.display();
  }
}
```

Ln 29, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\oject as arguments>java value
Enter the Value of a:
5
a:5
a:15
```

Recursion

What is Recursion?

- The process in which **a function calls itself directly** or **indirectly is called recursion** and **the corresponding function is called as recursive function.**
- **Using recursive algorithm**, **certain problems can be solved quite easily**.
- Examples of **such problems are**

    1. **Towers of Hanoi (TOH),**
    2. **Inorder/Preorder/Postorder Tree Traversals,**
    3. **DFS of Graph, etc.**

- **Recursion in java** is a process in which **a method calls itself continuously.**
- A **method in java that calls itself** is called **recursive method.**
- **It makes the code compact but complex to understand**.
- Recursion is a process by which **a function calls itself repeatedly**, **until some specified condition has been satisfied**.
- In order **to solve a problem recursively**, **two conditions must be satisfied**. **First**, **the problem must be written in a recursive form**, and **second**, **the problem statement must include a stopping condition.**

Java Recursion Example 1: Finite times

info - Notepad

File Edit Format View Help

```
import java.io.*;

class info
{
int count;

 info()
 {
  count=0;
 }

  public void p()
   {
    count++;
    if(count<=5)
      {
       System.out.println("hello "+count);
       p();
      }
    }

 public static void main(String args[])
   {

    info o=new info();
    o.p();
    }
}
```

Ln 22, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\recursion>java info
hello 1
hello 2
hello 3
hello 4
hello 5
```

Java Recursion Example 2: Factorial Number

factorial - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class factorial
{
int res=1;

  public int fact(int num)
  {

   if(num==1)
     {
      return 1;
     }
    else
     {
      res=num*fact(num-1);
     }
   return res;
   }

 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    factorial o=new factorial();
    System.out.println("Enter the number:");
    int num=io.nextInt();
    int res=o.fact(num);
    System.out.println("Factorial of "+num+" is:"+res);
    }
}
```

Ln 7, Col 2 | 100% | Windows (CRLF) | UTF-8

Working of above program:

```
factorial(5)
   factorial(4)
      factorial(3)
         factorial(2)
            factorial(1)
               return 1
            return 2*1 = 2
         return 3*2 = 6
      return 4*6 = 24
   return 5*24 = 120
```

```
D:\java\programs\module-2\recursion>java factorial
Enter the number:
5
Factorial of 5 is:120
```

Java Recursion Example 3: Fibonacci Series

fibonacci - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class fibonacci
{
int res=1;
   public int fib(int num)
   {
      if(num==0)
      {
       return 0;
      }
     else if(num==1)
      {
      return 1;
      }
     else
      {
       return(fib(num-1)+fib(num-2));
      }
   }
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    fibonacci o=new fibonacci();
    System.out.println("Enter the number of terms in the series:");
    int n=io.nextInt();
    int i;
    for(i=0;i<n;i++)
    {
     int res=o.fib(i);
     System.out.print(res+",");
    }
  }
}
```

Ln 21, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\recursion>java fibonacci
Enter the number of terms in the series:
10
0,1,1,2,3,5,8,13,21,34,
```

GCD (a,b) by using Recursion

GCD (Example of recursive algorithm)

The **Greatest Common Divisor** of two positive integers a and b is the greatest number that divides both a and b.

Given two integers a and b, the greatest common **greatest common divisor** (GCD) is recursively found using the formula:

$$\gcd(a,b) = \begin{cases} a & \text{if } b=0 \\ b & \text{if } a=0 \\ \gcd(b,\ a \bmod b) & \end{cases}$$

Base case

General case

Algorithm:

GCD(a, b)

Input: integers a > 0, b ≥ 0

Output: GCD (a, b)

1. If b = 0 then return (a)
2. else return GCD(b, a mod b)

divisor - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class divisor
{

   public int gcd(int a,int b)
   {
      if(b==0)
      {
       return a;
      }
      else
      {
       return(gcd(b,a%b));
      }
   }
 public static void main(String args[])
   {
    Scanner io=new Scanner(System.in);
    divisor o=new divisor();
    System.out.println("Enter the First number ");
    int a=io.nextInt();
    System.out.println("Enter the Second number ");
    int b=io.nextInt();
    //int res=;
     System.out.print("gcd("+a+","+b+")="+o.gcd(a,b));
  }
}
```

Ln 29, Col 8 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\recursion>java divisor
Enter the First number
7
Enter the Second number
49
gcd(7,49)=7
D:\java\programs\module-2\recursion>java divisor
Enter the First number
14
Enter the Second number
28
gcd(14,28)=14
D:\java\programs\module-2\recursion>java divisor
Enter the First number
4
Enter the Second number
0
gcd(4,0)=4
```

Advantages of recursion

1. **The code may be easier to write.**
2. **To solve such problems which are naturally recursive such as tower of Hanoi.**
3. **Reduce unnecessary calling of function.**
4. **Extremely useful when applying the same solution.**
5. **Recursion reduce the length of code.**
6. **It is very useful in solving the data structure problem.**
7. **Stacks evolutions and infix, prefix, postfix evaluations etc.**

Disadvantages of recursion

1. Recursive functions are generally slower than non-recursive function.

2. It may require a lot of memory space to hold intermediate results on the system stacks.

3. Hard to analyze or understand the code.

4. It is not more efficient in terms of space and time complexity.

5. The computer may run out of memory if the recursive calls are not properly checked.

Recursion vs Iteration

Recursion	Iteration
Recursion refers to a situation where a function calls itself again and again until some base condition is not reached.	Iteration refers to a situation where some statements are executed again and again using loops until some condition is true.
The stack is used to store the set of new local variables and parameters each time the function is called.	Doesn't use stack.
Size of code is comparatively smaller in recursion.	Iteration makes the code size bigger.
Recursion is always applied to functions.	Iteration is applied to iteration statements or "loops".
Slow in execution.	Fast in execution.
Not all problems have recursive solution.	Any recursive problem can be solved iteratively.
It is less efficient in terms of memory utilization and execution speed.	It is more efficient in terms of memory utilization and execution speed.
Example: *int fact(int n)* *{* *if (n==0)* *return 1;* *else* *return n*fact(n-1);* *}*	Example: *int fact(int n)* *{* *int i, result=1;* *if (n==0)* *return 1;* *else* *for(i=1;i<=n;i++)* *result=result*i;* *return result;* *}*

Call by Value and Call by Reference in Java

- **There is only call by value in java, <u>not</u> call by reference.**
- **If we call a method passing a value, it is known as call by value.**
- **The changes being done in the called method, <u>is not affected</u> in the calling method.**

Example of call by value in java

- In case of **call by value <u>original value is not changed</u>**.
- Let's take a simple example:

Operation - Notepad

File Edit Format View Help

```
import java.io.*;
class Operation
{
int data=50;

 public void changes(int data)
   {
    data=data+100;
    System.out.println("In method After change data="+data);
   }
 public static void main(String args[])
   {
    Operation o=new Operation();
    System.out.println("Before change data="+o.data);
    o.changes(500);
    System.out.println("After change data="+o.data);
   }
}
```

Ln 2, Col 1 | 100% | Windows (CRLF) | UTF-8

```
Before change data=50
In method After change data=600
After change data=50
```

Another Example of call by value in java

- In case of **call by reference original value is changed if we made changes in the called method**.
- **If we pass object in place of any primitive value, original value will be changed**.
- In this example we are **passing object as a value**.
- Let's take a simple example:

Operation1 - Notepad

File Edit Format View Help

```
import java.io.*;
class Operation1
{
int data=50;

 public void changes(Operation1 o)
   {
    o.data=o.data+100;
    System.out.println("In method After change data="+o.data);
   }
 public static void main(String args[])
   {
    Operation1 o=new Operation1();
    System.out.println("Before change data="+o.data);
    o.changes(o);
    System.out.println("After change data="+o.data);
   }
}
```

Ln 15, Col 16 | 100% | Windows (CRLF) | UTF-8

```
Before change data=50
In method After change data=150
After change data=150
```

Access Modifiers in Java

- There are **two types of modifiers in Java**:
    1. **access modifiers** and
    2. **non-access modifiers**.
- The **access modifiers in Java specifies the accessibility** or **scope** of a **field**, **method**, **constructor**, or **class**.
- **We can change the access level** of **fields, constructors, methods**, and **class by applying the access modifier** on it.
- There are **four types** of **Java access modifiers**:

1. **Private:** The access level of a **private modifier** is **only within the class**.

- It **cannot be accessed from outside the class**.

2. Default: The access level of a **default modifier** is **only within the package**.

- It **cannot be accessed from outside the package**.
- If you **<u>do not specify any access level</u>**, **it will be the default**.

3. Protected: The access level of a **protected modifier** is **within the package** and **outside the package through child class**.

- If you **do not make the child class**, **it cannot be accessed from**

outside the package.

4. Public: The access level of a **public modifier is everywhere**.

- It **can be accessed from within the class**, **outside the class**, **within**

the package and **outside the package**.

- **There are many non-access modifiers**, **such as static**, **abstract**, **synchronized**, **native**, **volatile**, **transient**, etc. Here, **we are going to learn the access modifiers only.**

Understanding Java Access Modifiers

Let's understand the access modifiers in Java by a simple table.

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

1) Private

- **The private access modifier is accessible only within the class.**
- **Private**: The private access modifier is specified using the keyword **private**.

  - The **methods** or **data members declared as private** **<u>are accessible only within the class</u>** in which they are declared.
  - **Any other class of** the **same package** **<u>will not be able to access</u>** **these members.**
  - **Top-level classes** or **interfaces can not be declared as private because**
    - **private** means "**only visible within the enclosing class**".
    - **protected** means "**only visible within the enclosing class** and **any subclasses**"

Simple example of private access modifier

- In this example, we have created **two classes A** and **Simple**.
- **A class contains private data member** and **private method.**
- We are **accessing these private members from outside the class**, so there is a **compile-time error**.

Simple - Notepad

File Edit Format View Help

```
import java.io.*;
class A
{
private int data=40;
private void msg()
   {
    System.out.println("Hello Java");
   }
}

class Simple
{
 public static void main(String args[])
   {
    A o=new A();
    System.out.println("Value of data="+o.data);
    o.msg();
   }
}
```

```
D:\java\programs\module-2\access specifier>javac Simple.java
Simple.java:16: error: data has private access in A
    System.out.println("Value of data="+o.data);
                                          ^
Simple.java:17: error: msg() has private access in A
    o.msg();
     ^
2 errors
```

OR

```
class A{
private int data=40;
private void msg(){System.out.println("Hello java");}
}

public class Simple{
 public static void main(String args[]){
   A obj=new A();
   System.out.println(obj.data);//Compile Time Error
   obj.msg();//Compile Time Error
   }
}
```

Role of Private Constructor

- **If you make any class constructor private**, you **cannot create the instance of that class from outside the class**.
- For example:

Simple1 - Notepad

File Edit Format View Help

```
import java.io.*;
class A
{
private A()
         {
         }
private void msg()
   {
    System.out.println("Hello Java");
   }
}

class Simple1
{
 public static void main(String args[])
   {
    A o=new A();
   }
}
```

Ln 13, Col 14 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\access specifier>javac Simple1.java
Simple1.java:17: error: A() has private access in A
    A o=new A();
        ^
1 error
```

OR

```
class A{
private A(){}//private constructor
void msg(){System.out.println("Hello java");}
}
public class Simple{
 public static void main(String args[]){
   A obj=new A();//Compile Time Error
 }
}
```

2) Default

- If you don't use any modifier, it is treated as **default** by default.
- The default modifier is accessible **only within package**.
- It **cannot be accessed from outside the package**.
- It **provides more accessibility than private**.
- But, **it is more restrictive than protected**, and **public**.
- **Default**: When **no access modifier is specified** for a class, method, or data member – It is said to be having the **default access modifier by default**.
    - The data members, class or methods which are **not declared using any access modifiers** i.e. having default access modifier are accessible **only within the same package**.

Example of default access modifier

- **In this example, we have created two packages pack and mypack. We are accessing the A class from outside its package, since A class is not public, so it cannot be accessed from outside the package.**

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
   A obj = new A();//Compile Time Error
   obj.msg();//Compile Time Error
  }
}
```

- **In the above example, the scope of class A and its method msg() is default so it cannot be accessed from outside the package.**

3) Protected

- The **protected access modifier** is **accessible within package** and **outside the package but through inheritance only.**
- The **protected access modifier can be applied on the data member**, **method** and **constructor**.
- It **can't be applied on the class**.
- It **provides more accessibility than the default modifier.**
- **protected**: The protected access modifier is specified using the keyword **protected**.
    - The methods or data members declared as protected are **accessible within the same package or subclasses in different packages.**

Example of protected access modifier

- In this example, we have created the two packages pack and mypack.
- The A class of pack package is public, so can be accessed from outside the package.
- But msg method of this package is declared as protected, so it can be accessed from outside the class only through inheritance.

```java
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```java
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

```
Output:Hello
```

4) Public

- The **public access modifier is accessible everywhere.**
- It has the **widest scope among all other modifiers**.
- **public**: The public access modifier is specified using the keyword **public**.
    - The public access modifier has the **widest scope among all other access modifiers.**
    - **Classes**, **methods**, or **data members** that **are declared as public** are **accessible from everywhere in the program**.
    - **There is no restriction on the scope of public data members**.
-

Example of public access modifier

```
//save by A.java

package pack;
public class A{
public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

Java Access Modifiers with Method Overriding

- **If you are overriding any method, overridden method (i.e. declared in subclass) must not be more restrictive.**

```
class A{
protected void msg(){System.out.println("Hello java");}
}

public class Simple extends A{
void msg(){System.out.println("Hello java");}//C.T.Error
 public static void main(String args[]){
   Simple obj=new Simple();
   obj.msg();
   }
}
```

- **The default modifier is more restrictive than protected. That is why, there is a compile-time error.**

Java static keyword

- The **static keyword** in Java **is used for memory management mainly.**
- **We can apply static keyword with variables**,

methods, **blocks** and **nested classes**.

- The static can be:

  1. **Variable (also known as a class variable)**
  2. **Method (also known as a class method)**
  3. **Block**

4. **Nested class**

- **To create a static member**(block,variable,method,nested class), **precede its declaration with the keyword *static***.
- **When a member is declared static**, **<u>it can be accessed</u> before any objects of its class are created**, and **<u>without reference to any object</u>**.
- For example, in below java program, we **are accessing static method *m1()* without creating any object of *Test* class**.

without using static

Test1 - Notepad

File Edit Format View Help

```
import java.io.*;
class Test1
{

 void m1()
   {
   System.out.println("From m1");
   }

public static void main(String args[])
   {
     m1();
    }
}
```

Ln 5, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\static>javac Test1.java
Test1.java:12: error: non-static method m1() cannot be referenced from a static context
    m1();
    ^
1 error
```

using static

Test - Notepad

File Edit Format View Help

```
import java.io.*;
class Test
{

static void m1()
  {
  System.out.println("From m1");
  }

public static void main(String args[])
  {
    m1();
   }
}
```

Ln 14, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\static>java Test
From m1
```

1) Java static variable

- If you **declare any variable as static**, **it is known as** a **static variable**.
- The **static variable** **can be used to refer to the common property of all objects** (which is not unique for each object), for example, **the company name of employees**, **college name of students**, etc.
- The **static variable** **gets memory only once** in the class area **at the time of class loading**.

Advantages of static variable

- It makes your program **memory efficient** (i.e., it **saves memory**).

Understanding the problem without static variable

- Suppose there are 500 students in my college, now all instance **data members will get memory each time when the** **object is created**.
- **All students have** its **unique rollno** and **name**, so **instance data member is good in such case**.
- Here, "**college**" refers to the **common property of all** **objects**.
- If **we make it static**, **this field will** **get the memory** **only once**.
- Java **static property is shared to all objects**.

Understanding the problem static variable

```
class Student
{
    int rollno;
    String name;
    Static String college;
}
```

If **we make it static**, **this field will get the memory only once**.

```
Student o1=new Student();
Student o2=new Student();
Student o3=new Student();
```

Example of static variable

```
//Java Program to demonstrate the use of static variable
class Student{
  int rollno;//instance variable
  String name;
  static String college ="ITS";//static variable
  //constructor
  Student(int r, String n){
  rollno = r;
  name = n;
  }
  //method to display the values
  void display (){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to show the values of objects
public class TestStaticVariable1{
 public static void main(String args[]){
 Student s1 = new Student(111,"Karan");
 Student s2 = new Student(222,"Aryan");
 //we can change the college of all objects by the single line of code
 //Student.college="BBDIT";
 s1.display();
 s2.display();
 }
}
```

Output:

```
111 Karan ITS
222 Aryan ITS
```

college=ITS
Class Area
id=222;
name=Aryan;
id=111;
name=Karan;
s2
s1
Stack Memory
Heap Memory

Eg:2

info - Notepad

File Edit Format View Help

```
import java.io.*;
class Student
{
int rno;
String name;
static String college="TIST";
 Student(int r,String n)
    {
     rno=r;
     name=n;
    }
public void display()
  {
  System.out.print("Roll No:"+rno);
  System.out.print("\tName:"+name);
  System.out.print("\tCollege:"+college);
  System.out.println(" ");
  }
}
class info
{
public static void main(String args[])
  {
     Student o1=new Student(101,"Anil");
     Student o2=new Student(102,"Sunil");
     Student o3=new Student(103,"Surya");
     o1.display();
     o2.display();
     o3.display();
   }
}
```

Ln 25, Col 39 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\static>java info
Roll No:101     Name:Anil       College:TIST
Roll No:102     Name:Sunil      College:TIST
Roll No:103     Name:Surya      College:TIST
```

Program of the counter without static variable

counter - Notepad

File Edit Format View Help

```
import java.io.*;
class Counter
{
int count=0;

 Counter()
    {
     count++;
     System.out.println("count:"+count);
    }
public static void main(String args[])
  {
     Counter o1=new Counter();
     Counter o2=new Counter();
     Counter o3=new Counter();

   }
}
```

Ln 2, Col 14 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\static>java Counter
count:1
count:1
count:1
```

Program of counter by static variable

- As we have mentioned above, static variable will get the memory only once, if any object changes the value of the static variable, it will retain its value.

Counter1 - Notepad

File Edit Format View Help

```
import java.io.*;
class Counter1
{
static int count=0;

 Counter1()
    {
     count++;
     System.out.println("count:"+count);
    }
public static void main(String args[])
  {
     Counter1 o1=new Counter1();
     Counter1 o2=new Counter1();
     Counter1 o3=new Counter1();

   }
}
```

Ln 18, Col 2 | 100% | Windows (CRLF) | UTF-8

- When a variable is declared as static, then a **single copy of variable is created** and **shared among all objects** at class level.
- **Static variables** are, essentially, **global variables**.
- **All instances of the class share the same static variable**.

Important points for static variables :-

- **We can create static variables at class-level only**. See here
- **static block** and **static variables are executed in order they are present in a program.**

2) Java static method

- If you **apply static keyword with any method**, **it is known as static method**.
- A static method belongs to the class rather than the object of a class.
- A **static method can be invoked without the need for creating an instance of a class.**
- A **static method can access static data member** and **can change the value of it**.

Example of static method

```
//Java Program to demonstrate the use of a static method.
class Student{
    int rollno;
    String name;
    static String college = "ITS";
    //static method to change the value of static variable
    static void change(){
    college = "BBDIT";
    }
    //constructor to initialize the variable
    Student(int r, String n){
    rollno = r;
    name = n;
    }
    //method to display values
    void display(){System.out.println(rollno+" "+name+" "+college);}
}
//Test class to create and display the values of object
public class TestStaticMethod{
   public static void main(String args[]){
   Student.change();//calling change method
   //creating objects
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   Student s3 = new Student(333,"Sonoo");
```

```
    //calling display method
    s1.display();
    s2.display();
    s3.display();
    }
}
```

Test it Now

```
Output:111 Karan BBDIT
       222 Aryan BBDIT
       333 Sonoo BBDIT
```

Eg:2

info1 - Notepad

File Edit Format View Help

```
import java.io.*;
class Student
{
int rno;
String name;
static String college="Tist";
 Student(int r,String n)
    {
     rno=r;
     name=n;
    }
static void change()
{
college="Toc H Institute of Science and Technology";
}
public void display()
  {
  System.out.print("Roll No:"+rno);
  System.out.print("\tName:"+name);
  System.out.print("\tCollege:"+college);
  System.out.println(" ");
  }
}
class info1
{
public static void main(String args[])
  {
     Student.change();
     Student o1=new Student(101,"Anil");
     Student o2=new Student(102,"Sunil");
     Student o3=new Student(103,"Surya");
     o1.display();
     o2.display();
     o3.display();
   }
}
```

Ln 28, Col 23 | 100% | Windows (CRLF) | UTF-8

A **static method "change()" can access static data member "college"** and **<u>can change the value of it</u>**.

```
D:\java\programs\module-2\static>java info1
Roll No:101     Name:Anil       College:Toc H Institute of Science and Technology
Roll No:102     Name:Sunil      College:Toc H Institute of Science and Technology
Roll No:103     Name:Surya      College:Toc H Institute of Science and Technology
```

Another example of a static method that performs a normal calculation

```
//Java Program to get the cube of a given number using the static method

class Calculate{
  static int cube(int x){
  return x*x*x;
  }

  public static void main(String args[]){
  int result=Calculate.cube(5);
  System.out.println(result);
  }
}
```

```
Output:125
```

Restrictions for the static method

- Methods declared as static have several restrictions:

1. **The static method can not use non static data member or call non-static method directly.**
2. **this and super cannot be used in static context.**
3. **They can only directly call other static methods.**
4. **They can only directly access static data.**
5. **They cannot refer to <u>this</u> or <u>super</u> in any way**

When to use static variables and methods?

- **Use the static variable for the property that is common to all objects.**
- **For example, in class Student, all students shares the same college name.**
- **Use static methods for changing static variables.**
- **Consider the following java program, that illustrate the use of *static* keyword with variables and methods.**

Eg:1

```java
// A java program to demonstrate use of
// static keyword with methods and variables

// Student class
class Student
{
    String name;
    int rollNo;

    // static variable
    static String cllgName;

    // static counter to set unique roll no
    static int counter = 0;


    public Student(String name)
    {
        this.name = name;

        this.rollNo = setRollNo();
    }

    // getting unique rollNo
    // through static variable(counter)
    static int setRollNo()
    {
        counter++;
        return counter;
    }

    // static method
    static void setCllg(String name){
        cllgName = name ;
    }
```

```java
    // instance method
    void getStudentInfo(){
        System.out.println("name : " + this.name);
        System.out.println("rollNo : " + this.rollNo);

        // accessing static variable
        System.out.println("cllgName : " + cllgName);
    }
}

//Driver class
public class StaticDemo
{
    public static void main(String[] args)
    {
        // calling static method
        // without instantiating Student class
        Student.setCllg("XYZ");

        Student s1 = new Student("Alice");
        Student s2 = new Student("Bob");

        s1.getStudentInfo();
        s2.getStudentInfo();

    }
}
```

Output:

```
name : Alice
rollNo : 1
cllgName : XYZ
name : Bob
rollNo : 2
cllgName : XYZ
```

Q) Why is the Java main method static?

- Ans) It is because the **object is not required to call a static method.**
- If it were a **non-static method**, JVM **creates an object first** then call main() method that will lead the problem of **extra memory allocation**.

3) Java static block

- **Is used to initialize the static data member**.
- **It is executed before the main method at the time of class loading.**

Example of static block

A2 - Notepad

File Edit Format View Help

```
import java.io.*;

class A2
{
 static
   {
    System.out.println("Static block is Invoked");
   }

 public static void main(String args[])
   {
    System.out.println("Hello main");
  }
}
```

Ln 9, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\static>java A2
Static block is Invoked
Hello main
```

- **If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded. Consider the following java program demonstrating use of static blocks.**

Test2 - Notepad

File Edit Format View Help

```
import java.io.*;
class Test2
{
 static int a=4;
 static int b;
 static
   {
    System.out.println("Static block is Initialized");
    b=a*10;
   }
 public static void main(String args[])
   {
    System.out.println("from  main");
    System.out.println("value of a="+a);
    System.out.println("value of b="+b);
  }
}
```

Ln 11, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\static>java Test2
Static block is Initialized
from  main
value of a=4
value of b=40
```

Q) Can we execute a program without main() method?

Ans) No, one of the ways was the static block, but it was possible till JDK 1.6. Since JDK 1.7, it is not possible to execute a Java class without the <u>main method</u>.

final keyword in java

- ***final* keyword is used in different contexts**.
- First of all, ***final*** is a **<u>non-access modifier</u> <u>applicable only to a</u> variable, a method or a class**.
- Following are different contexts where final is used.

- The **final keyword can be applied with the variables.**
- a **final variable that have no value** it is called **blank final**

variable or **uninitialized final variable**.

It can be initialized in the **constructor only**.

- The **blank final variable can be static** also **which will be initialized** in the **<u>static block only.</u>**

1. Final variables

- When a variable is declared with ***final*** **keyword**, **its value** **<u>can't be modified</u>**, essentially, **a constant**.
- i.e, you cannot change the value of final variable(It will be constant).
- This also means that you **must initialize a final variable**.
- It is good practice to represent **final variables** in **all uppercase**, **using underscore to separate words**.

```
class Bike9{
 final int speedlimit=90;//final variable
 void run(){
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}//end of class
```

```
Output:Compile Time Error
```

Examples :

```
// a final variable
final int THRESHOLD = 5;
// a blank final variable
final int THRESHOLD;
// a final static variable PI
static final double PI = 3.141592653589793;
// a  blank final static  variable
static final double PI;
```

Initializing a final variable :

- We must initialize a final variable, otherwise compiler will throw compile-time error.
- A final variable can only be initialized once, either via an initializer or an assignment statement.
- There are **three ways to initialize** a **final variable** :

1. You can **initialize a final variable when it is declared**. **This approach is the most common.**

- A **final variable** is called **blank final variable**, if it is **not initialized while declaration.**
- Below are the **two ways to initialize a blank final variable**.

2. A **blank final variable can be initialized inside** instance-initializer block or **inside constructor.**

- If you have **more than one constructor** in your class then **it must be initialized in all of them**, **otherwise compile time error will be thrown**.

3. A **blank final static variable** can be **initialized inside static block.**

Let us see above different ways of initializing a final variable through an example.

```
//Java program to demonstrate different
// ways of initializing a final variable

class Gfg
{
    // a final variable
    // direct initialize
    final int THRESHOLD = 5;

    // a blank final variable
    final int CAPACITY;

    // another blank final variable
    final int  MINIMUM;

    // a final static variable PI
    // direct initialize
    static final double PI = 3.141592653589793;

    // a  blank final static  variable
    static final double EULERCONSTANT;

    // instance initializer block for
    // initializing CAPACITY
    {
        CAPACITY = 25;
    }

    // static initializer block for
    // initializing EULERCONSTANT
    static{
        EULERCONSTANT = 2.3;
    }
```

```
    // constructor for initializing MINIMUM
    // Note that if there are more than one
    // constructor, you must initialize MINIMUM
    // in them also
    public GFG()
    {
        MINIMUM = -1;
    }

}
```

When to use a final variable :

- The only **difference between** a **normal variable** and **a final variable** is that **we can re-assign value to a normal variable** but **we cannot change the value of a final variable once assigned**.
- Hence final variables must be used only for the values that we want to **remain constant throughout the execution of program**.

Note :

1. As discussed above, a final variable cannot be reassign, doing it will throw compile-time error.

```
// Java program to demonstrate re-assigning
// final variable will throw compile-time error

class Gfg
{
    static final int CAPACITY = 4;

    public static void main(String args[])
    {
        // re-assigning final variable
        // will throw compile-time error
        CAPACITY = 5;
    }
}
```

Output

```
Compiler Error: cannot assign a value to final variable CAPACITY
```

2. When a final variable is created inside a method/constructor/block, it is called local final variable, and it must initialize once where it is created. See below program for local final variable

```java
// Java program to demonstrate
// local final variable

// The following program compiles and runs fine

class Gfg
{
    public static void main(String args[])
    {
        // local final variable
        final int i;
        i = 20;
        System.out.println(i);
    }
}
```

Output:

```
20
```

3. Note the difference between **C++ *const* variables** and **Java *final* variables**.

- **const variables in C++ must be assigned a value <u>when declared</u>**.
- For **final variables in Java**, **it is not necessary** as we see in above examples.
- A **final variable can be <u>assigned value later</u>, <u>but only once.</u>**

2. Final classes

- When **a class is declared with *final* keyword**, it is called a **final class**.
- A **final class <u>cannot be</u> extended(inherited).**
- There are **two uses** of a **final class** :

1. One is definitely **to prevent <u>inheritance</u>**, as **final classes <u>cannot be extended</u>**.

- For example, all Wrapper Classes like Integer,Float etc. are final classes. We can not extend them.

- **If you make any class as final, you cannot extend it.**

Example of final class

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();
  }
}
```

```
Output:Compile Time Error
```

Eg:2

```
final class A
{
     // methods and fields
}
// The following class is illegal.
class B extends A
{
    // COMPILE-ERROR! Can't subclass A
}
```

2. The other use of final with classes is **to <u>create an immutable (unchangeable) class</u> like the predefined <u>String</u> class**.

- You **can not make a class immutable <u>without making it final</u>**.

3. Final methods

- When **a method is declared with *final* keyword**, it is called a **final method.**
- **If you make any method as final**, **<u>you cannot override it.</u>**
- i.e A **final method** cannot be **<u>overridden</u>**.
- The <u>Object</u> class does this—a number of its methods are final. We must declare methods with final keyword for which we required to follow the same implementation throughout all the derived classes. The following fragment illustrates final keyword with a method:

Eg:1

```
class A
{
    final void m1()
    {
        System.out.println("This is a final method.");
    }
}

class B extends A
{
    void m1()
    {
        // COMPILE-ERROR! Can't override.
        System.out.println("Illegal!");
    }
}
```

Eg:2

Example of final method

```
class Bike{
  final void run(){System.out.println("running");}
}

class Honda extends Bike{
   void run(){System.out.println("running safely with 100kmph");}

   public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
   }
}
```

```
Output:Compile Time Error
```

Eg: 3

info - Notepad

File Edit Format View Help

```
class read                          // Base class
{
 final void display()
  {
   System.out.println("From read class");
  }
}

class info extends read                          // Derived Class
{
 void display()
   {
    System.out.println("From info class");
   }
 public static void main(String args[])
 {
   info o=new info();

   o.display();
 }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\final>javac info.java
info.java:11: error: display() in info cannot override display() in read
 void display()
      ^
  overridden method is final
1 error
```

Q) Is final method inherited?

- **Ans) Yes, final method is inherited but you cannot override it. For Example:**

```
class Bike{
  final void run(){System.out.println("running...");}
}
class Honda2 extends Bike{
   public static void main(String args[]){
    new Honda2().run();
   }
}
```

```
Output:running...
```

Q) What is blank or uninitialized final variable?

- A **final variable** that is **not initialized at the time of declaration is known as blank final variable**.
- If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful.
- For example **PAN CARD number** of an employee.
- It can be **initialized only in constructor**.

Example of blank final variable

```
class Student{
int id;
String name;
final String PAN_CARD_NUMBER;
...
}
```

Q) Can we initialize blank final variable?

- **Yes**, **but only in constructor**. For example:

```
class Bike10{
  final int speedlimit;//blank final variable

  Bike10(){
  speedlimit=70;
  System.out.println(speedlimit);
  }

  public static void main(String args[]){
    new Bike10();
  }
}
```

```
Output: 70
```

static blank final variable

- A **static final variable** that is **not initialized at the time of declaration** is known as **static blank final variable**.
- It can be **initialized only in static block**.

Example of static blank final variable

```
class A{
  static final int data;//static blank final variable
  static{ data=50;}
  public static void main(String args[]){
    System.out.println(A.data);
 }
}
```

Q) What is final parameter?

- If you **declare any parameter as final**, **<u>you cannot change the value of it</u>**.

```java
class Bike11{
 int cube(final int n){
  n=n+2;//can't be changed as n is final
  n*n*n;
 }
 public static void main(String args[]){
   Bike11 b=new Bike11();
   b.cube(5);
 }
}
```

```
Output: Compile Time Error
```

Q) Can we declare a constructor final?

- **No**, **because constructor** is **never inherited**.

Nested and Inner classes

Java Inner Classes

- **Java inner class** or **nested class is a class which is declared <u>inside the class.</u>**
- **We use inner classes to logically group classes in one place** so that it can be **more readable** and **maintainable**.
- Additionally, **it can access all the members of outer class <u>including</u> private data members** and **methods**.

eg:

Syntax of Inner class

```
class Java_Outer_class{
 //code
 class Java_Inner_class{
  //code
 }
}
```

```
class info
   {
        class cal
            {

            }
   }
```

Advantage of java inner classes

There are basically **three advantages** of **inner classes** in java. They are as follows:

1) Nested classes represent a special type of relationship that is **it can access all the members (data members and methods) of outer class** including private.

2) Nested classes **are used to develop more readable and maintainable code <u>because</u> <u>it logically group classes in one place only</u>**.

3) **Code Optimization**: **It requires less code to write**.

Difference between nested class and inner class in Java

- In Java, it is possible **to define a class within another class**, such classes are known as ***nested*** **classes**.
- **Inner class** is a part of **nested class**.
- **Non-static nested classes are known as inner classes**.

Types of Nested classes

- There are **two types of nested classes non-static** and **static nested classes.**
- The **non-static nested classes** are **also known as inner classes**.

1. Non-static nested class (inner class)

1) **Member inner class**
2) **Anonymous inner class**
3) **Local inner class**

2.Static nested class

Type	Description
Member Inner Class	A class created within class and outside method.
Anonymous Inner Class	A class created for implementing interface or extending class. Its name is decided by the java compiler.
Local Inner Class	A class created within method.
Static Nested Class	A static class created within class.
Nested Interface	An interface created within class or interface.

1.Java Member inner class

- A **non-static class that is created inside a class but outside a method is called member inner class**.

Syntax:

```
class Outer{
 //code
 class Inner{
  //code
 }
}
```

Java Member inner class example

- In this example, we are creating msg() method in member inner class that is accessing the private data member of outer class.

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void msg(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.msg();
 }
}
```

Output:

```
data is 30
```

Eg:2

info - Notepad

File Edit Format View Help

```
import java.io.*;
class info
{
 private int rno=101;
  class student
    {
       void display()
         {
          System.out.println("rno:"+rno);
         }
    }
 public static void main(String args[])
    {
     info o1=new info();
      info.student o2=o1.new student();
      o2.display();
    }
}
```

Output

```
rno:101
```

Eg: 3

sinfo - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class sinfo
{
 private int rno;
 private String name;
 private int total;
    sinfo(int r,String s,int t)
      {
       rno=r;
       name=s;
       total=t;
      }
  class student
    {
       void display()
         {
          System.out.println("Roll No:"+rno);
          System.out.println("Name:"+name);
          System.out.println("Total:"+total);
         }
    }
```

Ln 22, Col 6 | 100% | Windows (CRLF) | UTF-8

sinfo - Notepad

File Edit Format View Help

```
 public static void main(String args[])
    {
     Scanner io=new Scanner(System.in);
     int r,t;
     String n;
     System.out.println("Enter the roll no:");
     r=io.nextInt();
     System.out.println("Enter the name:");
     n=io.next();
     System.out.println("Enter the Total Marks:");
     t=io.nextInt();
     sinfo o1=new sinfo(r,n,t);
      sinfo.student o2=o1.new student();
      o2.display();
    }
}
```

Ln 22, Col 6 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\nested class>java sinfo
Enter the roll no:
101
Enter the name:
Anil
Enter the Total Marks:
356
Roll No:101
Name:Anil
Total:356
```

2. Java Anonymous inner class

- **A class that have no name is known as anonymous inner class** in java.
- It should be used if you have **to override method of class**.
- **Java Anonymous inner class** can be created by **two ways:**
    1. **Class (may be abstract or concrete).**
    2. **Interface**

Java anonymous inner class example using class

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

Output:

```
nice fruits
```

3. Java Local inner class

- **A class** i.e. **created inside a method is called local inner class** in java.
- **If you want to invoke the methods of local inner class**, you must **instantiate this class inside the method**.

Java local inner class example

```
public class localInner1{
 private int data=30;//instance variable
 void display(){
  class Local{
   void msg(){System.out.println(data);}
  }
  Local l=new Local();
  l.msg();
 }
 public static void main(String args[]){
  localInner1 obj=new localInner1();
  obj.display();
 }
}
```

Output:

```
30
```

Eg: 2

einfo - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class einfo
{
 private int eno;
 private String name;
 private int salary;
    einfo(int en,String n,int s)
      {
       eno=en;
       name=n;
       salary=s;
      }
 void show()
  {
    class employee
      {
         void display()
           {
            System.out.println("Employee No:"+eno);
            System.out.println("Name:"+name);
            System.out.println("Salary:"+salary);
           }
      }

       employee o=new employee();
       o.display();
   }
```

Ln 42, Col 6 | 100% | Windows (CRLF) | UTF-8

einfo - Notepad

File Edit Format View Help

```
   public static void main(String args[])
    {
     Scanner io=new Scanner(System.in);
     int en,s;
     String n;
     System.out.println("Enter the Employe no:");
     en=io.nextInt();
     System.out.println("Enter the name:");
     n=io.next();
     System.out.println("Enter the Salary:");
     s=io.nextInt();
     einfo o1=new einfo(en,n,s);
     o1.show();
    }
}
```

Ln 42, Col 6 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-2\nested class>java einfo
Enter the Employe no:
101
Enter the name:
Anil
Enter the Salary:
45000
Employee No:101
Name:Anil
Salary:45000
```

Java static nested class

- A **static class** i.e. **created inside a class** is called **static nested class** in java.
- **It cannot access non-static data members** and **methods**.
- **It can be accessed by outer class name**.
- **It can access static data members of outer class including private**.
- **Static nested class cannot access non-static (instance) data member** or **method**.

Java static nested class example with instance method

```
class TestOuter1{
  static int data=30;
  static class Inner{
   void msg(){System.out.println("data is "+data);}
  }
  public static void main(String args[]){
  TestOuter1.Inner obj=new TestOuter1.Inner();
  obj.msg();
  }
}
```

Output:

```
data is 30
```

- In this example, you need to **create the instance of static nested class** because it has instance method msg(). But you **don't need to create the object of Outer class <u>because nested class is static</u>** and static properties, **methods** or **classes can be accessed <u>without object</u>**.

Eg:2

stinfo - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class stinfo
{
 static int rno=101;


 static class student
    {
       void display()
         {
          System.out.println("Roll No:"+rno);

         }
    }
 public static void main(String args[])
    {
      stinfo.student o1=new stinfo.student();
      o1.display();
    }
}
```

Ln 5, Col 9 | 100% | Windows (CRLF) | UTF-8

```
Output

Roll No: 101
```

----------**End of Module-2**----------

Module-3

Inheritance

Syllabus

- **Derived Class and Base Class,**
- **Usage of super keyword,**
- **Creating a Multilevel Hierarchy,**
- **Method Overriding,**
- **Dynamic Method Dispatch,**
- **Using Abstract Classes,**
- **Using final with Inheritance,**
- **Definition and application of Packages and Interfaces**

Inheritance

- **Inheritance** is the method by which **objects of one class get the properties of objects of another class.**
- In object-oriented programming, **inheritance provides the thought of reusability** and **reduce its redundancy**.

Why use inheritance in java

1. For **Method Overriding** (so **runtime polymorphism** can be achieved).
2. For **Code Reusability**.

Terms used in Inheritance

- **Class:** A class is a **group of objects which have common properties**. It is a **template** or **blueprint from which objects are created**.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a **derived class**, **extended class**, or **child class**.
- **Super Class/Parent Class: Superclass** is the class **from where** a **subclass inherits the features**. **It is also called** a **base class** or a **parent class.**
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

The syntax of Java Inheritance

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

- The **extends keyword** indicates that you are making a new class that derives from an existing class. The meaning of "extends" is to increase the functionality.
- In the terminology of Java, **a class which is inherited is called** a **parent** or **superclass**, and **the new class is called child** or **subclass**.

Java Inheritance Example

- As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

```
class Employee{
 float salary=40000;
}
class Programmer extends Employee{
 int bonus=10000;
 public static void main(String args[]){
   Programmer p=new Programmer();
   System.out.println("Programmer salary is:"+p.salary);
   System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

```
Programmer salary is:40000.0
Bonus of programmer is:10000
```

- In the above example**, Programmer object can access the field of own class as well as of Employee class** i.e. **code reusability**.

Eg:2

programmer - Notepad

File Edit Format View Help

```
import java.io.*;

class employee
{

float salary=45000;

}

class programmer extends employee
{

int bonus=2000;
 public static void main(String args[])
   {
    programmer o=new programmer();
    System.out.println("Programmer Salary is:"+o.salary);
    System.out.println("Bonus of Programmer is:"+o.bonus);
   }
}
```

Ln 18, Col 47 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\single inheritance>java programmer
Programmer Salary is:45000.0
Bonus of Programmer is:2000
```

Eg:3

Hospital - Notepad

File Edit Format View Help

```
import java.io.*;
class Doctor
{
 void Doctor_details()
    {
     System.out.println(" Dr.Arun ");
     System.out.println(" ID: 1001 ");
     System.out.println(" MBBS");
    }
}
class Surgeon extends Doctor
  {
   void Surgeon_details()
    {
     System.out.println(" Dr.Vinu ");
     System.out.println(" ID: 2001 ");
     System.out.println(" MBBS");
    }
  }
class Hospital
{
public static void main(String args[])
  {
    Surgeon o=new Surgeon();
    o.Doctor_details();
    o.Surgeon_details();
  }
}
```

Ln 28, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\single inheritance>java Hospital
 Dr.Arun
 ID: 1001
 MBBS
 Dr.Vinu
 ID: 2001
 MBBS
```

Types of inheritance in java

- On the basis of class, there can be **three types of inheritance in java**: **single**, **multilevel** and **hierarchical.**
- Note: **Multiple inheritance** and **hybrid inheritance** **is not supported in Java through class**.
- In java programming, **multiple** and **hybrid inheritance is supported through interface only**. We will learn about interfaces later.

- When **one class inherits multiple classes**, **it is known as multiple inheritance**. For Example:

Single Inheritance Example

- When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.

Eg:1

inheritance1 - Notepad

File Edit Format View Help

```
import java.io.*;
class Animal
{
 void eat()
   {
     System.out.println("eating");
   }
}
class Dog extends Animal
{
 void bark()
   {
    System.out.println("barking");
    }
}
class inheritance1
   {
   public static void main(String args[])
    {
     Dog o=new Dog();
     o.eat();
     o.bark();
    }
   }
```

Ln 16, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\single inheritance>java inheritance1
eating
barking
```

Lab Cycle Programme

Q.)Develop a Java program to read and print students data using inheritance Class person: name, age, gender Class student inherits from person: mark1, mark2, mark3, total marks, grade

single - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class person
{
String name;
int age;
String gender;

 void readp()
   {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Name");
   name=io.next();
   System.out.println("Enter the Age");
   age=io.nextInt();
   System.out.println("Enter the Gender");
   gender=io.next();
   }
 }

class student extends person
{
int mark1;
int mark2;
int mark3;
int total;
float per;
char grade;
 void read()
   {
    readp();
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the mark1");
   mark1=io.nextInt();
   System.out.println("Enter the mark2");
   mark2=io.nextInt();
   System.out.println("Enter the mark3");
   mark3=io.nextInt();
    cal();
   }
```

Ln 29, Col 1 | 100% | Windows (CRLF) | UTF-8

*single - Notepad

File Edit Format View Help

```
void cal()
 {
  total=mark1+mark2+mark3;
  per=total/3;
   if(per>=90)
    {
     grade='A';
    }
    else if(per>=80)
    {
     grade='B';
    }
    else if(per>=70)
    {
     grade='C';
    }
    else if(per>=60)
    {
     grade='D';
    }
    else if(per>=50)
    {
     grade='E';
    }
    else
     {
     grade='F';
    }
  }
void display()
  {
  System.out.println("Name :"+name);
  System.out.println("Age :"+age);
  System.out.println("Gender :"+gender);
  System.out.println("Physics :"+mark1);
  System.out.println("Chemistry :"+mark2);
  System.out.println("Mathematics:"+mark3);
  System.out.println("Total:"+total);
  System.out.println("Per:"+per);
  System.out.println("Grade:"+grade);
  }
}
```

Ln 84, Col 1 | 100% | Windows (CRLF) | UTF-8

*single - Notepad

File Edit Format View Help

```
class single
{
 public static void main(String args[])
   {
     student o=new student();
     o.read();
     o.display();
   }
}
```

```
D:\java\programs\module-3\single inheritance>java single
Enter the Name
Anil
Enter the Age
22
Enter the Gender
Male
Enter the mark1
67
Enter the mark2
78
Enter the mark3
90
Name :Anil
Age :22
Gender :Male
Physics :67
Chemistry :78
Mathematics:90
Total:235
Per:78.0
Grade:C
```

Multilevel Inheritance Example

- **When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see in the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.**

Eg:1

Multilevel - Notepad

File Edit Format View Help

```
import java.io.*;

class Animal
{
 void eat()
   {
     System.out.println("eating");
   }
}

class Dog extends Animal
{
 void bark()
   {
    System.out.println("barking");
    }
}
class BabyDog extends Dog
{
  void weep()
    {
     System.out.println("weeping");
    }
}

class Multilevel
   {
   public static void main(String args[])
    {
     BabyDog o =new BabyDog();
      o.eat();
      o.bark();
      o.weep();
    }
   }
```

```
D:\java\programs\module-3\single inheritance>java Multilevel
eating
barking
weeping
```

Hierarchical Inheritance Example

- **When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.**

Eg:1

inheritance - Notepad

File Edit Format View Help

```
import java.io.*;
class Animal
{
 void eat()
   {
     System.out.println("eating");
   }
}
class Dog extends Animal
{
 void bark()
   {
    System.out.println("barking");
    }
}
class Cat extends Animal
{
  void meow()
    {
     System.out.println("meowing");
    }
}
class inheritance
   {
   public static void main(String args[])
    {
     Cat o=new Cat();
     o.eat();
     o.meow();
     Dog o1=new Dog();
     o1.bark();
     o1.eat();
    }
   }
```

Ln 23, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\single inheritance>java inheritance
eating
meowing
barking
eating
```

Q) Why multiple inheritance is not supported in java?

- **To reduce the complexity** and **simplify the language**, **multiple inheritance** **is not supported in java**.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the **same method** and you call it from child class object**, there will be ambiguity to call the method** of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
class A{
void msg(){System.out.println("Hello");}
}
class B{
void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were

 public static void main(String args[]){
   C obj=new C();
   obj.msg();//Now which msg() method would be invoked?
}
}
```

```
Compile Time Error
```

Difference Between Superclass and Subclass

What is Superclass?

- In Inheritance, **the existing class from which the new classes are derived is known as** the **Superclass**. It is also known as the **parent class** or **base class**.
- There are different inheritance types. There are illustrated using the following examples. Consider A B and C as classes.

- According to the above diagrams, **Superclasses varies from each inheritance type**.
- In single-level inheritance, A is the Superclass.
- In Multilevel inheritance, A is the Superclass for B and B is the Superclass for C.
- In Hierarchical Inheritance A is the Superclass for both B and C.
- In multiple inheritances both A and B are Superclasses for C.
- Hybrid inheritance is a combination of multi-level and multiple inheritances. In the left-hand side diagram, A is the Superclass for B, C and B, C are the Superclasses for D. In the right-hand side diagram, A is the Superclass for B and B, D are Superclasses for C.

What is Subclass?

- When implementing inheritance, **the class that inherits the properties** and **methods from the Superclass** is the **Subclass**.
- According to the above diagrams, **Subclasses varies from each inheritance type**.
- In Single Inheritance, B is the Subclass.
- In multi-level inheritance, B is the Subclass of A and C is the Subclass of B.
- In Hierarchical Inheritance B and C are Subclasses of A.
- In multiple inheritances, C is the Subclass for A and B.
- In Hybrid inheritance, the diagram in the left, B and C are Subclasses of A. D is the Subclass of B and C. In the diagram to the right, B is the Subclass for A. C is the Subclass of B and D.

What is the Similarity Between Superclass and Subclass?

- **Both are related to Inheritance.**

What is the Difference Between Superclass and Subclass?

Superclass vs Subclass	
When implementing inheritance, the existing class from which the new classes are derived is the Superclass.	When implementing inheritance, the class that inherits the properties and methods from the Superclass is the Subclass.
Synonyms	
Superclass is known as base class, parent class.	Subclass is known as derived class, child class.
Functionality	
A superclass cannot use the properties and methods of the Subclass.	A subclass can use the properties and methods of the Superclass.
Single-Level-Inheritance	
There is one Superclass.	There is one Subclass.
Hierarchical Inheritance	
There is one Superclass	There are many Subclasses.
Multiple Inheritance	
There are many Superclasses.	There is one Subclass.

Super Keyword in Java

- The **super** keyword in Java **is a reference variable which is used to refer immediate parent class object.**
- Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be **used to refer immediate parent class instance variable(data members).**
2. super can be **used to invoke immediate parent class method.**
3. super() can be **used to invoke immediate parent class constructor.**

1) super is used to refer immediate parent class instance variable.

- We can use **super keyword to access the data member** or **field of parent class**.
- **It is used if parent class** and **child class** **<u>have same fields (data members)</u>**.

TestSuper1 - Notepad

File Edit Format View Help

```
import java.io.*;
class Animal
{

String color="white";
}

class Dog extends Animal
{
String color="black";

  void printColor()
     {
      System.out.println("color:"+color);      // prints color of Dog class
      System.out.println("color:"+super.color);    // prints color of Animal class
     }
}

class TestSuper1
{
 public static void main(String args[])
   {
    Dog o=new Dog();
    o.printColor();
   }
}
```

Ln 26, Col 3 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\super>java TestSuper1
color:black
color:white
```

- In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

2) super can be used to invoke parent class method

- The **super keyword** can also be **used to invoke parent class method**.
- It should **be used** if **subclass contains <u>the same method</u> as parent class**.
- **<u>In other words</u>**, it is used if **<u>method is overridden</u>**.

TestSuper2 - Notepad

File Edit Format View Help

```
import java.io.*;
class Animal
{

 void eat()
  {
  System.out.println("eating");
  }

}
class Dog extends Animal
{

void eat()
  {
  System.out.println("eating bread");
  }

 void bark()
  {
  System.out.println("barking");
  }

  void work()
     {
      super.eat();
      bark();
     }
}

class TestSuper2
{
 public static void main(String args[])
   {
    Dog o=new Dog();
    o.work();
   }
}
```

Ln 38, Col 3 100%

```
D:\java\programs\module-3\super>java TestSuper2
eating
barking
```

- In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.
- To call the parent class method, we need to use super keyword.

3) super is used to invoke parent class constructor.

- The **super keyword** can also **be used to invoke** the **parent class constructor**. Let's see a simple example:

TestSuper3 - Notepad

File Edit Format View Help

```
import java.io.*;

class Animal
{
  Animal()
   {
   System.out.println("Animal is created");
  }
}

class Dog extends Animal
{
  Dog()
  {
    super();
    System.out.println("Dog is created");
  }
}

class TestSuper3
{
 public static void main(String args[])
   {
    Dog o=new Dog();
   }
}
```

```
D:\java\programs\module-3\super>java TestSuper3
Animal is created
Dog is created
```

- Note: **super() is added <u>in each class constructor automatically</u> by compiler if there is no super()** or **this().**

- As we know well that **default constructor is provided by compiler automatically if there is no constructor**. But, **it also adds super() as the first statement.**

- **Another example of super keyword where super() is provided by the compiler implicitly.**

TestSuper3 - Notepad

File Edit Format View Help

```
import java.io.*;

class Animal
{
  Animal()
   {
   System.out.println("Animal is created");
  }
}

class Dog extends Animal
{
  Dog()
  {

    System.out.println("Dog is created");
  }
}

class TestSuper3
{
 public static void main(String args[])
   {
    Dog o=new Dog();
   }
}
```

Ln 2, Col 1 | 100% | Windows (CRLF)

```
D:\java\programs\module-3\super>java TestSuper3
Animal is created
Dog is created
```

super example: real use

- Let's see the real use of super keyword. Here, Emp class inherits Person class so all the properties of Person will be inherited to Emp by default. To initialize all the property, we are using parent class constructor from child class. In such way, we are reusing the parent class constructor.

TestSuper4 - Notepad

File Edit Format View Help

```
import java.io.*;
class Person
{
int id;
String name;
 Person(int id,String name)
  {
  this.id=id;
  this.name=name;
  }
}

class Emp extends Person
{
float salary;

  Emp(int id,String name,float salary)
  {
  super(id,name);
  this.salary=salary;
  }

 void display()
  {
   System.out.println("Name:"+name);
   System.out.println("Id:"+id);
   System.out.println("Salary:"+salary);
  }
}

class TestSuper4
{
 public static void main(String args[])
   {
    Emp o=new Emp(101,"Anil",45000);
    o.display();
   }
}
```

Ln 4, Col 37 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\super>java TestSuper4
Name:Anil
Id:101
Salary:45000.0
```

Creating a Multilevel Hierarchy

Multilevel Inheritance

Eg:1 A program that demonstrates a multilevel inheritance hierarchy in Java is given as follows:

multi1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class A
{
  void funA()
     {
      System.out.println("This is class A");
     }
}
class B extends A
{
  void funB()
   {
   System.out.println("This is class B");
   }
}
class C extends B
{
  void funC()
     {
     System.out.println("This is class C");
     }
}
class multi1
{
public static void main(String args[])
   {
     C o=new C();
     o.funA();
     o.funB();
     o.funC();
   }
}
```

Ln 28, Col 5 100%

```
D:\java\programs\module-3\multi>java multi1
This is class A
This is class B
This is class C
```

Eg 2:

multi - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Employee
{
protected String name;
protected float experience;
protected double salary;
}
class Department extends Employee
{
protected String dept;
public void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Employee Name:");
   name=io.next();
   System.out.println("Enter the experience:");
   experience=io.nextFloat();
   System.out.println("Enter the Salary:");
   salary=io.nextDouble();
   System.out.println("Enter the Department Name:");
   dept=io.next();
  }
}
class Senior extends Department
{
public void display()
   {
   System.out.print(name+"\t");
   System.out.print(experience+"\t\t");
   System.out.print(salary+"\t");
   System.out.print(dept+"\t");
   System.out.println(" ");
   }
}
class multi
{
public static void main(String args[])
   {
     Senior o=new Senior();
     o.read();
     System.out.println("Name\tExperience\tsalary\tDepartment");
     o.display();
   }
}
```

Ln 9, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\multi>java multi
Enter the Employee Name:
Anil
Enter the experience:
10
Enter the Salary:
45000
Enter the Department Name:
IT
Name    Experience      salary  Department
Anil    10.0            45000.0 IT
```

Lab Cycle Programme

Write a program to insert the details of 5 employees. Using array of objects, prepare the employee list and print the details of the employee with 10 plus experience. Use **multilevel inheritance** features.

Employee : Make a class with Employee with name, experience and salary as Protected data members

↓

Department : Make a class Department inherit from employee. Add dept_name as Data member and read values into the data members.

↓

Senior : Make a class Senior inherits from Department. Display the details of Employees with 10 plus experience.

*multilevel - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Employee
{
protected String name;
protected float experience;
protected double salary;
}

class Department extends Employee
{
protected String dept;
public void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Employee Name:");
   name=io.next();
   System.out.println("Enter the experience:");
   experience=io.nextFloat();
   System.out.println("Enter the Salary:");
   salary=io.nextDouble();
   System.out.println("Enter the Department Name:");
   dept=io.next();
  }

}
class Senior extends Department
{
public void display()
   {
   System.out.print(name+"\t");
   System.out.print(experience+"\t\t");
   System.out.print(salary+"\t");
   System.out.print(dept+"\t");
   System.out.println(" ");

   }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\multi>java multilevel
Enter the value of n:
3
Enter the 3Employees details:
Enter the Employee Name:
Anil
Enter the experience:
12
Enter the Salary:
45000
Enter the Department Name:
IT
Enter the Employee Name:
Binu
Enter the experience:
7
Enter the Salary:
40000
Enter the Department Name:
CS
Enter the Employee Name:
Vijay
Enter the experience:
14
Enter the Salary:
50000
Enter the Department Name:
CS
Name    Experience      salary  Department
Anil    12.0            45000.0 IT
Vijay   14.0            50000.0 CS
```

Hierarchical Inheritance

hierarchical - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class A
{
 void funA()
   {
   System.out.println("This is class A");
   }
}
class B extends A
{
  void funB()
   {
   System.out.println("This is class B");
   }
}
class C extends A
{
  void funC()
     {
     System.out.println("This is class C");
     }
}
class hierarchical
{
public static void main(String args[])
   {
     B o1=new B();
     o1.funA();
     o1.funB();

    C o2=new C();
     o2.funA();
     o2.funC();

   }
}
```

Ln 1, Col 1

```
D:\java\programs\module-3\multi>java hierarchical
This is class A
This is class B
This is class A
This is class C
```

Eg:2

hierarchical1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class Person
{
protected String name;
protected float experience;
protected double salary;
}

class Assistant_Professor extends Person
{
protected String dept;
protected String des;
public void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Employee Name:");
   name=io.next();
   System.out.println("Enter the experience:");
   experience=io.nextFloat();
   System.out.println("Enter the Salary:");
   salary=io.nextDouble();
   System.out.println("Enter the Department Name:");
   dept=io.next();
   System.out.println("Enter the Designation:");
   des=io.next();
  }
public void display()
   {
   System.out.print(name+"\t");
   System.out.print(experience+"\t\t");
   System.out.print(salary+"\t");
   System.out.print(dept+"\t\t");
   System.out.print(des+"\t");
   System.out.println(" ");

   }

}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

*hierarchical1 - Notepad

File Edit Format View Help

```
class Associate_Professor extends Person
{
protected String dept;
protected String des;
public void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Employee Name:");
   name=io.next();
   System.out.println("Enter the experience:");
   experience=io.nextFloat();
   System.out.println("Enter the Salary:");
   salary=io.nextDouble();
   System.out.println("Enter the Department Name:");
   dept=io.next();
   System.out.println("Enter the Designation:");
   des=io.next();
  }

public void display()
   {
   System.out.print(name+"\t");
   System.out.print(experience+"\t\t");
   System.out.print(salary+"\t");
   System.out.print(dept+"\t\t");
   System.out.print(des+"\t");
   System.out.println(" ");

   }
}
```

Ln 41, Col 1 | 100% | Windows (CRLF) | UTF-8

*hierarchical1 - Notepad

File Edit Format View Help

```
class Professor extends Person
{
protected String dept;
protected String des;
public void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Employee Name:");
   name=io.next();
   System.out.println("Enter the experience:");
   experience=io.nextFloat();
   System.out.println("Enter the Salary:");
   salary=io.nextDouble();
   System.out.println("Enter the Department Name:");
   dept=io.next();
   System.out.println("Enter the Designation:");
   des=io.next();
  }

public void display()
   {
   System.out.print(name+"\t");
   System.out.print(experience+"\t\t");
   System.out.print(salary+"\t");
   System.out.print(dept+"\t\t");
   System.out.print(des+"\t");
   System.out.println(" ");

   }
}
```

Ln 41, Col 1 | 100% | Windows (CRLF) | UTF-8

*hierarchical1 - Notepad

File Edit Format View Help

```
class hierarchical1
{
public static void main(String args[])
   {

     Assistant_Professor o1=new Assistant_Professor();

     o1.read();

     System.out.println("Name\tExperience\tsalary\tDepartment\tDesignation");

     o1.display();

    Associate_Professor o2=new Associate_Professor();

     o2.read();

     System.out.println("Name\tExperience\tsalary\tDepartment\tDesignation");

     o2.display();

     Professor o3=new Professor();

     o3.read();

     System.out.println("Name\tExperience\tsalary\tDepartment\tDesignation");

     o3.display();

   }
}
```

Ln 103, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\multi>java hierarchical1
Enter the Employee Name:
Anil
Enter the experience:
12
Enter the Salary:
45000
Enter the Department Name:
IT
Enter the Designation:
AssistProfessor
Name    Experience      salary  Department      Designation
Anil    12.0            45000.0 IT              AssistProfessor
Enter the Employee Name:
Binu
Enter the experience:
7
Enter the Salary:
50000
Enter the Department Name:
CS
Enter the Designation:
AssociateProfessor
Name    Experience      salary  Department      Designation
Binu    7.0             50000.0 CS              AssociateProfessor
Enter the Employee Name:
Vijay
Enter the experience:
8
Enter the Salary:
70000
Enter the Department Name:
CS
Enter the Designation:
Professor
Name    Experience      salary  Department      Designation
Vijay   8.0             70000.0 CS              Professor
```

Lab Cycle Programme

Q.9) Suppose your institution wishes to maintain a database of its employees. The database is divided into a number of classes whose (Assistant Professor, Associate Professor, Professor etc). Construct a Super Class Person to store personal information. Derive all the above classes from the Class. Develop a Java program to specify all the classes and define functions to create the database and retrieve individual information as and when required. Support at least 5 employees on each category.

hierarchical2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class Person
{
protected String name;
protected float experience;
protected double salary;
}

class Assistant_Professor extends Person
{
protected String dept;
protected String des;
public void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Employee Name:");
   name=io.next();
   System.out.println("Enter the experience:");
   experience=io.nextFloat();
   System.out.println("Enter the Salary:");
   salary=io.nextDouble();
   System.out.println("Enter the Department Name:");
   dept=io.next();
   System.out.println("Enter the Designation:");
   des=io.next();
  }
public void display()
   {
   System.out.print(name+"\t");
   System.out.print(experience+"\t\t");
   System.out.print(salary+"\t");
   System.out.print(dept+"\t\t");
   System.out.print(des+"\t");
   System.out.println(" ");

   }

}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

*hierarchical2 - Notepad

File Edit Format View Help

```
class Associate_Professor extends Person
{
protected String dept;
protected String des;
public void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Employee Name:");
   name=io.next();
   System.out.println("Enter the experience:");
   experience=io.nextFloat();
   System.out.println("Enter the Salary:");
   salary=io.nextDouble();
   System.out.println("Enter the Department Name:");
   dept=io.next();
   System.out.println("Enter the Designation:");
   des=io.next();
  }

public void display()
   {
   System.out.print(name+"\t");
   System.out.print(experience+"\t\t");
   System.out.print(salary+"\t");
   System.out.print(dept+"\t\t");
   System.out.print(des+"\t");
   System.out.println(" ");

   }
}
```

Ln 42, Col 1 | 100% | Windows (CRLF) | UTF-8

*hierarchical2 - Notepad

File Edit Format View Help

```
class Professor extends Person
{
protected String dept;
protected String des;
public void read()
  {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the Employee Name:");
   name=io.next();
   System.out.println("Enter the experience:");
   experience=io.nextFloat();
   System.out.println("Enter the Salary:");
   salary=io.nextDouble();
   System.out.println("Enter the Department Name:");
   dept=io.next();
   System.out.println("Enter the Designation:");
   des=io.next();
  }

public void display()
   {
   System.out.print(name+"\t");
   System.out.print(experience+"\t\t");
   System.out.print(salary+"\t");
   System.out.print(dept+"\t\t");
   System.out.print(des+"\t");
   System.out.println(" ");

   }
}
```

Ln 42, Col 1 | 100% | Windows (CRLF) | UTF-8

*hierarchical2 - Notepad

File Edit Format View Help

```
class hierarchical2
{
public static void main(String args[])
   {
     Scanner io=new Scanner(System.in);
     System.out.println("Enter the value of n:");
     int n=io.nextInt();
      Assistant_Professor o1[]=new Assistant_Professor[n];
     int i;
     for(i=0;i<n;i++)
       {
       o1[i]=new Assistant_Professor();
       }
      System.out.println("Enter the "+n+ " Assistant_Professor details:");
       for(i=0;i<n;i++)
       {
       o1[i].read();
       }
    Associate_Professor o2[]=new Associate_Professor[n];
   for(i=0;i<n;i++)
       {
       o2[i]=new Associate_Professor();
       }
      System.out.println("Enter the "+n+ " Associate_Professor details:");
       for(i=0;i<n;i++)
       {
       o2[i].read();
       }
   Professor o3[]=new Professor[n];
   for(i=0;i<n;i++)
       {
       o3[i]=new Professor();
       }
      System.out.println("Enter the "+n+ " Professor details:");
       for(i=0;i<n;i++)
       {
       o3[i].read();
       }
```

Ln 104, Col 1 | 100% | Windows (CRLF) | UTF-8

*hierarchical2 - Notepad

File Edit Format View Help

```
     System.out.println(n + "Assistant_Professor details:");
     System.out.println("Name\tExperience\tsalary\tDepartment\tDesignation");
      for(i=0;i<n;i++)
       {
        o1[i].display();
       }
     System.out.println(n + "Associate_Professor details:");
     System.out.println("Name\tExperience\tsalary\tDepartment\tDesignation");
     for(i=0;i<n;i++)
       {
       o2[i].display();
       }
     System.out.println(n + "Professor details:");
     System.out.println("Name\tExperience\tsalary\tDepartment\tDesignation");
    for(i=0;i<n;i++)
       {
        o3[i].display();
       }
    }
}
```

```
D:\java\programs\module-3\multi>java hierarchical2
Enter the value of n:
1
Enter the 1 Assistant_Professor details:
Enter the Employee Name:
Anil
Enter the experience:
10
Enter the Salary:
45000
Enter the Department Name:
IT
Enter the Designation:
AssistantProfessor
Enter the 1 Associate_Professor details:
Enter the Employee Name:
Binu
Enter the experience:
7
Enter the Salary:
50000
Enter the Department Name:
CS
Enter the Designation:
AssociateProfessor
```

```
Enter the 1 Professor details:
Enter the Employee Name:
Vijay
Enter the experience:
14
Enter the Salary:
80000
Enter the Department Name:
EC
Enter the Designation:
Professor
1Assistant_Professor details:
Name    Experience      salary  Department      Designation
Anil    10.0            45000.0 IT              AssistantProfessor
1Associate_Professor details:
Name    Experience      salary  Department      Designation
Binu    7.0             50000.0 CS              AssociateProfessor
1Professor details:
Name    Experience      salary  Department      Designation
Vijay   14.0            80000.0 EC              Professor
```

Method Overriding in Java

- Polymorphism **allows** the **same function to act differently in different classes**.

2. Runtime polymorphism:

- It is also known as **Dynamic Method Dispatch**. It is a process in which a function call to the overridden method is resolved at Runtime. This type of polymorphism is achieved by Method Overriding.
- **Method overriding,** on the other hand, occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

What is the difference between Overloading and Overriding?

- **Overloading** is about **same function have different signatures**. **Overriding** is about **same function**, **same signature** but **different classes connected through inheritance**.
- **Overloading** is an **example of compiler time polymorphism** and **overriding** is an example of **run time polymorphism**.

Method Overriding in Java

- If **subclass** (child class) has the **same method** as declared in the **parent class**, it is known as **method overriding in Java**.
- In other words, If a **subclass provides the specific implementation of the method that has been declared by one of its parent class**, it is known as **method overriding**.

Usage of Java Method Overriding

- Method overriding is **used to provide the specific implementation of a method which is already provided by its superclass**.
- Method overriding is **used for runtime polymorphism**

Rules for Java Method Overriding

1. **The method must have the same name as in the parent class**
2. **The method must have the same parameter as in the parent class**.
3. There **must be an IS-A relationship** (**inheritance**).

Understanding the problem without method overriding

- Let's understand the problem that we may face in the program if we don't use method overriding.

wover - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class A
{
  void funA()
     {
      System.out.println("This is from class A");
     }
}
class B extends A
{
  void funB()
   {
   System.out.println("This is from class B");
   }
}
class wover
{
public static void main(String args[])
   {
     B o=new B();
     o.funA();
     o.funB();
   }
}
```

```
D:\java\programs\module-3\overriding>javac wover.java

D:\java\programs\module-3\overriding>java wover
This is from class A
This is from class B
```

over - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class A
{
  void funA()
     {
      System.out.println("This is from class A");
     }
}
class B extends A
{
  void funA()
   {
   System.out.println("This is from class B");
   }
}
class over
{
public static void main(String args[])
   {
     B o=new B();
     o.funA();
   }
}
```

Ln 23, Col 4 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\overriding>javac over.java

D:\java\programs\module-3\overriding>java over
This is from class B
```

Eg:2

wover1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Vehicle
{
  void Run()
     {
      System.out.println("Vehicle is running");
     }
}
class Bike extends Vehicle
{
  void show()
   {
   System.out.println("This is from class Bike");
   }
}
class wover1
{
public static void main(String args[])
   {
     Bike o=new Bike();
     o.Run();
     o.show();
   }
}
```

```
D:\java\programs\module-3\overriding>javac wover1.java

D:\java\programs\module-3\overriding>java wover1
Vehicle is running
This is from class Bike
```

- **Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.**

Example of method overriding

- In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding.

over1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Vehicle
{
  void Run()
     {
      System.out.println("Vehicle is running");
     }
}
class Bike extends Vehicle
{
  void Run()
   {
   System.out.println("Bike is running safely");
   }
}
class over1
{
public static void main(String args[])
   {
     Bike o=new Bike();
     o.Run();
   }
}
```

```
D:\java\programs\module-3\overriding>javac over1.java

D:\java\programs\module-3\overriding>java over1
Bike is running safely
```

A real example of Java Method Overriding

- Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.

```java
//Java Program to demonstrate the real scenario of Java Method Overriding
//where three classes are overriding the method of a parent class.
//Creating a parent class.
class Bank{
int getRateOfInterest(){return 0;}
}
//Creating child classes.
class SBI extends Bank{
int getRateOfInterest(){return 8;}
}

class ICICI extends Bank{
int getRateOfInterest(){return 7;}
}
class AXIS extends Bank{
int getRateOfInterest(){return 9;}
}
//Test class to create objects and call the methods
class Test2{
public static void main(String args[]){
SBI s=new SBI();
ICICI i=new ICICI();
AXIS a=new AXIS();
System.out.println("SBI Rate of Interest: "+s.getRateOfInterest());
System.out.println("ICICI Rate of Interest: "+i.getRateOfInterest());
System.out.println("AXIS Rate of Interest: "+a.getRateOfInterest());
}
}
```

```
Output:
SBI Rate of Interest: 8
ICICI Rate of Interest: 7
AXIS Rate of Interest: 9
```

Can we override static method?

- No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Can we override java main method?

- No, because the main is a static method

Difference between method Overloading and Method Overriding in java

- There are many differences between method overloading and method overriding in java. A list of differences between method overloading and method overriding are given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

- **Java Method Overloading example**

```
class OverloadingExample{
static int add(int a,int b){return a+b;}
static int add(int a,int b,int c){return a+b+c;}
}
```

- **Java Method Overriding example**

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void eat(){System.out.println("eating bread...");}
}
```

Dynamic Method Dispatch or Runtime Polymorphism in Java

- **In Java polymorphism is mainly divided into two types:**
    1. **Compile time Polymorphism**
    2. **Runtime Polymorphism**

Runtime polymorphism:

- It is also known as **Dynamic Method Dispatch**.
- **It is a process in which a function call to the overridden method is resolved at Runtime**. This type of polymorphism is achieved by Method Overriding.

Dynamic Method Dispatch or Runtime Polymorphism in Java

- **Method overriding** is one of the ways in which Java supports **Runtime Polymorphism.**
- **Dynamic method dispatch is the mechanism** by which a call **to an overridden method is resolved at run time**, **rather than compile time**.
- When an **overridden method is called through a superclass reference**, **Java determines which** version(superclass/subclasses) of that **method is to be executed based upon the type of the object being referred to at the time the call occurs**. Thus, **this determination is made at run time.**
- **At run-time**, it depends on the **type of the object being referred** to (not the type of the reference variable) **that determines which version of an overridden method will be executed**
- **A superclass reference variable can refer to a subclass object**. This is also known as **upcasting**.
- Java uses this fact to resolve calls to overridden methods **at run time.**

- Therefore**, if a superclass contains a method that is overridden by a subclass**, **then** when **different types of objects are referred to through a superclass reference variable, <u>different versions of the method are executed</u>**.
- Here is an example that illustrates **dynamic method dispatch:**

Eg:1

dynamic1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class A
{
  void m1()
     {
      System.out.println("This is from class A");
     }
}
class B extends A
{
  void m1()
   {
   System.out.println("This is from class B");
   }
}
class C extends A
{
  void m1()
   {
   System.out.println("This is from class C");
   }
}
class dynamic1
{
public static void main(String args[])
   {
     A o1=new A();
     B o2=new B();
     C o3=new C();
     A ref;
     ref=o1;
     ref.m1();
     ref=o2;
     ref.m1();
     ref=o3;
     ref.m1();
   }
}
```

Ln 17, Col 18 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\dynamic>java dynamic1
This is from class A
This is from class B
This is from class C
```

Explanation :

- The above program creates one superclass called A and it's two subclasses B and C.

These subclasses overrides m1( ) method.

1. Inside the main() method in Dispatch class, initially objects of type A, B, and C are declared.

```
A a = new A(); // object of type A
B b = new B(); // object of type B
C c = new C(); // object of type C
```

```
ref = a; // r refers to an A object
ref.m1(); // calling A's version of m1()
```

```
ref = b; // now r refers to a B object
ref.m1(); // calling B's version of m1()
```

```
ref = c; // now r refers to a C object
ref.m1(); // calling C's version of m1()
```

Advantages of Dynamic Method Dispatch

1. **Dynamic method dispatch allow Java to support <u>overriding of methods</u>** which is central for **run-time polymorphism**.
2. **It allows a class to specify methods that will be common to all of its derivatives**, while allowing subclasses to define the specific implementation of some or all of those methods.
3. **It also allow subclasses to add its specific methods ,**subclasses to define the specific implementation of some.

Abstract class in Java

- **Abstraction is a process of hiding the implementation details and showing only functionality to the user.**
- **Another way, it shows only essential things to the user and hides the internal details.**
- **for example, sending SMS where you type the text and send the message. <u>You don't know</u> the internal processing about the message delivery.**
- **Abstraction lets you focus on what the <u>object</u> does instead of how it does it.**

Ways to achieve Abstraction

- **There are two ways to achieve abstraction in java**
    1. **Abstract class (0 to 100%)**
    2. **Interface (100%)**

1. Abstract class in Java

- **A class which is declared as abstract** is known as an **abstract class**.
- **It can have abstract** and **non-abstract methods**. It needs to be extended and its method implemented. It **cannot be instantiated**.

Points to Remember

1. An **abstract class** must be declared with an **abstract keyword**.
2. It can have **abstract** and **non-abstract methods.**
3. It **cannot** be **instantiated**.

4. It can have **constructors** and **static methods** also.
5. It can have **final methods** which will force the subclass **not to change the body of the method.**

Example of abstract class

```
abstract class A{}
```

Abstract Method in Java

- A **method which is declared as abstract** and **does not have implementation** is known as an **abstract method**.
- **Example of abstract method**

```
abstract void printStatus();//no method body and abstract
```

Example of Abstract class that has an abstract method

abst - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
abstract class Bike
{
  abstract void run();

}
class Honda extends Bike
{
  void run()
   {
   System.out.println("running safely");
   }
}

class abst
{
public static void main(String args[])
   {

   Bike o=new Honda();
   o.run();

   }
}
```

Ln 9, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\abstract>java abst
running safely
```

Understanding the real scenario of Abstract class

- In this example, **Shape** is the **abstract class**, and **its implementation is provided by the Rectangle** and **Circle classes**.
- Mostly, **we don't know about the implementation class** (which is hidden to the end user), and an object of the implementation class is provided by the **factory method**.
- A **factory method** is a method that returns the instance of the class. We will learn about the factory method later.
- In this example, if you create the instance of Rectangle class, draw() method of Rectangle class will be invoked.

abstraction - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
abstract class Shape
{
  abstract void draw();

}
class Rectangle extends Shape
{
  void draw()
   {
   System.out.println("drawing rectangle");
   }
}

class Circle extends Shape
{
  void draw()
   {
   System.out.println("drawing circle");
   }
}

class abstraction
{
public static void main(String args[])
   {

   Shape o1=new Rectangle();
   o1.draw();

   o1=new Circle();
   o1.draw();
   }
}
```

Ln 35, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\abstract>java abstraction
drawing rectangle
drawing circle
```

Another example of Abstract class in java

abstraction1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
abstract class Bank
{
  abstract int getRateOfInterest();

}
class SBI extends Bank
{
  int getRateOfInterest()
   {
   return(7);
   }
}

class PNB extends Bank
{
  int getRateOfInterest()
   {
   return(8);
   }
}

class abstraction1
{
public static void main(String args[])
   {

   Bank o1=new SBI();
   System.out.println("SBI Rate of Interest is:"+o1.getRateOfInterest()+"%");

   o1=new PNB();
   System.out.println("PNB Rate of Interest is:"+o1.getRateOfInterest()+"%");
   }
}
```

```
D:\java\programs\module-3\abstract>java abstraction1
SBI Rate of Interest is:7%
PNB Rate of Interest is:8%
```

Abstract class having constructor, data member and methods

- An **abstract class** can have a **data member**, **abstract method**, **method body (non-abstract method)**, **constructor**, and even **main() method**.

> Rule: If there is an abstract method in a class, that class must be abstract.

abstraction2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
abstract class Bike
{
  Bike()
    {
     System.out.println("Bike is created");
    }
  abstract void run();
  void changeGear()
    {
     System.out.println("gear changed");
    }

}
class Honda extends Bike
{
  void run()
   {
   System.out.println("running safely");
   }
}

class abstraction2
{
public static void main(String args[])
   {

   Bike o=new Honda();
   o.run();
   o.changeGear();
   }
}
```

Ln 31, Col 19 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\abstract>java abstraction2
Bike is created
running safely
gear changed
```

```
class Bike12{
abstract void run();
}
```

```
compile time error
```

Rule: If you are extending an abstract class that has an abstract method, you must either provide the implementation of the method or make this class abstract.

Another real scenario of abstract class

- The abstract class can also be used to provide some implementation of the interface. In such case, the end user may not be forced to override all the methods of the interface.

```
interface A{
void a();
void b();
void c();
void d();
}

abstract class B implements A{
public void c(){System.out.println("I am c");}
}

class M extends B{
public void a(){System.out.println("I am a");}
public void b(){System.out.println("I am b");}
public void d(){System.out.println("I am d");}
}

class Test5{
public static void main(String args[]){
A a=new M();
a.a();
a.b();
a.c();
a.d();
```

```
Output:I am a
       I am b
       I am c
       I am d
```

Lab cycle Program

- **Develop a java program to create an abstract class named Shape that contains three integers and an empty method named printArea(). Provide three classes named Rectangle, Triangle and Circle such that each one of the classes extends the class Shape. Each one of the classes contain only the method printArea( ) that prints the area of the given shape.**

abstraction3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
abstract class Shape
{
  abstract double printArea();
  double r,l,b,h;
}
class Rectangle extends Shape
{
  double printArea()
   {
   Scanner io=new Scanner(System.in);
    System.out.println("Enter the value of l and b:");
   l=io.nextDouble();
   b=io.nextDouble();
   double res;
   res=l*b;
   return res;
   }
}
class Triangle extends Shape
{
  double printArea()
   {
   Scanner io=new Scanner(System.in);
   System.out.println("Enter the value of b and h:");
   b=io.nextDouble();
   h=io.nextDouble();
   double res;
   res=0.5*b*h;
   return res;
  }
}
```

Ln 21, Col 1 | 100% | Windows (CRLF) | UTF-8

abstraction3 - Notepad

File Edit Format View Help

```
class Circle extends Shape
{
  double printArea()
   {
   Scanner io=new Scanner(System.in);
    System.out.println("Enter the value of r:");
   r=io.nextDouble();
   double res;
   res=3.14*r*r;
   return res;
   }
}


class abstraction3
{
public static void main(String args[])
   {

   Shape o1=new Rectangle();
   System.out.println("Area of Rectangle is="+o1.printArea());

   o1=new Triangle();
   System.out.println("Area of Triangle is="+o1.printArea());

   o1=new Circle();
   System.out.println("Area of Circle is="+o1.printArea());
   }
}
```

```
D:\java\programs\module-3\abstract>java abstraction3
Enter the value of l and b:
4
5
Area of Rectangle is=20.0
Enter the value of b and h:
4
5
Area of Triangle is=10.0
Enter the value of r:
2
Area of Circle is=12.56
```

Using final with Inheritance in Java

final keyword

- **final** is a keyword in java **used for restricting some functionalities**.
- The **final keyword** is final that is **we cannot change**.
- We can use **final keywords** for **variables**, **methods**, and **class**.
- If we use the **final keyword for the inheritance** that is **if we declare any method with the final keyword** in the **base class so the implementation of the final method will be the same as in derived class.**
- **We can declare** the **final method** in **any subclass** for which we want that **if any other class extends this subclass**.
- The **two uses of final** apply to inheritance. Both are examined here.

    1. **Using final to Prevent Inheritance**
    2. **Using final to Prevent Overriding**

1. Using final to Prevent Inheritance

- Sometimes you will want **to prevent a class from being inherited**. To do this, **precede the class declaration with final**.
- When a class is declared as final then it cannot be subclassed i.e. no any other class can extend it.
- This is particularly useful, for example, when creating an immutable class like the predefined String class. The following fragment illustrates **final** keyword with a class:

over - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
final class A
{
  void funA()
     {
      System.out.println("This is from class A");
     }
}
class B extends A
{
  void funA()
   {
   System.out.println("This is from class B");
   }
}
class over
{
public static void main(String args[])
   {
     B o=new B();
     o.funA();
   }
}
```

Ln 1, Col 18 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\final>javac over.java
over.java:10: error: cannot inherit from final A
class B extends A
                ^
1 error
```

- As the comments imply, **it is illegal** for **B** to inherit **A since A** is declared as **final**.

Java final class

- **If you make any class as final, <u>you cannot extend it</u>.**

Example of final class : eg:2

abst - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
final class Bike
{


}
class Honda extends Bike
{
  void run()
   {
   System.out.println("running safely");
   }
}

class abst
{
public static void main(String args[])
   {

   Honda o=new Honda();
   o.run();

   }
}
```

```
D:\java\programs\module-3\final>javac abst.java
abst.java:8: error: cannot inherit from final Bike
class Honda extends Bike
                    ^
1 error
```

Note :

- **Declaring a class as final implicitly declares all of its methods as final**, too.
- **It is illegal to declare a class as both abstract** and **final** since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

2. Using final to Prevent Overriding

- While method **overriding** is **one of Java's most powerful features**, there will be times when you will **<u>want to prevent it from occurring</u>**.
- When a **method is declared as final <u>then it cannot be</u> overridden by subclasses.**
- The Object class does this—a number of its methods are final. The following fragment illustrates **final** keyword with a method:

over1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class A
{
  final void funA()
     {
      System.out.println("This is from class A");
     }
}
class B extends A
{
  void funA()
   {
   System.out.println("This is from class B");
   }
}
class over1
{
public static void main(String args[])
   {
     B o=new B();
     o.funA();
   }
}
```

Ln 5, Col 9 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\final>javac over1.java
over1.java:12: error: funA() in B cannot override funA() in A
  void funA()
       ^
  overridden method is final
1 error
```

- Because **funA( )** is declared as **final**, **it cannot be overridden** in **B**.
- **If you attempt to do** so, a **compile-time error will result**.

Java final method

- If you **make any method as final**, **you cannot override it**.

Eg:2

abst1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
 class Bike
{
  final void run()
   {
   System.out.println("running");
   }

}
class Honda extends Bike
{
  void run()
   {
   System.out.println("running safely");
   }
}

class abst1
{
public static void main(String args[])
   {

   Honda o=new Honda();
   o.run();

   }
}
```

Ln 7, Col 31 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\final>javac abst1.java
abst1.java:13: error: run() in Honda cannot override run() in Bike
  void run()
       ^
  overridden method is final
1 error
```

Is final method inherited?

- Ans) Yes, **final method is inherited <u>but you cannot override it</u>**. For Example:

abst2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
 class Bike
{
  final void run()
   {
   System.out.println("running");
   }

}
class Honda extends Bike
{
  void show()
   {
   System.out.println("running safely");
   }
}

class abst2
{
public static void main(String args[])
   {

   Honda o=new Honda();
   o.run();

   }
}
```

Ln 13,

```
D:\java\programs\module-3\final>java abst2
running
```

Definition and application of Packages and Interfaces

Abstraction in Java

- **Abstraction** is a **process of hiding the implementation details** and **showing only functionality to the user**.

Ways to achieve Abstraction

- There are **two ways to achieve abstraction** in java

1. **Abstract class** (0 to 100%)
2. **Interface** (100%)

2. Interface in Java

- An **interface in Java** is a **blueprint of a class**. **It has static constants** and **abstract methods**.
- The **interface in Java is *a mechanism to achieve abstraction***. **There can be only abstract methods** in the **Java interface, not method body**.
- **It is used to achieve abstraction** and **multiple inheritance** in Java.
- In other words, you can say that **interfaces can have abstract methods** and **variables**. **It cannot have** a **method body**.
- **Java Interface** also **represents the IS-A relationship**.
- **It cannot be instantiated** just like the abstract class.
- Since Java 8, we can have **default and static methods** in an interface.
- Since Java 9, we can have **private methods** in an interface.

Why **use Java interface**?

- There are mainly **three reasons to use interface**. They are given below.

1. **It is used to achieve abstraction**.
2. **By interface**, **we can support** the functionality of **multiple inheritance.**
3. **It can be used to achieve loose coupling (the classes are independent of each other)**.

How to declare an interface?

- An **interface is declared by using** the **interface keyword**.
- **It provides total abstraction**; means **all the methods in an interface are declared <u>with the empty body</u>**, and **all the fields are public**, **static** and **final by default**.
- **A class that implements an interface** must **implement all the methods declared in the interface**.

Syntax:

```
interface <interface_name>{

    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

Java 8 Interface Improvement

- Since **<u>Java 8</u>, interface can have default** and **static methods** .
- Internal addition by the compiler

The Java compiler adds public and abstract keywords before the interface method. Moreover, it adds public, static and final keywords before data members.

In other words, Interface fields are public, static and final by default, and the methods are public and abstract.

The relationship between classes and interfaces

- As shown in the figure given below, **a class extends another class**, an **interface extends another interface**, **<u>but</u>** a **class implements an interface**.

Java Interface Example

- In this example, the **Printable interface has only one method**, and **its <u>implementation</u> is provided in the class A.**

inter1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

interface printable
{
   void print();
}

class A implements printable
{
  public void print()
   {
   System.out.println("Hello");
   }
}

class inter1
{
public static void main(String args[])
   {
   printable o=new A();
   o.print();
   }
}
```

Ln 24, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\interface>javac inter1.java

D:\java\programs\module-3\interface>java inter1
Hello
```

Java Interface Example: Drawable

- In this example, the Drawable interface has only one method. Its implementation is provided by Rectangle and Circle classes. In a real scenario, an interface is defined by someone else, but its implementation is provided by different implementation providers. Moreover, it is used by someone else. The implementation part is hidden by the user who uses the interface.

inter2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
interface Drawable
{
   void draw();

}
class Rectangle implements Drawable
{
  public void draw()
   {
   System.out.println("drawing rectangle");
   }
}

class Circle implements Drawable
{
  public void draw()
   {
   System.out.println("drawing circle");
   }
}

class inter2
{
public static void main(String args[])
   {

   Drawable o1=new Rectangle();
   o1.draw();

   o1=new Circle();
   o1.draw();
   }
}
```

Ln 10, Col 10

```
D:\java\programs\module-3\interface>javac inter2.java

D:\java\programs\module-3\interface>java inter2
drawing rectangle
drawing circle
```

Java Interface Example: Bank

- Let's see another example of java interface which provides the implementation of Bank interface.

inter3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
interface Bank
{
  float getRateOfInterest();

}
class SBI implements Bank
{
  public float getRateOfInterest()
   {
   return(9.15f);
   }
}

class PNB implements Bank
{
  public float getRateOfInterest()
   {
   return(9.7f);
   }
}

class inter3
{
public static void main(String args[])
   {

   Bank o1=new SBI();
   System.out.println("SBI Rate of Interest is:"+o1.getRateOfInterest()+"%");

   o1=new PNB();
   System.out.println("PNB Rate of Interest is:"+o1.getRateOfInterest()+"%");
   }
}
```

Ln 13, Col 5 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\interface>javac inter3.java

D:\java\programs\module-3\interface>java inter3
SBI Rate of Interest is:9.15%
PNB Rate of Interest is:9.7%
```

Multiple inheritance in Java by interface

- If **a class implements multiple interfaces, or** an **interface extends multiple interfaces, it is known as <u>multiple inheritance</u>**.

Multiple Inheritance in Java

inter4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

interface Printable
{
   void print();
}
interface Showable
{
   void show();
}

class A implements Printable,Showable
{
  public void print()
   {
   System.out.println("Hello");
   }
    public void show()
   {
   System.out.println("Welcome");
   }
}

class inter4
{
public static void main(String args[])
   {
   A o=new A();
   o.print();
   o.show();
   }
}
```

Ln 13, Col 1 100%

```
D:\java\programs\module-3\interface>javac inter4.java

D:\java\programs\module-3\interface>java inter4
Hello
Welcome
```

Q) Multiple inheritance is not supported through class in java, but it is possible by an interface, why?

- **multiple inheritance** is **not supported** **in the case of class because of ambiguity**.
- **However, it is supported** **in case of an interface because there is no ambiguity**.
- **It is because its implementation** **is provided by the implementation class**.

For example:

inter5 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

interface Printable
{
   void print();
}
interface Showable
{
   void print();
}

class A implements Printable,Showable
{
  public void print()
   {
   System.out.println("Hello");
   }

}

class inter5
{
public static void main(String args[])
   {
   A o=new A();
   o.print();
   }
}
```

Ln 22, Col 13

```
D:\java\programs\module-3\interface>javac inter5.java

D:\java\programs\module-3\interface>java inter5
Hello
```

- As you can see in the above example, **Printable** and **Showable interface have same methods but its implementation is provided by class A** , **so there is no ambiguity**.

Interface inheritance

- A **class implements an interface**, **but one interface extends another interface.**

inter6 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

interface Printable
{
   void print();
}
interface Showable extends Printable
{
   void show();
}

class A implements Showable
{
  public void print()
   {
   System.out.println("Hello");
   }
  public void show()
   {
   System.out.println("Welcome");
   }
}

class inter6
{
public static void main(String args[])
   {
   A o=new A();
   o.print();
   o.show();
   }
}
```

Ln 23, Col 1

```
D:\java\programs\module-3\interface>javac inter6.java

D:\java\programs\module-3\interface>java inter6
Hello
Welcome
```

Java 8 Default Method in Interface

- Since Java 8, **we can have method body in interface**. **But we need to make it default method**. Let's see an example:

inter7 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
interface Drawable
{
   void draw();

   default void msg()
    {
    System.out.println("default method");
    }
}

class Rectangle implements Drawable
{
  public void draw()
   {
   System.out.println("drawing rectangle");
   }
}

class inter7
{
public static void main(String args[])
   {

   Drawable o1=new Rectangle();
   o1.draw();
   o1.msg();
   }
}
```

Ln 12, Col 1

```
D:\java\programs\module-3\interface>javac inter7.java

D:\java\programs\module-3\interface>java inter7
drawing rectangle
default method
```

Java 8 Static Method in Interface

- Since Java 8, we can have **static method in interface**. Let's see an example:

inter8 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
interface Drawable
{
   void draw();

   static int cube(int x)
    {
    return(x*x*x);
    }
}

class Rectangle implements Drawable
{
  public void draw()
   {
   System.out.println("drawing rectangle");
   }
}

class inter8
{
public static void main(String args[])
   {

   Drawable o1=new Rectangle();
   o1.draw();
   System.out.println("cube of 3 is "+Drawable.cube(3));
   }
}
```

```
D:\java\programs\module-3\interface>java inter8
drawing rectangle
cube of 3 is 27
```

Q) What is marker or tagged interface?

- **An interface which has no member** is known as **a marker** or **tagged interface**, for example, Serializable, Cloneable, Remote, etc.
- **They are used to provide some essential information to the JVM so that JVM may perform some useful operation**.

```
//How Serializable interface is written?
public interface Serializable{
}
```

Nested Interface in Java

- Note: **An interface can have another interface which is known as** a **nested interface**.
- For example:

```
interface printable{
 void print();
 interface MessagePrintable{
   void msg();
 }
}
```

Difference between abstract class and interface

- **Abstract class** and **interface** **both are used to achieve abstraction** where we can declare the abstract methods.
- **Abstract class** and **interface both can't be instantiated**.
- But there are many **differences between abstract class** and **interface** that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract** methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.
5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class** can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class** can be extended using keyword "extends".	An **interface** can be implemented using keyword "implements".
8) A Java **abstract class** can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
	Example:
	public interface Drawable{
	void draw();

Example of abstract class and interface in Java

- Let's see a simple example where we are using interface and abstract class both

inter9 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
interface A
{
   void a();
   void b();
   void c();
   void d();
}
abstract class B implements A
{
 public void c()
  {
   System.out.println("I am c");
  }
}
class M extends B
{
  public void a()
  {
   System.out.println("I am a");
  }
 public void b()
  {
   System.out.println("I am b");
  }
 public void d()
  {
   System.out.println("I am d");
  }
}
class inter9
{
public static void main(String args[])
   {
   A o=new M();
   o.a();
   o.b();
   o.c();
   o.d();
   }
}
```

Ln 10, Col 1

```
D:\java\programs\module-3\interface>javac inter9.java

D:\java\programs\module-3\interface>java inter9
I am a
I am b
I am c
I am d
```

Java Package

- A **java package is a group of similar types of classes**, **interfaces** and **sub-packages**.
- **Package in java can be categorized in two form**, **built-in package** and **user-defined package**.
- **There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql** etc.
- Here, we will have the detailed **learning of creating** and **using user-defined packages.**

Advantage of Java Package

1) **Java package is used to categorize the classes** and **interfaces so that they can be easily maintained**.

2) **Java package provides access protection**.

3) **Java package removes naming collision**.

Simple example of java package

- The **package keyword** is used to create a package in java.

Simple - Notepad

File Edit Format View Help

```
package mypack;
public class Simple
{
public static void main(String args[])
    {
     System.out.println("Welcome to Package");
    }
}
```

Ln 8, Col 2 | 100% | Windows (CRLF) | UTF-8

How to compile java package

javac filename.java

Eg:

javac Simple.java

<u>How to Run java package</u>

java packagename.filename

Eg:

java mypack.Simple

```
D:\java\programs\module-3\package\mypack>javac Simple.java

D:\java\programs\module-3\package\mypack>cd ..

D:\java\programs\module-3\package>java mypack.Simple
Welcome to Package
```

How to access package from another package?

- There are **three ways to access the package from outside the package.**
    1. **import package.*;**
    2. **import package.classname;**
    3. **fully qualified name.**

1) Using packagename.*

- **If you use package.* then all the classes** and **interfaces of this package <u>will be accessible</u> but not subpackages**.
- The **import keyword is used to make the classes** and **interface of another package accessible to the current package**.
- Example of package that import the packagename.*

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

2) Using packagename.classname

- If you import package.classname then only declared class of this package will be accessible.
- Example of package by import package.classname

```
//save by A.java

package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.A;

class B{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

```
Output:Hello
```

3) Using fully qualified name

- **If you use fully qualified name then only declared class of this package will be accessible**.
- Now **there is no need to import**. But you need to use fully qualified name every time when you are accessing the class or interface.
- **It is generally used when two packages have same class name**

e.g. **java.util** and **java.sql packages contain Date class**.

Example of package by import fully qualified name

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

```
Output:Hello
```

Note: If you import a package, subpackages will not be imported.

- If you import a package, all the classes and interface of that package will be imported excluding the classes and interfaces of the subpackages. Hence, you need to import the subpackage as well.
- Note: Sequence of the program must be package then import then class.

Subpackage in java

- **Package inside the package is called the subpackage**. It should be created **to categorize the package further**.

Example of Subpackage

```
package com.javatpoint.core;
class Simple{
  public static void main(String args[]){
   System.out.println("Hello subpackage");
  }
}
```

To Compile: javac -d . Simple.java

To Run: java com.javatpoint.core.Simple

```
Output:Hello subpackage
```

How to put two public classes in a package?

- **If you want to put two public classes in a package, have two java source files containing one public class, but keep the package name same. For example:**

Dell - Notepad

File Edit Format View Help

```
package mypack;
public class Dell
{
public void disp()
    {
     System.out.println("From Dell class");
    }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

Hp - Notepad

File Edit Format View Help

```
package mypack;
public class Hp
{
public void display()
    {
     System.out.println("From Hp class");
    }
}
```

Show - Notepad

File Edit Format View Help

```
import mypack.*;

class Show
{
public static void main(String args[])
   {

    Dell obj1=new Dell();

    obj1.disp();

   Hp obj2=new Hp();
   obj2.display();
   }
}
```

Ln 13, Col 16 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\package>javac Show.java

D:\java\programs\module-3\package>java Show
From Dell class
From Hp class
```

Eg: 2

cube - Notepad

File Edit Format View Help

```
package mypack;

import java.io.*;
import java.util.*;

public class cube
{
public void cal()
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter a number");
    int num=io.nextInt();
    int res=num*num*num;
    System.out.println("Cube of "+num+" is:"+res);
   }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

factorial - Notepad

File Edit Format View Help

```
package mypack;

import java.io.*;
import java.util.*;

public class factorial
{
public void fact()
   {
    Scanner io=new Scanner(System.in);
    System.out.println("Enter a number");
    int num=io.nextInt();
    int res=1,temp=num;
    while(num>=1)
    {
     res=res*num;
     num--;
    }
    System.out.println("Factorial of "+temp+" is:"+res);
   }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

Show1 - Notepad

File Edit Format View Help

```
import mypack.*;

class Show1
{
public static void main(String args[])
   {

    cube obj1=new cube();

    obj1.cal();

   factorial obj2=new factorial();
   obj2.fact();
   }
}
```

Ln 15, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-3\package>java Show1
Enter a number
3
Cube of 3 is:27
Enter a number
5
Factorial of 5 is:120
```

----------End of Module-3----------

Module-4

Exception handling and Multithreaded Programming

Syllabus:

- **Fundamentals of exception handling,**
- **Exception Types,**
- **Using try and catch, throw, throws, finally,**
- **Java's Built-in Exceptions,**
- **Creating Exception subclasses,**
- **the Java Thread Model,**
- **Creating a Thread,**
- **Creating Multiple Threads,**
- **Thread Priorities, Synchronization, Interthread Communication**

Exception Handling in Java

- The **Exception Handling in Java** is **one of the powerful *mechanism* *to handle the runtime errors* so that the normal flow of the application can be maintained.**
- **Exception handling plays an important role in software development.**

What is Exception in Java?

- **Exception** is an **abnormal condition**.
- In Java, an **exception is an event that disrupts the normal flow of the program.**
- **It is an object which is thrown at runtime**.
- **Exception handling in java helps in minimizing exceptions** and **helps in recovering from exceptions**.
- It is **one of the powerful mechanisms to handle runtime exceptions** and **makes it bug-free**.
- **Exception handling helps in maintaining the flow of the program**

What is Exception Handling?

- **Exception Handling is a mechanism to handle runtime errors** such as **ClassNotFoundException, IOException, SQLException, RemoteException**, etc.

Advantage of Exception Handling

- The core advantage of exception handling is **to maintain the normal flow of the application**.
- An **exception normally disrupts the normal flow of the application**; **that is why we need to handle exceptions**.
- Let's consider a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

- Suppose there are 10 statements in a Java program and an **exception occurs at statement 5**; the **rest of the code will not be executed**, i.e., **statements 6 to 10 will not be executed**.
- However, when **we perform exception handling**, **the rest of the statements will be executed**. That is why **we use exception handling in Java.**

Hierarchy of Java Exception classes

- **The java.lang.Throwable class** is the **root class** of **Java Exception** hierarchy inherited by **two subclasses**: **Exception** and **Error**.
- The hierarchy of Java Exception classes is given below:

Types of Java Exceptions

- There are mainly **two types of exceptions**: **checked** and **unchecked**. An **error is considered as** the **unchecked exception**. However, according to Oracle, **there are three types of exceptions** namely:
    1. **Checked Exception**
    2. **Unchecked Exception**
    3. **Error**

Difference between Checked and Unchecked Exceptions

1) Checked Exception

- The classes which directly inherit Throwable class **except RuntimeException** and **Error are known as checked exceptions**.
- e.g. IOException, SQLException etc.
- **Checked exceptions are checked at compile-time**.
- **A checked exception is an exception** that is typically a **user error** or a **problem that cannot be foreseen by the programmer.**
- For example, if a file is to be opened, but the file cannot be found, an exception occurs.
- These exceptions **cannot simply be ignored at the time of compilation.**

2) Unchecked Exception

- The classes which inherit **RuntimeException are known as unchecked exceptions**
- e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc.
- **Unchecked exceptions are not checked at compile-time**, **but they are checked at runtime**.

3) Error

- **Error is irrecoverable**
- e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.
- These are not exceptions at all, but **problems that arise beyond the control of the user** or **the programmer**.
- Errors are typically ignored in your code because you can rarely do anything about an error.

- For example, if a **stack overflow occurs**, an **error will arise**. **They are also ignored <u>at the time of compilation.</u>**

Difference between Checked and Unchecked Exception

Checked Exceptions	Unchecked Exceptions
Occur at compile time.	Occur at runtime.
The compiler checks for a checked exception.	The compiler doesn't check for exceptions.
Can be handled at the compilation time.	Can't be caught or handled during compilation time.
The JVM requires that the exception be caught and handled.	The JVM doesn't require the exception to be caught and handled.
Example of Checked exception- 'File Not Found Exception'	Example of Unchecked Exceptions- 'No Such Element Exception'

Java Exception Keywords

- Java provides **five keywords** that **are used to handle the exception**. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java Exception Handling Example

- Let's see an example of Java Exception Handling in which we are using a try-catch statement to handle the exception.

JavaExceptionExample.java

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
     //code that may raise exception
     int data=100/0;
   }catch(ArithmeticException e){System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

- In the above example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

Java try-catch block

- Java **try block is used to enclose the code that might throw an exception**. **It must be used within the method**.
- If an **exception occurs at the particular statement <u>in the try block</u>, the rest of the block code <u>will not execute</u>**. So, it is recommended **not to keep the code in try block that <u>will not throw an exception</u>**.
- Java **try block must be followed by either catch** or **finally block.**

Syntax of Java try-catch

```
try{
//code that may throw an exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw an exception
}finally{}
```

Java catch block

- **Java catch block is used to handle the Exception by declaring the type of exception within the parameter**.
- **The declared exception <u>must be the parent class exception</u>** (i.e., Exception) or **the generated exception type**.
- However, **the good approach is to declare the generated type of exception.**
- The **catch block must be used after** the **try block only**.
- **You can use multiple catch block <u>with a single try block.</u>**

Internal Working of Java try-catch block

- The **JVM firstly checks** whether the **exception is handled** or **not**.
- If **exception is not handled**, **JVM provides** a **<u>default exception handler</u> that performs the following tasks**:

1. **Prints out exception description.**
2. **Prints the stack trace (Hierarchy of methods where the exception occurred).**
3. **Causes the program to terminate.**

- **But** if the **application programmer handles the exception**, **<u>the normal flow of the application is maintained</u>**, i.e., **rest of the code is executed.**

Problem without exception handling

- Let's try to understand the problem if we don't use a try-catch block.

exception1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;


class exception1
{
public static void main(String args[])
   {
    int data=100/0;
    System.out.println("Rest of the code");
  }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\Exception>java exception1
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exception1.main(exception1.java:9)
```

- As displayed in the above example, the **rest of the code** is not executed (in such case, the **rest of the code** statement is not printed).
- There might be 100 lines of code after the exception. If the exception is not handled, all the code below the exception won't be executed.

Solution by exception handling

- Let's see the solution of the above problem by a java try-catch block.

exception3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception3
{
public static void main(String args[])
   {
     try
     {
      int data=100/0;
     }
    catch(Exception e)
     {
     System.out.println(e);
     }

     System.out.println("Rest of the code");
     System.out.println("welcome to java exception program");
  }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\Exception>java exception3
java.lang.ArithmeticException: / by zero
Rest of the code
welcome to java exception program
```

- As displayed in the above example, the **rest of the code** is executed, i.e., the **rest of the code** statement is printed.

Example 3

- In this example, we also kept the code in a try block that will not throw an exception.

exception2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception2
{
public static void main(String args[])
   {
     try
     {
      int data=100/0;
      System.out.println("Rest of the code");
     }
    catch(Exception e)
     {
     System.out.println(e);
     }
  }
}
```

Ln 16, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\Exception>java exception2
java.lang.ArithmeticException: / by zero
```

- Here, we can see that if an exception occurs in the try block, the rest of the block code will not execute.

Example 4

- Here, we handle the exception using the parent class exception.

exception3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception3
{
public static void main(String args[])
   {
     try
     {
      int data=100/0;
     }
    catch(Exception e)
     {
     System.out.println(e);
     }

     System.out.println("Rest of the code");
     System.out.println("welcome to java exception program");
  }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\Exception>java exception3
java.lang.ArithmeticException: / by zero
Rest of the code
welcome to java exception program
```

Example 5

- Let's see an example to print a custom message on exception.

exception3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception3
{
public static void main(String args[])
   {
     try
     {
      int data=100/0;
     }
    catch(Exception e)
     {
     System.out.println("Can't divide by zero");
     }


  }
}
```

```
D:\java\programs\module-4\Exception>java exception3
Can't divide by zero
```

Example 6

- Let's see an example to resolve the exception in a catch block.

exception4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class exception4
{
public static void main(String args[])
   {
     int i=50;
     int j=0;

     try
     {
      int data=i/j;

     }

    catch(Exception e)
     {
     System.out.println(i/(j+2));
     }
  }
}
```

Ln 22, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\Exception>java exception4
25
```

Example 7

- In this example, along with try block, we also enclose exception code in a catch block.

exception5 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception5
{
public static void main(String args[])
   {

    try
     {
      int data=50/0;
     }

    catch(Exception e)
     {
     int data2=50/0;
     }
    System.out.println("rest of the code");
  }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\Exception>java exception5
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exception5.main(exception5.java:18)
```

- Here, we can see that the catch block didn't contain the exception code. So, enclose exception code within a try block and use catch block only to handle the exceptions.

Example 8

- In this example, we handle the generated exception (Arithmetic Exception) with a different type of exception class (ArrayIndexOutOfBoundsException).

exception5 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception5
{
public static void main(String args[])
   {

    try
     {
      int data=50/0;
     }

    catch(ArrayIndexOutOfBoundsException e)
     {
     System.out.println(e);
     }
    System.out.println("rest of the code");
  }
}
```

Ln 15, Col 28 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\Exception>java exception5
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at exception5.main(exception5.java:10)
```

Example 9

- Let's see an example to handle another unchecked exception.

exception7 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception7
{
public static void main(String args[])
   {

    try
     {
      int arr[]={1,3,5,7};
      System.out.println(arr[10]);
     }
    catch(ArrayIndexOutOfBoundsException e)
     {
     System.out.println(e);
     }

    System.out.println("rest of the code");
  }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\Exception>java exception7
java.lang.ArrayIndexOutOfBoundsException: 10
rest of the code
```

Example 10

- Let's see an example to handle checked exception.

TryCatchExample10.java

```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class TryCatchExample10 {

   public static void main(String[] args) {



      PrintWriter pw;
      try {
         pw = new PrintWriter("jtp.txt"); //may throw exception
         pw.println("saved");
      }
// providing the checked exception handler
 catch (FileNotFoundException e) {

         System.out.println(e);
      }
   System.out.println("File saved successfully");
   }
}
```

Output:

```
File saved successfully
```

Java Catch Multiple Exceptions

Java Multi-catch block

- **A try block can be followed by one** or **more catch blocks**.
- **Each catch block must contain** a **different exception handler**.
- **So,** if **you have to perform different tasks at the occurrence of different exceptions**, **use java multi-catch block**.

Points to remember

- **At a time only one exception occurs** and **at a time only one catch block is executed.**
- All catch blocks must be ordered from **most specific to most general**, i.e. catch for ArithmeticException must come before catch for Exception.

Flowchart of Multi-catch Block

Example 1

- Let's see a simple example of java multi-catch block.

exception1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception1
{
public static void main(String args[])
   {

    try
     {
      int a[]=new int[5];
      a[6]=30/0;
     }

    catch(ArithmeticException e)
     {
     System.out.println("Arithmetic Exception occurs");
     }

    catch(ArrayIndexOutOfBoundsException e)
     {
     System.out.println(" Array Index Out Of Bounds Exception occurs");
     }

   catch(Exception e)
     {
     System.out.println("Parent Exception occurs");
     }

    System.out.println("rest of the code");
  }
}
```

Ln 31, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\muticatch>java exception1
Arithmetic Exception occurs
rest of the code
```

Example 2

exception2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception2
{
public static void main(String args[])
   {

    try
     {
      int a[]=new int[5];
       System.out.println(a[10]);
     }
    catch(ArithmeticException e)
     {
     System.out.println("Arithmetic Exception occurs");
     }
    catch(ArrayIndexOutOfBoundsException e)
     {
     System.out.println(" Array Index Out Of Bounds Exception occurs");
     }

   catch(Exception e)
     {
     System.out.println("Parent Exception occurs");
     }

    System.out.println("rest of the code");
  }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\muticatch>java exception2
 Array Index Out Of Bounds Exception occurs
rest of the code
```

Example-3

- In this example, **try block contains two exceptions**. **But at a time only one exception occurs** and **its corresponding catch block is executed**.

exception3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception3
{
public static void main(String args[])
   {

    try
     {
      int a[]=new int[5];
      a[5]=30/0;
      System.out.println(a[10]);
     }
    catch(ArithmeticException e)
     {
     System.out.println("Arithmetic Exception occurs");
     }
    catch(ArrayIndexOutOfBoundsException e)
     {
     System.out.println(" Array Index Out Of Bounds Exception occurs");
     }

   catch(Exception e)
     {
     System.out.println("Parent Exception occurs");
     }

    System.out.println("rest of the code");
  }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\muticatch>java exception3
Arithmetic Exception occurs
rest of the code
```

Example 4

- In this example, we **generate NullPointerException**, **but didn't provide the corresponding exception type**.
- **In such case**, the **catch block containing the parent exception** class **Exception will invoked**.

exception4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception4
{
public static void main(String args[])
   {

    try
     {
      String s=null;
      System.out.println(s.length());
     }
    catch(ArithmeticException e)
     {
     System.out.println("Arithmetic Exception occurs");
     }
    catch(ArrayIndexOutOfBoundsException e)
     {
     System.out.println(" Array Index Out Of Bounds Exception occurs");
     }

   catch(Exception e)
     {
     System.out.println("Parent Exception occurs");
     }

    System.out.println("rest of the code");
  }
}
```

Ln 11, Col 5 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\muticatch>java exception4
Parent Exception occurs
rest of the code
```

Example 5

- Let's see an example, **to handle the exception without maintaining the order of exceptions** (i.e. **from most specific to most general**).

exception6 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception6
{
public static void main(String args[])
   {

    try
     {
      int a[]=new int[5];
      a[5]=30/0;
     }
    catch(Exception e)
     {
     System.out.println("common task completed");
     }
    catch(ArithmeticException e)
     {
     System.out.println("task 1 completed");
     }
    catch(ArrayIndexOutOfBoundsException e)
     {
     System.out.println(" task 2 completed");
     }

     System.out.println("rest of the code");
   }
}
```

Ln 16, Col 7 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\muticatch>javac exception6.java
exception6.java:17: error: exception ArithmeticException has already been caught
    catch(ArithmeticException e)
    ^
exception6.java:21: error: exception ArrayIndexOutOfBoundsException has already been caught
    catch(ArrayIndexOutOfBoundsException e)
    ^
2 errors
```

Java Nested try block

- In Java, using a **try block inside another try block is permitted**. **It is called as nested try block**. Every statement that **we enter a statement in** **<u>try block</u>**, **context of that** **<u>exception is pushed onto the stack.</u>**
- For example, the **inner try block** can be used to handle **ArrayIndexOutOfBoundsException** while the **outer try block** can handle the **ArithemeticException** (division by zero).

Why use nested try block

- Sometimes a situation may arise where a **part of a block may cause one error** and **the entire block itself may cause another error**. **In such cases**, **exception handlers have to be nested**.

Syntax:

```
....
//main try block
try
{
   statement 1;
   statement 2;
//try catch block within another try block
   try
   {
      statement 3;
      statement 4;
//try catch block within nested try block
      try
      {
         statement 5;
         statement 6;
   }
      catch(Exception e2)
      {
//exception message
      }
```

```
   }
   catch(Exception e1)
   {
//exception message
   }
}
//catch block of parent (outer) try block
catch(Exception e3)
{
//exception message
}
....
```

Java Nested try Example

Example 1

- **Let's see an example where we place a try block within another try block for two different exceptions.**

exception1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception1
{
public static void main(String args[])
 {
 try
  {

    try
     {
      System.out.println("going to divide by 0");
      int b=39/0;
     }
     catch(ArithmeticException e)
     {
     System.out.println(e);
     }

   try
     {
     int a[]=new int[5];
     a[5]=4;
     }

    catch(ArrayIndexOutOfBoundsException e)
     {
     System.out.println(e);
     }

    System.out.println("other statement");
}
catch(Exception e)
     {
     System.out.println("handled the exception (outer catch)");
      }
  System.out.println("normal flow");
  }
}
```

Ln 39, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\nested>java exception1
going to divide by 0
java.lang.ArithmeticException: / by zero
java.lang.ArrayIndexOutOfBoundsException: 5
other statement
normal flow
```

- When any **try block does not have a catch block for a particular exception**, **then** the **catch block of the outer (parent) try block are checked for that exception**, and **if it matches**, the **catch block of outer try block is executed.**
- If **none of the catch block** specified in the code **is unable to handle the exception**, **then** the **Java runtime system will handle the exception**. Then **it displays the system generated message for that exception.**

Example 2

- Let's consider the following example. Here the try block within nested try block (inner try block 2) do not handle the exception. The control is then transferred to its parent try block (inner try block 1). If it does not handle the exception, then the control is transferred to the main try block (outer try block) where the appropriate catch block handles the exception. It is termed as nesting.

exception2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class exception2
{
public static void main(String args[])
{

try
 {
      try
        {
           try
             {
              int a[]={1,2,3,4};
              System.out.println(a[10]);
             }
           catch(ArithmeticException e)
             {
             System.out.println("Arithmetic Exception");
             System.out.println("inner try block 2");
             }
        }

      catch(ArithmeticException e)
        {
         System.out.println("Arithmetic Exception");
         System.out.println("inner try block 1");
        }
 }
 catch(ArrayIndexOutOfBoundsException e)
    {
     System.out.println(e);
     System.out.println("Outer main try block 1");
    }

 catch(Exception e)
    {
     System.out.println("Exception");
     System.out.println("handled in main try block");
    }

  }
}
```

```
D:\java\programs\module-4\nested>java exception2
java.lang.ArrayIndexOutOfBoundsException: 10
Outer main try block 1
```

Java finally block

- **Java finally block is a block used to execute <u>important code</u> such as closing the connection**, etc.
- Java **finally block <u>is always executed</u> whether an exception is handled** or **not**. **Therefore**, **it contains all the necessary statements** that need to be printed regardless of the exception occurs or not.
- The **finally block follows the try-catch block.**

> Note: If you don't handle the exception, before terminating the program, JVM executes finally block (if any).

Flowchart of finally block

Why use Java finally block?

- **finally block in Java can be used to put "cleanup" code such as closing a file, closing connection**, etc.
- The **important statements to be printed can be placed in the finally block.**

Usage of Java finally

- Let's see the **different cases where** Java **finally block can be used**.

Case 1: When an exception does not occur

- Let's see the below example where the **Java program does not throw any exception**, and the **finally block is executed <u>after the try block</u>**.

final1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;


class final1
{
public static void main(String args[])
   {

    try
     {
      int data=25/5;
      System.out.println(data);
     }

    catch(NullPointerException e)
     {
     System.out.println(e);
     }

    finally
     {
     System.out.println("finally block is always executed");
     }

  System.out.println("rest of the code");
}
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\finally>java final1
5
finally block is always executed
rest of the code
```

Case 2: When an exception occur but not handled by the catch block

- Let's see the the fillowing example. Here, **the code throws an exception** however the **catch block cannot handle it**. Despite this, the **finally block is executed after the try block** and then **the program terminates abnormally.**

final2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class final2
{
public static void main(String args[])
   {

    try
     {
      System.out.println("Inside the try block");
      int data=25/0;
      System.out.println(data);
     }

    catch(NullPointerException e)
     {
     System.out.println(e);
     }

    finally
     {
     System.out.println("finally block is always executed");
     }

  System.out.println("rest of the code");
}
}
```

Ln 12, Col 24 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\finally>java final2
Inside the try block
finally block is always executed
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at final2.main(final2.java:12)
```

Case 3: When an exception occurs and is handled by the catch block

Example:

- Let's see the following example where the Java code throws an exception and the catch block handles the exception. Later the finally block is executed after the try-catch block. Further, the rest of the code is also executed normally.

final3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class final3
{
public static void main(String args[])
   {

    try
     {
      System.out.println("Inside the try block");
      int data=25/0;  //exception occur (arthmetic exception)
      System.out.println(data);
     }

    catch(ArithmeticException e)
     {
     System.out.println("Exception handled");
     System.out.println(e);
     }

    finally
     {
     System.out.println("finally block is always executed");
     }

  System.out.println("rest of the code");
}
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\finally>java final3
Inside the try block
Exception handled
java.lang.ArithmeticException: / by zero
finally block is always executed
rest of the code
```

Rule: For each try block there can be zero or more catch blocks, but only one finally block.

Note: The finally block will not be executed if the program exits (either by calling System.exit() or by causing a fatal error that causes the process to abort).

Java throw Exception

What is Exception in Java?

- **Exception is an abnormal condition.**
- **In Java, an exception is an event that disrupts the normal flow of the program.**

Exception Handling in Java

- The **Exception Handling in Java** is **one of the powerful *mechanism to handle the runtime errors* so that the normal flow of the application can be maintained.**
- **Exception handling plays an important role in software development.**

Java Exception Keywords

- Java provides **five keywords** that **are used to handle the exception**. The following table describes each.

Keyword	Description
try	The "try" keyword is used to specify a block where we should place an exception code. It means we can't use try block alone. The try block must be followed by either catch or finally.
catch	The "catch" block is used to handle the exception. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used to execute the necessary code of the program. It is executed whether an exception is handled or not.
throw	The "throw" keyword is used to throw an exception.
throws	The "throws" keyword is used to declare exceptions. It specifies that there may occur an exception in the method. It doesn't throw an exception. It is always used with method signature.

Java throw keyword

- The Java **throw keyword** is used to **throw an exception explicitly**.
- We specify the **exception object which is to be thrown**. The **Exception has some message** with it **that provides** the **error description**. **These exceptions may be related to user inputs**, **server**, etc.
- **We can throw either checked** or **unchecked exceptions** in Java **by throw keyword**. It is **mainly used to throw** a **custom exception**.
- **We can also define our own set of conditions** and **throw an exception explicitly using throw keyword**.
- For example, we can throw ArithmeticException if we divide a number by another number. Here, we just need to set the condition and throw exception using throw keyword.
- **The syntax** of the Java **throw keyword** is given below.
- throw Instance i.e.,

```
throw new exception_class("error message");
```

Let's see the example of throw IOException.

```
throw new IOException("sorry device error");
```

Java throw keyword Example

- Example 1: Throwing **Unchecked Exception**
- In this example, we have created a method named validate() that accepts an integer as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.
- **TestThrow1.java**
- In this example, we have created the validate method that takes integer value as a parameter. If the age is less than 18, we are throwing the ArithmeticException otherwise print a message welcome to vote.

throw2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class throw2
{
 public static void validate(int age)
   {
    if(age<18)
     {

      throw new ArithmeticException("Person is not eligible to vote")

     }
   else
    {
    System.out.println("Person is eligible to vote");
    }

  }

public static void main(String args[])
   {
      validate(13);
      System.out.println("rest of the code");
   }
}
```

Ln 5, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\throw>java throw2
Exception in thread "main" java.lang.ArithmeticException: Person is not eligible to vote
        at throw2.validate(throw2.java:10)
        at throw2.main(throw2.java:22)
```

The above code throw an unchecked exception. Similarly, we can also throw unchecked and user defined exceptions.

Note: If we throw unchecked exception from a method, it is must to handle the exception or declare in throws clause.

If we throw a checked exception using throw keyword, it is must to handle the exception using catch block or the method must declare it using throws declaration.

throw1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;


class throw1
{
public static void validate(int age)
   {
    if(age<18)
     {
      throw new ArithmeticException("Person is not eligible to vote");
     }
   else
    {
    System.out.println("Person is eligible to vote");
    }

  }

public static void main(String args[])
   {

    try
     {
      validate(13);
     }
    catch(ArithmeticException e)
     {
      System.out.println(e);
     }


  System.out.println("rest of the code");
}
}
```

Ln 23, Col 8 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\throw>java throw1
java.lang.ArithmeticException: Person is not eligible to vote
rest of the code
```

Example 2: Throwing Checked Exception

Note: Every subclass of Error and RuntimeException is an unchecked exception in Java. A checked exception is everything else under the Throwable class.

TestThrow2.java

```java
import java.io.*;

public class TestThrow2 {

    //function to check if person is eligible to vote or not
    public static void method() throws FileNotFoundException {

        FileReader file = new FileReader("C:\\Users\\Anurati\\Desktop\\abc.txt");
        BufferedReader fileInput = new BufferedReader(file);



        throw new FileNotFoundException();

    }
    //main method
    public static void main(String args[]){
        try
        {
            method();
        }
```

```
        catch (FileNotFoundException e)
        {
            e.printStackTrace();
        }
        System.out.println("rest of the code...");
    }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow2.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow2
java.io.FileNotFoundException
        at TestThrow2.method(TestThrow2.java:12)
        at TestThrow2.main(TestThrow2.java:22)
rest of the code...
```

Example 3: Throwing User-defined Exception

- exception is everything else under the Throwable class.

throw4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class UserDefinedException extends Exception
{

public UserDefinedException(String str)
   {
    super(str);
   }
}

class throw4
{



public static void main(String args[])
   {

      try
      {

      throw new UserDefinedException("This is uesr defined exception");
      }
     catch(UserDefinedException msg)
     {
      System.out.println("Caught the Exception");
      System.out.println(msg);
     }


      System.out.println("rest of the code");
}
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\throw>java throw4
Caught the Exception
UserDefinedException: This is uesr defined exception
rest of the code
```

Java throws keyword

- The **Java throws keyword is used to declare an exception**. It gives an information to the programmer that there may occur an exception. So, it is better for the programmer to provide the exception handling code so that the normal flow of the program can be maintained.
- Exception Handling is **mainly used to handle the checked exceptions**. If there occurs any unchecked exception such as NullPointerException, it is programmers' fault that he is not checking the code before it being used.

Which exception should be declared?

- **Ans: Checked exception only, because:**
- **unchecked exception: under our control so we can correct our code.**
- **error: beyond our control. For example, we are unable to do anything if there occurs VirtualMachineError or StackOverflowError.**

Advantage of Java throws keyword

- Now Checked Exception can be propagated (forwarded in call stack).
- It provides information to the caller of the method about the exception.

Java throws Example

- Let's see the example of Java throws clause which describes that checked exceptions can be propagated by throws keyword.

throw1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;


class throw1
{

  void m() throws IOException
       {
        throw new IOException("device error");
       }
   void n() throws IOException
        {
         m();
        }
  void p()
      {
        try
          {
           n();
          }
         catch(Exception e)
           {
            System.out.println("Exception handled");
           }
      }


public static void main(String args[])
   {
    throw1 o=new throw1();
    o.p();
    System.out.println("normal flow");
   }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\throws>java throw1
Exception handled
normal flow
```

Rule: If we are calling a method that declares an exception, we must either caught or declare the exception.

- **There are two cases:**

1. **Case 1:** We have caught the exception i.e. we have handled the exception using try/catch block.
2. **Case 2:** We have declared the exception i.e. specified throws keyword with the method.

Case 1: Handle Exception Using try-catch block

- In case we handle the exception, the code will be executed fine whether exception occurs during the program or not.

throw2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class throw1
{

  void method() throws IOException
       {
        throw new IOException("device error");
       }
}

class throw2
{

public static void main(String args[])
   {
     try
       {

        throw1 o=new throw1();
        o.method();
       }
     catch(Exception e)
       {
        System.out.println("Exception handled");
        }
    System.out.println("normal flow");
   }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\throws>java throw2
Exception handled
normal flow
```

Case 2: Declare Exception

- In case we declare the exception, **if exception does not occur**, the **code will be executed fine**.
- In case we declare the exception and the exception occurs, it will be thrown at runtime because **throws** does not handle the exception.
- Let's see examples for both the scenario.

A) If exception does not occur

throw3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;


class throw1
{

  void method() throws IOException
       {
        System.out.println("device operation performed");
       }
}

class throw3
{

public static void main(String args[]) throws IOException
   {
    throw1 o=new throw1();
    o.method();
    System.out.println("normal flow");
   }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\throws>java throw3
device operation performed
normal flow
```

B) If exception occurs

throw4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class throw1
{

  void method() throws IOException
       {
        throw new IOException("device error");
       }
}

class throw4
{

public static void main(String args[]) throws IOException
   {
    throw1 o=new throw1();
    o.method();
    System.out.println("normal flow");
   }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\throws>java throw4
Exception in thread "main" java.io.IOException: device error
        at throw1.method(throw4.java:8)
        at throw4.main(throw4.java:18)
```

Difference between throw and throws in Java

- The **throw** and **throws is the concept of exception handling** where the **throw keyword throw the exception explicitly from a method** or **a block of code whereas** the **throws keyword is used in signature of the method. (**In Java, a method signature is part of the method declaration. It's the combination of the method name and the parameter list.)
- There are **many differences between throw** and **throws** keywords.
- A list of **differences between throw** and **throws are given below:**

Sr. no.	Basis of Differences	throw	throws
1.	Definition	Java throw keyword is used throw an exception explicitly in the code, inside the function or the block of code.	Java throws keyword is used in the method signature to declare an exception which might be thrown by the function while the execution of the code.
2.	Type of exception Using throw keyword, we can only propagate unchecked exception i.e., the checked exception cannot be propagated using throw only.	Using throws keyword, we can declare both checked and unchecked exceptions. However, the throws keyword can be used to propagate checked exceptions only.	
3.	Syntax	The throw keyword is followed by an instance of Exception to be thrown.	The throws keyword is followed by class names of Exceptions to be thrown.
4.	Declaration	throw is used within the method.	throws is used with the method signature.
5.	Internal implementation	We are allowed to throw only one exception at a time i.e. we cannot throw multiple exceptions.	We can declare multiple exceptions using throws keyword that can be thrown by the method. For example, main() throws IOException, SQLException.

Java throw Example

TestThrow.java

```java
public class TestThrow {
   //defining a method
   public static void checkNum(int num) {
      if (num < 1) {
         throw new ArithmeticException("\nNumber is negative, cannot calculate square");
      }
      else {
         System.out.println("Square of " + num + " is " + (num*num));
      }
   }
   //main method
   public static void main(String[] args) {
         TestThrow obj = new TestThrow();
         obj.checkNum(-3);
         System.out.println("Rest of the code..");
   }
}
```

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrow.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrow
Exception in thread "main" java.lang.ArithmeticException:
Number is negative, cannot calculate square
        at TestThrow.checkNum(TestThrow.java:6)
        at TestThrow.main(TestThrow.java:16)
```

Java throws Example

TestThrows.java

```
public class TestThrows {
   //defining a method
   public static int divideNum(int m, int n) throws ArithmeticException {
      int div = m / n;
      return div;
   }
   //main method
   public static void main(String[] args) {
      TestThrows obj = new TestThrows();
      try {
         System.out.println(obj.divideNum(45, 0));
      }
      catch (ArithmeticException e){
         System.out.println("\nNumber cannot be divided by 0");
      }

      System.out.println("Rest of the code..");
   }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrows

Number cannot be divided by 0
Rest of the code..
```

Java throw and throws Example

TestThrowAndThrows.java

```
public class TestThrowAndThrows
{
   // defining a user-defined method
   // which throws ArithmeticException
   static void method() throws ArithmeticException
   {
      System.out.println("Inside the method()");
      throw new ArithmeticException("throwing ArithmeticException");
   }
   //main method
   public static void main(String args[])
   {
      try
      {
         method();
      }
      catch(ArithmeticException e)
      {
         System.out.println("caught in main() method");
      }
   }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestThrowAndThrows.java

C:\Users\Anurati\Desktop\abcDemo>java TestThrowAndThrows
Inside the method()
caught in main() method
```

Lab Cycle Program

Construct a Stack Class with proper exception handling mechanisms. While doing a Pop operation, if the stack is empty then display an error message. While doing a Push operation, if the stack is full then display corresponding error message.

nstack - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class StackException extends Exception
{
public StackException(String str)
 {
  super(str);
 }
}
interface stack
{
 public void push(int num) throws StackException;
 public void pop()throws StackException;
}
class operation implements stack
{
 int top;
 int A[];
 int size;

    operation(int n)
     {
       top=-1;
       A=new int[n];
       size=n;
     }
  public void push(int num) throws StackException
    {
      if(top==size-1)
        {
         throw new StackException("Stack is Full");
         }
      else
         {
           A[++top]=num;
         }
   }
```

Ln 46, Col 47 | 100% | Windows (CRLF) | UTF-8

*nstack - Notepad

File Edit Format View Help

```
public void pop()throws StackException
  {
   int re;
    if(top==-1)
     {
     throw new StackException("Stack is Empty");
     }
    else
     {
      re=A[top];
      System.out.println("removed element from the stack is :"+re);
      top--;
     }
  }
public void display()
  {

  int i;
  if(top==-1)
    {
    System.out.println("Stack is empty");
    }
  else
    {
     for(i=top; i>=0;i-- )
     {
       System.out.println(A[i]);
     }
    }
  }
}
```

Ln 72, Col 1 | 100% | Windows (CRLF) | UTF-8

*nstack - Notepad

File Edit Format View Help

```
class nstack
{
public static void main(String args[])
   {
     Scanner io=new Scanner(System.in);
     System.out.println("Enter the value of n:");
     int n=io.nextInt();
     int ch,num;
     char c;
     operation o=new operation(n);
       do
        {
          System.out.println("\n1.push\n2.pop\n3.display");
          System.out.println("Enter your choice(1-3):");
          ch=io.nextInt();
        try
          {
            switch(ch)
               {
                case 1:System.out.println("Enter the number:");
                        num=io.nextInt();
                        o.push(num);
                       break;
                 case 2:o.pop();
                       break;
                 case 3:o.display();
                       break;
                 default:System.out.println("Invalid Choice");
               }
            }
          catch(StackException e)
             {
              System.out.println(e);
             }
            System.out.println("Do you want to continue(y/n):");
            c=io.next().charAt(0);
       }while(c=='Y'||c=='y');
   }
}
```

Ln 111, Col 1 100% Windows (CRLF) UTF-8

```
D:\java\programs\module-4\lab>java nstack
Enter the value of n:
3

1.push
2.pop
3.display
Enter your choice(1-3):
1
Enter the number:
10
Do you want to continue(y/n):
y

1.push
2.pop
3.display
Enter your choice(1-3):
1
Enter the number:
20
Do you want to continue(y/n):
y

1.push
2.pop
3.display
Enter your choice(1-3):
1
Enter the number:
30
Do you want to continue(y/n):
y
```

```
1.push
2.pop
3.display
Enter your choice(1-3):
1
Enter the number:
40
StackException: Stack is Full
Do you want to continue(y/n):
y

1.push
2.pop
3.display
Enter your choice(1-3):
3
30
20
10
Do you want to continue(y/n):
y

1.push
2.pop
3.display
Enter your choice(1-3):
2
removed element from the stack is :30
Do you want to continue(y/n):
y

1.push
2.pop
3.display
Enter your choice(1-3):
2
```

```
Do you want to continue(y/n):
y

1.push
2.pop
3.display
Enter your choice(1-3):
3
10
Do you want to continue(y/n):
y

1.push
2.pop
3.display
Enter your choice(1-3):
2
removed element from the stack is :10
Do you want to continue(y/n):
y

1.push
2.pop
3.display
Enter your choice(1-3):
2
StackException: Stack is Empty
Do you want to continue(y/n):
```

Java's Built-in Exceptions

Exceptions

- Inside the standard package **java.lang**.
- **Java defines several exception classes**.
- The **most general of these exceptions are subclasses of the standard type RuntimeException.**
- **Exceptions** can be categorized into **two ways**:

1. **Built-in Exceptions**
    1. Checked Exception
    2. Unchecked Exception
2. **User-Defined Exceptions**

Built-in Exception

- **Exceptions** that are **already available** in **Java libraries** are referred to as **built-in exception**.
- These exceptions **are able to define the error situation so that we can understand the reason of getting this error**.
- **It can be categorized into two** broad categories, i.e., **checked exceptions** and **unchecked exception**.

Checked Exception

- **Checked exceptions** are called **compile-time exceptions because these exceptions are checked at compile-time by the compiler**.
- The **compiler ensures whether the programmer handles the exception** or **not**.
- **The programmer should have to handle the exception**; **otherwise**, **the system has shown** a **compilation error**.
- **Common checked exceptions include IOException**, DataAccessException, InterruptedException, etc.

Unchecked Exceptions

- The **unchecked exceptions are just opposite to the checked exceptions**.
- The **compiler will not check these exceptions at compile time**.
- In simple words, if a program throws an unchecked exception, and even if we didn't handle or declare it, the program would not give a compilation error. Usually, it occurs when the user provides bad data during the interaction with the program.
- **Common unchecked exceptions include** ArithmeticException, InvalidClassException, NullPointerException, etc. These exceptions are propagated using the throws keyword.

Creating Exception subclasses (Java Custom Exception)

- In Java, we can create our own exceptions that are derived classes of the Exception class.
- Creating our own Exception is known as custom exception or user-defined exception.
- Basically, Java custom exceptions are used to customize the exception according to user need.
- Consider the example 1 in which InvalidAgeException class extends the Exception class.
- Using the custom exception, we can have your own exception and message. Here, we have passed a string to the constructor of superclass i.e. Exception class that can be obtained using getMessage() method on the object we have created.
- In this section, we will learn how custom exceptions are implemented and used in Java programs.

TestCustomException1.java

```
// class representing custom exception
class InvalidAgeException  extends Exception
{
   public InvalidAgeException (String str)
   {
      // calling the constructor of parent Exception
      super(str);
   }
}

// class that uses custom exception InvalidAgeException
public class TestCustomException1
{

   // method to check the age
   static void validate (int age) throws InvalidAgeException{
     if(age < 18){

      // throw an object of user defined exception
      throw new InvalidAgeException("age is not valid to vote");
   }
```

```
        else {
        System.out.println("welcome to vote");
        }
    }

    // main method
    public static void main(String args[])
    {
        try
        {
            // calling the method
            validate(13);
        }
        catch (InvalidAgeException ex)
        {
            System.out.println("Caught the exception");

            // printing the message from InvalidAgeException object
            System.out.println("Exception occured: " + ex);
        }

        System.out.println("rest of the code...");
    }
}
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException1.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException1
Caught the exception
Exception occured: InvalidAgeException: age is not valid to vote
rest of the code...
```

Example 2:

```
// class representing custom exception
class MyCustomException extends Exception
{

}

// class that uses custom exception MyCustomException
public class TestCustomException2
{
   // main method
   public static void main(String args[])
   {
      try
      {
         // throw an object of user defined exception
         throw new MyCustomException();
      }
      catch (MyCustomException ex)
      {
         System.out.println("Caught the exception");
         System.out.println(ex.getMessage());
      }

      System.out.println("rest of the code...");
   }
```

Output:

```
C:\Users\Anurati\Desktop\abcDemo>javac TestCustomException2.java

C:\Users\Anurati\Desktop\abcDemo>java TestCustomException2
Caught the exception
null
rest of the code...
```

the Java Thread Model

Thread Concept in Java

- Before introducing the **thread concept**, we were **unable to run <u>more than one task</u> in parallel**. It was a **drawback**, and **to remove that drawback**, **Thread Concept was introduced**.
- A **Thread** is a **very light-weighted process**, or we can say the smallest part of the process **that allows a program to operate more efficiently by running <u>multiple tasks</u> simultaneously**.
- In order **to perform <u>complicated tasks</u> in the background**, we used the **Thread concept in Java**. **All the tasks are executed <u>without affecting the main program</u>**. In a program or process, **all the threads have their own separate path for execution**, so **each thread of a process is independent.**

- **Another benefit of using thread** is that **if a thread gets an exception** or **an error at the time of its execution**, **<u>it doesn't affect the execution</u> of the other threads**.
- **All the threads share a common memory** and **have their own stack**, **local variables** and **program counter**.
- When **multiple threads are executed <u>in parallel</u> at the same time**, this process is known as **<u>Multithreading</u>**

In a simple way, a Thread is a:

- Feature through which **we can perform multiple activities within a single process.**
- **Lightweight process**.
- **Series of executed statements**.
- **Nested sequence of method calls**.

Thread Model/ Life cycle of a Thread /Thread States

- The life cycle of the thread in java is controlled by JVM.
- A thread can be in one of the five states.
- The java thread states are as follows:

  1. **New**
  2. **Runnable**
  3. **Running**
  4. **Non-Runnable (Blocked)**
  5. **Terminated**

1) New

- The thread is in **new state if you create an instance of Thread class but before the invocation of start() method**.

2) Runnable

- The thread is in **runnable state after invocation of start() method**, **but** the **thread scheduler has not selected** it **to be the running thread**.

3) Running

- The thread is in **running state if** the **thread scheduler has selected it**.

4) Non-Runnable (Blocked)

- This is the **state when** the thread is **still alive**, **but is currently not eligible to run**. A thread is in the **Blocked** state **when it is waiting for resources**.

5) Terminated

- A thread is in **terminated** or **dead state when** its **run() method exits.**

Creating Thread

- **A thread is created either by "creating** or **implementing"** the **Runnable Interface or** by extending the **Thread class**.
- These are the **only two ways through which we can create a thread**.
- Let's dive into details of both these way of creating a thread:

1. Thread Class

- A **Thread class has several methods** and **constructors which allow us to perform various operations on a thread.**
- The **Thread class extends** the **Object class**.
- The **Object class implements the Runnable interface**.
- The **thread class has the following constructors that are used to perform various operations.**

- **Thread()**
- **Thread(Runnable, String name)**
- **Thread(Runnable target)**
- **Thread(ThreadGroup group, Runnable target, String name)**
- **Thread(ThreadGroup group, Runnable target)**
- **Thread(ThreadGroup group, String name)**
- **Thread(ThreadGroup group, Runnable target, String name, long stackSize)**

Runnable Interface(run() method)

- The **Runnable interface is required to be implemented by that class** whose instances are intended to be executed by a thread.
- The **runnable interface gives us the run() method <u>to perform an action for the thread.</u>**

start() method

- The **start() method is used for starting a thread** that we have newly created. **It starts a new thread** with a new callstack.
- **After executing the start() method**, **the thread changes <u>the state</u> from New to Runnable**.
- **It executes the run() method <u>when</u> the thread gets the correct time to execute it**.
- Let's take an example to understand how we can create a **<u>Java</u>** thread by extending the Thread class:

Method 1: Thread Creation by extending Thread Class

test1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test1 extends Thread
{

public void run()
{
System.out.println("My thread is in running state");
}

public static void main(String args[])
   {
    test1 o=new test1();
    o.start();
  }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\thread>java test1
My thread is in running state
```

Java Thread Example by extending Thread class

test2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test2 extends Thread
{

public void run()
{
int a=10;
int b=12;
int result=a+b;
System.out.println("Thread started running...");
System.out.println("Sum of two numbers is"+result);
}

public static void main(String args[])
   {
    test2 o=new test2();
    o.start();
  }
}
```

Ln 5, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\thread>java test2
Thread started running...
Sum of two numbers is22
```

2. Creating thread by implementing the runnable interface

- In Java, we can also create a thread by implementing the runnable interface. The **runnable interface provides us both the run() method and** the **start() method**.
- Let's takes an example to understand how we can create, start and run the thread using the runnable interface.

2) Java Thread Example by implementing Runnable interface

test3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class test3 implements Runnable
{

public void run()
{
System.out.println("My thread is in running state");
}

public static void main(String args[])
   {
    test3 o=new test3();
    Thread t1=new Thread(o);
    t1.start();
  }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\thread>java test3
My thread is in running state
```

Multithreading in Java

- When **multiple threads are executed in parallel at the same time**, this process is known as **Multithreading**.
- Java **Multithreading** is **mostly used in games**, **animation**, etc.

Advantages of Java Multithreading

1) It **doesn't block the user because threads are independent** and you **can perform multiple operations at the same time.**

2) You **can perform many operations together, so it saves time**.

3) Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

- **Multitasking** is a **process of executing multiple tasks simultaneously**.
- **We use multitasking to utilize the CPU**.
- Multitasking can be achieved in **two ways**:
    1. **Process-based Multitasking (Multiprocessing)**
    2. **Thread-based Multitasking (Multithreading)**

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- **A process is heavyweight**.
- **Cost of communication between the process is high.**
- **Switching from one process to another requires some time** for **saving** and **loading registers**, **memory maps**, **updating lists**, etc.

2) Thread-based Multitasking (Multithreading)

- Threads **share the same address space**.
- A thread is **lightweight.**
- **Cost of communication between the thread** is **low**.

Note: At least one process is required for each thread.

1. Thread creation by extending the Thread class

- We create a class that extends the **java.lang.Thread** class. This class overrides the run() method available in the Thread class. A thread begins its life inside run() method. We create an object of our new class and call start() method to start the execution of a thread. Start() invokes the run() method on the Thread object.

Java Thread sleep() method

- The **sleep() method** of thread class is **used to sleep a thread for the specified amount of time.**

Syntax

```
public static void sleep(long milis)throws InterruptedException
public static void sleep(long milis, int nanos)throws InterruptedException
```

Parameter

millis: It defines the length of time to sleep in milliseconds

nanos: 0-999999 additional nanoseconds to sleep

Return

It does not **return** any value.

Throws

- **IllegalArgumentException:** If the value of millis **is negative** or the value of nanos is **not in the range** 0-999999.
- **InterruptedException:** If any thread has interrupted the current thread. The interrupted status of the current thread is cleared when this exception is thrown.

Example 1

```
public class SleepExp1 extends Thread
{
    public void run()
    {
        for(int i=1;i<5;i++)
        {
            try
            {
                Thread.sleep(500);
            }catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[])
    {
        SleepExp1 t1=new SleepExp1();
        SleepExp1 t2=new SleepExp1();
        t1.start();
        t2.start();
    }
}
```

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

- As you know that **at a time only one thread is executed**.
- **If you sleep a thread for the specified time, the thread scheduler <u>picks up another thread</u>** and so on.

Example 2: when sleep time is negative

```
public class SleepExp2 extends Thread
{
    public void run()
    {
        for(int i=1;i<5;i++)
        {
            try
            {
                Thread.sleep(-500); // sleep time is negative
            }catch(InterruptedException e){System.out.println(e);}
            System.out.println(i);
        }
    }
    public static void main(String args[])
    {
        SleepExp2 t1=new SleepExp2();
        SleepExp2 t2=new SleepExp2();
        t1.start();
        t2.start();
    }
}
```

Output:

```
Exception in thread "Thread-0" Exception in thread "Thread-1" java.lang.IllegalArgumentException:
        at java.lang.Thread.sleep(Native Method)
        at SleepExp2.run(SleepExp2.java:9)
java.lang.IllegalArgumentException: timeout value is negative
        at java.lang.Thread.sleep(Native Method)
        at SleepExp2.run(SleepExp2.java:9)
```

Thread creation by implementing the Runnable Interface

- We create a new class which implements java.lang.Runnable interface and override run() method. Then we instantiate a Thread object and call start() method on this object.

test4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test5 implements Runnable
{
String name;
   test5(String name)
    {
     this.name=name;
     Thread t=new Thread(this,name);
     System.out.println("A New thread:"+t+"is created\n")
     t.start();
     }
public void run()
{
 try
   {
    for(int j=5;j>0;j--)
     {
     System.out.println(name+":"+j);
     Thread.sleep(1000);
     }
   }
  catch(InterruptedException e)
   {
    System.out.println(name+" thread interrupted");
  }
  System.out.println(name+" thread exiting");
}

}

class test4
{

public static void main(String args[])
   {
   new test5("1st");
   new test5("2nd");
   new test5("3rd");

  }
}
```

Ln 28, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\thread>java test4
A New thread:Thread[1st,5,main]is created

A New thread:Thread[2nd,5,main]is created

A New thread:Thread[3rd,5,main]is created

2nd:5
1st:5
3rd:5
1st:4
3rd:4
2nd:4
2nd:3
1st:3
3rd:3
3rd:2
1st:2
2nd:2
2nd:1
1st:1
3rd:1
3rd thread exiting
1st thread exiting
2nd thread exiting
```

- In the above example, we perform the Multithreading by implementing the runnable interface.

Lab Cycle Program

Develop a Java application that executes two threads. One thread displays —"Hello" in every 1000 milliseconds and other displays — "World" in every 3000 milliseconds. Create the threads by extending the Thread class

test4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class test5 extends Thread
{

public void run()
{

Hello t1=new Hello();
t1.start();
World t2=new World();
t2.start();
}
}

class Hello extends Thread
 {
  public void run()
   {
    for(int i=1;i<=10;i++)
    {
    try
     {
     Thread.sleep(1000);
     }
     catch(InterruptedException e)
      {
      System.out.println(e);
      }
    System.out.println("hello");
    }
   }
}
```

Ln 36, Col 1 | 100% | Windows (CRLF) | UTF-8

test4 - Notepad

File Edit Format View Help

```
class World extends Thread
{
 public void run()
   {
for(int i=1;i<=10;i++)
    {
   try
     {
     Thread.sleep(3000);
     }
     catch(InterruptedException e)
      {
      System.out.println(e);
      }
    System.out.println("world");
 }
   }

}

class test4
{

public static void main(String args[])
   {
    test5 t1=new test5();

    t1.start();

  }
}
```

Ln 36, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\multithread>java test4
hello
hello
world
hello
hello
hello
world
hello
hello
hello
world
hello
hello
world
world
world
world
world
world
world
```

Lab Program

Develop a java program that implements a multi-thread application that has three threads. First thread generates random integer for every 1 second and if the value is even, second thread computes the square of the number and prints and if the value is odd, the third thread will print the value of cube of the number

test2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test1 extends Thread
{
        public void run()
                        {
                        Random num1=new Random();  //create a random number

                        for (int i = 0; i < 5; i++)
                          {
                          int num2=num1.nextInt(10);

                          System.out.println("Random Integer generated : " + num2);
                          if((num2%2)==0)
                                {
                                 Square t2=new Square(num2);
                                 t2.start();
                                }
                         else
                               {
                                 Cube t3 = new Cube(num2);
                                 t3.start();
                              }

                         try
                            {
                                 Thread.sleep(1000);
                             }
                         catch (InterruptedException e)
                               {
                                 System.out.println(e);
                               }
                }
        }
}
```

Ln 3, Col 1 | 100% | Windows (CRLF) | UTF-8

test2 - Notepad

File Edit Format View Help

```
class Square extends Thread
{
        int number;

        Square(int n)
        {
                number=n;
        }

        public void run()
          {
                System.out.println("Square of " + number + " = " + (number * number));
         }
}

class Cube extends Thread
 {
        int number;

        Cube(int n)
         {
                number=n;
        }

        public void run()
         {
                System.out.println("Cube of " + number + " = " + number * number * number);
        }
}

class test2
{
        public static void main(String args[])
        {
                test1 t1 = new test1();
                t1.start();
        }
}
```

```
D:\java\programs\module-4\lab>java test2
Random Integer generated : 5
Cube of 5 = 125
Random Integer generated : 9
Cube of 9 = 729
Random Integer generated : 2
Square of 2 = 4
Random Integer generated : 7
Cube of 7 = 343
Random Integer generated : 8
Square of 8 = 64
```

Priority of a Thread (Thread Priority):

Thread Priority

- **Each thread have a priority**.
- **Priorities are represented by a number between** 1 and **10**.
- **In most cases**, **thread schedular schedules the threads** **according to their priority** (known as preemptive scheduling).
- But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

3 constants defined in Thread class:

1. public static int **MIN_PRIORITY**
2. public static int **NORM_PRIORITY**
3. public static int **MAX_PRIORITY**

- **Default priority of a thread is 5** (**NORM_PRIORITY**).
- **The value of MIN_PRIORITY** is **1** and
- **the value of MAX_PRIORITY** is **10**.

Java Thread getPriority() method

- The **getPriority() method of thread class** **is used to check the priority of the thread**.
- **When we create a thread**, it has **some priority assigned to it**.
- **Priority of thread** can **either be assigned by** the **JVM** or **by the programmer** explicitly **while creating the thread**.
- The **thread's priority is in the range of 1** to **10**.
- The **default priority of a thread is 5**.

Syntax

```
public final int getPriority()
```

Return

It returns the **thread's priority**.

test1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

public class test1 extends Thread
{
    public void run()
    {
        System.out.println("running thread name is:"+Thread.currentThread().getName());
    }
    public static void main(String args[])
    {
        // creating two threads
        test1 t1 = new test1();
        test1 t2 = new test1();
        // print the default priority value of thread
        System.out.println("t1 thread priority : " + t1.getPriority());
        System.out.println("t2 thread priority : " + t2.getPriority());
        // this will call the run() method
        t1.start();
        t2.start();
    }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\threadPriority>java test1
t1 thread priority : 5
t2 thread priority : 5
running thread name is:Thread-0
running thread name is:Thread-1
```

Java Thread setPriority() method

- The **setPriority() method of thread class is used to change the thread's priority**.
- **Every thread has a priority which is represented by the integer number between 1** to **10**.
- Thread class provides 3 constant properties:

1. **public static int MIN_PRIORITY:** It is the maximum priority of a thread. The value of it is 1.
2. **public static int NORM_PRIORITY:** It is the normal priority of a thread. The value of it is 5.

3. **public static int MAX_PRIORITY:** It is the minimum priority of a thread. The value of it is 10.

- We can also set the priority of thread between 1 to 10. This priority is known as custom priority or user defined priority.

Syntax

```
public final void setPriority(int a)
```

Parameter

- **a**: It is the priority to set this thread to.

Return

- It does not return any value.

Exception

- **IllegalArgumentException:** This exception throws if the priority is not in the range MIN_PRIORITY to MAX_PRIORITY.
- **SecurityException:** This exception throws if the current thread cannot modify this thread.

Example 1: Maximum Priority Thread

test3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class test3 extends Thread
{
    public void run()
    {
        System.out.println("Priority of thread is: "+Thread.currentThread().getPriority());
    }
    public static void main(String args[])
    {
        // creating one thread
        test3 t1=new test3();

        // print the maximum priority of this thread
        t1.setPriority(Thread.MAX_PRIORITY);
        // call the run() method
        t1.start();
    }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\threadPriority>java test3
Priority of thread is: 10
```

Example 2: Minimum Priority Thread

test4 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test4 extends Thread
{
    public void run()
    {
        System.out.println("Priority of thread is: "+Thread.currentThread().getPriority());
    }
    public static void main(String args[])
    {
        // creating one thread
        test4 t1=new test4();
        // print the minimum priority of this thread
        t1.setPriority(Thread.MIN_PRIORITY);
        // This will call the run() method
        t1.start();
    }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\threadPriority>java test4
Priority of thread is: 1
```

Example 3: Normal Priority Thread

test5 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test5 extends Thread
{
    public void run()
    {
        System.out.println("Priority of thread is: "+Thread.currentThread().getPriority());
    }
    public static void main(String args[])
    {
        // creating one thread
        test5 t1=new test5();
        // print the normal priority of this thread
        t1.setPriority(Thread.NORM_PRIORITY);
        // starting the thread
        t1.start();
    }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\threadPriority>java test5
Priority of thread is: 5
```

Example 4: User define Priority Thread

test6 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class test6 extends Thread
{
    public void run()
    {
        System.out.println("running...");
    }
    public static void main(String args[])
    {
        // creating one thread
        test6 t1=new test6();
        test6 t2=new test6();
        // set the priority
        t1.setPriority(4);
        t2.setPriority(7);
        // print the user defined priority
        System.out.println("Priority of thread t1 is: " + t1.getPriority()); //4
        System.out.println("Priority of thread t2 is: " + t2.getPriority()); //7
        // this will call the run() method
        t1.start();
    }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\threadPriority>java test6
Priority of thread t1 is: 4
Priority of thread t2 is: 7
running...
```

Example 5: When priority is greater than 10

```java
public class JavaSetPriorityExp5 extends Thread
{
    public void run()
    {
        System.out.println("running...");
    }
    public static void main(String args[])
    {
        // creating one thread
        JavaSetPriorityExp5 t1=new JavaSetPriorityExp5();
        JavaSetPriorityExp5 t2=new JavaSetPriorityExp5();
        // set the priority
        t1.setPriority(12);
        t2.setPriority(7);
        // print exception because priority of t1 is greater than 10
        System.out.println("Priority of thread t1 is: " + t1.getPriority());
        System.out.println("Priority of thread t2 is: " + t2.getPriority());
        // call the run() method
        t1.start();
    }
}
```

```
Exception in thread "main" java.lang.IllegalArgumentException
        at java.lang.Thread.setPriority(Thread.java:1089)
        at JavaSetPriorityExp5.main(JavaSetPriorityExp5.java:13)
```

Example of priority of a Thread:

test9 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class test9 extends Thread
{
 public void run()
   {
   System.out.println("running thread name is:"+Thread.currentThread().getName());
   System.out.println("running thread priority is:"+Thread.currentThread().getPriority());
   }
 public static void main(String args[])
  {
  test9 m1=new test9();
  test9 m2=new test9();
  m1.setPriority(Thread.MIN_PRIORITY);
  m2.setPriority(Thread.MAX_PRIORITY);
  m1.start();
  m2.start();
  }
}
```

```
D:\java\programs\module-4\threadPriority>java test9
running thread name is:Thread-0
running thread priority is:1
running thread name is:Thread-1
running thread priority is:10
```

Synchronization in Java

- **Synchronization in Java is the capability to control the access of multiple threads to any shared resource.**
- **Java Synchronization is better option where we want <u>to allow only one thread to access</u> the shared resource.**

Why use Synchronization?

- The synchronization is mainly used to
    1. **To prevent thread interference**.
    2. **To prevent consistency problem**.

Types of Synchronization

- There are **two types of synchronization**
    1. **Process Synchronization**
    2. **Thread Synchronization**

Here, we will **discuss only thread synchronization**.

Thread Synchronization

- There are **two types of thread synchronization mutual exclusive** and **inter-thread communication**.

1. **Mutual Exclusive**
    1. **Synchronized method.**
    2. **Synchronized block.**

3. **Static synchronization.**

2. **Cooperation (Inter-thread communication in java)**

Mutual Exclusive

- Mutual Exclusive helps **keep threads from <u>interfering with one another</u> while sharing data**.
- It can be achieved by using the following **three ways**:

    1. **By Using Synchronized Method**
    2. **By Using Synchronized Block**
    3. **By Using Static Synchronization**

Concept of Lock in Java

- Synchronization is built around an internal entity known as the **lock** or **monitor**.
- **Every object has a lock** associated with it.
- By convention, a thread that needs consistent access to an object's fields has to **acquire the object's lock before accessing them**, and **then release the lock when it's done with them**.
- From Java 5 the package java.util.concurrent.locks contains several lock implementations.

Understanding the problem without Synchronization

- In this example, there is **no synchronization**, so output is inconsistent. Let's see the example:

test1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Table
{
void printTable(int n)    //method not synchronized
       {
       for(int i=1;i<=5;i++)
         {
          System.out.println(n*i);
           try{
               Thread.sleep(400);
              }
           catch(Exception e)
              {
               System.out.println(e);
              }
         }

      }
}
class MyThread1 extends Thread
{
Table t;
MyThread1(Table t)
{
this.t=t;
}

public void run()
{
t.printTable(5);
}

}
```

Ln 33, Col 1 | 100% | Windows (CRLF) | UTF-8

*test1 - Notepad

File Edit Format View Help

```
class MyThread2 extends Thread
{
Table t;

MyThread2(Table t)
{
this.t=t;
}

public void run()
{
t.printTable(100);
}

}

class test1
{
public static void main(String args[])
{
Table o1= new Table();//only one object
MyThread1 t1=new MyThread1(o1);
MyThread2 t2=new MyThread2(o1);
t1.start();
t2.start();
}
}
```

Ln 37, Col 1 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\synchronization>java test1
5
100
200
10
300
15
400
20
500
25
```

Java Synchronized Method

- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

test2 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Table
{
synchronized  void printTable(int n)    //synchronized  method
       {
       for(int i=1;i<=5;i++)
         {
          System.out.println(n*i);
           try{
               Thread.sleep(400);
              }
           catch(Exception e)
              {
               System.out.println(e);
              }
         }

      }
}
class MyThread1 extends Thread
{
Table t;
        MyThread1(Table t)
        {
        this.t=t;
        }
        public void run()
        {
        t.printTable(5);
        }
}
```

Ln 57, Col 2 | 100% | Windows (CRLF) | UTF-8

test2 - Notepad

File Edit Format View Help

```
class MyThread2 extends Thread
{
Table t;
        MyThread2(Table t)
        {
        this.t=t;
        }

        public void run()
        {
        t.printTable(100);
        }
}

class test2
{
        public static void main(String args[])
        {
        Table o1= new Table();//only one object
        MyThread1 t1=new MyThread1(o1);
        MyThread2 t2=new MyThread2(o1);
        t1.start();
        t2.start();
        }
}
```

Ln 58, Col 4 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\synchronization>java test2
5
10
15
20
25
100
200
300
400
500
```

Synchronized Block in Java

- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.
- If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

Points to Remember

- Synchronized block is used to lock an object for any shared resource.
- Scope of synchronized block is smaller than the method.
- A Java synchronized block doesn't allow more than one JVM, to provide access control to a shared resource.
- The system performance may degrade because of the slower working of synchronized keyword.
- Java synchronized block is more efficient than Java synchronized method.

Syntax

```
synchronized (object reference expression) {
  //code block
}
```

Example of Synchronized Block

- Let's see the simple example of synchronized block.

test3 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;
class Table
{
  void printTable(int n)
    {
       synchronized(this)    //synchronized  block
       {
         for(int i=1;i<=5;i++)
           {
            System.out.println(n*i);
              try{
                  Thread.sleep(400);
                  }
               catch(Exception e)
                  {
                 System.out.println(e);
                 }
           }
       }
   }
}

class MyThread1 extends Thread
{
        Table t;
        MyThread1(Table t)
        {
        this.t=t;
        }

        public void run()
        {
        t.printTable(5);
        }
}
```

Ln 9, Col 1 | 100% | Windows (CRLF) | UTF-8

*test3 - Notepad

File Edit Format View Help

```
class MyThread2 extends Thread
{
        Table t;
        MyThread2(Table t)
        {
        this.t=t;
        }
        public void run()
        {
        t.printTable(100);
        }
}

class test3
{
        public static void main(String args[])
        {
        Table o1= new Table();//only one object
        MyThread1 t1=new MyThread1(o1);
        MyThread2 t2=new MyThread2(o1);
        t1.start();
        t2.start();
        }
}
```

Ln 61, Col 3 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\synchronization>java test3
5
10
15
20
25
100
200
300
400
500
```

Inter-thread Communication in Java

Synchronization in Java

- Synchronization in Java is the capability **to control the access of multiple threads to any shared resource**.
- **Java Synchronization is better option** where we want **<u>to allow only one thread to access</u> the shared resource**.

Types of Synchronization

- There are **two types of synchronization**
    1. **Process Synchronization**
    2. **Thread Synchronization**

Here, we will **discuss only thread synchronization**.

Thread Synchronization

- There are **two types of thread synchronization mutual exclusive** and **inter-thread communication**.

1. **Mutual Exclusive**
    1. Synchronized method.
    2. Synchronized block.
    3. Static synchronization.
2. **Cooperation (Inter-thread communication in java)**

- **Inter-thread communication** or **Co-operation** is all about **allowing synchronized threads <u>to communicate with each other</u>**.
- **Cooperation** (**Inter-thread communication**) **is a mechanism in which** a **thread is paused running in its critical section** and **<u>another thread is allowed to enter</u>** (or lock) **in the same critical section to be executed**.
- It is implemented by following methods of **Object class**:

- wait()
- notify()
- notifyAll()

1) wait() method

- **The wait() method causes <u>current thread to release the lock</u>** and **wait until either another thread <u>invokes</u> the notify() method** or the **notifyAll() method** for this object, or a specified amount of time has elapsed.
- The current thread must own this object's monitor, so it must be called from the synchronized method only otherwise it will throw exception.

Method	Description
public final void wait()throws InterruptedException	It waits until object is notified.
public final void wait(long timeout)throws InterruptedException	It waits for the specified amount of time.

2) notify() method

- The **notify() method <u>wakes up a single thread</u>** that is waiting on this object's monitor.
- If any threads are waiting on this object, one of them is chosen to be awakened. The choice is arbitrary and occurs at the discretion of the implementation.

Syntax:

```
public final void notify()
```

3) notifyAll() method

- **Wakes up all threads** that are waiting on this object's monitor.

Syntax:

```
public final void notifyAll()
```

Understanding the process of inter-thread communication

The point to point explanation of the above diagram is as follows:

1. **Threads enter to acquire lock**.
2. **Lock is acquired by one thread**.
3. **Now thread goes to waiting state** if you call **wait() method** on the object. Otherwise it releases the lock and exits.
4. If you call **notify()** or **notifyAll()** method, **thread moves to the notified state (runnable state**).
5. Now thread is available to acquire lock.
6. After completion of the task, thread releases the lock and exits the monitor state of the object.

Why wait(), notify() and notifyAll() methods are defined in Object class not Thread class?

- It is because they are related to lock and object has a lock.

Difference between wait and sleep?

- Let's see the important differences between wait and sleep methods.

wait()	sleep()
The wait() method releases the lock.	The sleep() method doesn't release the lock.
It is a method of Object class	It is a method of Thread class
It is the non-static method	It is the static method
It should be notified by notify() or notifyAll() methods	After the specified amount of time, sleep is completed.

Example of Inter Thread Communication in Java

- **Let's see the simple example of inter thread communication.**

test1 - Notepad

File Edit Format View Help

```
import java.io.*;
import java.util.*;

class Customer
{
int amount=10000;

  synchronized void withdraw(int amount)
   {
     System.out.println("going to withdraw...");

      if(this.amount<amount)
         {
          System.out.println("Less balance; waiting for deposit...");
             try
               {
                wait();
               }
              catch(Exception e)
               {}
          }
    this.amount-=amount;
    System.out.println("withdraw completed...");
   }

   synchronized void deposit(int amount)
    {
    System.out.println("going to deposit...");
     this.amount+=amount;
    System.out.println("deposit completed... ");
     notify();
    }
}
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

test1 - Notepad

File Edit Format View Help

```
class test1
{
        public static void main(String args[])
        {
        Customer c=new Customer();

        new Thread()
        {
        public void run()
                {
                c.withdraw(15000);
                }
        }.start();


        new Thread()
        {
        public void run()
                {
                c.deposit(10000);
                }
        }.start();
     }
}
```

Ln 59, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-4\interthread>java test1
going to withdraw...
Less balance; waiting for deposit...
going to deposit...
deposit completed...
withdraw completed...
```

----------End of Module-4----------

Module-5

Event Handling, AWT and UML

Syllabus

- **Event Handling-delegation event model,**
- **event classes, sources, listeners.**
- **String class - basics.**
- **Applet basics and methods,**
- **AWT- working with frames, graphics, color, font.**
- **AWT Control fundamentals.**
- **Swing overview,**
- **Introduction to Object Oriented Modelling ,**
- **Unified Modeling Language,**
- **UML class diagram,**
- **Use-case diagram,**
- **Familiarisation of UMLtools,**
- **Case study**

Event Handling

Delegation Event Model in Java

- The **Delegation Event model is defined to handle events in GUI programming languages**
- **The GUI stands for Graphical User Interface**, where **a user graphically/visually interacts with the system**.
- **The GUI programming is inherently event-driven**; **whenever a user initiates an activity such as** a **mouse activity**, **clicks**, **scrolling**, etc., **each is known as an event that is mapped to a code to respond to functionality to the user**. **This is known as event handling**.
- In this section, we will discuss event processing and how to implement the delegation event model in Java .
- We will also discuss the different components of an Event Model.

Event Processing in Java

- **Java support event processing since** Java 1.0.
- It provides support for **AWT (Abstract Window Toolkit)** , **which is an API used to develop the Desktop application**.
- **The below image demonstrates the event processing.**

- **the modern approach for event processing is based on the Delegation Model**.
- It defines **a standard** and **compatible mechanism to generate** and **process events**.
- In this model, a **source generates an event** and **forwards it to one** or **more listeners**.
- **The listener waits until it receives an event**.
- **Once it receives the event, it is processed by the listener** and **returns it**.
- The **key advantage of the Delegation Event Model is that the application logic is completely separated from the interface logic**.
- In this model, **the listener must be connected with a source to receive the event notifications.**
- **In the older model**, an event was propagated up the containment until a component was handled. This needed components to receive events that were not processed, and it **took lots of time**. **The Delegation Event model overcame this issue.**

- Basically, an **Event Model is based on the following three components:**

    1. **Events**
    2. **Events Sources**
    3. **Events Listeners**

1. Events

- **The Events are the objects that define state change in a source**.
- An **event can be generated** as a **reaction of a user while interacting with GUI elements.**
- **Some of the event generation activities are moving the mouse pointer**, **clicking on a button**, **pressing the keyboard key**, **selecting an item from the list**, and **so on**. **We can also consider many other user operations as events**.
- The **Events may also occur that may be not related to user interaction**, **such as** a **timer expires**, **counter exceeded**, **system failures**, or **a task is completed**, etc. We can define events for any of the applied actions.

2. Event Sources

- A **source is an object that causes** and **generates an event**.
- **It generates an event when** the **internal state of the object is changed.**
- **The sources are allowed to generate several different types of events.**
- **A source must register a listener to receive notifications for a specific event.**
- **Each event contains its registration method**.
- Below is an example:

```
public void addTypeListener (TypeListener e1)
```

- From the above syntax, the **Type is the name of the event**, and **e1 is a reference to the event listener**.
- For example, for a keyboard event listener, the method will be called as **addKeyListener()**.
- For the mouse event listener, the method will be called as **addMouseMotionListener()**.

- When the specified event occurs, it will be notified to the registered listener. This process is known as **unicasting** events.
- Below is an example of the method that will remove the event from the listener.

```
public void removeTypeListener(TypeListener e2?)
```

- From the above syntax, the Type is an event name, and e2 is the reference of the listener.
- For example, to remove the keyboard listener, the **removeKeyListener()** method will be called.
- The **source provides the methods to add** or **remove listeners that generate the events**.

3.Event Listeners

- An **event listener is an object that is invoked when an event triggers**.
- The **listeners require two things**; **first**, **it must be registered with a source**; **Second**, **it must implement the methods to receive** and **process the received notifications**.
- The methods that deal with the events are defined in a set of interfaces. These interfaces can be found in the java.awt.event package.
- For example, the **MouseMotionListener** interface provides two methods when the mouse is dragged and moved. Any object can receive and process these events if it implements the MouseMotionListener interface.

Types of Events

- The events are categories into the following **two categories**:

The Foreground Events:

- The foreground events **are those events that require <u>direct interaction of the user</u>**.
- **These types of events are generated as a result of user interaction with the GUI component.**
- **For example**, **clicking on a button**, **mouse movement**, **pressing a keyboard key**, **selecting an option from the list**, etc.

The Background Events :

- The Background events **are those events that result from the interaction of the end-user**.
- **For example**, an **Operating system interrupts system failure** (Hardware or Software).
- To handle these events, we need an event handling mechanism that provides control over the events and responses.

Design Goals

The design goals of the event delegation model are as following:

- It is **easy to learn** and **implement**
- **It supports a clean separation between application** and **GUI code**.
- It provides robust event handling program code which is **less error-prone** (strong compile-time checking)
- **It is Flexible**, can enable different types of application models for event flow and propagation.
- It enables run-time discovery of both the component-generated events as well as observable events.
- It provides support for the backward binary compatibility with the previous model.

Event classes

Event and Listener (Java Event Handling)

- Changing the state of an object is known as an event.
- For example, click on button, dragging mouse etc.
- The **java.awt.event package provides many event classes** and **Listener interfaces <u>for event handling.</u>**

Java Event classes and Listener interfaces

Event Classes	Listener Interfaces
ActionEvent	ActionListener
MouseEvent	MouseListener and MouseMotionListener
MouseWheelEvent	MouseWheelListener
KeyEvent	KeyListener
ItemEvent	ItemListener
TextEvent	TextListener
AdjustmentEvent	AdjustmentListener
WindowEvent	WindowListener
ComponentEvent	ComponentListener
ContainerEvent	ContainerListener
FocusEvent	FocusListener

Steps to perform Event Handling

- Following steps are required to perform event handling:

1. Register the component with the Listener

Registration Methods

- For registering the component with the Listener, many classes provide the registration methods. For example:
- **Button**
  - public void addActionListener(ActionListener a){}
- **MenuItem**
  - public void addActionListener(ActionListener a){}
- **TextField**
  - public void addActionListener(ActionListener a){}
  - public void addTextListener(TextListener a){}
- **TextArea**
  - public void addTextListener(TextListener a){}
- **Checkbox**
  - public void addItemListener(ItemListener a){}
- **Choice**
  - public void addItemListener(ItemListener a){}
- **List**
  - public void addActionListener(ActionListener a){}
  - public void addItemListener(ItemListener a){}

Java Event Handling Code

- **We can put the event handling code into one of the following places:**

  1. **Within class**
  2. **Other class**
  3. **Anonymous class**

1. Java event handling by implementing ActionListener

- **public void setBounds(int xaxis, int yaxis, int width, int height); have been used in the above example that sets the position of the component it may be button, textfield etc.**

AEvent - Notepad

File Edit Format View Help

```
import java.awt.*;
import java.awt.event.*;
class AEvent extends Frame implements ActionListener
{
TextField tf;
        AEvent(){
                //create components
                tf=new TextField();
                tf.setBounds(60,50,170,20);
                Button b=new Button("click me");
                b.setBounds(100,120,80,30);
                //register listener
                b.addActionListener(this);//passing current instance
                //add components and set size, layout and visibility
                add(tf);
                add(b);
                setSize(300,300);
                setLayout(null);
                setVisible(true);
                }

        public void actionPerformed(ActionEvent e)
        {
        tf.setText("Welcome");
        }

        public static void main(String args[])
        {
        AEvent o=new AEvent();
        }
}
```

Ln 31, Col 3 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-5\event>java AEvent
```

Sources of Events

- **Table 24-2 lists some of the user interface components that can generate the events described in the previous section.**
- **In addition to these graphical user interface elements, any class derived from Component, such as Applet, can generate events.**
- **For example, you can receive key and mouse events from an applet. (You may also build your own components that generate events.)**
- **In this chapter, we will be handling only mouse and keyboard events, but the following two chapters will be handling events from the sources shown in Table 24-2.**

Event Source : Description

Button : Generates action events when the button is pressed.

Check box : Generates item events when the check box is selected or deselected.

Choice : Generates item events when the choice is changed.

List : Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.

Menu item : Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.

Scroll bar : Generates adjustment events when the scroll bar is manipulated.

Text components : Generates text events when the user enters a character.

Window : Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Event Source	Description
Button	Generates action events when the button is pressed.
Check box	Generates item events when the check box is selected or deselected.
Choice	Generates item events when the choice is changed.
List	Generates action events when an item is double-clicked; generates item events when an item is selected or deselected.
Menu item	Generates action events when a menu item is selected; generates item events when a checkable menu item is selected or deselected.
Scroll bar	Generates adjustment events when the scroll bar is manipulated.
Text components	Generates text events when the user enters a character.
Window	Generates window events when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24-2 Event Source Examples

Event Listener Interface

Interface	Description
ActionListener	Defines one method to receive action events.
AdjustmentListener	Defines one method to receive adjustment events.
ComponentListener	Defines four methods to recognize when a component is hidden, moved, resized, or shown.
ContainerListener	Defines two methods to recognize when a component is added to or removed from a container.
FocusListener	Defines two methods to recognize when a component gains or loses keyboard focus.
ItemListener	Defines one method to recognize when the state of an item changes.
KeyListener	Defines three methods to recognize when a key is pressed, released, or typed.
MouseListener	Defines five methods to recognize when the mouse is clicked, enters a component, exits a component, is pressed, or is released.
MouseMotionListener	Defines two methods to recognize when the mouse is dragged or moved.
MouseWheelListener	Defines one method to recognize when the mouse wheel is moved.
TextListener	Defines one method to recognize when a text value changes.
WindowFocusListener	Defines two methods to recognize when a window gains or loses input focus.
WindowListener	Defines seven methods to recognize when a window is activated, closed, deactivated, deiconified, iconified, opened, or quit.

Table 24-3 Commonly Used Event Listener Interfaces

1. Java ActionListener Interface

- **The Java ActionListener is notified whenever you click on the button or menu item.**
- **It is notified against ActionEvent.**
- **The ActionListener interface is found in java.awt.event package**
- **. It has only one method: actionPerformed().**

actionPerformed() method

- **The actionPerformed() method is invoked automatically whenever you click on the registered component.**

```
public abstract void actionPerformed(ActionEvent e);
```

How to write ActionListener

- **The common approach is to implement the ActionListener. If you implement the ActionListener class, you need to follow 3 steps:**
- **1) Implement the ActionListener interface in the class:**

```
public class ActionListenerExample Implements ActionListener
```

2) Register the component with the Listener:

```
component.addActionListener(instanceOfListenerclass);
```

3) Override the actionPerformed() method:

```
public void actionPerformed(ActionEvent e){
        //Write the code here
}
```

Java ActionListener Example: On Button click

```
import java.awt.*;
import java.awt.event.*;
//1st step
public class ActionListenerExample implements ActionListener{
public static void main(String[] args) {
    Frame f=new Frame("ActionListener Example");
    final TextField tf=new TextField();
    tf.setBounds(50,50, 150,20);
    Button b=new Button("Click Here");
    b.setBounds(50,100,60,30);
    //2nd step
    b.addActionListener(this);
    f.add(b);f.add(tf);
    f.setSize(400,400);
    f.setLayout(null);
    f.setVisible(true);
}
//3rd step
public void actionPerformed(ActionEvent e){
        tf.setText("Welcome to Javatpoint.");
}
}
```

Output:

2. The AdjustmentListener Interface

This interface defines the **adjustmentValueChanged( )** method that is invoked when an adjustment event occurs. Its general form is shown here:

void adjustmentValueChanged(AdjustmentEvent *ae*)

The ComponentListener Interface

This interface defines four methods that are invoked when a component is resized, moved, shown, or hidden. Their general forms are shown here:

void componentResized(ComponentEvent *ce*)

void componentMoved(ComponentEvent *ce*)

void componentShown(ComponentEvent *ce*)

void componentHidden(ComponentEvent *ce*)

The ContainerListener Interface

This interface contains two methods. When a component is added to a container, componentAdded( ) is invoked.

When a component is removed from a container, componentRemoved( ) is invoked. Their general forms are shown here:

void componentAdded(ContainerEvent *ce*)

void componentRemoved(ContainerEvent *ce*)

The FocusListener Interface

This interface defines two methods. When a component obtains keyboard focus, focusGained( ) is invoked.

When a component loses keyboard focus, focusLost( ) is called. Their general forms are shown here:

void focusGained(FocusEvent *fe*)

void focusLost(FocusEvent *fe*)

The ItemListener Interface

This interface defines the itemStateChanged( ) method that is invoked when the state of an item changes. Its general form is shown here:

void itemStateChanged(ItemEvent *ie*)

The KeyListener Interface

This interface defines three methods. The keyPressed( ) and keyReleased( ) methods are invoked when a key is pressed and released, respectively.

The keyTyped( ) method is invoked when a character has been entered.

For example, if a user presses and releases the a key, three events are generated in sequence: key pressed, typed, and released.

If a user presses and releases the home key, two key events are generated in sequence: key pressed and released.

The general forms of these methods are shown here:

void keyPressed(KeyEvent *ke*)

void keyReleased(KeyEvent *ke*)

void keyTyped(KeyEvent *ke*)

The MouseListener Interface

- **This interface defines five methods. If the mouse is pressed and released at the same point, mouseClicked( ) is invoked. When the mouse enters a component, the mouseEntered( ) method is called. When it leaves, mouseExited( ) is called. The mousePressed( ) and mouseReleased( ) methods are invoked when the mouse is pressed and released, respectively.**
- **The general forms of these methods are shown here:**

void mouseClicked(MouseEvent *me*)

void mouseEntered(MouseEvent *me*)

void mouseExited(MouseEvent *me*)

void mousePressed(MouseEvent *me*)

void mouseReleased(MouseEvent *me*)

The MouseMotionListener Interface

- **This interface defines two methods. The mouseDragged( ) method is called multiple times as the mouse is dragged. The mouseMoved( ) method is called multiple times as the mouse is moved. Their general forms are shown here:**
- **void mouseDragged(MouseEvent *me*)**
- **void mouseMoved(MouseEvent *me*)**

Java MouseListener Interface

- **The Java MouseListener is notified whenever you change the state of mouse. It is notified against MouseEvent. The MouseListener interface is found in java.awt.event package. It has five methods.**

The MouseWheelListener Interface

- **This interface defines the mouseWheelMoved( ) method that is invoked when the mouse wheel is moved. Its general form is shown here:**
- **void mouseWheelMoved(MouseWheelEvent *mwe*)**

The TextListener Interface

- **This interface defines the textValueChanged( ) method that is invoked when a change occurs in a text area or text field. Its general form is shown here:**
- **void textValueChanged(TextEvent *te*)**

The WindowFocusListener Interface

- **This interface defines two methods: windowGainedFocus( ) and windowLostFocus( ). These are called when a window gains or loses input focus. Their general forms are shown here:**
- **void windowGainedFocus(WindowEvent *we*)**
- **void windowLostFocus(WindowEvent *we*)**

The WindowListener Interface

- **This interface defines seven methods. The windowActivated( ) and windowDeactivated( ) methods are invoked when a window is activated or deactivated, respectively. If a window is iconified, the windowIconified( ) method is called. When a window is deiconified,**
- **the windowDeiconified( ) method is called. When a window is opened or closed, the windowOpened( ) or windowClosed( ) methods are called, respectively. The windowClosing( ) method is called when a window is being closed. The general forms of these methods are**

void windowActivated(WindowEvent *we*)

void windowClosed(WindowEvent *we*)

void windowClosing(WindowEvent *we*)

Java String Class Methods

- **The java.lang.String class provides a lot of built-in methods that are used to manipulate string in Java. By the help of these methods, we can perform operations on String objects such as trimming, concatenating, converting, comparing, replacing strings etc.**
- **Java String is a powerful concept because everything is treated as a String if you submit any form in window based, web based or mobile application.**
- **Let's use some important methods of String class.**

Java String toUpperCase() and toLowerCase() method

- **The Java String toUpperCase() method converts this String into uppercase letter and String toLowerCase() method into lowercase letter.**

Stringoperation1.java

```
public class Stringoperation1
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.toUpperCase());//SACHIN
System.out.println(s.toLowerCase());//sachin
System.out.println(s);//Sachin(no change in original)
}
}
```

```
SACHIN
sachin
Sachin
```

Java String trim() method

- **The String class trim() method eliminates white spaces before and after the String.**

Stringoperation2.java

```
public class Stringoperation2
{
public static void main(String ar[])
{
String s="  Sachin  ";
System.out.println(s);//  Sachin
System.out.println(s.trim());//Sachin
}
}
```

```
Sachin
Sachin
```

Java String startsWith() and endsWith() method

- **The method startsWith() checks whether the String starts with the letters passed as arguments and endsWith() method checks whether the String ends with the letters passed as arguments.**

Stringoperation3.java

```
public class Stringoperation3
{
public static void main(String ar[])
{
String s="Sachin";
 System.out.println(s.startsWith("Sa"));//true
 System.out.println(s.endsWith("n"));//true
}
}
```

```
true
true
```

Java String charAt() Method

- **The String class charAt() method returns a character at specified index.**

Stringoperation4.java

```
public class Stringoperation4
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.charAt(0));//S
System.out.println(s.charAt(3));//h
}
}
```

```
S
h
```

Java String length() Method

```
public class Stringoperation5
{
public static void main(String ar[])
{
String s="Sachin";
System.out.println(s.length());//6
}
}
```

```
6
```

Java String intern() Method

- **A pool of strings, initially empty, is maintained privately by the class String.**
- **When the intern method is invoked, if the pool already contains a String equal to this String object as determined by the equals(Object) method, then the String from the pool is returned. Otherwise, this String object is added to the pool and a reference to this String object is returned.**

Stringoperation6.java

```
public class Stringoperation6
{
public static void main(String ar[])
{
String s=new String("Sachin");
String s2=s.intern();
System.out.println(s2);//Sachin
}
}
```

```
Sachin
```

Java String valueOf() Method

- **The String class valueOf() method coverts given type such as int, long, float, double, boolean, char and char array into String.**

Stringoperation7.java

```
public class Stringoperation7
{
public static void main(String ar[])
{
int a=10;
String s=String.valueOf(a);
System.out.println(s+10);
}
}
```

```
1010
```

Java String replace() Method

- **The String class replace() method replaces all occurrence of first sequence of character with second sequence of character.**

Stringoperation8.java

```
public class Stringoperation8
{
public static void main(String ar[])
{
String s1="Java is a programming language. Java is a platform. Java is an Island.";
String replaceString=s1.replace("Java","Kava");//replaces all occurrences of "Java" to "Kava"
System.out.println(replaceString);
}
}
```

Output:

```
Kava is a programming language. Kava is a platform. Kava is an Island.
```

Applet basics and methods

Java Applet

- **Applet is a special type of program that is embedded in the webpage to generate the dynamic content.**
- **It runs inside the browser and works at client side.**

Advantage of Applet

- **There are many advantages of applet. They are as follows:**
- **It works at client side so <u>less response time</u>.**
- **Secured**
- **It can be executed by browsers <u>running under many plateforms</u>, including Linux, Windows, Mac Os etc.**

Drawback of Applet

- **Plugin is required at client browser to execute applet.**

Lifecycle of Java Applet

1. Applet is initialized.
2. Applet is started.
3. Applet is painted.
4. Applet is stopped.
5. Applet is destroyed.

Lifecycle methods for Applet:

- **The java.applet.Applet class provides 4 life cycle methods and java.awt.Component class provides 1 life cycle methods for an applet.**

java.applet.Applet class

- **For creating any applet java.applet.Applet class must be inherited. It provides 4 life cycle methods of applet.**

1. **public void init(): is used to initialized the Applet. It is invoked only once.**
2. **public void start(): is invoked after the init() method or browser is maximized. It is used to start the Applet.**
3. **public void stop(): is used to stop the Applet. It is invoked when Applet is stop or browser is minimized.**
4. **public void destroy(): is used to destroy the Applet. It is invoked only once.**

java.awt.Component class

- **The Component class provides 1 life cycle method of applet.**

1. **public void paint(Graphics g):** is used to paint the Applet. It provides Graphics class object that can be used for drawing oval, rectangle, arc etc.

Who is responsible to manage the life cycle of an applet?

- **Java Plug-in software.**

How to run an Applet?

- **There are two ways to run an applet**
    1. **By html file.**
    2. **By appletViewer tool (for testing purpose).**

Simple example of Applet by html file:

- **To execute the applet by html file, create an applet and compile it. After that create an html file and place the applet code in html file. Now click the html file.**

```java
//First.java
import java.applet.Applet;
import java.awt.Graphics;
public class First extends Applet{

public void paint(Graphics g){
g.drawString("welcome",150,150);
}

}
```

myapplet.html

```html
<html>
<body>
<applet code="First.class" width="300" height="300">
</applet>
</body>
</html>
```

Note: class must be public because its object is created by Java Plugin software that resides on the browser.

Simple example of Applet by appletviewer tool:

- **To execute the applet by appletviewer tool, create an applet that contains applet tag in comment and compile it. After that run it by: appletviewer First.java. Now Html file is not required but it is for testing purpose only.**

To execute the applet by appletviewer tool, write in command prompt:

Lab Cycle Program

Develop a java program that simulates a traffic light using AWT. The program lets the user select one of three lights: red, yellow, or green with radio buttons. On selecting a button, an appropriate message with "stop" or "ready" or "go" should appear above the buttons in a selected colour. Initially there is no message shown.

TrafficLightsExample - Notepad

File Edit Format View Help

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


public class TrafficLightsExample extends Applet implements ItemListener
{

        CheckboxGroup grp = new CheckboxGroup();
        Checkbox redLight, yellowLight, greenLight;
        Label msg;
        public void init()
               {
                redLight = new Checkbox("Red", grp, false);
                yellowLight = new Checkbox("Yellow", grp, false);
                greenLight = new Checkbox("Green", grp, false);

                msg = new Label("");

                redLight.addItemListener(this);
                yellowLight.addItemListener(this);
                greenLight.addItemListener(this);

                add(redLight);
                add(yellowLight);
                add(greenLight);
                add(msg);
                msg.setFont(new Font("Serif", Font.BOLD, 20));
        }
```

Ln 31, Col 1 | 100% | Windows (CRLF) | UTF-8

TrafficLightsExample - Notepad

File Edit Format View Help

```
        public void itemStateChanged(ItemEvent ie)
             {
                redLight.setForeground(Color.BLACK);
                yellowLight.setForeground(Color.BLACK);
                greenLight.setForeground(Color.BLACK);

                if(redLight.getState() == true)
                 {
                        redLight.setForeground(Color.RED);
                        msg.setForeground(Color.RED);
                        msg.setText("STOP");
                }
                else if(yellowLight.getState() == true)
                 {
                        yellowLight.setForeground(Color.YELLOW);
                        msg.setForeground(Color.YELLOW);
                        msg.setText("READY");
                }
                else
                 {
                        greenLight.setForeground(Color.GREEN);
                        msg.setForeground(Color.GREEN);
                        msg.setText("GO");
                }
        }
}

/*
  <applet code = "TrafficLightsExample.class" width ="1000" height="500">
  </applet>
*/
```

Ln 31, Col 1 | 100% | Windows (CRLF) | UTF-8

Lab Cycle Program

Develop a program that creates a user interface to perform integer divisions. The user enters two numbers in the text fields, Num1 and Num2. The division of Num1 and Num2 is displayed in the Result field when the Divide button is clicked. If Num1 or Num2 were not an integer, the program would throw a NumberFormatException. If Num2 were Zero, the program would throw an Arithmetic Exception Display the exception in a message dialog box.

test6 - Notepad

File Edit Format View Help

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

public class test6 extends Applet implements ActionListener

 {

  Label l1,l2,l3;
  TextField num1,num2,res;
  Button div;
  String msg;
  public void init()
   {
   l1=new Label("Dividend");
   l2=new Label("Divisor");
   l3=new Label("Result");
   num1=new TextField(10);
   num2=new TextField(10);
   res=new TextField(10);
   div=new Button("Click");
   div.addActionListener(this);
    add(l1);
    add(num1);
    add(l2);
    add(num2);
    add(l3);
    add(res);
    add(div);
   }
```

Ln 1, Col 1 | 100% | Windows (CRLF) | UTF-8

test6 - Notepad

File Edit Format View Help

```
public void actionPerformed(ActionEvent e)
  {
   String arg=e.getActionCommand();
   if(arg.equals("Click"))
     {
       if(this.num1.getText().isEmpty() || this.num2.getText().isEmpty())
          {
          msg="Enter the valid numbers";
          repaint();
         }
      else
          {
            try
              {
               int num1,num2;
               num1=Integer.parseInt(this.num1.getText());
               num2=Integer.parseInt(this.num2.getText());
               int num3=num1/num2;
               res.setText(String.valueOf(num3));
               msg="Operation Successfully";
               repaint();
              }
            catch(ArithmeticException ae)
              {
               System.out.println("Can't be devided by zero");
               msg="Can't be devided by zero";
               res.setText("");
               repaint();
              }
            catch(NumberFormatException ne)
              {
               System.out.println("NumberFormatException -Non Numeric");
               msg="NumberFormatException -Non Numeric";
               res.setText("");
               repaint();
              }
            }
       }
    }
```

Ln 72, Col 1 | 100% | Windows (CRLF) | UTF-8

test6 - Notepad

File Edit Format View Help

```
public void paint(Graphics g)
   {
    g.setColor(Color.RED);
    g.drawString(msg,30,200);
   }
}

/*

<applet code="test6.class" width="200" height="250">
</applet>

*/
```

Ln 85, Col 3 | 100% | Windows (CRLF) | UTF-8

Displaying Graphics in Applet

- **java.awt.Graphics class provides <u>many methods</u> for graphics programming.**

Commonly used methods of Graphics class:

1. **public abstract void drawString(String str, int x, int y):** is used to draw the specified string.
2. **public void drawRect(int x, int y, int width, int height):** draws a rectangle with the specified width and height.
3. **public abstract void fillRect(int x, int y, int width, int height):** is used to fill rectangle with the default color and specified width and height.
4. **public abstract void drawOval(int x, int y, int width, int height):** is used to draw oval with the specified width and height.
5. **public abstract void fillOval(int x, int y, int width, int height):** is used to fill oval with the default color and specified width and height.
6. **public abstract void drawLine(int x1, int y1, int x2, int y2):** is used to draw line between the points(x1, y1) and (x2, y2).
7. **public abstract boolean drawImage(Image img, int x, int y, ImageObserver observer):** is used draw the specified image.
8. **public abstract void drawArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used draw a circular or elliptical arc.
9. **public abstract void fillArc(int x, int y, int width, int height, int startAngle, int arcAngle):** is used to fill a circular or elliptical arc.
10. **public abstract void setColor(Color c):** is used to set the graphics current color to the specified color.
11. **public abstract void setFont(Font font):** is used to set the graphics current font to the specified font.

Example of Graphics in applet:

GraphicsDemo - Notepad

File Edit Format View Help

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo extends Applet
{
        public void paint(Graphics g)
        {

        g.setColor(Color.red);
        g.drawString("Welcome",50, 50);
        g.drawLine(20,30,20,300);
        g.drawRect(70,100,30,30);
        g.fillRect(170,100,30,30);
        g.drawOval(70,200,30,30);
        g.setColor(Color.pink);
        g.fillOval(170,200,30,30);
        g.drawArc(90,150,30,30,30,270);
        g.fillArc(270,150,30,30,0,180);
        }

}
  /*
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
*/
```

Ln 18, Col 2 | 100% | Windows (CRLF) | UTF-8

```
D:\java\programs\module-5\color>appletviewer GraphicsDemo.java
```

Java Color Codes

- **Java programming language allows us to create different types of applications like windows application or web application.**
- **The user interface is an important factor while developing an application.**
- **The GUI of the Java application can be made interactive using different colors available in Java programming.**

Java Color Constants

- **The color constants in Java are values that cannot be changed and can be used with different Java programs.**
- **The following table shows the color constants available in the Java programming.**
- **The all-capital version depicts a constant value. But lowercase version also works fine.**

Java Color Constants	
BLACK	black
BLUE	blue
CYAN	cyan
DARK_GRAY	darkGray
GRAY	gray
GREEN	green
LIGHT_GRAY	lightGray
MAGENTA	magenta
ORANGE	orange
PINK	pink
RED	red
WHITE	white
YELLOW	yellow

Java Color Codes

- **The basic colors of color system are red, green, and blue.**
- **Java provides the Color class constructor with different RGB color codes as arguments.**
- **Many developer tools are available that helps in picking up the correct RGB value.**

The following table shows some color code combinations using different RGB values.

Color	RGB value
Black	0-0-0
Very light red	255-102-102
Light red	255-51-51
Red	255-0-0
Dark red	204-0-0
Very dark red	153-0-0
Very light blue	51-204-255
Light blue	51-153-255
Blue	0-0-255
Dark blue	0-0-204
Very dark blue	0-0-153
Very light green	102-255-102
Light green	0-255-51
Green	0-204-0
Dark green	0-153-0

Green	0-204-0
Dark green	0-153-0
Very dark green	0-102-0
White	255-255-255

Java AWT Color Class Program

- **Java Abstract Window Toolkit is an API for developing GUI based Java applications. Let's create a Java program and use the Color class.**

test5 - Notepad

File Edit Format View Help

```
import java.awt.*;

class test5 extends Frame
{
    test5()
    {

        /* create an instance of Color class. */
        /* RGB value 0, 0, 204 for Blue color. */
        Color c1 = new Color(255, 102, 102);

        /* create an instance of JPanel. */
        Panel p = new Panel();

        /* Set the background of the JPanel to the specified Color. */
          p.setBackground(c1);

        setSize(200, 200);

        add(p);

        show();
    }


    public static void main(String args[])
    {
        test5 c = new test5();
    }
}
```

Example : 2

GraphicsDemo - Notepad

File Edit Format View Help

```
import java.applet.Applet;
import java.awt.*;
public class GraphicsDemo extends Applet
{
public void paint(Graphics g)
{
Color c1=new Color(255,100,100);
g.setColor(c1);
g.drawString("Welcome",50, 50);

Color c2=new Color(153,0,0);
g.setColor(c2);
g.fillOval(170,200,30,30);
g.drawLine(20,30,20,300);
Color c3=new Color(0,255,51);
g.setColor(c3);
g.fillRect(170,100,30,30);
g.drawOval(250,100,30,30);
g.setColor(Color.pink);
g.drawArc(90,150,30,30,30,270);
g.fillArc(270,150,30,30,0,180);
}

}
  /*
<applet code="GraphicsDemo.class" width="300" height="300">
</applet>
*/
```

UTF-8

AWT Control Fundamentals

The AWT supports the following types of controls:

- **Labels**
- **Push buttons**
- **Check boxes**
- **Choice lists**
- **Lists**
- **Scroll bars**
- **Text Editing**

- **These controls are subclasses of Component. Although this is not a particularly rich set of controls, it is sufficient for simple applications. (Note that both Swing and JavaFX provide a substantially larger, more sophisticated set of controls.)**

Adding and Removing Controls

- **To include a control in a window, you must add it to the window. To do this, you must first create an instance of the desired control and then add it to a window by calling add( ), which is defined by Container. The add( ) method has several forms. The following form is the one that is used for the first part of this chapter:**
- **Component add(Component *compRef*)**
- **Here, *compRef* is a reference to an instance of the control that you want to add. A reference to the object is returned. Once a control has been added, it will automatically be visible whenever its parent window is displayed.**
- **Sometimes you will want to remove a control from a window when the control is no longer needed. To do this, call remove( ). This method is also defined by Container. Here is one of its forms:**
- **void remove(Component *compRef*)**
- **Here, *compRef* is a reference to the control you want to remove. You can remove all controls by calling removeAll( ).**

Responding to Controls

- **Except for labels, which are passive, all other controls generate events when they are accessed by the user. For example, when the user clicks on a push button, an event is sent that identifies the push button. In general, your program simply implements the appropriate interface and then registers an event listener for each control that you need to monitor. As explained in Chapter 24, once a listener has been installed, events are automatically sent to it. In the sections that follow, the appropriate interface for each control is specified.**

The HeadlessException

- Most of the AWT controls described in this chapter have constructors that can throw a **HeadlessException** when an attempt is made to instantiate a GUI component in a non-interactive environment (such as one in which no display, mouse, or keyboard is present). You can use this exception to write code that can adapt to non-interactive environments. (Of course, this is not always possible.) This exception is not handled by the programs in this chapter because an interactive environment is required to demonstrate the AWT controls.

Unified Modeling Language (UML)

- UML (Unified Modeling Language) is a **general-purpose**, **graphical modeling language in the field of Software Engineering.**
- **It helps in designing** and **characterizing**, **especially those software systems that include the concept of Object orientation**.
- **It describes the working of both the software** and **hardware systems**.
- The UML was developed in 1994-95 by Grady Booch, Ivar Jacobson, and James Rumbaugh at the Rational Software. In 1997, it got adopted as a standard by the Object Management Group (OMG).

Goals of UML

- Since **it is a general-purpose modeling language**, **it can be utilized by all the modelers**.
- **UML came into existence after the introduction of object-oriented concepts to systemize** and **consolidate the object-oriented development**, **due to the absence of standard methods at that time**.
- The **UML diagrams are made for business users**, **developers**, **ordinary people**, or **anyone who is looking forward to understand the system**, such that the system can be software or non-software.
- Thus it can be concluded that the **UML is a simple modeling approach that is used to model all the practical systems**.

Characteristics of UML

- **The UML has the following features:**
- **It is a generalized modeling language.**
- **It is distinct from other programming languages like C++, Python, etc.**
- **It is interrelated to object-oriented analysis and design.**
- **It is used to visualize the workflow of the system.**
- **It is a pictorial language, used to generate powerful modeling artifacts.**

UML-Diagrams

- The **UML diagrams are categorized into structural diagrams, behavioral diagrams,** and **also interaction overview diagrams.**
- The diagrams are hierarchically classified in the following figure:

1. Structural Diagrams

- **Structural diagrams describe** a **static view or structure of a system**.
- **It is widely used in the documentation of software architecture**.
- **It consists of** class diagrams, composite structure diagrams, component diagrams, deployment diagrams, object diagrams, and package diagrams.
- **It presents an <u>outline for</u> the system**.
- It stresses the **elements to be present that are to be modeled**.

Class Diagram:

- Class diagrams are **one of the most widely used diagrams**.
- **It is the backbone of all the object-oriented software systems**.
- **It depicts the static structure of the system**.
- **It displays the system's class**, **attributes**, and **methods**.
- **It is helpful in recognizing the relation between different objects as well as classes**.

Eg:

- A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**).
- Each attribute has a type.
- Each **operation** has a **signature**.
- *The class name is the **only mandatory information***.

Eg:2

- **Composite Structure Diagram:** The composite structure diagrams show parts within the class. It displays the relationship between the parts and their configuration that ascertain the behavior of the class. It makes full use of ports, parts, and connectors to portray the internal structure of a structured classifier. It is similar to class diagrams, just the fact it represents individual parts in a detailed manner when compared with class diagrams.
- **Object Diagram:** It describes the static structure of a system at a particular point in time. It can be used to test the accuracy of class diagrams. It represents distinct instances of classes and the relationship between them at a time.
- **Component Diagram:** It portrays the organization of the physical components within the system. It is used for modeling execution details. It determines whether the desired functional requirements have been considered by the planned development or not, as it depicts the structural relationships between the elements of a software system.
- **Deployment Diagram:** It presents the system's software and its hardware by telling what the existing physical components are and what software components are running on them. It produces information about system software. It is incorporated whenever software is used, distributed, or deployed across multiple machines with dissimilar configurations.
- **Package Diagram:** It is used to illustrate how the packages and their elements are organized. It shows the dependencies between distinct packages. It manages UML diagrams by making it easily understandable. It is used for organizing the class and use case diagrams.

2. Behavioral Diagrams:

- **Behavioral diagrams describe** a **dynamic view of a system** or **the behavior of a system**, **which describes the functioning of the system**.
- **It includes use case diagrams**, **state diagrams**, and **activity diagrams**.
- **It defines the interaction <u>within</u> the system**.

Use Case Diagram:

- It represents the **functionality of a system by utilizing actors** and **use cases**.
- **It encapsulates the functional requirement of a system** and **its association with actors.**
- **It describes the use case view of a system**.

Eg:2

- **State Machine Diagram:** It is a behavioral diagram. it portrays the system's behavior utilizing finite state transitions. It is also known as the **State-charts** diagram. It models the dynamic behavior of a class in response to external stimuli.
- **Activity Diagram:** It models the flow of control from one activity to the other. With the help of an activity diagram, we can model sequential and concurrent activities. It visually depicts the workflow as well as what causes an event to occur.

3. Interaction Diagrams

- **Interaction diagrams are a subclass of behavioral diagrams that give importance to object interactions** and **also depicts the flow between various use case elements of a system**.
- In simple words, it shows **how objects interact with each other** and **how the data flows within them**.
- **It consists of** communication, interaction overview, sequence, and timing diagrams.
- **Sequence Diagram:** It shows the interactions between the objects in terms of messages exchanged over time. It delineates in what order and how the object functions are in a system.
- **Communication Diagram:** It shows the interchange of sequence messages between the objects. It focuses on objects and their relations. It describes the static and dynamic behavior of a system.
- **Timing Diagram:** It is a special kind of sequence diagram used to depict the object's behavior over a specific period of time. It governs the change in state and object behavior by showing the time and duration constraints.
- **Interaction Overview diagram:** It is a mixture of activity and sequence diagram that depicts a sequence of actions to simplify the complex interactions into simple interactions.

UML Class Diagram

- Class diagrams are **one of the most widely used diagrams**.
- **It is the backbone of all the object-oriented software systems**.
- **It depicts the static structure of the system**.
- **It displays the system's class**, **attributes**, and **methods**.
- **It is helpful in recognizing the relation between different objects as well as classes**.

components of a Class Diagram

- The class diagram is made up of three sections:
    - **Upper Section:**
    - **Middle Section:**
    - **Lower Section:**

Eg:

- A class represent a concept which encapsulates state (**attributes**) and behavior (**operations**).
- Each attribute has a type.
- Each **operation** has a **signature**.
- *The class name is the **only mandatory information**.*

Class Visibility

- The +, - and # **symbols before an attribute** and **operation name in a class denote the visibility of the attribute** and **operation**.

- + denotes **public attributes** or **operations**
- - denotes **private attributes** or **operations**
- # denotes **protected attributes** or **operations**

Eg:

For example, a Person class and a Book class might be modeled like this.

Relationships between classes

- UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?
- If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

Inheritance (or Generalization):

- A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.
- Represents an "is-a" relationship.
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of SuperClass.
- The figure below shows an example of inheritance hierarchy. SubClass1 and SubClass2 are derived from SuperClass. The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element.

Inheritance Example - Shapes

- The figure below shows an inheritance example with two styles. Although the connectors are drawn differently, they are semantically equivalent.

Simple Association

- A **structural link between two peer classes**.
- There is an association between Class1 and Class2
- The figure below shows an example of simple association. There is an association that connects the <<control>> class Class1 and <<boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.

Cardinality

Cardinality is expressed in terms of:

- **one to one**
- **one to many**
- **many to many**

Aggregation

- A special type of association.
- It represents a **"part of"** relationship.
- **Class2** is **part of Class1**.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.
- The figure below shows an example of aggregation. The relationship is displayed as a solid line with a unfilled diamond at the association end, which is connected to the class that represents the aggregate.

Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of **Class2 live and die with Class1**.
- **Class2 cannot stand by itself**.
- The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.

Dependency

- An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.
- A special type of association.
- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- **Class1 depends on Class2**
- The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow.

- The figure below shows another example of dependency. The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).

Realization

- Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.
- For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways.

Eg:2

Vehicle UML Diagram

Class diagram for an ATM system

- ATMs are deceptively simple: although customers only need to press a few buttons to receive cash, there are many layers of security that a safe and effective ATM must pass through to prevent fraud and provide value for banking customers. The various human and inanimate parts of an ATM system are illustrated by this easy-to-read diagram—every class has its title, and the attributes are listed beneath. You can edit, save, and share this chart by opening the document and signing up for a free Lucidchart account.

Class Diagram Example: Order System

Class diagram for a hotel management system

- A class diagram can show the relationships between each object in a hotel management system, including guest information, staff responsibilities, and room occupancy. The example below provides a useful overview of the hotel management system. Get started on a class diagram by clicking the template below.

Class Diagram: Exercise & Solution

Question 1:

A company consists of departments. Departments are located in one or more offices. One office acts as a headquarter. Each department has a manager who is recruited from the set of employees. Your task is to model the system for the company.

Task:

Draw a class diagram which consists of all the classes in your system their attributes and operations, relationships between the classes, multiplicity specifications, and other model elements that you find appropriate.

Class diagrams for railway reservation system

2. Behavioral Diagrams:

- **Behavioral diagrams describe** a **dynamic view of a system** or **the behavior of a system**, **which describes the functioning of the system**.
- **It includes use case diagrams**, **state diagrams**, and **activity diagrams**.
- **It defines the interaction <u>within</u> the system**.

Use Case Diagram:

- It represents the **functionality of a system by utilizing actors** and **use cases**.
- **It encapsulates the functional requirement of a system** and **its association with actors.**
- **It describes the use case view of a system**.

Eg:1

Eg:2 Use case diagram for student attendance management system

Eg:3

Use Case Diagram for Online Shopping Website
Login
ADD Category
ADD Item
Manage Item
Manage Order
Registration
View Item
Make Order
Make Payment
Change Password
ADMIN
USER

Eg:4

Use Case Diagram - Payroll System
ADD Category/Class
ADD Employee
Apply Leave
Approve / Reject Leave
Generate Salary
My Account
Admin Login
Change Password
Leave Report
Salary Report
Employee Report
ADMIN
Employee
www.meeraacademy.com

Eg:5

Use Case Diagram - Job Portal System
Registration
Login
Search
Apply Job
Like Job List
Feed Back
Profile Match
Manage Account
Post job
Manage Job Active / Inactive
Send / Receive Message
Change Password
Student
Company
Admin
www.meeraacademy.com

Eg:6 Use case diagrams for railway reservation system

Eg: 7

UML Tools

- Since UML is a Unified Modeling Language, it is used to create meaningful, object-oriented models for a software application. It clearly represents the working of any hardware/ software system. There are numerous tools, both commercial and open-source, which are available for designing UML diagrams, are enlisted below:

1. StarUML
2. Umbrello
3. UML designer tool
4. Altova
5. Umple
6. Visual Paradigm
7. WhitestarUML