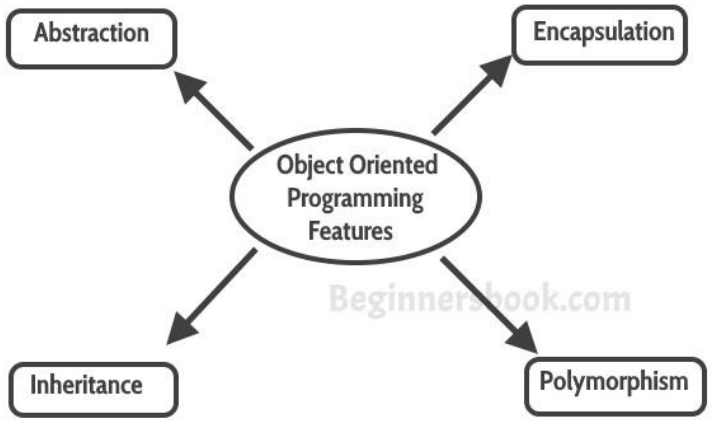
JAVA Unit 1

1. What are OOPs Concepts in Java

- OOPs concepts include the following object-oriented programming concepts:

- Object
- Class
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism



1. Object

- An object can be represented as an entity that has state and behavior. For example: A car is an object that has states such as color, model, price and behavior such as speed, start, gear change, stop etc.

- Let's understand the **difference between state and behaviour.** The state of an object is a data item that can be represented in value such as price of car, color, consider them as variables in programming.
- The behaviour is like a method of the class, it is a group of actions that together can perform a task. For example, gear change is a behaviour as it involves multiple subtasks such as speed control, clutch, gear handle movement.
  Let's take few more examples of Objects:

Examples of states and behaviors
Example 1:
Class: House
State: address, color, area
Behaviour: Open door, close door

Let's see how can we write these state and behaviours in a java program. States can be represented as instance variables and behaviours as methods of the class.

```
class House {
   String address;
   String color;
   double area;
   void openDoor() {
      //Write code here
   }
   void closeDoor() {
      //Write code here
   }
 ...
 ...
}
```

Example 2:
Class: Car
State: color, brand, weight, model
Behaviour: Break, Accelerate, Slow Down, Gear change.

Note: As we have seen in the above example, the states and behaviours of an object can be represented by variables and methods in the class.

2. Class

- A class can be considered as a **blueprint** which **you can use to create as many objects as you like**. For example, here we have a class `Website` that has two data members.
- This is just a blueprint, it does not represent any website, however using this we can create Website objects that represents the websites. We have created two objects, while creating objects we provided separate properties to the objects using constructor.

```
class Student{
 //defining fields
 int id;//field or data member or instance variable
 String name;
 //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

Output:

```
0
null
```

3. Abstraction

- Abstraction is a process where you show only “relevant” data and “hide” unnecessary details of an object from the user.
- For example, when you login to your bank account online, you enter your user_id and password and press login, what happens when you press login, how the input data sent to server, how it gets verified is all abstracted away from you. Read more about it here: Abstraction in Java.

Abstract Class Example:

- Here we have an abstract class Animal that has an abstract method `animal Sound()`, since the animal sound differs from one animal to another, there is no point in giving the implementation to this method as every child class must override this method to give its own implementation details. That's why we made it abstract.
- Now each animal must have a sound, by making this method abstract we made it compulsory to the child class to give implementation details to this method. This way we ensures that every animal has a sound.

```
//abstract class
abstract class Animal{
   //abstract method
   public abstract void animalSound();
}
public class Dog extends Animal{

   public void animalSound(){
        System.out.println("Woof");
   }
   public static void main(String args[]){
        Animal obj = new Dog();
        obj.animalSound();
   }
}
```

Output:

```
Woof
```

4. Encapsulation

Encapsulation simply means binding object state(fields) and behaviour(methods) together. If you are creating class, you are doing encapsulation.

Example

1) Make the instance variables private so that they cannot be accessed directly from outside the class. You can only set and get values of these variables through the methods of the class.
2) 2) Have getter and setter methods in the class to set and get the values of the fields.

```
class EmployeeCount
{
   private int numOfEmployees = 0;
   public void setNoOfEmployees (int count)
   {
       numOfEmployees = count;
   }
   public double getNoOfEmployees ()
   {
       return numOfEmployees;
   }
}
public class EncapsulationExample
{
   public static void main(String args[])
   {
      EmployeeCount obj = new EmployeeCount ();
      obj.setNoOfEmployees(5613);
      System.out.println("No Of Employees: "+(int)obj.getNoOfEmployees());
    }
}
```

Output:

```
No Of Employees: 5613
```

The class `EncapsulationExample` that is using the Object of class `EmployeeCount` will not able to get the NoOfEmployees directly. It has to use the setter and getter methods of the same class to set and get the value.

What is the benefit of using encapsulation in java programming?
Well, at some point of time, if you want to change the implementation details of the class EmployeeCount, you can freely do so without affecting the classes that are using it.

5. Inheritance

The process by which one class acquires the properties and functionalities of another class is called inheritance. Inheritance provides the idea of reusability of code and each sub class defines only those features that are unique to it, rest of the features can be inherited from the parent class.

1. Inheritance is a process of defining a new class based on an existing class by extending its common data members and methods.
2. Inheritance allows us to reuse of code, it improves reusability in your java application.

3. The parent class is called the **base class** or **super class**. The child class that extends the base class is called the derived class or **sub class** or **child class**.

Note: The biggest advantage of Inheritance is that the code in base class need not be rewritten in the child class.
The **variables** and **methods** of the base class can be used in the **child class** as well.

Syntax: Inheritance in Java:

To inherit a class we use extends keyword. Here class A is child class and class B is parent class.

```
class A extends B
{
}
```

Inheritance Example

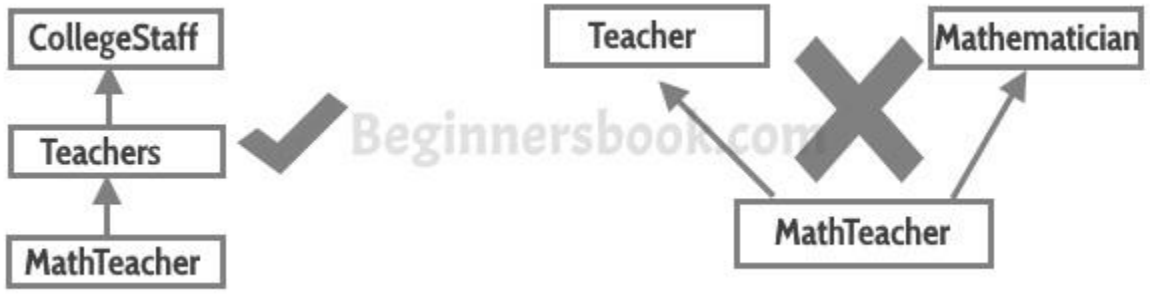
- In this example, we have a parent class `Teacher` and a child class `MathTeacher`. In the `MathTeacher` class we need not to write the same code which is already present in the present class.
- Here we have college name, designation and does() method that is common for all the teachers, thus MathTeacher class does not need to write this code, the common data members and methods can inherited from the `Teacher` class.

```
class Teacher {
   String designation = "Teacher";
   String college = "Acharya";
   void does(){
        System.out.println("Teaching");
   }
}
public class MathTeacher extends Teacher{
   String mainSubject = "Maths";
   public static void main(String args[]){
      MathTeacher obj = new MathTeacher();
      System.out.println(obj.college);
      System.out.println(obj.designation);
      System.out.println(obj.mainSubject);
      obj.does();
   }
}
```

Output:

```
Acharya
Teacher
Maths
Teaching
```

Note: Multi-level inheritance is allowed in Java but **multiple inheritances** not allowed as shown in the following diagram.



Types of Inheritance:

Single Inheritance: refers to a child and parent class relationship where a class extends the another class.

Multilevel inheritance: refers to a child and parent class relationship where a class extends the child class. For example class A extends class B and class B extends class C.

Hierarchical inheritance: refers to a child and parent class relationship where more than one classes extends the same class. For example, class B extends class A and class C extends class A.

Multiple Inheritance: refers to the concept of one class extending more than one classes, which means a child class has two parent classes. Java doesn't support multiple inheritance, read more about it here.

Most of the new **OO languages** like Small Talk, Java, C# do not support Multiple inheritance. Multiple Inheritance is supported in C++.

6. Polymorphism

- Polymorphism is a object oriented programming feature that allows us to perform a single action in different ways.
- For example, let's say we have a class `Animal` that has a method `animalSound()`, here we cannot give implementation to this method as we do not know which Animal class would extend `Animal` class. So, we make this method abstract like this:

```
public abstract class Animal{
   ...
   public abstract void animalSound();
}
```

Now suppose we have two Animal classes `Dog` and `Lion` that extends `Animal` class. We can provide the implementation detail there.

```
public class Lion extends Animal{
...
    @Override
    public void animalSound(){
        System.out.println("Roar");
    }
}
```

and

```
public class Dog extends Animal{
...
    @Override
    public void animalSound(){
        System.out.println("Woof");
    }
}
```

2. Procedural Programming vs Object-Oriented Programming

- Below are some of the differences between procedural and object-oriented programming:

Procedural Oriented Programming	Object-Oriented Programming
In procedural programming, the program is divided into small parts called ***functions***.	In object-oriented programming, the program is divided into small parts called ***objects***.
Procedural programming follows a ***top-down approach***.	Object-oriented programming follows a ***bottom-up approach***.
There is no access specifier in procedural programming.	Object-oriented programming has access specifiers like private, public, protected, etc.
Adding new data and functions is not easy.	Adding new data and function is easy.
Procedural programming does not have any proper way of hiding data so it is ***less secure***.	Object-oriented programming provides data hiding so it is ***more secure***.
In procedural programming, overloading is not possible.	Overloading is possible in object-oriented programming.
In procedural programming, there is no concept of data hiding and inheritance.	In object-oriented programming, the concept of data hiding and inheritance is used.
In procedural programming, the function is more important than the data.	In object-oriented programming, data is more important than function.

Procedural Oriented Programming	Object-Oriented Programming
Procedural programming is based on the ***unreal world***.	Object-oriented programming is based on the ***real world***.
Procedural programming is used for designing medium-sized programs.	Object-oriented programming is used for designing large and complex programs.
Procedural programming uses the concept of procedure abstraction.	Object-oriented programming uses the concept of data abstraction.
Code reusability absent in procedural programming,	Code reusability present in object-oriented programming.
Examples: C, FORTRAN, Pascal, Basic, etc.	**Examples:** C++, Java, Python, C#, etc.

3. Principles of Object-Oriented Programming

- Object-Oriented Principles mainly include the 4 pillars that together make the OOP a very powerful concept. That is –

1. *Abstraction*
2. *Encapsulation*
3. *Inheritance*
4. *Polymorphism*

- OOP is based on real-life engineered things. Like the approaches followed in the other engineering – Electronic Engineers are making some device, Automobile engineers are making some vehicle, etc. So the design approaches followed their, the same replicated to the software engineering with the concept of these 4 Object-Oriented Programming principles.

- If we analyze the programming concept then we will find that there are two elements of programming –

    - *Data*
    - *Operations on Data [functions]*

- So If we develop software then that software is meant for performing tasks on data only. And if any function is performed on data, then there might be chances that in the future also the same operation can be performed either in the same way or in some modified way. So that is very smoothly done using the concept of OOP principles. Let's see the principles:

Abstraction

- Abstraction can be defined as hiding internal implementation and showing only the required features or set of services that are offered. This is the most essential part of Object-Oriented programming.Let's understand with the help of an example –

- **Example 1 –** We all have been using ATMs. And if we want to withdraw the money from that then we simply interact with the Use Interface Screen where we have been asking for details.
  As a user, we only want that I can use that ATM for the services. We are not interested in knowing the internal implementation that what is the process happening inside the ATM Machine like getting card details, validation, etc kind of things. We only have been provided an interface to interact with that and simply ask for the work. And the internal implementation we don't need to know. So that is the Abstraction.

Explanation-

- The above image shows the GUI screen of the ATM. In this when we want to withdraw then we simply use that. For the implementation part, we don't care.
- The same concept is applicable in the programming implementation of the Abstraction. Although different programming has a different syntax of implementation of Abstraction. But the most popular programming languages like JAVA use a keyword called abstract which applies to the class and methods to make a class abstract and achieve abstraction. And it can also be achieved using the interfaces in java.
- C++ achieves it by grouping the attributes and using an access specifier allowing what data is to be visible outside.

- Access Specifiers are the keywords that state, from where the (attributes and methods) of a class can be accessed. For example – private can only be accessible within the class, the public can be accessed from anywhere, etc.
- Similarly, other programming languages follow their own implementation for achieving abstraction.

```
class Television{
   void changeChannel(){
      //Implementation
   }
   void adjustVolume(){
      //Implementation
   }
   void otherFeatues(){
      //Implementation
   }
}
```

- This pseudo-code example states that the television class is providing features like channel change, adjust volume, etc. so as a user we only need to use that functions. We need not know the internal implementation like how the function is working for changing channels, how the codes manage to adjust the volume, etc. We have been provided the method to do that task, So just utilize that.If we consider an example more specific to programming then we can say that the functions included in the Math library like – pow(), min(), max(), etc are there, and we are only using that to get our result. It's not the case where we check its implementation. So it is also an abstraction.

Features of Abstraction –

1. Security- With Abstraction, the Outside persons don't know the internal implementation so it makes the classes more secure, so the unauthorized user will not have access to the data.
2. Easy Enhancement- Suppose in the future we want to change the logic of the implementation in class, So we don't have to change the entire external logic which is

there for the user. Just we need to make changes for the methods and it will not affect the functionality.

3. Improves Easiness- It helps to use the features without knowing implementation so it improves easiness for the users to use that.
4. Maintainability will improve- Without affecting the user, it can able to perform any types of changes internally. SO maintainability will improve.

• Encapsulation

- Encapsulation can be defined as the binding of data and attributes or methods and data members in a single unit. In classes, we have Data and attributes that perform operations on that data. So according to the OOPs principle of Encapsulation, that data can be merged into a single unit. Encapsulation enhances more security of the data as everything related to a single task must be grouped and access to the data is given as per need.
- And this can be achieved using the concept of **Data Hiding.**

Encapsulation = Data Hiding + Abstraction.

- **Data Hiding** – It means hiding the data of the class and restricting access to the outside world. **Example** – Using the access specifier keywords like **private** that restricts the data to only accessible and modifiable in the same class. Outside users cannot access the data.
- Let's understand with the help of an example-

Example 1- In the market, Capsules are available to cure different medical problems. Consider capsule for fever. Now in that capsules, there are different compositions are grouped to make a complete medicine that cures fever. So the grouping of that compositions into a single unit in the capsule is a form of encapsulation. Here, consider fever as data, and the capsule as the operations on the data. So everything is grouped here.

Explanation-

- In the above image, all the dots represent the compositions for curing the fever (as per the example). And these are grouped into a capsule. Similarly in programming, when there are multiple operations related to a particular data are present, then grouping that into a simple unit is Encapsulation.

Example 2- In a car, we have all the required things to make a car complete. Like engines, gear shift, Wheels, etc, and these are grouped into a single body that makes a car. It is not like the wheels are not attached to the car, Engines are misplaced and dragged outside, etc. Everything is grouped into a single body. So in the OOP Principle also the same principle applies that whatever operations are there for a particular group, must be capsuled to a single unit.

Explanation-

- In the image we can see that the car has an engine, steering to control, gear shift, paddles to race and stop the car are present, and these are grouped into a complete car.
  So we want the same things in programming also, And that's what OOP Principle makes it possible. Similarly, if we take an example more specific to programming then we can state an example of a LinkedList. We create a node that contains data and this node is related to the LinkedList class. And Operation on that node like – Add, Delete, Search, etc are grouped into a single class.

```
class LinkedList{
    private class Node{
        data;
        next pointer;
    }
    public createNode(){}
    public addNode(){}
    public deleteNode(){}
    .
    .
}
```

- If any components follow Data Hiding and Abstraction, such type of component is said to be encapsulated component.

Features of Encapsulation –

- It provides extra security by the concept of Data Hiding.
- It groups the data and operation in that data into a single unit.
- It attaches an extra security layer to the data and allows to access data only to authorized ones.

• Inheritance:

- Inheritance is the method of acquiring features of the existing class into the new class. Suppose there is a class, considered as the parent class, that has some methods associated with it. And we declare a new class, considered as the child class, that has its own methods.

- So, when a child class is inherited from the parent class then it will have all the methods associated with parent class are available in the child class along with the child class own methods also. So that is Inheritance.

Example 1 – We all have seen the updates which receive on gadgets both in terms of hardware and software. Like firstly mobile phones are their only to talk and then used for media access, Internet, Camera, etc. So these are evolution that arrives by adding some extra feature and making a new product without rebuilding the complete functionality so it is an example of inheritance. Similarly, in software, regular updates are with some new features are also the inheritance.

Explanation-

- In the above image, we see that from old mobile new mobile is derived by adding new features without affecting the old functionality, so it is an inheritance.

Note- This is just an example of what inheritance is. In OOPs it has some differences.

- **Now, In terms of Object-Oriented Programming Inheritance can be stated as –**
- There are two keywords involved in the inheritance – Base and Derived Class.
- **Base Class –** It is also termed the parent class. It is the main class that has the basic properties and methods which is defined.

- **Derived Class –** This is the extension of the base class and it is also called the child class. It has the properties and methods that are in the base class with its own features in it.

Inheritance has multiple types that depend on programming languages implementation. Some of the common types of inheritance are-

1. **Single Inheritance** – When there is only one derived class.
2. **Multiple Inheritance** – When child class is inheriting more than one base class.
3. **Multilevel Inheritance** – When there is a level of inheritance. From class A to class B to class C.
4. **Hierarchical Inheritance** – When more than one derived class is inheriting one base class.

Programming Implementation of Inheritance-

```
class Base{
   data;
   feature1(){}
   feature2(){}
}
class Derived extends Base{
   feature3(){}
}
```

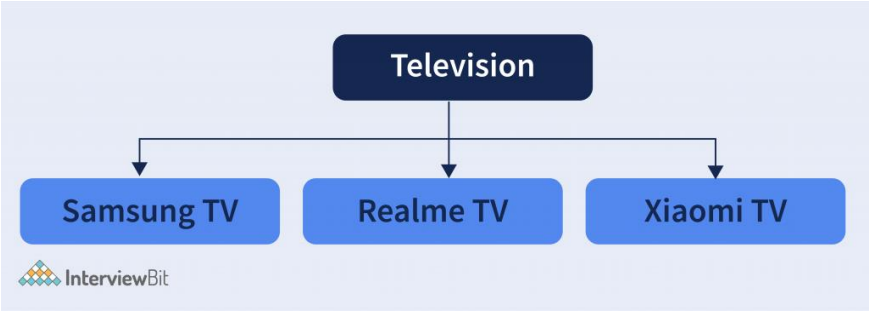
- In the above snippet of code there is a base class that has some data and features. And now the derived class is implementing the base class which has an additional feature. So overall derived class have all the feature which are in the base class with its own feature. So this is inheritance.

Derived class = feature1 (from base class),
feature2 (from base class),
data (from base class), and
feature3 (it's own).

Note- Different programming languages used their different keyword for inheritance. Like java uses **extends** keyword, c++ uses **:** (colon to inherit), Python uses **()** Parenthesis, etc.

The biggest advantage of inheritance is code reusability. The methods which are already declared in the base class need not be rewritten in the derived class, They can be directly used. It is automatically available for use on the derived class.

- **Polymorphism**
  - Polymorphism is the most essential concept of the Object-Oriented Programming principle. It has meaning '**poly**' – many, '**morph**' – forms. So polymorphism means many forms.
  - In Object-Oriented Programming, any object or method has more than one name associated with it. That is nothing but polymorphism.
  - **For Objects –** When any object can hold the reference of its parent object then it is a polymorphism. To understand this concept more clear, let's consider an example-
  - **Example –** We have Television from different companies, Let's say Samsung TV, LG TV, Xiaomi TV, etc. Now if we have to call it then mostly we don't call it with its brand name. We simply call it a Television, And it can be of a different brand. But in general, we call it television. So this can be said to be polymorphism.



Explanation-

- In the above image we have given these different brands of Television a common name which we call **Television.** Here the term Polymorphism is followed.
- It can also be stated as the **Generalization** because we have given a general name that can apply to all the subclasses it has been derived from.
  And yes, It can be achieved using inheritance. And the derived class object can also be called with the name of the base class.

Example-

```
class Television{
public void TVOn(){}
public void TVOff(){}
}
class SamsungTV extends Television{
public void surfInternet(){}
}
```

```
Television TV = new SamsungTV(); //Yes, It is absolutely correct.
```

Explanation – In the above snippet the SamsungTV implemented Television class and method **TVOn()** and **TVOff()** is available in SamsungTV class.

- Now the object we have created of reference of Television class and created the object of the SamsungTV class. Now, if we call TV.TVOn() then this works perfectly well. So it is the way to achieve Polymorphism.
- Because here it is polymorphism. And we can say that the SamsungTV is a Television. Because the Television, as it has all the methods related to that. If we call Television.on() then the SamsungTV will on as we have objected to that.

For Methods – Methods can also have multiple names associated with them. That we also consider as the types of polymorphism.

OOPs states 2 types of Polymorphism –

1. Compile Time Polymorphism. Also called Static binding.
2. Runtime Polymorphism. Also called Dynamic binding.

Compile Time Polymorphism is achieved using Method **Overloading**.

- **Method Overloading –** When more than one method is declared with the same name but with a different number of parameters or different data-type of parameter, that is called method overloading.
- **Example –**

```
class Base{
    public void method1(int x){ }
}
class Derived extends base{
    //Overloaded Method
    public void method1(float y){ }
}
```

- In the above snippet their is method1 declared twice but both have different signature. One is Accepting integer value as a parameter and the other is accepting the float value as a parameter.
- It is called Static Binding because, during the compilation time itself, it's been clear which method to call. It is recognized with the help of Data Type and during the compilation time of the program.

Similarly, Runtime polymorphism is achieved using **Method Overriding.**

- **Method Overriding –** When more than one method is declared as the same name and same signature but in a different class. Then the derived class method will override the base class method. This means the Base class method will be shadowed by the derived class method. And when the method is called then it can be called based on the overridden method.
- **Example –**

```
class Base{
    public void method1(int x){ } // Shadowed Method
}
class Derived extends base
    public void method1(int y){ }//Overriden Method
}
```

- In the above snippet, method1 is overridden in derived class. And when the object of the derived class will be created and method1 will be called then the Derived class method1 will be called. The Base class method1 will be shadowed.
- It is called Dynamic binding because, when the object gets the memory during runtime of the program and based on the object the overridden method will be called. So it happens during the execution of the program.

4. Benefits of OOPS:

- OOPs concept in Java offers several advantages that are not available in procedural programming like C, Pascal, etc. Some of the major benefits of object-oriented programming in java are as follows:

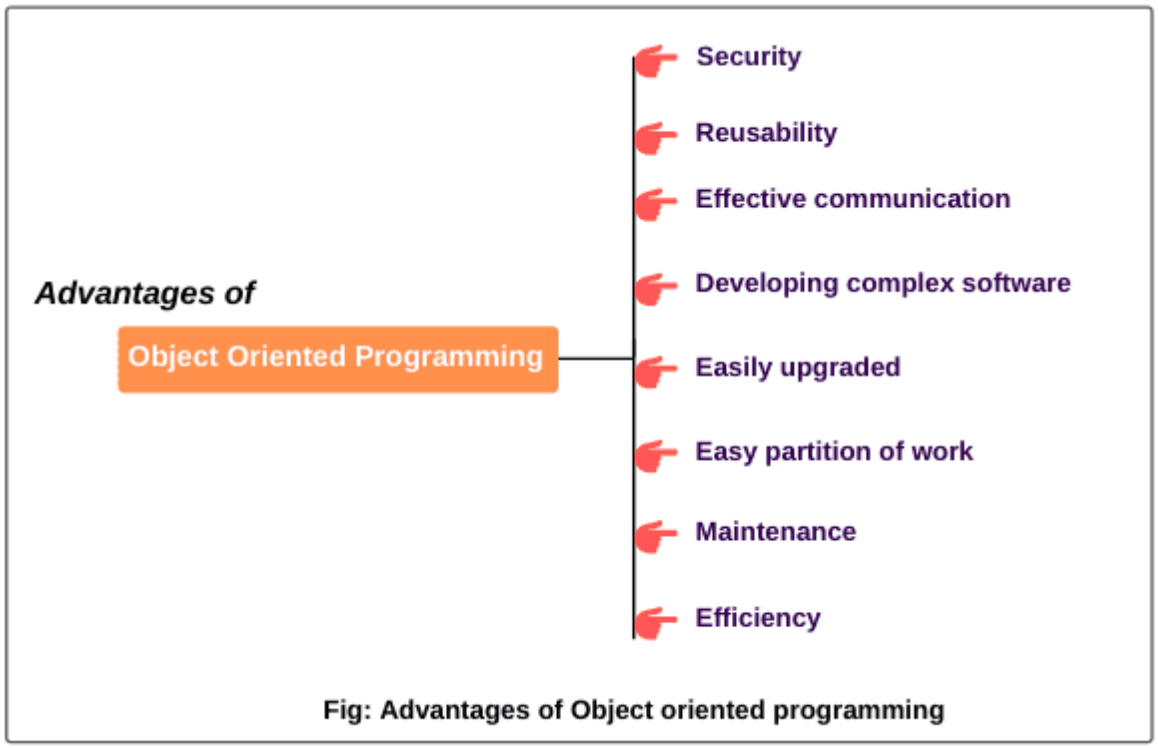


Fig: Advantages of Object oriented programming

***Security*:** In OOP, Data is encapsulated with methods in the class so that data is protected and secured from accidental modification by other external non-member methods.

Reusability: Through inheritance, we can use the features of an existing class in a new class without repeating existing code that saves a lot of time for developers, and also increases productivity.

Effective communication: In OOP, objects can communicate via message passing technique that makes interface descriptions with outside systems much simpler.

***Developing complex software*:** OOPs is the most suitable approach for developing complex software because it minimizes the complexity through the feature of inheritance.

Complex software Object-oriented system can be easily upgraded from small to large systems because OOP uses bottom-up approach.

Complex software it is easy to partition complicated work in a project based on objects.

Complex software the maintenance of object-oriented code is easier.

Complex software the concepts of OOP provide better efficiency and an easy development process.

5. Application Of OOPS:

- Now we have a basic idea of what object-oriented programming means, now let's look at some of the applications of OOPs.

1. Real Time Systems

- The term "real-time system" refers to any information processing system with hardware and software components that perform real-time application functions and can respond to events within predictable and specific time constraints.
- For real-time computing, timeliness and time synchronization are the two requirements. Timeliness means the ability to produce the expected result by a specific deadline and time synchronization means the capability of agents to coordinate independent clocks and operate together in unison.
- Using Object-oriented technology, we can develop real-time systems, this will offer adaptability, ease of modifications, reusability for the code. There is a lot of complexity involved in designing real-time systems, OOP techniques make it easier to handle those complexities.

2. Client Server System

- The client-server systems are those that involve a relationship between cooperating programs in an application. In general, the clients will initiate requests for services and the servers will

provide that functionality. The client and server either reside in the same system or communicate with each other through a computer network or the internet.

- The concepts of OOPs are quite useful when designing client-server systems. Object-oriented client-server systems are used to provide the IT Infrastructure that creates Object-oriented server internet or the OCSI applications. The infrastructure refers to the operating systems, networks, and hardware. OCSI contains the following main technologies

- The Client Server
- Object Oriented Programming
- The Internet

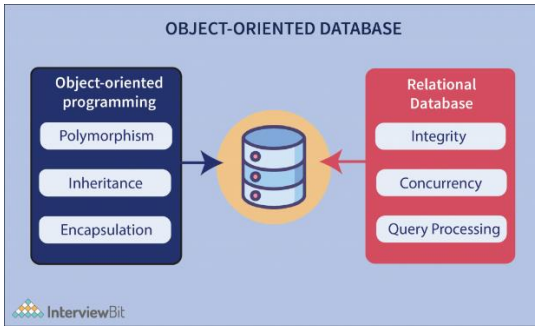
3. Hypertext and Hypermedia

- Hypertext is non-linear, multi-sequential, a cross-referencing tool that connects the links to other texts, Example of Hypertext is that InterviewBit, when we read one article it uses hypertext to link other pages and when we click on that hypertext it takes to us to that page so that we can gather more information related to the topic.

- Hypermedia is the extension to hypertext including multiple forms of media like graphics, text, audio, and video, etc. An example of hypermedia is that when we use an e-commerce site say Amazon and when we click on any product it takes us to the specific product page which belongs to that. So here the link is embedded in the image.OOP also helps in laying the framework for hypertext and hypermedia

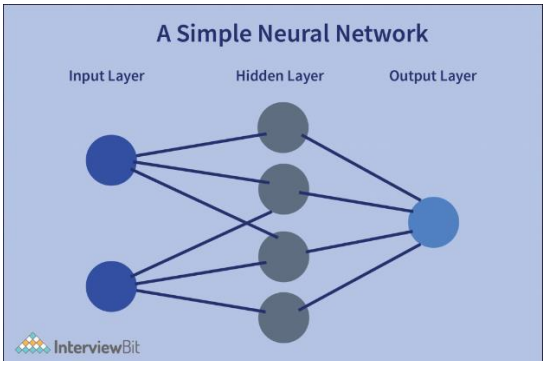
4. Object Oriented Database

- Nowadays each and every data is being stored and processed, the traditional model of storing data i.e the relational model stores each and every piece of data in tables that consist of rows and columns. However as complexity grows, storing in the form of tables becomes quite cumbersome, here the need for storing in the form of real-world objects comes into the picture.



- These databases try to maintain a direct correspondence between the real-world and database objects in order to let the object retain its identity and integrity. They can then be identified and operated upon.
- A popular example of object-oriented databases is MongoDB.

5. Neural Networks and Parallel Programming

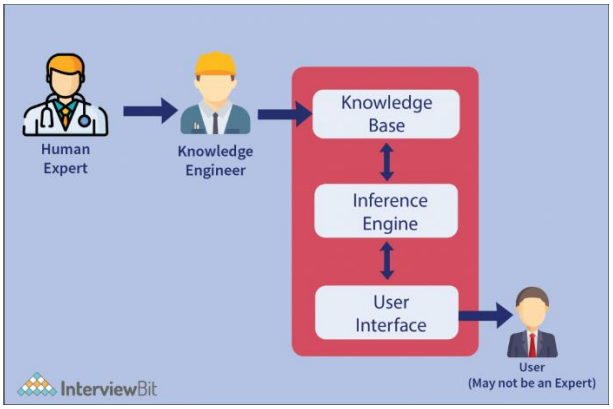


- A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain

operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature.

- Parallel programming involves the division of a problem into smaller subproblems, the subproblems can be executed at the same time using multiple computing resources. OOPs, are used to simplify the process by simplifying the approximation and prediction ability of the network.

6. AI Expert Systems



- An AI Expert system is the one that simulates the decision-making ability of a human expert, the expert knowledge can be increased using different add-ons to the knowledge base or in simple words the addition of rules. For example, PXDES is an expert system that predicts the degree and type of lung cancer. There are many expert systems out there.
- OOPs, power the development of such AI Expert systems, To AI systems use forward and backward chaining to reach a conclusion. Basically, the chaining involves a chain of conditions and derivations to deduce the outcome. An AI system has to be reliable, highly responsive, and offers a high performance, to power such capabilities OOPs are used.

7. Simulation and Modeling System

- Modeling a complex system is quite difficult owing to the varying specifications of the variables, such types of systems are prevalent in medicine and in other areas of natural science, like zoology, ecology, and agronomic systems. Simulation and modeling

systems are the imitations of the real world product. The system's workings can be checked and analyzed using object-oriented programming.

- A good example of a simulation and modeling system is of automobiles such as cars, Once the model of the car structure is developed by the engineer's team, as and when they feel that the product is good to go they can release the product. OOP provides an appropriate approach for simplifying these complex models.

8. Office Automation Systems

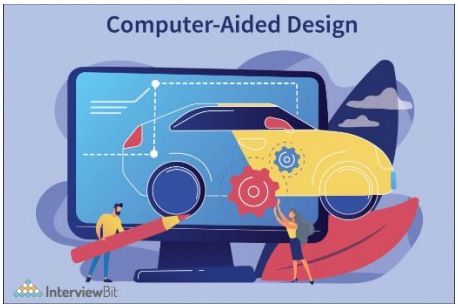
- Nowadays companies use automated systems to share information and communication to and from the people inside and outside the organization. There are various works in the office for which the employees can be replaced by automated robots and these works are generally time-consuming and hard work. The company can reduce its cost expenses by developing such a system.
- To develop office automation systems, robot programming automation or RPA is used. RPA uses Object-oriented programming.

9. CIM/CAD/CAM Systems

- OOP can also be used in manufacturing and designing applications as it allows people to reduce the efforts involved. For instance, it can be used while designing blueprints and flowcharts. So it makes it possible to produce these flowcharts and blueprint accurately.

10. Computer Aided Designs

- As per Wikipedia, Computer-aided design (CAD) is the use of computers (or workstations) to aid in the creation, modification, analysis, or optimization of a design. In mechanical design, it is known as mechanical design automation (MDA), which includes the process of creating a technical drawing with the use of computer software.



- One good example of computer-aided design is Mat lab. It is used by programmers to solve difficult mathematical models, these models are then used in bigger system designs to check whether the system will function as expected or not

JAVA (UNIT-II)

I. INTRODUCTION TO JAVA APPLICATION:

Technology is constantly going through an evolution and so are the languages that are used to develop them. Java is one of the popular programming languages having n number of applications. Through this article, I will be listing down the top 10 applications of Java.

1. ***Mobile Applications***
2. ***Desktop GUI Applications***
3. ***Web-based Applications***
4. ***Enterprise Applications***
5. ***Scientific Applications***
6. ***Gaming Applications***
7. ***Big Data technologies***
8. ***Business Applications***
9. ***Distributed Applications***
10. ***Cloud-based Applications***

1) Mobile Applications:

- Java is considered as the official programming language for mobile app development. It is compatible with software such as Android Studio and Kotlin. Now you must be wondering why only Java? The reason is that it can run on Java Virtual Machine(JVM), whereas Android uses DVK(Dalvik Virtual Machine) to execute class files. These files are further bundled as an Android application package(APK). With Java and its OOPs principles, it provides better security and ease of simplicity with Android.

2) Desktop GUI Applications

- All desktop applications can easily be developed in Java. Java also provides GUI development capability through various means mainly Abstract Windowing Toolkit (AWT), Swing, and JavaFX. While AWT holds a number of pre-assembled components like a menu, listand , button. Swing is a GUI widget toolkit, it provides certain advanced elements like trees, scroll panes, tables, tabbed panels, and lists.

3) Web-based Applications

- Java is also used to develop web applications. It provides vast support for web applications through Servlets, Struts, or JSPs. With the help of these technologies, you can develop any kind of web application that you require. The easy coding and high security offered by this programming language allow the development of a large number of applications for health, social security, education, and insurance.

4) Enterprise Applications

- Java is the first choice of many software developers for writing applications and Java Enterprise Edition (Java EE) is a very popular platform that provides API and runtime environment for scripting. It also includes network applications and web-services. JavaEE is also considered as the backbone for a variety of banking applications which have Java running on the UI to back server end.

5) Scientific Applications

- Software developers see Java is the weapon of choice when it comes to coding the scientific calculations and mathematical operations. These programs are designed to be highly secure and lighting fast. they support a higher degree of portability and offer low maintenance. Some of the most powerful applications like the MATLAB use Java for interacting user interface as well as part of the core system.

6) Gaming Applications

- Java has the support of the open-source most powerful 3D-Engine, the jMonkeyEngine that has the unparalleled capability when it comes to the designing of 3D games. However, it does cause an occasional latency issue for games as garbage collection cycles can cause noticeable pauses. This issue will be solved in the newer versions of JVMs.

7) Big Data technologies

- Java is the reason why the leading Big Data technologies like Hadoop have become a reality and also the most powerful programming languages like Scala are existing. It is crystal clear that Java is the backbone when it comes to developing Big Data using Java.

8) Business Applications:

- Java EE platform is designed to help developers create large-scale, multi-tiered, scalable, reliable, and secure network applications. These applications are designed to solve the problems encountered by large enterprises. The features that make enterprise applications powerful, like security and reliability, often make these applications complex. The Java EE platform reduces the complexity of enterprise application development by providing a development model, API, and runtime environment that allow developers to concentrate on functionality.

9) Distributed Applications:

- Distributed applications have several common requirements that arise specifically because of their distributed nature and of the dynamic nature of the system and platforms they operate on. Java offers options to realize these applications. The Jini (Java Intelligent Networking Infrastructure) represents an infrastructure to provide, register, and find distributed services based on its specification. One integral part of Jini is JavaSpaces, a mechanism that supports distribution, persistence, and migration of objects in a network.

10) Cloud-Based Applications:

- Cloud computing means on-demand delivery of IT resources via the internet with pay-as-you-go pricing. It provides a solution for IT infrastructure at a low cost. Java provides you with features that can help you build applications meaning that it can be used in the SaaS, IaaS and PaaS development. It can serve the companies to build their applications remotely or help companies share data with others, whatever the requirement.

II. JAVA CLASSES/OBJECTS

- Java is an object-oriented programming language.
- Everything in Java is associated with classes and objects, along with its attributes and methods. For example: in real life, a car is an object. The car has attributes, such as weight and colour, and methods, such as drive and brake.
- A Class is like an object constructor, or a "blueprint" for creating objects.

SYNTAX FOR CLASS:

```
class class_name
{
// member variables
// class methods
}
```

CREATE A CLASS

To create a class, use the keyword `class`:

Main.java

```
public class Main {

  int x = 5;

}
```

USING MULTIPLE CLASSES

- You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the `main()` method (code to be executed)).
- Remember that the name of the java file should match the class name.

EXAMPLE:

```
public class Main {

  int x = 5;

}

class Second {

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

```
    Main myObj = new Main();

    System.out.println(myObj.x);

  }

}
```

CREATE AN OBJECT

- In Java, an object is created from a class. We have already created the class named `Main`, so now we can use this to create objects.
- To create an object of `Main`, specify the class name, followed by the object name, and use the keyword `new`

SYNTAX:

```
className object = new className();
```

Example

Create an object called "`myObj`" and print the value of x:

```
public class Main {

  int x = 5;

  public static void main(String[] args) {

    Main myObj = new Main();

    System.out.println(myObj.x);

  }

}
```

MULTIPLE OBJECTS

- You can create multiple objects of one class:

Example

Create two objects of `Main`:

```
public class Main {
  int x = 5;

  public static void main(String[] args) {
    Main myObj1 = new Main();  // Object 1
    Main myObj2 = new Main();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

EXAMPLE PROGRAM FOR CLASS, OBJECT:

```
class Student //Defining a Student class.
{
 int id;  //field or data member or instance variable
 String name;
   //creating main method inside the Student class
 public static void main(String args[]){
  //Creating an object or instance
  Student s1=new Student();//creating an object of Student
  //Printing values of the object
  System.out.println(s1.id);//accessing member through reference variable
  System.out.println(s1.name);
 }
}
```

III. METHOD:

- A **method** is a block of code that only runs when it is called.
- You can pass data, known as parameters, into a method.
- Methods are used to perform certain actions, and they are also known as **functions**.
- Why use methods? To reuse code: define the code once, and use it many times.

CREATE A METHOD:

- A method must be declared within a class. It is defined with the name of the method, followed by parentheses **()**. Java provides some pre-defined methods, such as System.out.println(), but you can also create your own methods to perform certain actions:

Example

Create a method inside Main:

```
public class Main {

 static void myMethod() {

  // code to be executed

 }

}
```

Example Explained

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter.

CALL A METHOD:

- To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**
- In the following example, `myMethod()` is used to print a text (the action), when it is called:

Example

Inside `main`, call the `myMethod()` method:

```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}
```

A method can also be called multiple times:

Example

```
public class Main {
  static void myMethod() {
    System.out.println("I just got executed!");
  }
  public static void main(String[] args) {
    myMethod();
    myMethod();
    myMethod();
}}
```

EXAMPLE PROGRAM FOR STUDENT DETAILS USING METHODS:

```
import java.io.*;

class Student

{    int id;

     String name;

 void getdetails(int x,String y)

   {        id= x;

    name= y;

    }

    void display()

    {

        System.out.println("The student details "+id+ " " +name);

    }

}


class Main

{    public static void main(String args[])

   {

        Student s1=new Student();

        Student s2=new Student();

        s1.id=9;

        s1.name="XYZ";

        s1.display();

        s2.getdetails(10,"ABC");

        s2.display();

    }}
```

IV. JAVA STRINGS:

- Strings are used for storing text.
- A `String` variable contains a collection of characters surrounded by double quotes:

Example

Create a variable of type String and assign it a value:

```
public class Main {
  public static void main(String[] args) {
    String Str = "Hello";
    System.out.println(Str);
  }
}
```

STRING LENGTH:

- A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the `length()` method:

  EXAMPLE:

```
public class Main {
  public static void main(String[] args) {
    String txt = "Hello World";
    System.out.println(txt.toUpperCase());
    System.out.println(txt.toLowerCase());
  }}
```

STRING CONCATENATION:

- The `+` operator can be used between strings to combine them. This is called **concatenation**:

Example

```
String firstName = "TAJ";

String lastName = "MAHAL";

System.out.println(firstName + " " + lastName);
```

ADDING NUMBERS AND STRINGS:

WARNING!

- Java uses the + operator for both addition and concatenation.
- Numbers are added. Strings are concatenated.

If you add two numbers, the result will be a number:

Example

```
int x = 10;

int y = 20;

int z = x + y;  // z will be 30 (an integer/number)
```

METHODS OF JAVA STRING:

- Besides those mentioned above, there are various string methods present in Java. Here are some of those methods:

Methods	Description
contains()	checks whether the string contains a substring
substring()	returns the substring of the string
join()	join the given strings using the delimiter

replace()	replaces the specified old character with the specified new character
replaceAll()	replaces all substrings matching the regex pattern
replaceFirst()	replace the first matching substring
charAt()	returns the character present in the specified location
getBytes()	converts the string to an array of bytes
indexOf()	returns the position of the specified character in the string
compareTo()	compares two strings in the dictionary order
compareToIgnoreCase()	compares two strings ignoring case differences
trim()	removes any leading and trailing whitespaces
format()	returns a formatted string
split()	breaks the string into an array of strings
toLowerCase()	converts the string to lowercase
toUpperCase()	converts the string to uppercase
valueOf()	returns the string representation of the specified argument
toCharArray()	converts the string to a `char` array

matches()	checks whether the string matches the given regex
startsWith()	checks if the string begins with the given string
endsWith()	checks if the string ends with the given string
isEmpty()	checks whether a string is empty of not
intern()	returns the canonical representation of the string
contentEquals()	checks whether the string is equal to charSequence
hashCode()	returns a hash code for the string
subSequence()	returns a subsequence from the string

- Java compiler executes the code from top to bottom. The statements in the code 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. It is one of the fundamental features of Java, which provides a smooth flow of the program.

V. Java Control Statements

- Java provides three types of control flow statements.
- **Decision Making statements**
  - if statements
  - switch statement

- **Loop statements**
    - do while loop
    - while loop
    - for loop
    - for-each loop
- **Jump statements**
    - break statement
    - continue statement

DECISION-MAKING STATEMENTS:

- As the name suggests, decision-making statements decide which statement to execute and when. Decision-making statements evaluate the Boolean expression and control the program flow depending upon the result of the condition provided. There are two types of decision-making statements in Java, i.e., If statement and switch statement.

1) IF STATEMENT:

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

- ***Simple if statement***
- ***if-else statement***
- ***if-else-if ladder***
- ***Nested if-statement***

Let's understand the if-statements one by one.

1) Simple if statement:

It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

Syntax of if statement is given below.

```
if(condition) {

statement 1; //executes when condition is true
}
```

Consider the following example in which we have used the **if** statement in the java code.

Student.java

```
public class Student {

public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y > 20) {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

```
x + y is greater than 20
```

2) if-else statement

The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

Syntax:

```
if(condition) {

statement 1; //executes when condition is true  }

else{

statement 2; //executes when condition is false

}
```

Student.java

```
public class Student {
public static void main(String[] args) {
int x = 10;
int y = 12;
if(x+y < 10) {
System.out.println("x + y is less than      10");
}   else {
System.out.println("x + y is greater than 20");
}
}
}
```

Output:

x + y is greater than 20

3) if-else-if ladder:

The if-else-if statement contains the if-statement followed by multiple else-if statements. In other words, we can say that it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. We can also define an else statement at the end of the chain.

Syntax:

```
if(condition 1) {
statement 1; //executes when condition 1 is true
}
else if(condition 2) {
statement 2; //executes when condition 2 is true
}
else {
statement 2; //executes when all the conditions are false
 }
```

Student.java

```
public class Student {
public static void main(String[] args) {
String city = "Delhi";
if(city == "Meerut") {
System.out.println("city is meerut");
}else if (city == "Noida") {
System.out.println("city is noida");
}else if(city == "Agra") {
System.out.println("city is agra");
}else {
System.out.println(city);
}
}
}
```

Output:

Delhi

4. Nested if-statement

In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

Syntax:

```
if(condition 1) {
statement 1; //executes when condition 1 is true
if(condition 2) {
statement 2; //executes when condition 2 is true   }
else{
statement 2; //executes when condition 2 is false   } }
```

Consider the following example.

Student.java

```
public class Student {

public static void main(String[] args) {

String address = "Delhi, India";

if(address.endsWith("India")) {

if(address.contains("Meerut")) {

System.out.println("Your city is Meerut");

}else if(address.contains("Noida")) {

System.out.println("Your city is Noida");

}else {

System.out.println(address.split(",")[0]);

}   }else {

System.out.println("You are not living in India");

}   }   }
```

Output:

Delhi

Switch Statement:

In Java, Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

- The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
- Cases cannot be duplicate
- Default statement is executed when any of the case doesn't match the value of expression. It is optional.
- Break statement terminates the switch block when the condition is satisfied. It is optional, if not used, next case is executed.

- While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

Syntax:

```
switch (expression){
case value1:
statement1;
break;
.
.
.
case valueN:
statementN;
break;
default:
default statement;
}
```

Consider the following example to understand the flow of the switch statement.

Student.java

```
public class Student implements Cloneable {
public static void main(String[] args) {
int num = 2;
switch (num){
case 0:
System.out.println("number is 0");
break;
case 1:
System.out.println("number is 1");
```

```
break;
default:
System.out.println(num);
}
}
}
```

Output:

2

While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value. The switch permits only int, string, and Enum type variables to be used.

Loop Statements

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order. The execution of the set of instructions depends upon a particular condition.

In Java, we have three types of loops that execute similarly. However, there are differences in their syntax and condition checking time.

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

Let's understand the loop statements one by one.

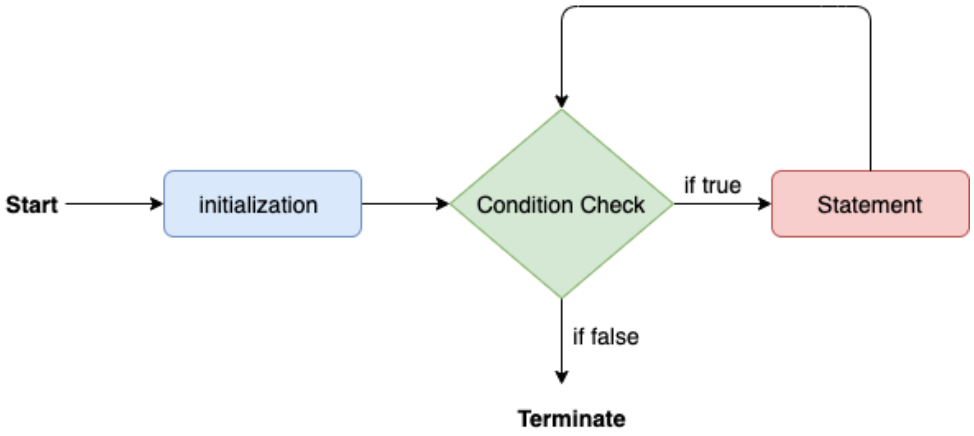
Java for loop

- In Java, for loop is similar to C and C++.
- It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

SYNTAX:

```
for(initialization, condition, increment/decrement) {
//block of statements
}
```

The flow chart for the for-loop is given below.



Consider the following example to understand the proper functioning of the for loop in java.

Calculation.java

```
public class Calculattion {
public static void main(String[] args) {
// TODO Auto-generated method stub
int sum = 0;
for(int j = 1; j<=10; j++) {
sum = sum + j;
}
System.out.println("The sum of first 10 natural numbers is " + sum);
}
}
```

Output:

The sum of first 10 natural numbers is 55

Java for-each loop

- Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

```
for(data_type var : array_name/collection_name)

{   //statements

}
```

Consider the following example to understand the functioning of the for-each loop in Java.

Calculation.java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
String[] names = {"Java","C","C++","Python","JavaScript"};
System.out.println("Printing the content of the array names:\n");
for(String name:names) {
System.out.println(name);
}
}
}
```

Output:

```
Printing the content of the array names:

Java
C
C++
Python
JavaScript
```

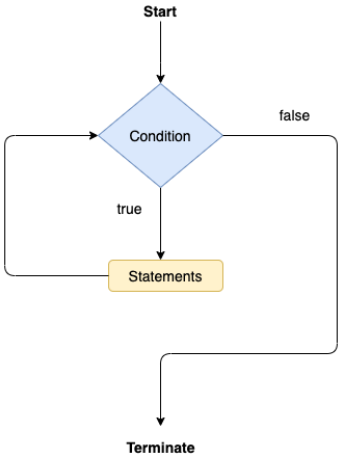
Java while loop

- The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop.

- It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.
- The syntax of the while loop is given below.

```
while(condition)
{
//looping statements
}
```

The flow chart for the while loop is given in the following image.



Consider the following example.

Calculation .java

```
public class Calculation {
public static void main(String[] args) {
// TODO Auto-generated method stub
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
while(i<=10) {
System.out.println(i);
i = i + 2;
}
}
}
```

Output:

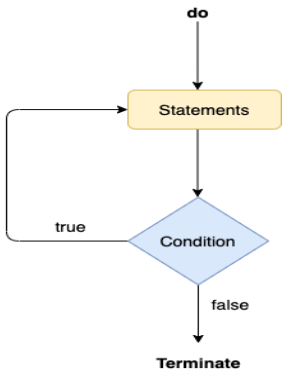
```
Printing the list of first 10 even numbers

0
2
4
6
8
10
```

Java do-while loop

- The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iteration is not known and we have to execute the loop at least once, we can use do-while loop.
- It is also known as the exit-controlled loop since the condition is not checked in advance.
- The syntax of the do-while loop is given below.

```
do {
//statements
} while (condition);
```

The flow chart of the do-while loop is given in the following image.



Calculation.java

```
public class Calculation {
public static void main(String[] args) {
int i = 0;
System.out.println("Printing the list of first 10 even numbers \n");
do {
System.out.println(i);
i = i + 2;
}while(i<=10);    }    }
```

Output:

```
Printing the list of first 10 even numbers
0
2
4
6
8
10
```

Jump Statements

- Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program.
- There are two types of jump statements in Java, i.e., break and continue.

Java break statement

- A name suggests, the break statement The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

The break statement example with for loop

Consider the following example in which we have used the break statement with the for a loop.

BreakExample.java

```
public class BreakExample {

public static void main(String[] args) {
// TODO Auto-generated method stub
for(int i = 0; i<= 10; i++) {
System.out.println(i);
if(i==6) {
break;
}
}
}
}
```

Output:

```
0
1
2
3
4
5
6
```

break statement example with labeled for loop

Calculation.java

```
public class Calculation {

public static void main(String[] args) {
// TODO Auto-generated method stub
a:
for(int i = 0; i<= 10; i++) {
b:
for(int j = 0; j<=15;j++) {
c:
for (int k = 0; k<=20; k++) {
System.out.println(k);
if(k==5) {
break a;
} } } } }
```

Output:

```
0
1
2
3
4
5
```

Java continue statement

- Unlike the break statement, the continue statement doesn't break the loop, whereas, it skips the specific part of the loop and jumps to the next iteration of the loop immediately.
- Consider the following example to understand the functioning of the continue statement in Java.

```
public class ContinueExample {

public static void main(String[] args) {
// TODO Auto-generated method stub

for(int i = 0; i<= 2; i++) {

for (int j = i; j<=5; j++) {

if(j == 4) {
continue;
}
System.out.println(j);
}  }}}
```

Output:

```
0
1
2
3
5
1
2
3
5
2
3
5
```

VI. 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.

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.

```
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]);  }}
```

Output:

```
10
20
70
40
50
```

Declaration, Instantiation and Initialization of Java Array:

- We can declare, instantiate and initialize the java array together by:

```
int a[]={33,3,4,5};//declaration, instantiation and initialization
```

Let's see the simple example to print this array.

```
//Java Program to illustrate the use of declaration, instantiation
//and initialization of Java array in a single line
class Testarray1{
public static void main(String args[]){
int a[]={33,3,4,5};//declaration, instantiation and initialization
//printing array
for(int i=0;i<a.length;i++)//length is the property of array
System.out.println(a[i]);
}}
```

Output:

```
33
3
4
5
```

For-each Loop for Java Array

- We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.
- The syntax of the for-each loop is given below:

```
for(data_type variable:array){

//body of the loop

}
```

Let us see the example of print the elements of Java array using the for-each loop.

```
//Java Program to print the array elements using for-each loop
class Testarray1{
public static void main(String args[]){
int arr[]={33,3,4,5};
//printing array using for-each loop
for(int i:arr)
System.out.println(i);
}}
```

Output:

```
33
3
4
5
```

2. Multidimensional Array in Java:

In such cases, data is stored in a 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 to initialize Multidimensional Array in Java

```
arr[0][0]=1;
arr[0][1]=2;
arr[0][2]=3;
arr[1][0]=4;
arr[1][1]=5;
arr[1][2]=6;
```

```
arr[2][0]=7;
arr[2][1]=8;
arr[2][2]=9;
```

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();
}
}}
```

Output:

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

Addition of 2 Matrices in Java

Let's see a simple example that adds two matrices.

//Java Program to demonstrate the addition of two matrices in Java

```
class Testarray5{
public static void main(String args[]){
//creating two matrices
int a[][]={{1,3,4},{3,4,5}};
int b[][]={{1,3,4},{3,4,5}};
```

```
//creating another matrix to store the sum of two matrices
int c[][]=new int[2][3];

//adding and printing addition of 2 matrices
for(int i=0;i<2;i++){
for(int j=0;j<3;j++){
c[i][j]=a[i][j]+b[i][j];
System.out.print(c[i][j]+" ");
}
System.out.println();//new line
}

}}
```

Output:

```
2 6 8
6 8 10
```

VII. Constructors in Java

- 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 the 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.
- It calls a default constructor if there is no constructor available in the class. In such cases, the Java compiler provides a default constructor by default.

Rules for creating Java constructor

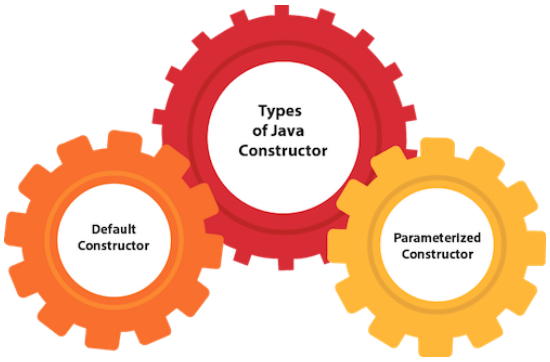
There are two rules defined for the constructor.

1. Constructor name must be the same as its class name
2. A Constructor must have no explicit return type

Rules for creating Java constructor

There are two 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



1.Java Default Constructor

A constructor is called "Default Constructor" when it doesn't have any parameter.

Example of default constructor

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

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

Output:

```
Bike is created
```

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.

```
//Java Program to demonstrate the use of the parameterized constructor.
class Student4{
    int id;
    String name;
    //creating a parameterized constructor
    Student4(int i,String n){
    id = i;
    name = n;
    }
    //method to display the values
    void display(){System.out.println(id+" "+name);}

    public static void main(String args[]){
    //creating objects and passing values
    Student4 s1 = new Student4(111,"Karan");
    Student4 s2 = new Student4(222,"Aryan");
    //calling method to display the values of object
    s1.display();
    s2.display();
   }
}
```

Output:

```
111 Karan
222 Aryan
```

Constructor Overloading in Java

- In Java, a constructor is just like a method but without return type. It can also be overloaded like Java methods.
- Constructor overloading in java is a technique of having more than one constructor with different parameter lists. They are arranged in a way that each constructor performs a different task. They are differentiated by the compiler by the number of parameters in the list and their types.

Example of Constructor Overloading

```
//Java program to overload constructors
class Student5{
   int id;
   String name;
   int age;
   //creating two arg constructor
   Student5(int i,String n){
   id = i;
   name = n;
   }
   //creating three arg constructor
   Student5(int i,String n,int a){
   id = i;
   name = n;
   age=a;
   }
   void display(){System.out.println(id+" "+name+" "+age);}
   public static void main(String args[]){
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
   }}
```

Output:

```
111 Karan 0
222 Aryan 25
```

Difference between constructor and method in Java

There are many differences between constructors and methods. They are given below.

Java Constructor	Java Method
A constructor is used to initialize the state of an object.	A method is used to expose the behaviour 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 the same as the class name.	The method name may or may not be the same as the class name.

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, arrays and interfaces.

Data types are divided into two groups:

- Primitive data types -
  includes `byte`, `short`, `int`, `long`, `float`, `double`, `boolean` and `char`
- Non-primitive data types - such as `String`, Arrays and Classes (you will learn more about these in a later chapter)

1.Primitive Data Types

- A primitive data type specifies the size and type of variable values, and it has no additional methods.
- There are eight primitive data types in Java:

Data Type	Size	Description
`byte`	1 byte	Stores whole numbers from -128 to 127
`short`	2 bytes	Stores whole numbers from -32,768 to 32,767
`int`	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
`double`	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal

		digits
`boolean`	1 bit	Stores true or false values
`char`	2 bytes	Stores a single character/letter or ASCII values

EXAMPLE OF PRIMITIVE DATATYPES:

```
public class Main {
  public static void main(String[] args) {
    int myNum = 5;            // integer (whole number)
    float myFloatNum = 5.99f;   // floating point number
    char myLetter = 'D';        // character
    boolean myBool = true;      // boolean
    String myText = "Hello";    // String
    System.out.println(myNum);
    System.out.println(myFloatNum);
    System.out.println(myLetter);
    System.out.println(myBool);
    System.out.println(myText);
  }
}
```

OUTPUT

```
5
5.99
D
true
Hello
```

2.Non-Primitive Data Types

Non-primitive data types are called **reference types** because they refer to objects.

The main difference between **primitive** and **non-primitive** data types are:

- Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for `String`).
- Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
- A primitive type has always a value, while non-primitive types can be `null`.
- A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
- The size of a primitive type depends on the data type, while non-primitive types have all the same size.

Operators in Java

Operator in Java

is a symbol that is used to perform operations. For example: +, -, *, / etc.

There are many types of operators in Java which are given below:

- Unary Operator,
- Arithmetic Operator,
- Shift Operator,
- Relational Operator,
- Bitwise Operator,
- Logical Operator,
- Ternary Operator and
- Assignment Operator.

Java Operator Precedence

Operator Type	Category	Precedence
Unary UNARY	postfix	*expr*++ *expr*--
	prefix	++*expr* --*expr* +*expr* -*expr* ~ !

Arithmetic ARITHMETIC	multiplicative	* / %
	additive	+ -
SHIFT	shift	<< >> >>>
RELATIONAL RELATIONAL	comparison	< > <= >= instanceof
	equality	== !=
BITWISE	bitwise AND	&
	bitwise exclusive OR	^
	bitwise inclusive OR	|
Logical 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 --

```
public 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 !

```
public 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
}}
```

Output:

```
-11
9
false
true
```

2. Java Arithmetic Operators

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

Java Arithmetic Operator Example

```
public 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
```

Java Arithmetic Operator Example: Expression

```
public class OperatorExample{
public static void main(String args[]){
System.out.println(10*10/5+3-1*4/2);
}}
```

Output:

```
21
```

3. Java Left Shift Operator

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

Java Left Shift Operator Example

```
public 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 the value of the left operand to right by the number of bits specified by the right operand.

Java Right Shift Operator Example

```
public 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
```

Java Shift Operator Example: >> vs >>>

```
public class OperatorExample{
public static void main(String args[]){
   //For positive number, >> and >>> works same
   System.out.println(20>>2);
```

```
    System.out.println(20>>>2);
    //For negative number, >>> changes parity bit (MSB) to 0
    System.out.println(-20>>2);
    System.out.println(-20>>>2);
}}
```

Output:

```
5
5
-5
1073741819
```

4. Java AND Operator Example: Logical && and Bitwise &

The logical && operator doesn't check the second condition if the first condition is false. It checks the second condition only if the first one is true.

The bitwise & operator always checks both conditions whether first condition is true or false.

```
public 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 AND Operator Example: Logical && vs Bitwise &

```
public 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);//10 because second condition is not checked
System.out.println(a<b&a++<c);//false && true = false
System.out.println(a);//11 because second condition is checked
}}
```

Output:

```
false
10
false
11
```

Java OR Operator Example: Logical || and Bitwise |

The logical || operator doesn't check the second condition if the first condition is true. It checks the second condition only if the first one is false.

The bitwise | operator always checks both conditions whether first condition is true or false.

```
public 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
```

5. Java Ternary Operator

Java Ternary operator is used as one line 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

```
public 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:

```
public 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
```

6. Java Assignment Operator

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

Java Assignment Operator Example

```
public 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
```

Inheritance in Java

- **Inheritance in Java** is a mechanism in which one object acquires all the properties and behaviours of a parent object. It is an important part of OOPs (Object Oriented programming system).

- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

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.

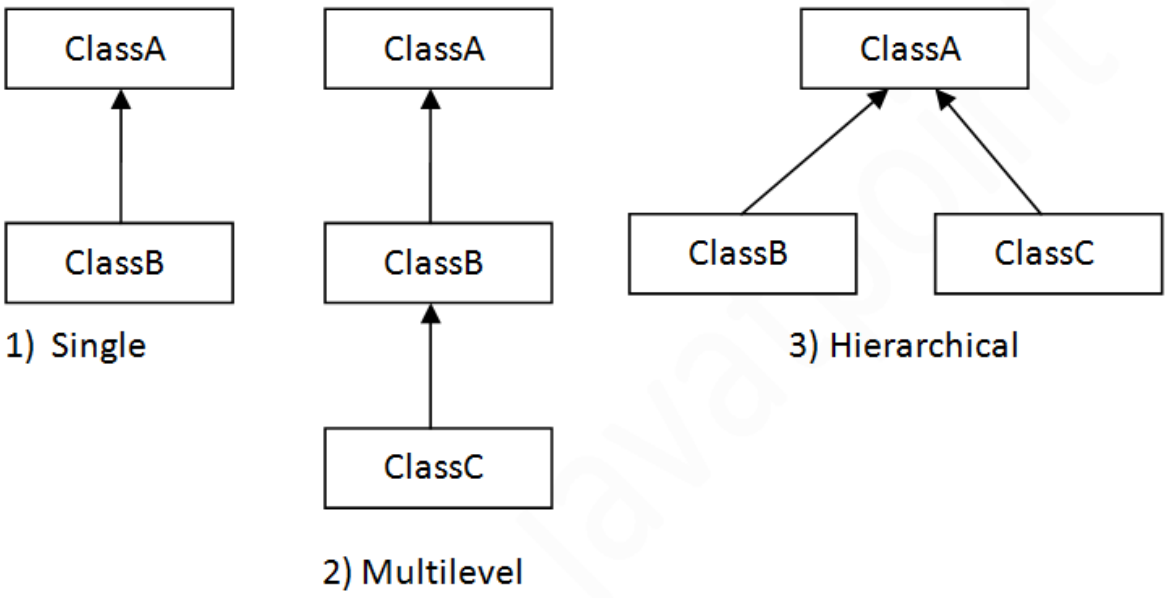
The syntax of Java Inheritance

```
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.

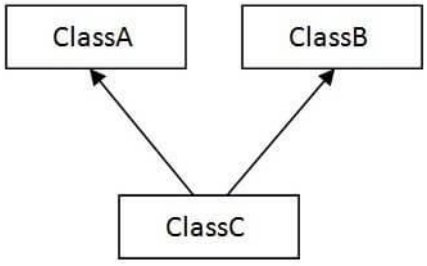
Types of inheritance in java

- On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical.
- In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

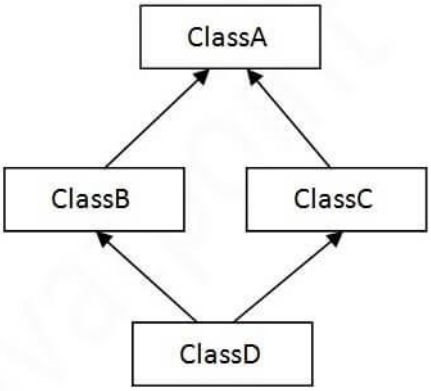


Note: Multiple inheritance is not supported in Java through class.

When one class inherits multiple classes, it is known as multiple inheritance. For Example:



4) Multiple



5) Hybrid

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.

File: TestInheritance.java

```
class Animal{
void eat(){System.out.println("eating...");}
}
class Dog extends Animal{
void bark(){System.out.println("barking...");}
}
class TestInheritance{
public static void main(String args[]){
Dog d=new Dog();
d.bark();
d.eat();
}}
```

Output:

```
barking...
eating...
```

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.

File: TestInheritance2.java

```
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 TestInheritance2{
public static void main(String args[]){
BabyDog d=new BabyDog();
d.weep();
d.bark();
d.eat();
}}
```

Output:

```
weeping...
barking...
eating...
```

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.

File: TestInheritance3.java

```
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 TestInheritance3{
public static void main(String args[]){
Cat c=new Cat();
c.meow();
c.eat();  }}
```

Output:

```
meowing...
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?
}
}
```

OUTPUT:

```
Compile Time Error
```

VIII. Polymorphism in Java

- The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.
- **Real-life Illustration:** Polymorphism
- A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person possesses different behavior in different situations. This is called polymorphism.
  Polymorphism is considered one of the important features of Object-Oriented Programming. Polymorphism allows us to perform a single action in different ways. In other words, polymorphism allows you to define one interface and have multiple implementations. The word "poly" means many and "morphs" means forms, So it means many forms.

Types of polymorphism

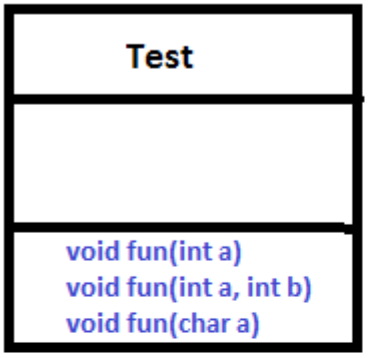
In Java polymorphism is mainly divided into two types:

- Compile-time Polymorphism
- Runtime Polymorphism

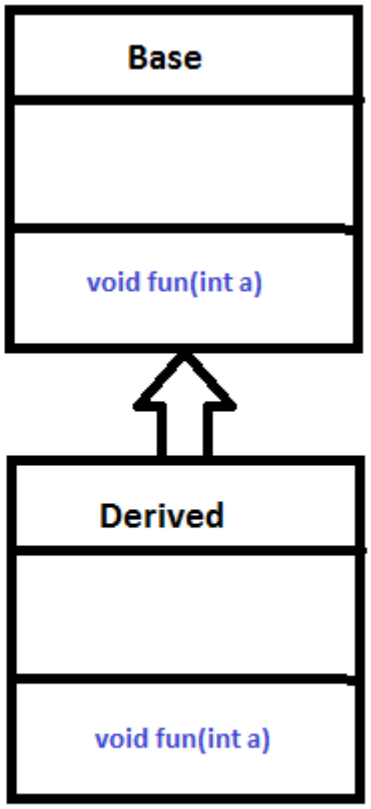
Type 1: Compile-time polymorphism

It is also known as static polymorphism. This type of polymorphism is achieved by function overloading or operator overloading.

Note: *But Java doesn't support the Operator Overloading.*



Overloading



Overriding

COMPILE TIME POLYMORPHISM:

Method Overloading: When there are multiple functions with the same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by change in the number of arguments or/and a change in the type of arguments.

Example 1

```
// Java Program for Method overloading
// By using Different Types of Arguments

// Class 1
// Helper class
class Helper {

    // Method with 2 integer parameters
    static int Multiply(int a, int b)
    {

        // Returns product of integer numbers
        return a * b;
    }
```

```
    // Method 2
    // With same name but with 2 double parameters
    static double Multiply(double a, double b)
    {

        // Returns product of double numbers
        return a * b;
    }
}

// Class 2
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(5.5, 6.3));
    }
}
```

Output:

8

34.65

Example 2

```
// Java program for Method Overloading
// by Using Different Numbers of Arguments

// Class 1
// Helper class
class Helper {

    // Method 1
    // Multiplication of 2 numbers
    static int Multiply(int a, int b)
    {

        // Return product
        return a * b;
    }

    // Method 2
    // // Multiplication of 3 numbers
    static int Multiply(int a, int b, int c)
    {
```

```
        // Return product
        return a * b * c;
    }
}

// Class 2
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Calling method by passing
        // input as in arguments
        System.out.println(Helper.Multiply(2, 4));
        System.out.println(Helper.Multiply(2, 7, 3));
    }
}
```

Output:

8

42

Type 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**.

Example

```
// Java Program for Method Overriding

// Class 1
// Helper class
class Parent {

    // Method of parent class
    void Print()
    {
```

```
        // Print statement
        System.out.println("parent class");
    }
}

// Class 2
// Helper class
class subclass1 extends Parent {

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

// Class 3
// Helper class
class subclass2 extends Parent {

    // Method
    void Print()
    {

        // Print statement
        System.out.println("subclass2");
    }
}

// Class 4
// Main class
class GFG {

    // Main driver method
    public static void main(String[] args)
    {

        // Creating object of class 1
        Parent a;

        // Now we will be calling print methods
        // inside main() method

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

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

Output:

```
subclass1

subclass2
```

Output explanation:

Here in this program, When an object of child class is created, then the method inside the child class is called. This is because The method in the parent class is overridden by the child class. Since The method is overridden, This method has more priority than the parent method inside the child class. So, the body inside the child class is executed.

IX. 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.

- It is used to achieve abstraction.
- By interface, we can support the functionality of multiple inheritance.
- It can be used to achieve loose coupling.

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 with the empty body, 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.

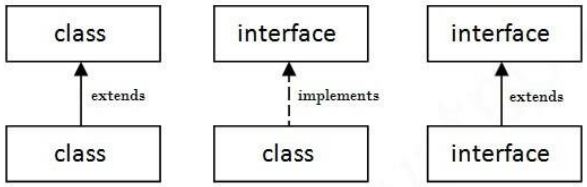
Syntax:

```
interface <interface_name>{
```

```
    // declare constant fields
    // declare methods that abstract
    // by default.
}
```

The relationship between classes and interfaces

As shown in the figure given below, a class extends another class, an interface extends another interface, but a **class implements an interface**.



Java Interface Example

In this example, the Printable interface has only one method, and its implementation is provided in the A6 class.

```
interface printable{
void print();
}
class A6 implements printable{
public void print(){System.out.println("Hello");}

public static void main(String args[]){
A6 obj = new A6();
obj.print();
 }
}
```

Output:

```
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.

File: TestInterface1.java

```
//Interface declaration: by first user
interface Drawable{
void draw();
}
//Implementation: by second user
class Rectangle implements Drawable{
public void draw(){System.out.println("drawing rectangle");}
}
class Circle implements Drawable{
public void draw(){System.out.println("drawing circle");}
}
//Using interface: by third user
class TestInterface1{
public static void main(String args[]){
Drawable d=new Circle();//In real scenario, object is provided by method e.g. getDrawable()
d.draw();
}}
```

Output:

```
drawing circldrawing circle
```

Java Interface Example: Bank

Let's see another example of java interface which provides the implementation of Bank interface.

File: TestInterface2.java

```
interface Bank{
float rateOfInterest();
}
```

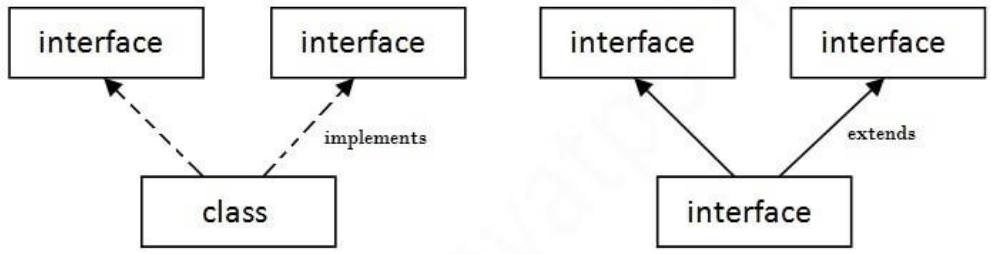
```
class SBI implements Bank{
public float rateOfInterest(){return 9.15f;}
}
class PNB implements Bank{
public float rateOfInterest(){return 9.7f;}
}
class TestInterface2{
public static void main(String[] args){
Bank b=new SBI();
System.out.println("ROI: "+b.rateOfInterest());
}}
```

Output:

```
ROI: 9.15
```

Multiple inheritance in Java by interface

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Multiple Inheritance in Java

```
interface Printable{
void print();
}
interface Showable{
void show();
}
class A7 implements Printable,Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}
```

```
public static void main(String args[]){
A7 obj = new A7();
obj.print();
obj.show();
 }
}
```

```
Output:Hello
       Welcome
```

Interface inheritance

A class implements an interface, but one interface extends another interface.

```
interface Printable{
void print();
}
interface Showable extends Printable{
void show();
}
class TestInterface4 implements Showable{
public void print(){System.out.println("Hello");}
public void show(){System.out.println("Welcome");}

public static void main(String args[]){
TestInterface4 obj = new TestInterface4();
obj.print();
obj.show();  }  }
```

Output:

```
Hello
Welcome
```

X. 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.

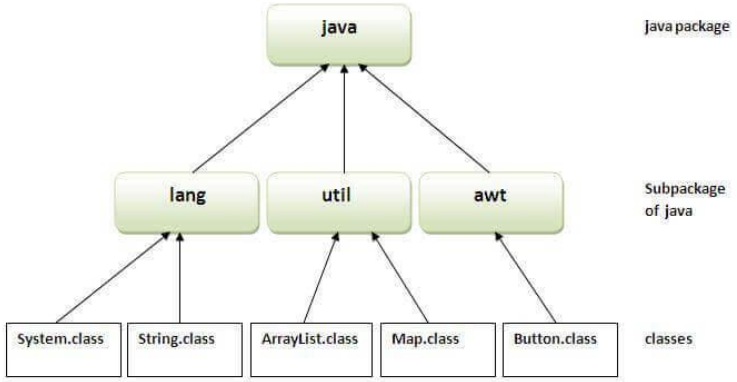
Here, we will have the detailed learning of creating and using user-defined packages.

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.

```
//save as Simple.java
package mypack;
public class Simple{
 public static void main(String args[]){
   System.out.println("Welcome to package");
  }
}
```

How to compile java package

If you are not using any IDE, you need to follow the **syntax** given below:

javac -d directory javafilename

How to run java package program

You need to use fully qualified name e.g. mypack.Simple etc to run the class.

To Compile: javac -d . Simple.java

To Run: java mypack.Simple

```
Output:Welcome to package
```

The -d is a switch that tells the compiler where to put the class file i.e. it represents destination.

The . represents the current folder.

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 will be accessible 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();
  }
}
```

```
Output:Hello
```

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
```

XI. 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.

In this tutorial, we will learn about Java exceptions, it's types, and the difference between checked and unchecked exceptions.

What is Exception in Java?

Dictionary Meaning: 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.

dvantage 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 that directly inherit the Throwable class except RuntimeException and Error are known as checked exceptions. For example, IOException, SQLException, etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

The classes that inherit the RuntimeException are known as unchecked exceptions. For example, ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException, etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

Error is irrecoverable. Some example of errors are OutOfMemoryError, VirtualMachineError, AssertionError etc.

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.

1. GUI COMPONENT:

- A GUI component is an object that represents a screen element such as a button or a text field
- GUI-related classes are defined primarily in the java.awt and the javax.swing packages
- The Abstract Windowing Toolkit (AWT) was the original Java GUI package
- The Swing package provides additional and more versatile components
- Both packages are needed to create a Java GUI-based program
- GUI Component classes, such as Button, TextField, and Label.
- GUI Container classes, such as Frame and Panel. Layout managers, such as FlowLayout, BorderLayout and GridLayout.
- Custom graphics classes, such as Graphics, Color and Font.

GUI Containers

- A GUI container is a component that is used to hold and organize other components
- A frame is a container displayed as a separate window with a title bar
- It can be repositioned and resized on the screen as needed
- A panel is a container that cannot be displayed on its own but is used to organize other components
- A panel must be added to another container (like a frame or another panel) to be displayed.
- A GUI container can be classified as either heavyweight or lightweight
- A heavyweight container is one that is managed by the underlying operating system
- A lightweight container is managed by the Java program itself
- Occasionally this distinction is important
- A frame is a heavyweight container and a panel is a lightweight container.

2. OVERVIEW OF SWING COMPONENTS:

DEFINITION: SWING:

- Swing is the collection of user interface components for Java programs. It is part of the Java
- Foundation classes are referred to as JFC. Swing is the graphical user interface toolkit that is
- Used for developing windows based java applications or programs. It is the successor of AWT,
- which is known as the Abstract window toolkit API for Java, and AWT components are mainly
- heavyweight.

FEATURES OF SWING:

1. **Platform Independent:** It is platform-independent; the swing components that are used to build the program are not platform-specific. It can be used on any platform and anywhere.
2. **Lightweight:** Swing components are lightweight, which helps in creating the UI lighter. The swings component allows it to plug into the operating system user interface framework, including the mappings for screens or devices and other user interactions like keypress and mouse movements.
3. **Plugging:** It has a powerful component that can be extended to provide support for the user interface which helps in a good look and feel to the application. It refers to the highly modular-based architecture that allows it to plug into other customized implementations and frameworks for user interfaces. Its components are imported through a package called java. Swing.
4. **Manageable:** It is easy to manage and configure. Its mechanism and composition pattern also allows changing the settings at run time. The uniform changes can be provided to the user interface without doing any changes to the application code.
5. **MVC:** They mainly follow the concept of MVC which is the Model View Controller. With the help of this, we can do changes in one component without impacting or touching other components. It is known as loosely coupled architecture as well.

6. **Customizable:** Swing controls can be easily customized. It can be changed, and the visual appearance of the swing component application is independent of its internal representation.

Swing Components

- A component is independent visual control, and Java Swing Framework contains a large set of these components, providing rich functionalities and allowing high customization. They all are derived from JComponent class. All these components are lightweight components. This class offers some standard functionality like pluggable look and feel, support for accessibility, drag and drop, layout, etc.
- A container holds a group of components. It delivers a space where a component can be managed and displayed. Containers are of two types:

Top-level Containers	It inherits the Component and Container of AWT.
	We cannot contain it within other containers.
	Heavyweight.
	Example: JFrame, JDialog, JApplet
Light weight container	It inherits JComponent class.
	It is a general-purpose container.
	We can use it to organize related components together.
	Example: JPanel

JButton

- We use JButton class to create a push button on the UI. The button can include some display text or images. It yields an event when clicked and double-clicked. We can implement a JButton in the application by calling one of its constructors.

Syntax:

```
JButton okBtn = new JButton("Click");
```

This constructor returns a button with the text Click on it.

```
JButton homeBtn = new JButton(carIcon);
```

Returns a button with a car Icon on it.

```
JButton homeBtn2 = new JButton(carIcon, "Car");
```

Returns a button with the car icon and text as Car.

Display:

JLabel

- We use JLabel class to render a read-only text label or images on the UI. It does not generate any event.

Syntax:

```
JLabel textLabel = new JLabel("This is 1st L...");
```

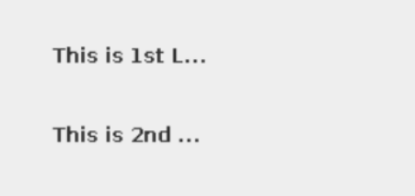
- This constructor returns a label with specified text.

```
JLabel imgLabel = new JLabel(carIcon);
```

- It returns a label with a car icon.
- The JLabel Contains four constructors. They are as follows:

1. JLabel()
2. JLabel(String s)
3. JLabel(Icon i)
4. JLabel(String s, Icon i, int horizontalAlignment)

Display:



JTextField

- The JTextField renders an editable single-line text box. Users can input non-formatted text in the box. We can initialize the text field by calling its constructor and passing an optional integer parameter. This parameter sets the box width measured by the number of columns. Also, it does not limit the number of characters that can be input into the box.

Syntax:

```
JTextField txtBox = new JTextField(50);
```

- It is the most widely used text component. It has three constructors:

1. JTextField(int cols)
2. JTextField(String str, int cols)
3. JTextField(String str)

Note: cols represent the number of columns in the text field.

Display:

JCheckBox

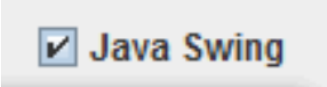
- The JCheckBox renders a check-box with a label. The check-box has two states, i.e., on and off. On selecting, the state is set to "on," and a small tick is displayed inside the box.

Syntax:

```
CheckBox chkBox = new JCheckBox("Java Swing", true);
```

- It returns a checkbox with the label Pepperoni pizza. Notice the second parameter in the constructor. It is a boolean value that denotes the default state of the check-box. True means the check-box defaults to the "on" state.

Display:



JRadioButton

- A radio button is a group of related buttons from which we can select only one. We use JRadioButton class to create a radio button in Frames and render a group of radio buttons in the UI. Users can select one choice from the group.

Syntax:

```
JRadioButton jrb = new JRadioButton("Easy");
```

Display:

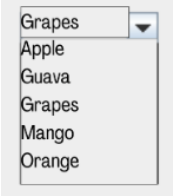
JComboBox

- The combo box is a combination of text fields and a drop-down list. We use JComboBox component to create a combo box in Swing.

Syntax:

```
JComboBox jcb = new JComboBox(name);
```

Display:



JTextArea

- In Java, the Swing toolkit contains a JTextArea Class. It is under package javax.swing.JTextArea class. It is used for displaying multiple-line text.

Declaration:

```
public class JTextArea extends JTextComponent
```

Syntax:

```
JTextArea textArea_area=new JTextArea("Ninja! please write something in the text area.");
```

- The JTextArea Contains four constructors. They are as follows:

1. JTextArea()
2. JTextArea(String s)
3. JTextArea(int row, int column)
4. JTextArea(String s, int row, int column)

Display:

JPasswordField

- In Java, the Swing toolkit contains a JPasswordField Class. It is under package javax.swing.JPasswordField class. It is specifically used for the password, and we can edit them.

Declaration:

```
public class JPasswordField extends JTextField
```

Syntax:

```
JPasswordField password = new JPasswordField();
```

- The JPasswordFieldContains 4 constructors. They are as follows:

1. JPasswordField()
2. JPasswordField(int columns)
3. JPasswordField(String text)
4. JPasswordField(String text, int columns)

Display:

JTable

- In Java, the Swing toolkit contains a JTable Class. It is under package javax.swing.JTable class. It is used to draw a table to display data.

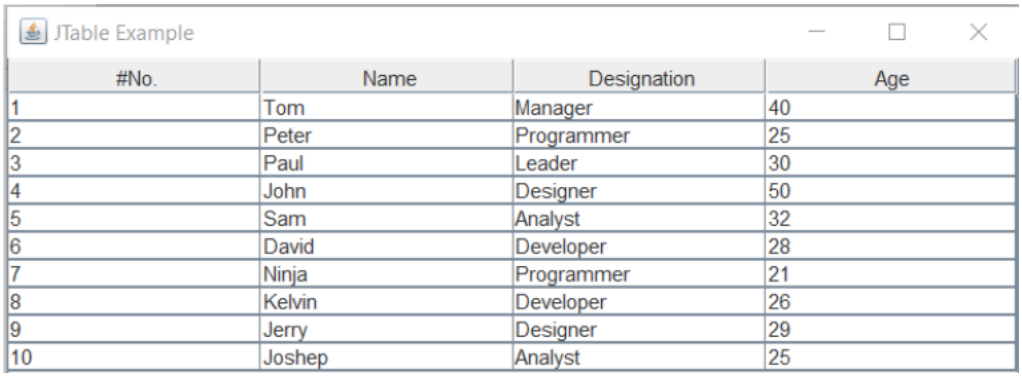
Syntax:

```
JTable table = new JTable(table_data, table_column);
```

- The JTable contains two constructors. They are as follows:

1. JTable()
2. JTable(Object[][] rows, Object[] columns)

Display:



#No.	Name	Designation	Age
1	Tom	Manager	40
2	Peter	Programmer	25
3	Paul	Leader	30
4	John	Designer	50
5	Sam	Analyst	32
6	David	Developer	28
7	Ninja	Programmer	21
8	Kelvin	Developer	26
9	Jerry	Designer	29
10	Joshep	Analyst	25

JList

In Java, the Swing toolkit contains a JList Class. It is under package javax.swing.JList class. It is used to represent a list of items together. We can select one or more than one items from the list.

Declaration:

```
public class JList extends JComponent implements Scrollable, Accessible
```

Syntax:

```
DefaultListModel<String> list1 = new DefaultListModel<>();
```

```
list1.addElement("Apple");
list1.addElement("Orange");
list1.addElement("Banan");
list1.addElement("Grape");
JList<String> list_1 = new JList<>(list1);
```

The JListContains 3 constructors. They are as follows:

1. JList()
2. JList(ary[] listData)
3. JList(ListModel<ary> dataModel)

JOptionPane

- In Java, the Swing toolkit contains a JOptionPane Class. It is under package javax.swing.JOptionPane class. It is used for creating dialog boxes for displaying a message, confirm box, or input dialog box.
- Declaration:
- public class JOptionPane extends JComponent implements Accessible

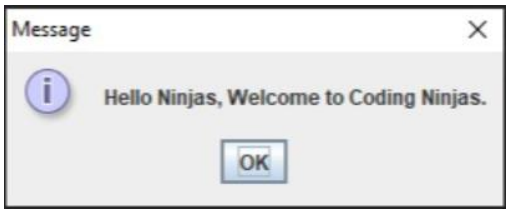
Syntax:

```
JOptionPane.showMessageDialog(jframe_obj, "Good Morning, Evening & Night.");
```

- The JOptionPaneContains 3 constructors. They are as following:

1. JOptionPane()
2. JOptionPane(Object message)
3. JOptionPane(Object message, intmessageType)

Display:



JScrollBar

- In Java, the Swing toolkit contains a JScrollBar class. It is under package javax.swing.JScrollBar class. It is used for adding horizontal and vertical scrollbars.

Declaration:

- public class JScrollBar extends JComponent implements Adjustable, Accessible

Syntax:

```
JScrollBar scrollBar = new JScrollBar();
```

- The JScrollBarContains 3 constructors. They are as following:

1. JScrollBar()
2. JScrollBar(int orientation)
3. JScrollBar(int orientation, int value, int extent, int min_, intmax_)

Display:

JMenuBar, JMenu and JMenuItem

- In Java, the Swing toolkit contains a JMenuBar, JMenu, and JMenuItem class. It is under package javax.swing.JMenuBar, javax.swing.JMenu and javax.swing.JMenuItem class. The JMenuBar class is used for displaying menubar on the frame. The JMenu Object is used to pull down the menu

bar's components. The JMenuItem Object is used for adding the labeled menu item.

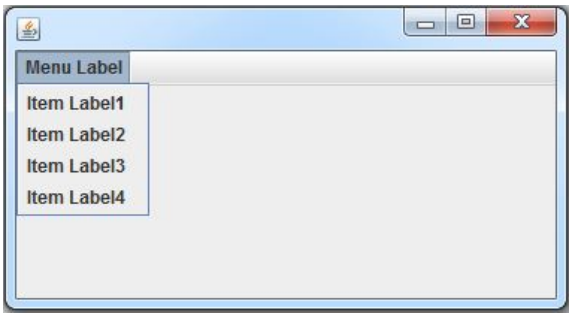
JMenuBar, JMenu and JMenuItem Declarations:

```
public class JMenuBar extends JComponent implements MenuElement, Accessible

public class JMenu extends JMenuItem implements MenuElement, Accessible

public class JMenuItem extends AbstractButton implements Accessible, MenuElement
```

Syntax:

```
JMenuBar menu_bar = new JMenuBar();
JMenu menu = new JMenu("Menu");
menuItem1 = new JMenuItem("Never");
menuItem2 = new JMenuItem("Stop");
menuItem3 = new JMenuItem("Learing");
menu.add(menuItem1);
menu.add(menuItem2);
menu.add(menuItem3);
```

Display:



JPopupMenu

- In Java, the Swing toolkit contains a JPopupMenu Class. It is under package javax.swing.JPopupMenu class. It is used for creating popups dynamically on a specified position.

Declaration:

```
public class JPopupMenu extends JComponent implements Accessible, MenuElement
```

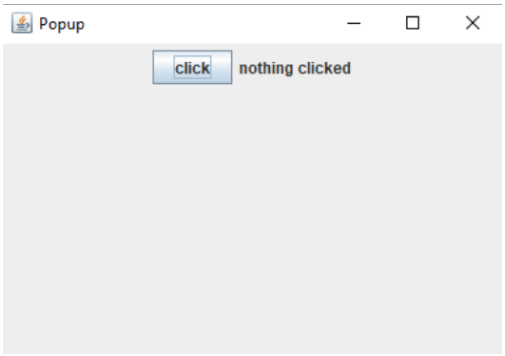
Syntax:

```
final JPopupMenu popupmenu1 = new JPopupMenu("Edit");
```

- The JPopupMenuContains 2 constructors. They are as follows:

1. JPopupMenu()
2. JPopupMenu(String label)

Display:



JCheckBoxMenuItem

- In Java, the Swing toolkit contains a JCheckBoxMenuItem Class. It is under package javax.swing.JCheckBoxMenuItem class. It is used to create a checkbox on a menu.

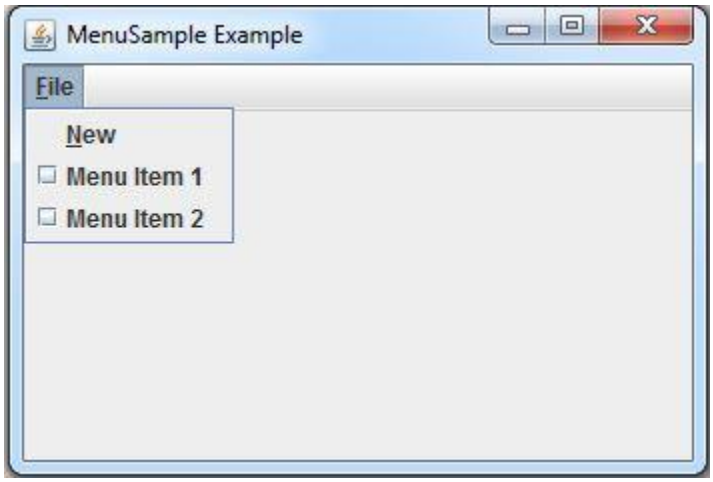
Syntax:

```
JCheckBoxMenuItem item = new JCheckBoxMenuItem("Option_1");
```

- The JCheckBoxMenuItemContains 2 constructors. They are as following:

1. JCheckBoxMenuItem()
2. JCheckBoxMenuItem(Action a)
3. JCheckBoxMenuItem(Icon icon)
4. JCheckBoxMenuItem(String text)
5. JCheckBoxMenuItem(String text, boolean b)
6. JCheckBoxMenuItem(String text, Icon icon)
7. JCheckBoxMenuItem(String text, Icon icon, boolean b)

Display:



JSeparator

- In Java, the Swing toolkit contains a JSeparator Class. It is under package javax.swing.JSeparator class. It is used for creating a separator line between two components.

Declaration:

```
public class JSeparator extends JComponent implements SwingConstants, Accessible
```

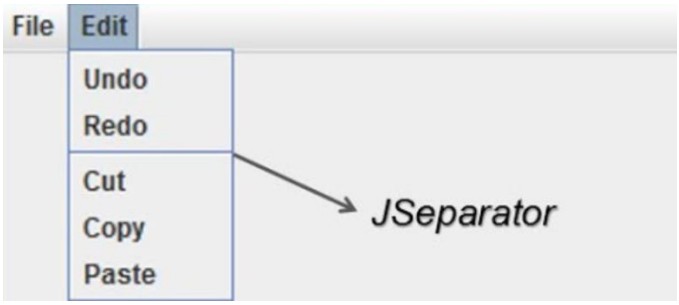
Syntax:

```
jmenu_Item.addSeparator();
```

- The JSeparatorContains 2 constructors. They are as following:

1. JSeparator()
2. JSeparator(int orientation)

Display:



JProgressBar

- In Java, the Swing toolkit contains a JProgressBar Class. It is under package javax.swing.JProgressBarclass. It is used for creating a progress bar of a task.

Declaration:

```
public class JProgressBar extends JComponent implements SwingConstants, Accessible
```

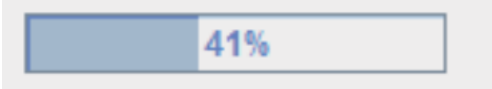
Syntax:

```
JProgressBar progressBar = new JProgressBar(0,2000);
```

The JProgressBarContains 4 constructors. They are as following:

1. JProgressBar()
2. JProgressBar(int min, int max)
3. JProgressBar(int orient)
4. JProgressBar(int orient, int min, int max)

Display:



JTree

- In Java, the Swing toolkit contains a JTree Class. It is under package javax.swing.JTreeclass. It is used for creating tree-structured data. It is a very complex component.

Declaration:

```
public class JTree extends JComponent implements Scrollable, Accessible
```

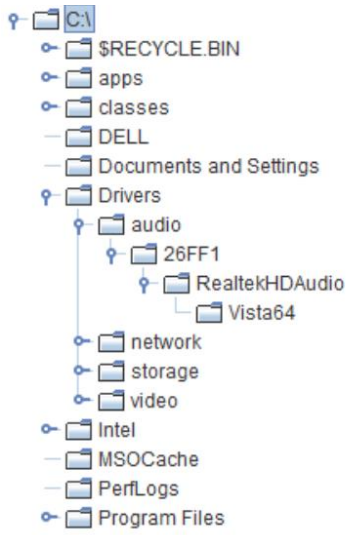
Syntax:

```
JTree tree = new JTree(tree_style);
```

- The JTreeContains 3 constructors. They are as follows:

1. JTree()
2. JTree(Object[] value)
3. JTree(TreeNode root)

Display:



JFileChooser

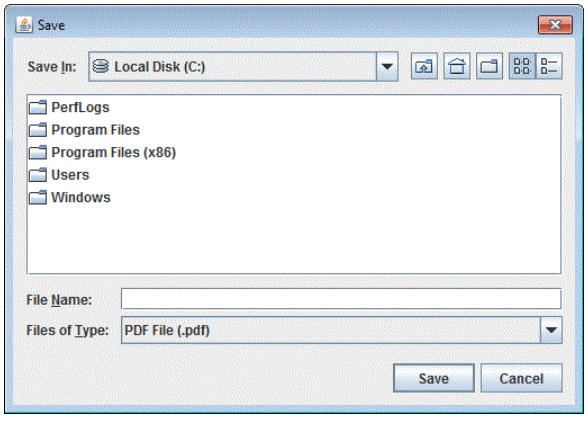
- The JFileChooser class renders a file selection utility. This component lets a user select a file from the local system.

Syntax:

```
JFileChooser fileChooser = new JFileChooser();
JButton fileBtn = new JButton("Select File");
fileBtn.AddEventListner(new ActionListner(){
        fileChooser.showOpenDialog();
})
var selectedFile = fileChooser.getSelectedFile();
```

- The above code creates a file chooser dialog and attaches it to the button. The button click would open the file chooser dialog. The selected file is returned through the get file method chosen.

Display:



JTabbedPane

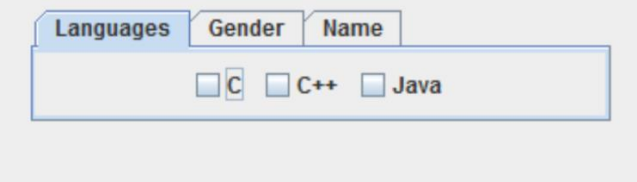
- The JTabbedPane is another beneficial component that lets the user switch between tabs in an application. It is a handy utility as it allows users to browse more content without navigating to different pages.

Syntax:

```
JTabbedPane jtabbedPane = new JTabbedPane();
jtabbedPane.addTab("Tab_1", new JPanel());
jtabbedPane.addTab("Tab_2", new JPanel());
```

The above code creates a two-tabbed panel with headings Tab_1 and Tab_2.

Display:



JSlider

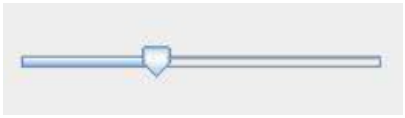
The JSlider component displays a slider that the user can drag to change its value. The constructor takes three arguments: minimum, maximum, and initial.

Syntax:

```
JSlider volumeSlider = new JSlider(0, 100, 20);
var volumeLevel = volumeSlider.getValue();
```

The above code makes a slider from 0 to 100 with an initial value set to 20. The value specified by the user is returned by the getValue method.

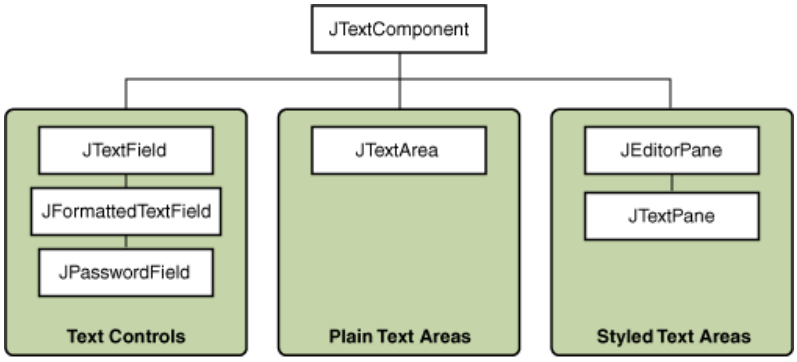
Display:



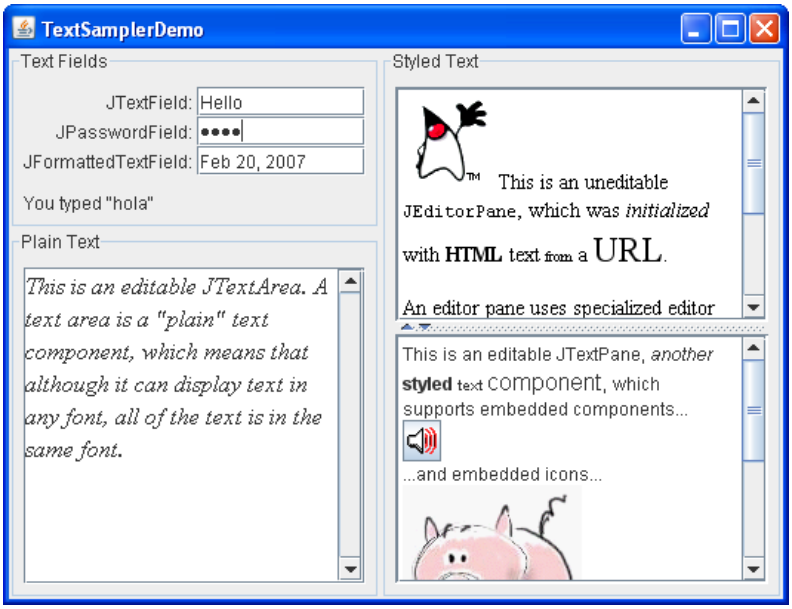
3. Using Text Components

- This section provides background information you might need when using Swing text components. If you intend to use an unstyled text component — a text field, password field, formatted text field, or text area — go to its how-to page and return here only if necessary. If you intend to use a styled text component, see How to Use Editor Panes and Text Panes, and read this section as well. If you do not know which component you need, read on.

- Swing text components display text and optionally allow the user to edit the text. Programs need text components for tasks ranging from the straightforward (enter a word and press Enter) to the complex (display and edit styled text with embedded images in an Asian language).
- Swing provides six text components, along with supporting classes and interfaces that meet even the most complex text requirements. In spite of their different uses and capabilities, all Swing text components inherit from the same superclass, `JTextComponent`, which provides a highly-configurable and powerful foundation for text manipulation.
- The following figure shows the `JTextComponent` hierarchy.



The following picture shows an application called `TextSamplerDemo` that uses each Swing text component.



Group	Description	Swing Classes

Text Controls	Also known simply as text fields, text controls can display only one line of editable text. Like buttons, they generate action events. Use them to get a small amount of textual information from the user and perform an action after the text entry is complete.	`JTextField` and its subclasses `JPasswordField` and `JFormattedTextField`
Plain Text Areas	`JTextArea` can display multiple lines of editable text. Although a text area can display text in any font, all of the text is in the same font. Use a text area to allow the user to enter unformatted text of any length or to display unformatted help information.	`JTextArea`
Styled Text Areas	A styled text component can display editable text using more than one font. Some styled text components allow embedded images and even embedded components. Styled text components are powerful and multi-faceted components suitable for high-end needs, and offer more avenues for customization than the other text components. Because they are so powerful and flexible, styled text components typically require more initial programming to set up and use. One exception is that editor panes can be easily loaded with formatted text from a URL, which makes them useful for displaying uneditable help information.	`JEditorPane` and its subclass `JTextPane`

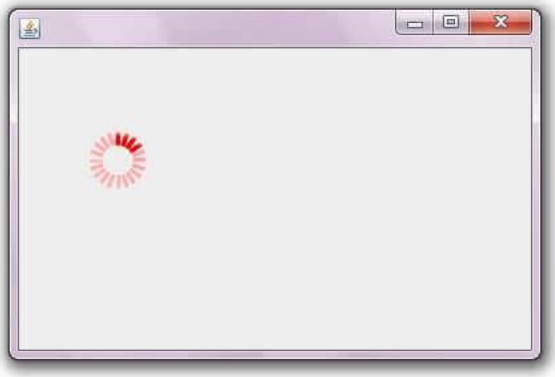
4. Displaying image in swing:

- For displaying image, we can use the method drawImage() of Graphics class.
- drawImage() method is used draw the specified image.

Example of displaying image in swing:

```
import java.awt.*;
import javax.swing.JFrame;

public class MyCanvas extends Canvas{

  public void paint(Graphics g) {
```

```
        Toolkit t=Toolkit.getDefaultToolkit();
```



```
        Image i=t.getImage("p3.gif");
        g.drawImage(i, 120,100,this);

    }
        public static void main(String[] args) {
        MyCanvas m=new MyCanvas();
        JFrame f=new JFrame();
        f.add(m);
        f.setSize(400,400);
        f.setVisible(true);
    }

}
```

OUTPUT:

Java JTextField

The object of a JTextField class is a text component that allows the editing of a single line text. It inherits JTextComponent class.

JTextField class declaration

Let's see the declaration for javax.swing.JTextField class.

public class JTextField **extends** JTextComponent **implements** SwingConstants .

Commonly used Constructors:

Constructor	Description
JTextField()	Creates a new TextField
JTextField(String text)	Creates a new TextField initialized with the specified text.
JTextField(String text, int columns)	Creates a new TextField initialized with the specified text and columns.
JTextField(int columns)	Creates a new empty TextField with the specified number of columns.

Commonly used Methods:

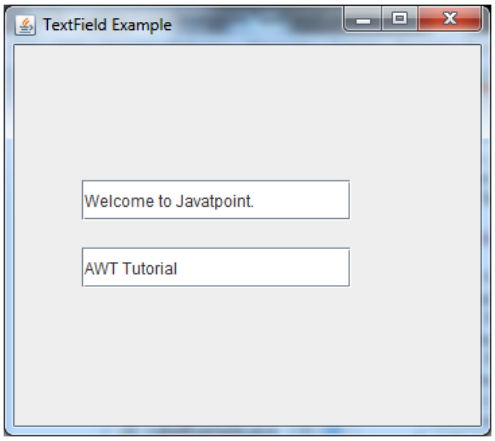
Methods	Description
void addActionListener(ActionListener l)	It is used to add the specified action listener to receive action events from this textfield.
Action getAction()	It returns the currently set Action for this ActionEvent source, or null if no Action is set.
void setFont(Font f)	It is used to set the current font.
void removeActionListener(ActionListener l)	It is used to remove the specified action listener so that it no longer receives action events from this textfield.

Java JTextField Example

```
import javax.swing.*;
class TextFieldExample
{
public static void main(String args[])
    {
    JFrame f= new JFrame("TextField Example");
    JTextField t1,t2;
    t1=new JTextField("Welcome to Javatpoint.");
    t1.setBounds(50,100, 200,30);
    t2=new JTextField("AWT Tutorial");
    t2.setBounds(50,150, 200,30);
```

```
f.add(t1); f.add(t2);
f.setSize(400,400);
f.setLayout(null);
f.setVisible(true);
}
}
```

Output:

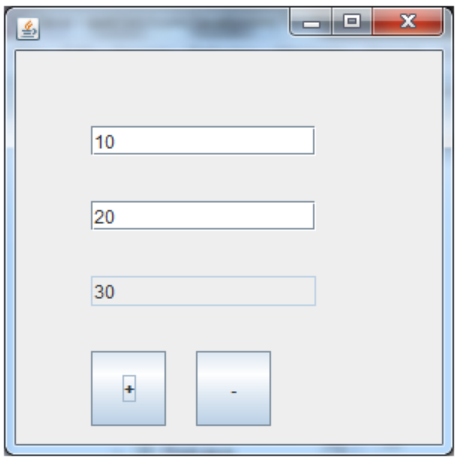


Java JTextField Example with ActionListener

```
import javax.swing.*;
import java.awt.event.*;
public class TextFieldExample implements ActionListener{
  JTextField tf1,tf2,tf3;
  JButton b1,b2;
  TextFieldExample(){
    JFrame f= new JFrame();
    tf1=new JTextField();
    tf1.setBounds(50,50,150,20);
    tf2=new JTextField();
    tf2.setBounds(50,100,150,20);
    tf3=new JTextField();
    tf3.setBounds(50,150,150,20);
    tf3.setEditable(false);
```

```
        b1=new JButton("+");
        b1.setBounds(50,200,50,50);
        b2=new JButton("-");
        b2.setBounds(120,200,50,50);
        b1.addActionListener(this);
        b2.addActionListener(this);
        f.add(tf1);f.add(tf2);f.add(tf3);f.add(b1);f.add(b2);
        f.setSize(300,300);
        f.setLayout(null);
        f.setVisible(true);
    }
    public void actionPerformed(ActionEvent e) {
        String s1=tf1.getText();
        String s2=tf2.getText();
        int a=Integer.parseInt(s1);
        int b=Integer.parseInt(s2);
        int c=0;
        if(e.getSource()==b1){
            c=a+b;
        }else if(e.getSource()==b2){
            c=a-b;
        }
        String result=String.valueOf(c);
        tf3.setText(result);
    }
public static void main(String[] args) {
    new TextFieldExample();
} }
```

Output:



5. Java Event Handling

Changing the state of an object is known as an event. For example, click on button, dragging mouse etc. Th
event classes and Listener interfaces for event handling.

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

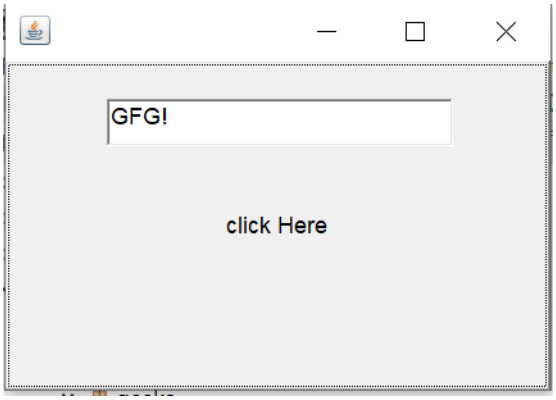
Event Handling Within Class

- Java

```
// Java program to demonstrate the
// event handling within the class

import java.awt.*;
import java.awt.event.*;

class GFG extends Frame implements ActionListener {

    TextField textField;

    GFGTop()
    {
        // Component Creation
        textField = new TextField();

        // setBounds method is used to provide
```

```
        // position and size of the component
        textField.setBounds(60, 50, 180, 25);
        Button button = new Button("click Here");
        button.setBounds(100, 120, 80, 30);

        // Registering component with listener
        // this refers to current instance
        button.addActionListener(this);

        // add Components
        add(textField);
        add(button);

        // set visibility
        setVisible(true);
    }

    // implementing method of actionListener
    public void actionPerformed(ActionEvent e)
    {
        // Setting text to field
        textField.setText("GFG!");
    }

    public static void main(String[] args)
    {
      new GFGTop();
    }
}
```

Output



After Clicking, the text field value is set to GFG!

Explanation

- Firstly extend the class with the applet and implement the respective listener.
- Create Text-Field and Button components.
- Registered the button component with respective event. i.e. ActionEvent by addActionListener().
- In the end, implement the abstract method.

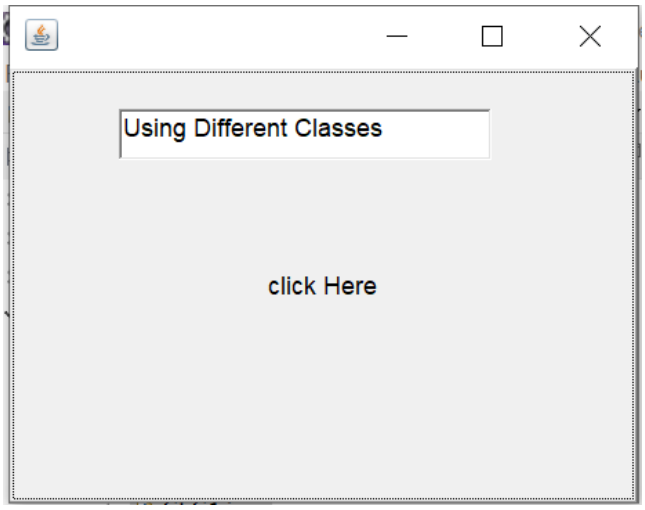
Event Handling by Other Class

- Java

```
// Java program to demonstrate the
// event handling by the other class

import java.awt.*;
import java.awt.event.*;

class GFG1 extends Frame {

    TextField textField;

    GFG2()
    {
        // Component Creation
        textField = new TextField();

        // setBounds method is used to provide
```

```
        // position and size of component
        textField.setBounds(60, 50, 180, 25);
        Button button = new Button("click Here");
        button.setBounds(100, 120, 80, 30);

        Other other = new Other(this);

        // Registering component with listener
        // Passing other class as reference
        button.addActionListener(other);

        // add Components
        add(textField);
        add(button);

        // set visibility
        setVisible(true);
    }

    public static void main(String[] args)
    {
      new GFG2();
    }
}

/// import necessary packages
import java.awt.event.*;

// implements the listener interface
class Other implements ActionListener {

    GFG2 gfgObj;

    Other(GFG1 gfgObj)
    {
      this.gfgObj = gfgObj;
    }

    public void actionPerformed(ActionEvent e)
    {
        // setting text from different class
        gfgObj.textField.setText("Using Different Classes");
}}
```

Output



Handling event from different class

Event Handling By Anonymous Class

- Java

```
// Java program to demonstrate the
// event handling by the anonymous class

import java.awt.*;
import java.awt.event.*;

class GFG3 extends Frame {

    TextField textField;

    GFG3()
    {
        // Component Creation
        textField = new TextField();

        // setBounds method is used to provide
        // position and size of component
```

```
        textField.setBounds(60, 50, 180, 25);
        Button button = new Button("click Here");
        button.setBounds(100, 120, 80, 30);

        // Registering component with listener anonymously
        button.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                // Setting text to field
                textField.setText("Anonymous");
            }
        });

        // add Components
        add(textField);
        add(button);

        // set visibility
        setVisible(true);
    }

    public static void main(String[] args)
    {
        new GFG3();
    }
```

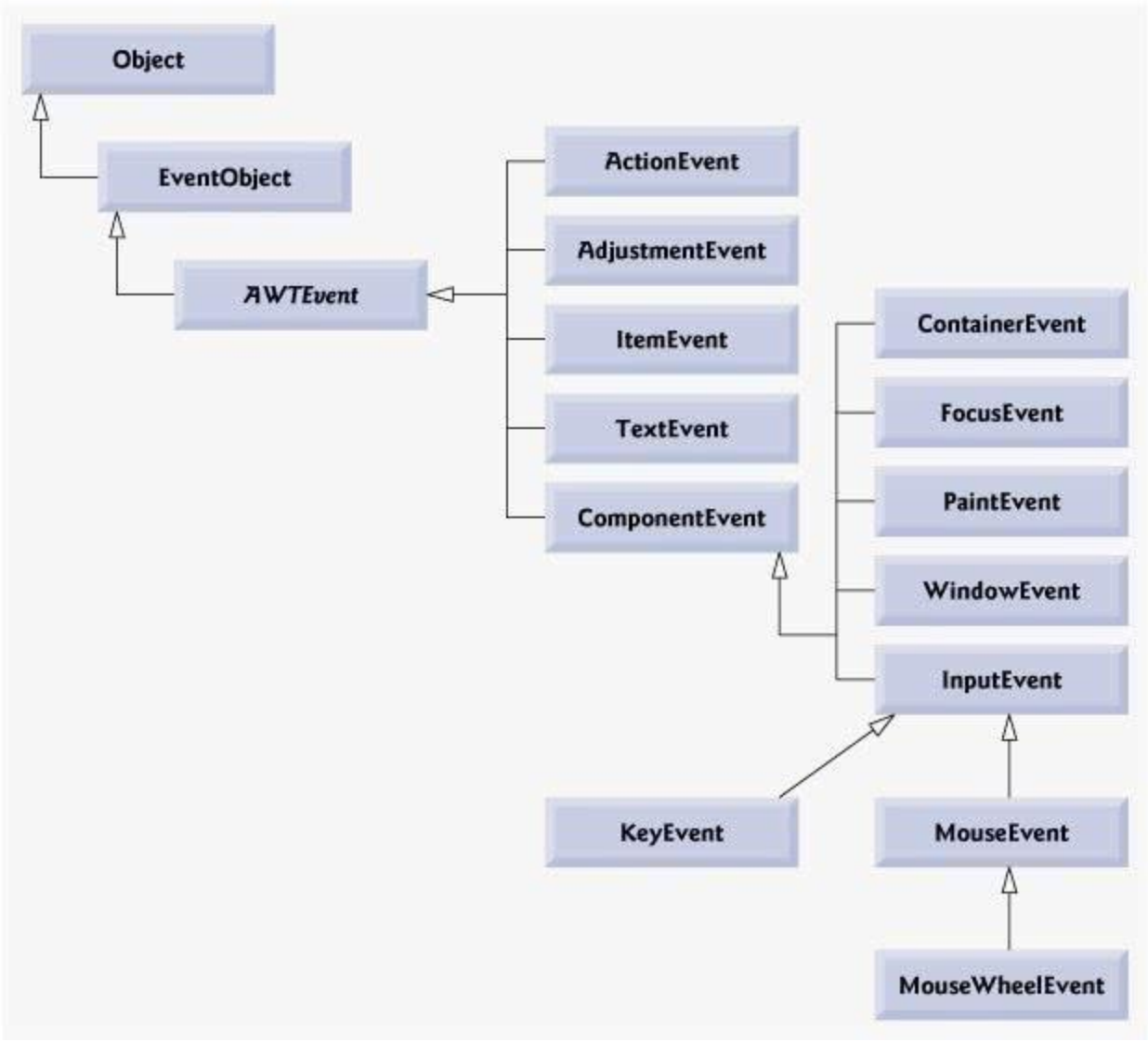
}

Output

Handling anonymously

Common GUI Event Types and Listener Interfaces

[View full size image]

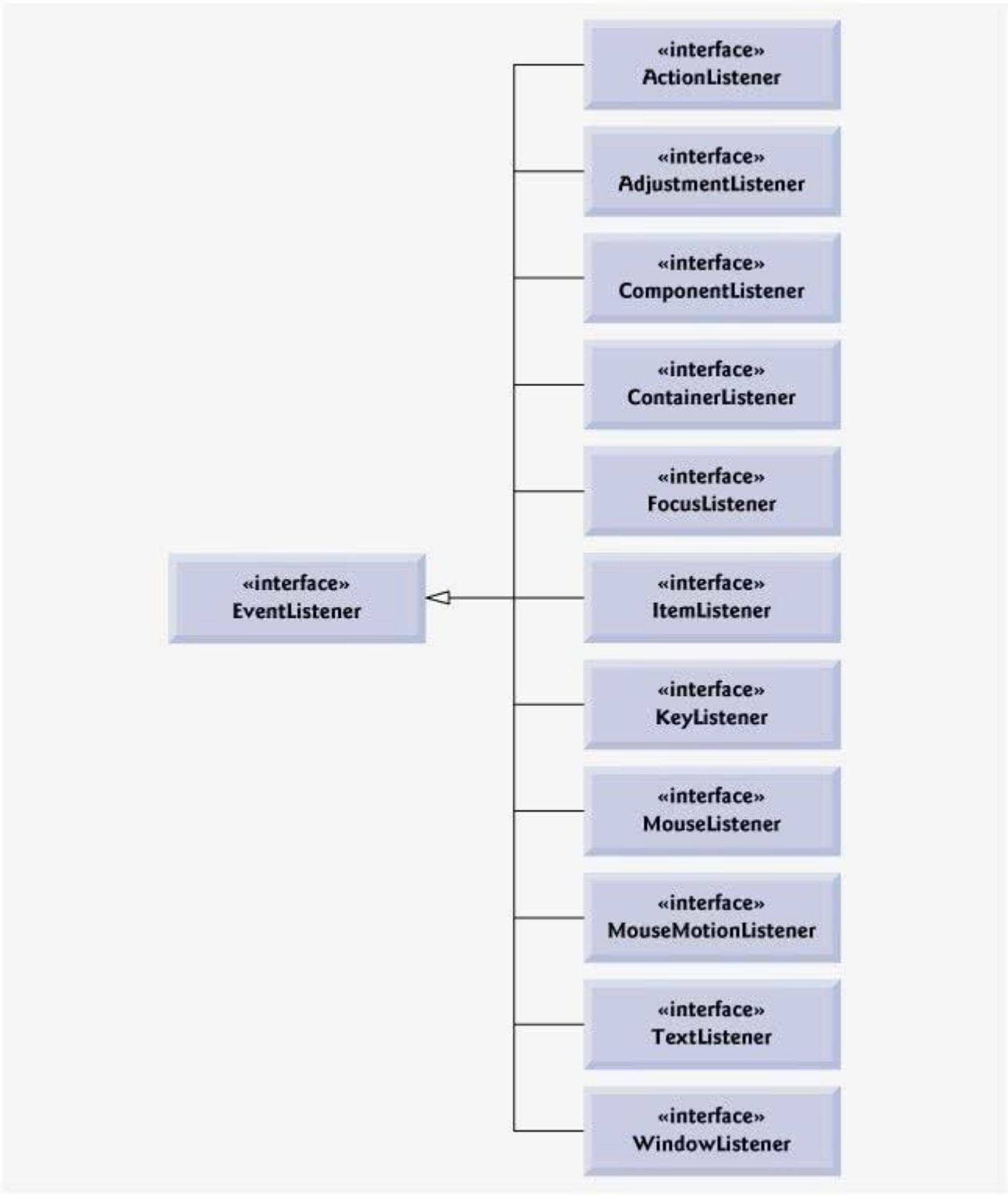


Let's summarize the three parts to the event-handling mechanism that you saw in the event source, the event object and the event listener. The event source is

the particular GUI component with which the user interacts. The event object encapsulates information about the event that occurred, such as a reference to the event source and any event-specific information that may be required by the event listener for it to handle the event. The event listener is an object that is notified by the event source when an event occurs; in effect, it "listens" for an event and one of its methods executes in response to the event. A method of the event listener receives an event object when the event listener is notified of the event. The event listener then uses the event object to respond to the event. The event-handling model described here is known as the **delegation event model**an event's processing is delegated to a particular object (the event listener) in the app. For each event-object type, there is typically a corresponding event-listener interface. An event listener for a GUI event is an object of a class that implements one or more of the event-listener interfaces from packages `java.awt.event` and `javax.swing.event`. Many of the event-listener types are common to both Swing and AWT components. Such types are declared in package `java.awt.event`, and some of them are shown in Screenshot. Additional event-listener types that are specific to Swing components are declared in package `javax.swing.event`.

Screenshot Some common event-listener interfaces of package `java.awt.event`.

[View full size
image]


«interface»
EventListener
«interface»
ActionListener
«interface»
AdjustmentListener
«interface»
ComponentListener
«interface»
ContainerListener
«interface»
FocusListener
«interface»
ItemListener
«interface»
KeyListener
«interface»
MouseListener
«interface»
MouseMotionListener
«interface»
TextListener
«interface»
WindowListener

Each event-listener interface specifies one or more event-handling methods that must be declared in the class that implements the interface. Recall from that any class which implements an interface must declare all the `abstract` methods of that interface; otherwise, the class is an `abstract` class and cannot be used to create objects.

When an event occurs, the GUI component with which the user interacted notifies its registered listeners by calling each listener's appropriate event-handling method. For example, when the user presses the Enter key in a `JTextField`, the registered listener's `actionPerformed` method is called. How did the event handler get registered? How does the GUI component know to call `actionPerformed` rather than another event-handling method? We answer these questions and diagram the interaction in the next section.

Previous Next

JAVA - FILES AND I/O

• The java.io package contains nearly every class you might ever need to perform input and output

• (I/O) in Java. All these streams represent an input source and an output destination. The stream in
• the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

• A stream can be defined as a sequence of data. There are two kinds of Streams –

• InPutStream − The InputStream is used to read data from a source.

• OutPutStream − The OutputStream is used for writing data to a destination.

• Java provides strong but flexible support for I/O related to files and networks but this tutorial

• covers very basic functionality related to streams and I/O. We will see the most commonly used.

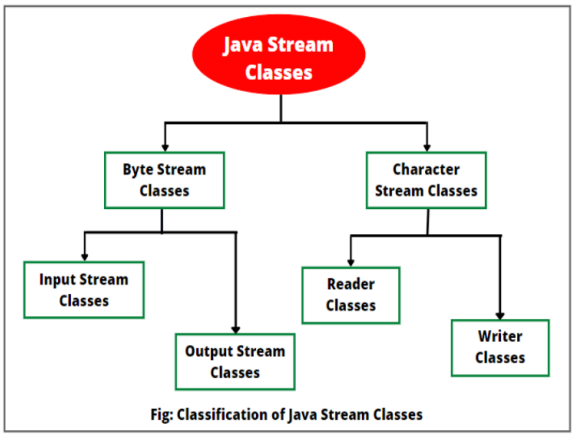
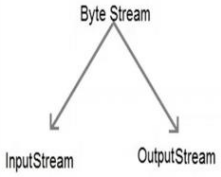


Fig: Classification of Java Stream Classes

Java Byte Streams



- The InputStream and OutputStream classes and their subclasses are used for dealing with data in byte format.

Input / Output Streams

- Stream is a path along which the data flows.
- Every stream has a source and destination.
- The source and destination for a stream may be two other streams in the same program or two different programs(client and server programs) or two physical devices (Hard disk and console screen).
- Streams are input streams and output streams.
- An output stream writes data into a file.
- An input stream is used to read data from a file.
- Java encapsulates Stream under java.io package.
- Java defines two types of streams.
  - Byte Stream : It provides a convenient means for handling input and output of byte.
  - Character Stream : It provides a convenient means for handling input and output of characters. Character stream uses Unicode.

Subclasses of OutputStream class

Name of the class	Functionality
OutputStream	Performing output operations
BufferedOutputStream	Buffering output
ByteArrayOutputStream	Writing to an array
FilterOutputStream	Filtering the output
FileOutputStream	Writing to a file
PrintStream	Outputs the Unicode format of the primitive types to the console
PipedOutputStream	Writing to a pipe
DataOutputStream	Writing primitive data types

Program to write the user inputs to file using FileOutputStream class

```
import java.io.*;
import java.util.*;
class filedemo
{       public static void main(String args[]) throws IOException
        {
        Scanner s = new Scanner (System.in);
        String str;
        str = s.nextLine();
        int j;
        byte b[]=str.getBytes();
        System.out.println(b.length);
        FileOutputStream fos= new FileOutputStream("f:/s1.txt");
        for(j=0; j<b.length; j++)
        {
        fos.write(b[j]);
        }
        fos.close();
        }
}
```

```
>javac filedemo.java
>java filedemo
welcome to department of IT
27
To see the content of txt file
F:\>type s1.txt
welcome to department of IT
```


Methods in OutputStream class

Syntax of the method	Usage
write()	To write a byte to the output stream
write(byte[]b)	To write all bytes in the array b to the output stream
write(byte b[],int n, int m)	To write m bytes from array b starting from n^{th} byte.
close()	To close the output stream.
flush()	To flush (clear) the output stream.

Subclasses of InputStream class

- An input stream is used to read data from a file.

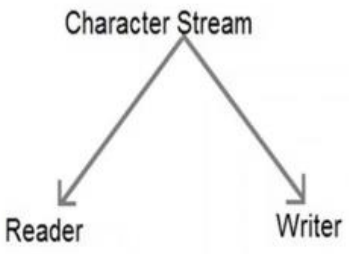
Name of the class	Functionality
InputStream	Performing input operations
BufferedInputStream	Buffering input
LineNumberInputStream	Keeping track of how many lines are read
ByteArrayInputStream	Reading from an array
FilterInputStream	Filtering the input
FileInputStream	Reading from a file
PipedInputStream	Reading from a pipe
DataInputStream	Reading primitive types

Methods in InputStream class

Syntax of the method	Usage
read()	To read a byte from the input stream
read(byte b[])	To read an array of b.length bytes into array b.
read(byte b[], int n, int m)	To read m bytes from b starting from nth byte
available()	To give the number of bytes available in the input.
skip(n)	To skip over and discard n bytes from the input stream.
reset()	To go back to the beginning of the stream.
close()	To close the input stream.

Java Character Stream classes

- **Character Stream :** It provides a convenient means for handling input and output of characters. Character stream uses Unicode.



Subclasses of Reader class

Name of the class	Functionality
Reader	Performing input operations
BufferedReader	Buffering input
LineNumberReader	Keeping track of line number
CharArrayReader	Reading from an array
FilterReader	Filtering the input
InputStreamReader	Translating a byte stream into a character stream
FileReader	Reading from a file
PipedReader	Reading from a pipe
StringReader	Reading from a string

Program to read the contents of file using FileReader class and display into the monit

```
import java.io.*;
class filedemo1
{
public static void main(String args[]) throws IOException
{
  FileReader fr= new FileReader("f:/s2.txt");
  int c;
  while( (c=fr.read())!=-1)   // -1 represents EOF
  {
    System.out.print((char)c);
  }
  fr.close();
}
}
```

```
>javac filedemo1.java
>java filedemo1
Department of Information Technology.
```

Find the output of the Program

```
import java.io.*;
class filedemo2
{
public static void main(String args[]) throws IOException
{
    FileWriter fw= new FileWriter("f:/sample1.txt");
    for(char i=65;i<=90;i++)
    {
    fw.write(i);
    }
    fw.close();
}
}
```

ASCII value of A to Z – 65 to 90

```
>javac filedemo2.java
>java filedemo2
>type sample1.txt
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Differences :

Character Streams	Byte Streams
These handle data in 16 bit Unicode.	These handle data in bytes (8 bits).
Common classes are FileReader and FileWriter	Common classes are FileInputStream and FileOutputStream
Works with character data (i.e. read and write text data only)	Works with non-character data (i.e. store characters, videos, audios, images etc.)

APPLET

Introduction:

- The applet is a special type of program that is embedded in the webpage to generate dynamic content. It runs inside the browser and works on the client side.

Advantages of Applet:

- There are many advantages of the applet. They are as follows:
- It works on the client side so less response time.
- Secured
- It can be executed by browsers running under many platforms, including Linux, Windows, Mac OS, etc.

A drawback of Applet:

- The plugin is required in the client browser to execute the applet.

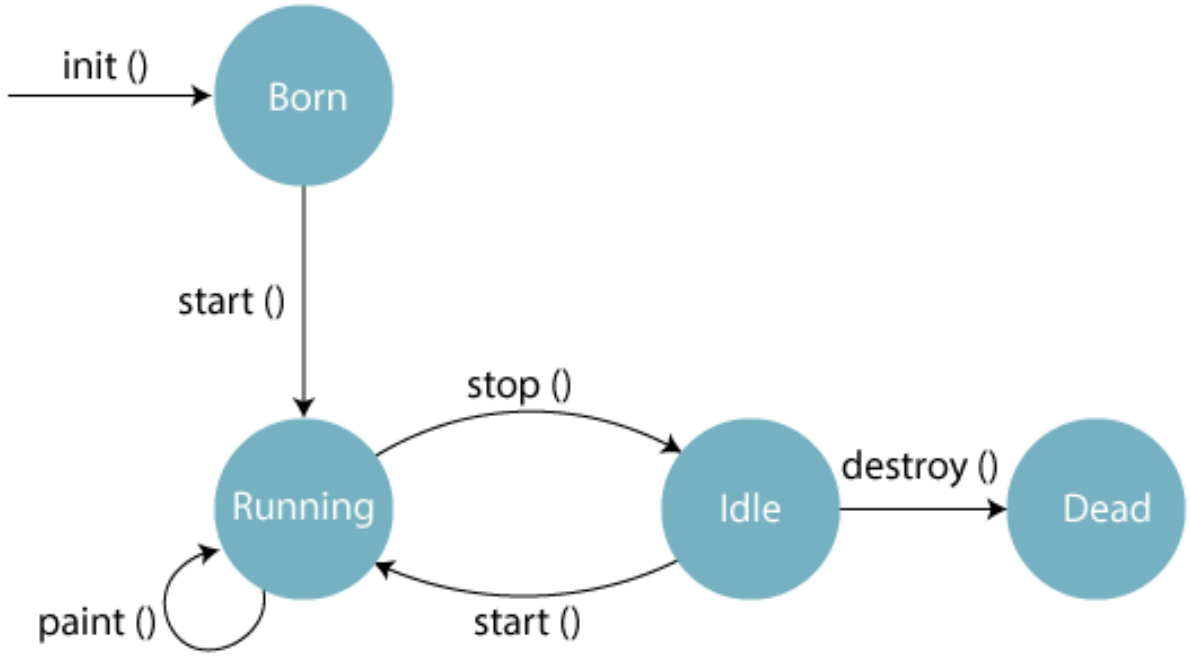
APPLET LIFE CYCLE

- In Java, an applet is a special type of program embedded in the web page to generate dynamic content. The applet is a class in Java.

- The applet life cycle can be defined as the process of how the object is created, started, stopped, and destroyed during the entire execution of its application. It basically has five core methods namely init(), start(), stop(), paint(), and destroy(). These methods are invoked by the browser to execute.

- Along with the browser, the applet also works on the client side, thus having less processing time.

Methods of Applet Life Cycle



FIVE METHODS OF LIFE CYCLE

- **init():** The init() method is the first method to run that initializes the applet. It can be invoked only once at the time of initialization. The web browser creates the initialized objects, i.e., the web browser (after checking the security settings) runs the init() method within the applet.

- **start():** The start() method contains the actual code of the applet and starts the applet. It is invoked immediately after the init() method is invoked. Every time the browser is loaded or refreshed, the start() method is invoked. It is also invoked whenever the applet is maximized, restored, or moving from one tab to another in the browser. It is in an inactive state until the init() method is invoked.

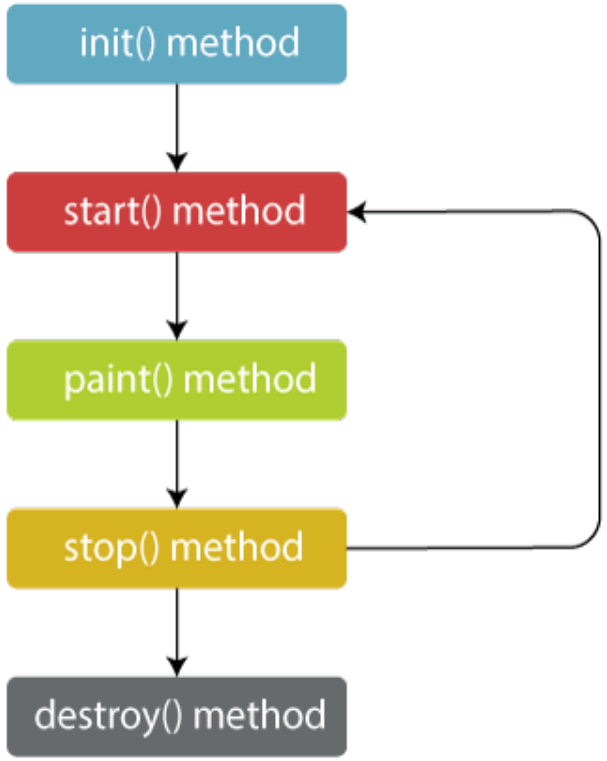
- **stop():** The stop() method stops the execution of the applet. The stop () method is invoked whenever the applet is stopped, minimized, or moving from one tab to another in the browser, the stop() method is invoked. When we go back to that page, the start() method is invoked again.

- **destroy():** The destroy() method destroys the applet after its work is done. It is invoked when the applet window is closed or when the tab containing the webpage is closed. It removes the applet object from memory and is executed only once. We cannot start the applet once it is destroyed.

- **paint():** The paint() method belongs to the Graphics class in Java. It is used to draw shapes like circle, square, trapezium, etc., in the applet. It is executed after the start() method and when the browser or applet windows are resized.

Flow of Applet Life Cycle:

- These methods are invoked by the browser automatically. There is no need to call them explicitly.



Syntax of the entire Applet Life Cycle in Java

```
class TestAppletLifeCycle extends Applet {
public void init() {
// initialized objects
}
public void start() {
// code to start the applet
}
public void paint(Graphics graphics) {
// draw the shapes
}
public void stop() {
// code to stop the applet
}
public void destroy() {
// code to destroy the applet
}
}
```

HTML TAGES

What are HTML Tags?

- HTML tags are simple instructions that tell a web browser how to format text. You can use tags to format italics, line breaks, objects, bullet points, and more.

- These tags live in every webpage's HTML (or the Hypertext Markup Language). But, HTML is the language of web pages

How Do Web Pages Read HTML Tags?

- Servers read HTML code to understand and render content. It will read the HTML from top to bottom, much like how you're reading this guide.
- You can use as many or as few tags as you like to format content. However, there are a few essential HTML tags and rules you'll need to follow.
- An HTML tag must contain three parts:
- An opening tag — this will start with a < > symbol
- Content — the short instructions on how to display the on-page element
- A closing tag — this will end with a </ > symbol
- However, some HTML tags can be unclosed. That means that the HTML tag does not need to be closed with a </ >. You'll typically use unclosed tags for metadata or line breaks.

Examples Of HTML Tags

** Paragraph Tag **

The and are the HTML tags and “Paragraph Tag” is the HTML element, i.e. the on-page text.

This tag formats any text between the opening tag and the closing tag as a standard paragraph or main body text.

<h2> Heading Tag </h2>

In this example, <h2> and </h2> are the HTML tags and “Heading Tag” is the HTML element, i.e. the on-page heading.

Using this tag will format any text between the opening <h2> tag and the closing </h2> tag as a Heading 2 (a type of subheading.)

** Bold Tag **

Here the and are the HTML tags and “Bold Tag” is the HTML element, i.e. the on page text.

This tag will format any text between the opening tag and the closing tag as bold.

** Italic Tag **

Here, the and are the HTML tags and “Italic Tag” is the HTML element (the on-page text.)

This tag will format any text between the opening tag and the closing tag as italic.

<u> Underline Tag</u>

Here the <u> and </u> are the HTML tags and “Underline Tag” is the HTML element, i.e. the on page text.

This tag will format any text between the opening <u> tag and the closing </u> tag as underlined.

The Most Common HTML Tags

Tag	Name	Code Example
<!--	comment	**<!--**This can be viewed in the HTML part of a document**-->**
<A -	anchor	**<A** HREF="http://www.yourdomain.com/">Visit Our Site**</A>**
<B>	bold	**<B>**Example**</B>**
<BODY>	body of document	**<BODY>**The content of your page**</BODY>**
 	line break	The contents of your page** **The contents of your page
<CENTER>	center	**<CENTER>**This will center your contents**</CENTER>**
<FONT>	font	**<FONT** FACE="Times New Roman">Example**</FONT>**

<HEAD>	heading of document	**<HEAD>**Contains elements describing the document**</HEAD>**
<HTML>	hypertext markup language	**<HTML>**<HEAD><META><TITLE>Title of your webpage</TITLE></HEAD><BODY>Webpage contents</BODY>**</HTML>**
<I>	italic	**<I>**Example**</I>**
<IMG>	image	**<IMG** SRC="Earth.gif" WIDTH="41" HEIGHT="41" BORDER="0" ALT="a sentence about your site">

EXAMPLE PROGRAM

HTML Images

HTML images are defined with the img tag:


```
<!DOCTYPE html>
<html>
<body>

<h2>HTML Images</h2>
<p>HTML images are defined with the img tag:</p>

<img src="w3schools.jpg" alt="W3Schools.com" width="104" height="142">

</body>
</html>
```

MULTITHREAD

UNIT 5

WHAT IS THREAD?

- A thread in Java is **the direction or path that is taken while a program is being executed**.

- Generally, all the programs have at least one thread, known as the main thread, that is provided by the JVM or Java Virtual Machine at the starting of the program's execution.

MULTITHREAD

- In Java, Multithreading refers to a process of executing two or more threads simultaneously for maximum utilization of the CPU.
- A thread in Java is a ***lightweight process*** requiring fewer resources to create and share the process resources.

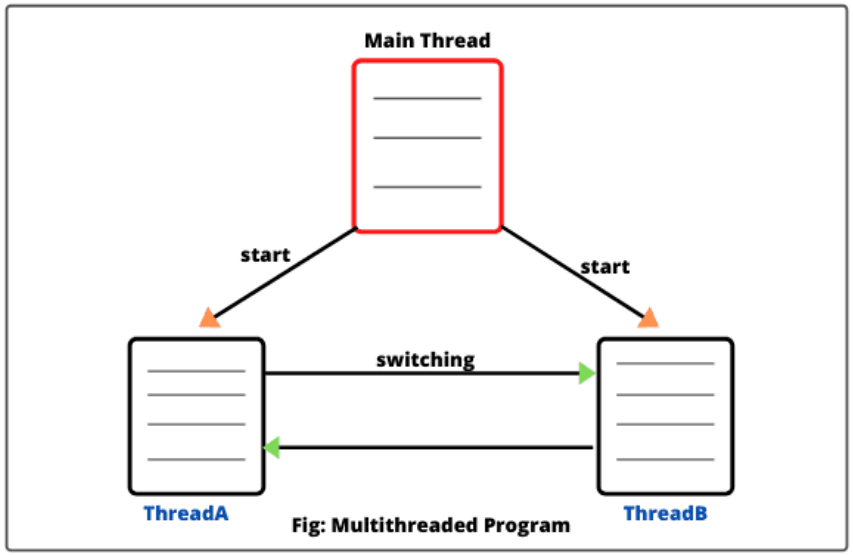


Fig: Multithreaded Program

Threads can be created by using two mechanisms :

- Extending the Thread class
- Implementing the Runnable Interface

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.

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.

Thread creation by extending the Thread class

OUTPUT:

Thread 15 is running
Thread 14 is running
Thread 16 is running
Thread 12 is running
Thread 11 is running
Thread 13 is running
Thread 18 is running
Thread 17 is running

```
// Java code for thread creation by extending
// the Thread class
class MultithreadingDemo extends Thread {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
public class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            MultithreadingDemo object
                = new MultithreadingDemo();
            object.start();
        }
    }
}
```

Thread creation by implementing the Runnable Interface

OUTPUT

Thread 13 is running
Thread 11 is running
Thread 12 is running
Thread 15 is running
Thread 14 is running
Thread 18 is running
Thread 17 is running
Thread 16 is running

```
/ Java code for thread creation by implementing
// the Runnable Interface
class MultithreadingDemo implements Runnable {
    public void run()
    {
        try {
            // Displaying the thread that is running
            System.out.println(
                "Thread " + Thread.currentThread().getId()
                + " is running");
        }
        catch (Exception e) {
            // Throwing an exception
            System.out.println("Exception is caught");
        }
    }
}

// Main Class
class Multithread {
    public static void main(String[] args)
    {
        int n = 8; // Number of threads
        for (int i = 0; i < n; i++) {
            Thread object
                = new Thread(new MultithreadingDemo());
            object.start();
        }
    }
}
```

THREAD LIFE CYCLE

Life Cycle of a Thread

- Every thread moves through several states from its creation to its termination.
- The possible states of a thread are
  - Newborn state
  - Runnable (ready) state
  - Running state
  - Blocked (waiting) state
  - Dead state

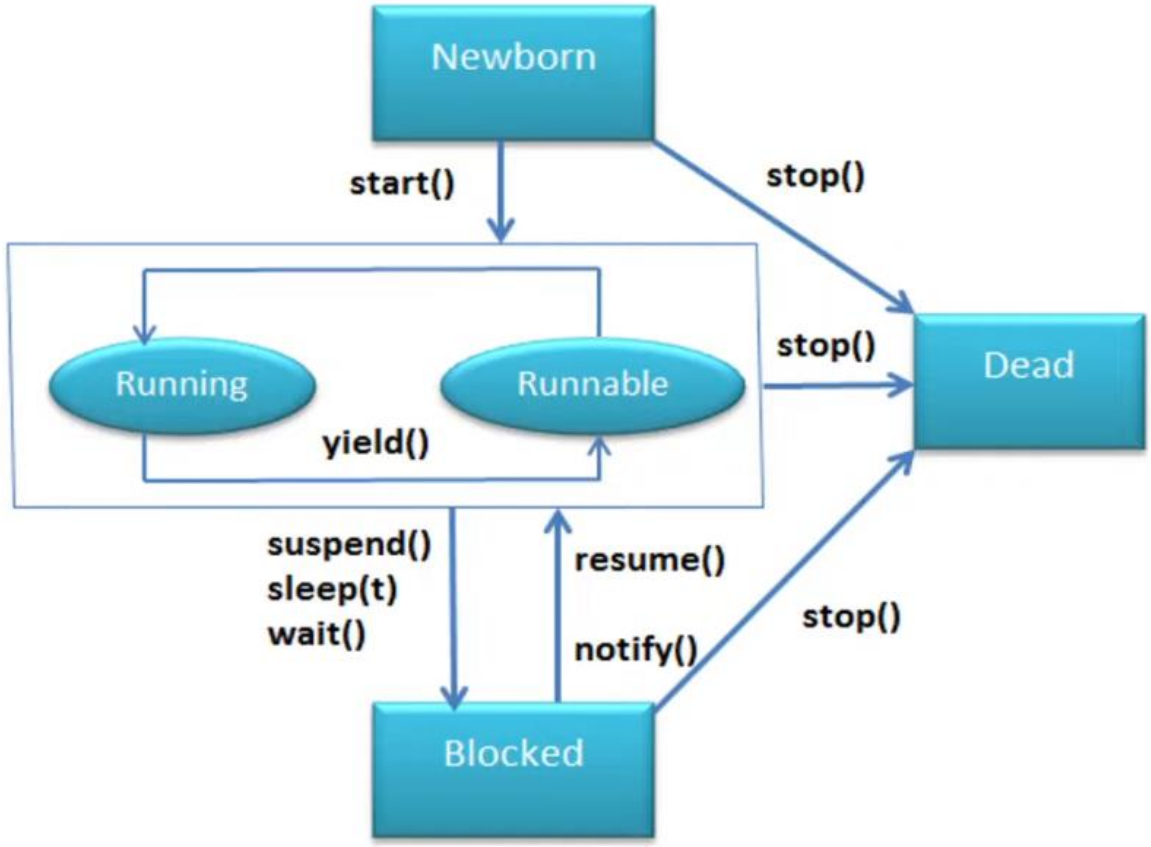


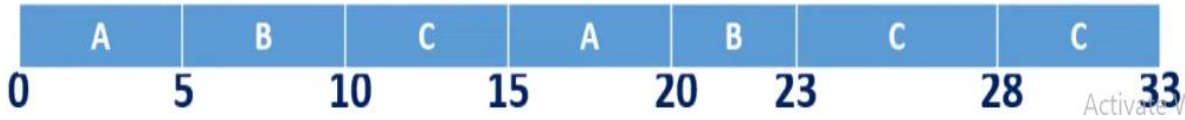
Fig: Life Cycle of Thread

Newborn state:

- When we create a thread object, the thread is born and is said to be in newborn state. At this state, we can do only one of the following:
    1. Schedule it for running using start() method
    2. Kill it using stop() method

Runnable State:

- The runnable state means that the thread is ready for execution and is waiting for the processor. The thread has joined the queue of threads that are waiting for execution.
- If all the threads have equal priority, then they are given time slots for execution in round robin fashion i.e. first come, first-serve manner. The thread releases processor and joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as time-slicing.
- A – 10sec, B – 8sec, C-15sec Assume Time slice is 5 sec. Processor execution is as follows



Running State:

- Running means that the processor has given its time to the thread for its execution.
- The thread runs until it releases control on its own or given chance to a higher priority thread.
- A running thread may releases its control in one of the following situations.
  1. It has been suspended using suspend() method. A suspended thread restarted by calling the resume() method.
  2. We can put a thread to sleep for a specified time period using the method sleep(time) where time is in milliseconds. This means that the thread is out of the queue during the time period. The thread re-enters the runnable state as soon as this time period is elapsed. Ex sleep(5000) – thread is idle for 5 secs.
  3. Thread has been wait until some event occurs. This is done using the wait() method. The thread can be restarted by calling the notify() method.

Blocked State:

- This state happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements.
- A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.
- A blocked thread is entered into the runnable state and subsequently the running state.

Dead State:

- When the thread completes its execution, it will move to the dead state. It is a natural death.
- We can kill thread by calling the stop() method at any state. It is a premature death.0

THREAD PRIORITY

METHODS IN THREAD CLASS

Syntax of the Method	Purpose
void start()	To start the specified thread.
void run()	This method encapsulates the functionality of a thread.
Boolean isAlive()	To return the value true if a thread has been started and has not yet died. Otherwise, it returns the value false.
void setName(String s)	To set the name of the specified thread to s.
String getName()	To return the name of the specified thread.
void setPriority(int p)	To set the priority p to specified thread.
int getPriority()	To return the priority of the specified thread.

Thread Priority

- Whenever multiple threads are ready for execution, the Java system chooses the highest priority thread and execute it.
- We can assign different priorities to the threads defined in a program by using a setPriority() method.
- Syntax:
- Threadname.setPriority(int number or priority constants);
- Priority constants are MAX_PRIORITY, MIN_PRIORITY and NORM_PRIORITY.
- MAX_PRIORITY represents 10 points on a relative scale, MIN_PRIORITY represents just 1 point. The NORM_PRIORITY represents 5 points.
- Default priority is NORM_PRIORITY.
- The priority number takes a value between 1 and 10.

Program -1 using Thread class

```
class A  extends Thread
{
        public  void  run()
        {
                for( int i=1;i<=5; i++)
                {
                System.out.println("From thread A :i =" +i);
                }
                System.out.println("Exit from A");
        }
```

```
class B  extends Thread
{
        public  void  run()
        {
                for( int j=1;j<=5; j++)
                {
                System.out.println("From thread B :j =" +j);
                }
                System.out.println("Exit from B");
        }
```

```
class C extends Thread
{
        public  void  run()
        {
                for( int k=1;k<=5; k++)
                {
                System.out.println("From thread C : k=" +k);
                }
                System.out.println("Exit from C");
        }
}
```

```
class testthread
{       public static void main (String args[])
   {
        A ta= new A();
        ta.setPriority(Thread.MIN_PRIORITY+1);
        B tb = new B();
        tb.setPriority(6);
        C tc = new C();
        tc.setPriority(Thread.MAX_PRIORITY);
        ta.start();
        tb.start();
        tc.start();
        try
        {
         ta.join();
         tb.join();
         tc.join();
        }
       catch(InterruptedException e)
       { }
   }
}
```

```
>javac testthread.java
>java testthread
From thread C :k =1
From thread C :k =2
From thread C :k =3
From thread C :k =4
From thread C :k =5
Exit from C
From thread B :j =1
From thread B :j =2
From thread B :j =3
From thread B :j =4
From thread B :j =5
Exit from B
From thread A :i =1
From thread A :i =2
From thread A :i =3
From thread A :i =4
From thread A :i =5
Exit from A
```

Program -2 using Runnable Interface

```
class square implements Runnable
{
    public  void  run()
    {
          for( int i=1;i<=5; i++)
          {
           System.out.println("From thread one :Square of " +i +"=" +(i*i));
          }
          System.out.println("Exit from Square class");
    }
```

```
class mythread
{
    public static void main (String args[])
    {
        square s = new square();
        Thread ta=new Thread(s);
        ta.setPriority(8);
        cube c = new cube();
        Thread tb=new Thread(c);
        tb.setPriority(4);
        ta.start();
        tb.start();
        try
        {
         ta.join();
         tb.join();
        }
       catch(InterruptedException e)
       { }
   }
}
```

```
>javac mythread.java
>java mythread
From thread one :Square of 1=1
From thread one :Square of 2=4
From thread one :Square of 3=9
From thread one :Square of 4=16
From thread one :Square of 5=25
Exit from Square class
From thread two :Cube of 1=1
From thread two :Cube of 2=8
From thread two :Cube of 3=27
From thread two :Cube of 4=64
From thread two :Cube of 5=125
Exit from Cube class
```

Multitasking vs Multithreading

Multitasking	Multithreading
CPU executes multiple tasks at the same time.	CPU execute multiple threads of a process.
CPU switches between tasks.	CPU switches between the threads frequently.
Process don't share same resources. Each process is allocated with separate resources.	Multiple threads of the process shares the same memory and resources allocated to the process.
Slower	Faster
Termination of process takes more time.	Termination of threads takes less time.
It supports for multiprocessing.	It does not support for multiprocessing.
High cost	Low cost