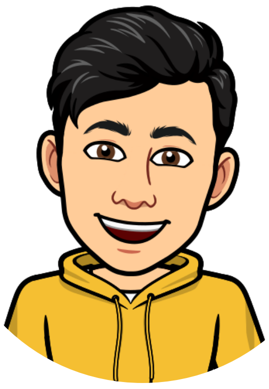
# CORE JAVA COMPLETE NOTES

*For Overnight Exam Rivison*



@Developers_Society

- Sourabh Chaudhari

65K+ Community on Instagram

@Developers_Society

# CORE JAVA COMPLETE NOTES

## Modules/ Sections

# Introduction to Java Programming Language

## 1.1 What is Java?

- Java is a programming language and a platform.

  **Note** - Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

- Java is a high level, robust, object-oriented and secure programming language.
- Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995.
- James Gosling is known as the father of Java.

---

## 1.2 History of java

- The history of Java starts with the Green Team. Java team members also known as Green Team.
- Team initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was best suited for internet programming. Later, Java technology was incorporated by Netscape.

- The principles for creating Java programming were:

1. Simple
2. Robust
3. Portable
4. Platform-independent
5. Secured
6. High Performance
7. Multithreaded
8. Architecture Neutral
9. Object-Oriented
10. Interpreted, and Dynamic

- Currently, Java is used in internet programming, mobile devices, games, ebusiness solutions, etc.

- Following are given significant points that describe the history of Java.

1. **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991.

2. Initially it was designed for small, embedded systems in electronic appliances like set-top boxes.
3. Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.
4. After that, it was called Oak and was developed as a part of the Green project.

   Why Java was named as "Oak"?

   Oak is a symbol of strength and chosen as a national tree of many countries like the U.S.A., France, Germany, Romania, etc.

5. In 1995, Oak was renamed as "Java" because it was already a trademark by Oak Technologies.

   Why Java Programming named "Java"?

   According to James Gosling, "Java was one of the top choices along with Silk". Since Java was so unique, most of the team members preferred Java than other names. Java is an island in Indonesia where the first coffee was produced (called Java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having a cup of coffee nearby his office.Notice that Java is just a name, not an acronym.

6. Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.

---

## **1.3 Main Features of JAVA**

1. Simple

Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

2. Object-oriented

   Java is an object-oriented programming language. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behavior.Basic concepts of OOPs are:

- Object
- Class

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation

3. Platform Independent

- Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.
- There are two types of platforms software-based and hardware-based. Java provides a software-based platform.
- The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on top of other hardware-based platforms.
  It has two components:
  0. Runtime Environment
  1. API(Application Programming Interface)
- Java code can be executed on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere (WORA).

4. Robust

Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems. ○ Java provides automatic garbage collection which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore. ○ There are exception handling and the type checking mechanism in Java. All these points make Java robust.

5. Portable

Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementation.

6. Distributed

Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet. 7. Distributed

A thread is like a separate program, executing concurrently. We can write Java programs that deal with many tasks at once by defining multiple threads. The main advantage of multi-threading is that it doesn't occupy memory for each thread. It shares a common memory area. Threads are important for multimedia, Web applications, etc.

8. Dynamic

Java is a dynamic language. It supports the dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.Java supports dynamic compilation and automatic memory management (garbage collection).

## 1.4 Write your first JAVA program

To create a simple Java program, you need to create a class that contains the main method. Let's understand the requirement first.

The requirement to write JAVA program

For executing any Java program, the following software or application must be properly installed.

- Install the JDK if you don't have installed it, download the JDK and install it.
- Set path of the jdk/bin directory. See below how to set path of java?
- Create the Java program
- Compile and run the Java program Let's create JAVA program

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

Save the above code as Simple.java.

To compile:

```
javac Simple.java
```

To execute:

```
 java Simple
```

Parameters used in First Java Program
Let's see what is the meaning of class, public, static, void, main, String[], System.out.println().

- **class** keyword is used to declare a class in Java.
- **public** keyword is an access modifier that represents visibility. It means it is visible to all.
- **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main() method is executed by the JVM, so it doesn't require creating an object to invoke the main() method. So, it saves memory.
- **void** is the return type of the method. It means it doesn't return any value.
- **main** represents the starting point of the program.
- **String[] args or String args[]** is used for command linSystem.out.println() is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class. argument. We will discuss it in coming section.
- **System.out.println()** is used to print statement. Here, System is a class, out is an object of the PrintStream class, println() is a method of the PrintStream class.

---

## Setting up the environment in Java

Here are few things which must be clear before setting up the environment

- **JDK(Java Development Kit) :** JDK is intended for software developers and includes development tools such as the Java compiler, Javadoc, Jar, and a debugger.
- **JRE(Java Runtime Environment) :** JRE contains the parts of the Java libraries required to run Java programs and is intended for end users. JRE can be view as a subset of JDK.
- **JVM:** JVM (Java Virtual Machine) is an abstract machine. It is a specification that provides runtime environment in which java bytecode can be executed. JVMs are available for many hardware and software platforms.

  Follow the given steps:

  - Java8 JDK is available at Download Java 8.
  - Click second last link for **Windows(32 bit)** and last link for **Windows(64 bit)** as highlighted below.
  - After download, run the **.exe file** and follow the instructions to install Java on your machine. Once you installed Java on your machine, you have to setup environment variable. o Go to **Control Panel -> System and Security -> System**.

# Creating Classes and their Objects in Java

**What is class in JAVA?**

- A class is a group of objects which have common properties.
- It is a template or blueprint from which objects are created.
- It is a logical entity. It can't be physical.

A class in Java can contain:

- Fields
- Methods
- Constructors
- Blocks
- Nested class and interface

**Syntax to declare a class:**

```
class  class_name
  {
     field OR Instance Variables;
     method;
  }
```

---

**Instance variable in Java**

- A variable which is created inside the class but outside the method is known as an instance variable.
- Instance variable doesn't get memory at compile time. It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

---

**Method in Java**

In Java, a method is like a function which is used to expose the behavior of an object.

Advantage of Method

- Code Reusability
- Code Optimization

---

**new keyword in Java**

The new keyword is used to allocate memory at runtime. All objects get memory in Heap memory area.

## What is an object in Java?

- An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc.
- It can be physical or logical (tangible and intangible).
- The example of an tangible object is chair, bike, marker.
- The example of an intangible object is the banking system, shopping system.
- An object is an instance of a class. A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.

- An object has three characteristics:
    0. **State:** represents the data (value) of an object.
    1. **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
    2. **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.
- *For Example,** Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.

## Examples of Class and Object

Here we are going to explain you two example:

- Object and Class Example: main method within the class
- Object and Class Example: main method outside the class

1. Object and Class Example: main method within the class

In this example, we have created a Student class which has two fields or instance variables id and name. We are creating the object of the Student class by new keyword and printing the object's value.

Here, we are creating a main() method inside the class.

```
// This program maintains the student record
          //Defining a Student class.

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

Output

```
   0
          null
```

2. Object and Class Example: main method outside the class

In real time development, we create classes and use it from another class. It is a better approach than previous one.

```
//Creating Student class.
        class Student
        {   //field or instance variable
          int id = 1;
          String name= "Kanishak";
        }
        //Creating another class TestStudent1 which contains the main method
        class TestStudent1
        {   //creating main method
         public static void main(String args[]){

        //creating an object of Student
          Student s1=new Student();

        //Printing values of the object
          System.out.println(s1.id);
          System.out.println(s1.name);
         }
        }
```

Output

```
     1
          Kanishak
```

# Using constructors for creating Objects

## What is constructor 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 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 case, Java compiler provides a default constructor by default.

---

## Why it is called constructor?

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

---

## Need of Constructor?

Think of a Box. If we talk about a box class then it will have some class variables (say length, breadth, and height). But when it comes to creating its object(i.e Box will now exist in the computer's memory), then can a box be there with no value defined for its dimensions. The answer is no. So constructors are used to assign values to the class variables at the time of object creation, either explicitly done by the programmer or by Java itself (default constructor).

---

## Rules for creating Java constructor

There are three rules defined for the constructor.

- A constructor name must be the same as its class name
- A constructor must have no explicit return type
- A constructor cannot be abstract, static, final, and synchronized

---

## Types of Java constructors

There are two types of constructors in Java:

1. Default constructor (No-argument constructor)

2. Parameterized constructor

**1. Java Default Constructor or No-argument constructor** • A constructor that has no

parameter is known as the default constructor.

- If we don't define a constructor in a class, then the compiler creates default constructor(with no arguments) for the class.
- If we write a constructor with arguments or no-arguments then the compiler does not create a default constructor.
- Default constructor provides the default values to the object like 0, null, etc. depending on the type.

Example of Default Constructor or No-argument constructor

Java Program to illustrate calling a no-argument constructor

```
//This is a java Input/Output package
import java.io.*;
class ABC
{     //This is instance variable or fields of this class.
     int num;
     String name;

    // this would be invoked while an object of that class is created.
    ABC()
    {
        System.out.println("Constructor called");
    }
}
class GFG
{
    public static void main (String[] args)
    {
    // this would invoke default constructor.
        ABC obj1 = new ABC();

    // Default constructor provides the default values to the object like 0, null
        System.out.println(obj1.name);
        System.out.println(obj1.num);
    }
}
```

OUTPUT

```
Constructor called
null
0
```

## 2. Java Parameterized Constructor

- A constructor that has parameters is known as parameterized constructor.
- If we want to initialize fields of the class with your own values, then use 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

Java Program to illustrate calling of parameterized constructor. In this example, we have created the constructor of class ABC that have two parameters name and id. We can have any number of parameters in the constructor.

```
    import java.io.*;

        class ABC
        {
            //This is instance variable or fields of this class.
             String name;
              int id;

            // constructor would initialize instance variable or fields with the
values of passed arguments while object of that class created.
         ABC(String name, int id)
        {
                this.name = name;
                this.id = id;
        }
          }
          class GFG
          {
              public static void main (String[] args)
              {
                  // this would invoke the parameterized
constructor.
                  ABC obj1 = new ABC("adam", 1); // Here you are passing arguments
to the parameterized constructor
                  System.out.println("Name :" + obj1.name + " and Id :" +
obj1.id);
              }
          }
```

OUTPUT

```
Name :adam and Id :1
```

## What is 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 Student{
    int id;      String name;         int age;
    //creating two argument constructor
    Student(int i,String n)
    {
      id = i;        name = n;
    }
    //creating three argument constructor
    Student(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[])
    {
    Student s1 = new Student(111,"Karan");
    Student s2 = new Student(222,"Aryan",25);

    s1.display();
    s2.display();
   }
}
```

OUTPUT

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

# Understanding Inheritance in Java

## Inheritance in Java

- It is an important part of OOPs (Object Oriented programming system).
- It is the mechanism in java by which one class is allowed to inherit the features(fields and methods) of another class.
- 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.

## Why use inheritance in java?

- For Method Overriding (so runtime polymorphism can be achieved).
- For Code Reusability.

## Important terminology:

- **Super Class:** The class whose features are inherited is known as superclass(or a base class or a parent class).
- **Sub Class:** The class that inherits the other class is known as a subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods in addition to the superclass fields and methods.
- **Reusability:** Inheritance supports the concept of "reusability", i.e. when we want to create a new class and there is already a class that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

## How to use inheritance in Java?

The keyword used for inheritance is **extends**. 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.

## The syntax of Inheritance in JAVA

```
class Subclass-name extends Superclass-name
{
   //methods and fields
}
```

## Simple Example of Inheritance

In this example, Programmer is the subclass and Employee is the superclass.

```
class Employee
{
 float salary=40000;
}
// Here subclass Programmer extends the feature of Superclass Employee
class Programmer extends Employee
{
 int bonus=10000;
 public static void main(String args[])
{
     Programmer p=new Programmer();
     System.out.println("Programmer salary is:"+p.salary);
     System.out.println("Bonus of Programmer is:"+p.bonus);   }
     }
```

OUTPUT

```
Programmer salary is:40000.0
          Bonus of programmer is:10000
```

## Types of inheritance in java

On the basis of class, there can be three types of inheritance in java:

- Single
- Multilevel
- Hierarchical

In java programming, multiple and hybrid inheritance is supported through interface only. We will learn about interfaces later.

**1. Single Inheritance:** 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.

```
class Animal
{
void eat()
{
System.out.println("eating...");
}
}
// Here subclass Dog inherits the property of superclass Animal
class Dog extends Animal
{
void bark()
{
  System.out.println("barking...");
    }
    }
  class TestInheritance
  {
  public static void main(String args[])
  {
  Dog d=new Dog();  // Here we are creating object of Dog class
  d.bark();
  d.eat();
  }
  }
```

**OUTPUT**

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

**2. Multilevel Inheritance:** When there is a chain of inheritance, it is known as **multilevel inheritance**. In the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.

```
  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();
            }
            }
```

<u>**OUTPUT**</u>

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

**3. Hierarchical Inheritance:** 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.

```
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();
//c.bark();//C.T.(Complie time) Error
}}
```

OUTPUT

```
meowing...
          eating...
```

## 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
        {
         public static void main(String args[]){
         C obj=new C();
        obj.msg();//Now which msg() method would be invoked?
        }
        }
```

OUTPUT

```
Compile Time Error
```

## Implementing Method Overloading through Polymorphism

### Java Method Overloading

- If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.
- If we have to perform only one operation, having same name of the methods increases the readability of the program.
- Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

You can create same method name with different argument list example:

```
voidfunc() { ... }
       voidfunc(int a) { ... }
       floatfunc(double a) { ... }
       floatfunc(int a, float b) { ... }
```

### Why method overloading?

Suppose, you have to perform the addition of given numbers but there can be any number of arguments (let's say either 2 or 3 arguments for simplicity). In order to accomplish the task, you can create two methods sum2num(int, int) and sum3num(int, int, int) for two and three parameters respectively. The better way to accomplish this task is by overloading methods. And, depending upon the argument passed, one of the overloaded methods is called. This helps to increase the readability of the program.

### Advantage of method overloading

- Method overloading increases the readability of the program.
- We don't have to create and remember different names for functions doing the same thing.

### How to perform method overloading in Java?

- Overloading by changing the number of arguments
- By changing the datatype of parameters

1. Overloading by changing the number of arguments

Java program to demonstrate working of method overloading in Java. If overloading was not supported by Java, we would have to create method names like sum1, sum2, … or sum2Int, sum3Int, … etc.

```java
public class Sum {

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

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

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

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

OUTPUT

```
30
          60
          31.0
```

2. By changing the datatype of parameters

```
class MethodOverloading
        {
            // this method accepts int
        private static void display(int a)
        {
        System.out.println("Got Integer data.");
        }

        // this method  accepts String object
        private static void display(String a)
        {
        System.out.println("Got String object.");
        }
```

OUTPUT

```
Got Integer data.
        Got String object.
```

---

**Can we overload java main() method?**

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But JVM calls main() method which receives string array as arguments only. Let's see the simple example:

```
class TestOverloading4
        {
        public static void main(String[] args)
        {
        System.out.println("main with String[]");
        }
        public static void main(String args)
        {
        System.out.println("main with String");
        }
        public static void main()
        {
        System.out.println("main without args");
        }
        }
```

OUTPUT

```
main with String[]
```

# Implementing Overriding through Polymorphism

## Method Overriding in Java

- Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes.
- When a method in a subclass has the same name, same parameters or signature, and same return type(or sub-type) as a method in its super-class, then the method in the subclass is said to override the method in the super-class.
- Method overriding is one of the way by which java achieve Run Time Polymorphism.

## Usage of Java Method Overriding

- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism.

## Rules for Java Method Overriding

- The method must have the same name as in the parent class.
- The method must have the same parameter as in the parent class.
- There must be an IS-A relationship (inheritance).

## Understanding the problem without method overriding

Let's understand the problem that we may face in the program if we don't use method overriding.

```
//Java Program to demonstrate why we need method overriding
   //Here, we are calling the method of parent class with child class object.

   //Creating a parent class
          class Vehicle
          {
            void run()
            {
            System.out.println("Vehicle is running");
            }
          }

          //Creating a child class
          class Bike extends Vehicle
          {
            public static void main(String args[])
            {
            //creating an instance of child class
            Bike obj = new Bike();
            //calling the method with child class instance
            obj.run();
            }
          }
```

Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

OUTPUT

```
Vehicle is running
```

---

## Example of method overriding

In this example, we have defined the run method in the subclass as defined in the parent class but it has some specific implementation. The name and parameter of the method are the same, and there is IS-A relationship between the classes, so there is method overriding

```
//Java Program to illustrate the use of Java Method Overriding
          //Creating a parent class.
          class Vehicle
          {
            //defining a method
            void run()
            {
            System.out.println("Vehicle is running");
            }
          }
          //Creating a child class
          class Bike2 extends Vehicle
          {
            //defining the same method as in the parent class
            void run()
            {
            System.out.println("Bike is running safely");
            }

            public static void main(String args[])
            {
            Bike2 obj = new Bike2();//creating object
            obj.run();//calling method
             }
          }
```

OUTPUT

```
Bike is running safely
```

---

## Real life example of method overriding

Lets consider an example that, A Son inherits his Father's public properties e.g. home and car and using it. At later point of time, he decided to buy and use his own car, but, still he wants to use his father's home. So, what he can do? He can use method overriding feature and use his own car. See below example, how he has overridden his own car method.

```
class Father
           {
          public void home()
          {
          System.out.println("Father's home");
          }
          public void car()
          {
          System.out.println("Father's Car");
          }
          }

          class Son extends Father
          {

          //Override
          public void car()
          {
          System.out.println("Son's Car");
          }
          }

          public class TestOverriding
          {
          public static void main(String[] args)
          {
          Son s = new Son();
          s.home();
          s.car();
          }
          }
```

OUTPUT

```
Father’s home
          Son’s Car
```

## Difference between method overloading and method overriding in java

| No. | Method Overloading | Method Overriding |
|---|---|---|
| 1. | Method overloading is used to increase the readability of the program. | Method overriding is used to provide the specific implementation of the method that is already provided by its super class. |
| 2. | Method overloading is performed within class. | Method overriding occurs in two classes that have IS-A (inheritance) relationship. |
| 3. | In case of method overloading, parameter must be different. | In case of method overriding, parameter must be same. |
| 4. | Method overloading is the example of compile time polymorphism. | Method overriding is the example of run time polymorphism. |
| 5. | Return type can be same or different in method overloading | Return type must be same or covariant in method overriding. |

# Learning Abstraction through Interface

## Interface in Java

Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).

- Interfaces specify what a class must do and not how. It is the blueprint of the class.
- An Interface is about capabilities like a Player may be an interface and any class implementing Player must be able to (or must implement) move(). So it specifies a set of methods that the class has to implement.
- If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.
- It is used to achieve abstraction and multiple inheritance in Java.
- To declare an interface, use interface keyword. It is used to provide total abstraction.That means all the methods in an interface are declared with an empty body and are public and all fields are public, static and final by default.

## Why use Java interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes? The reason is, abstract classes may contain non-final variables, whereas variables in interface are final, public and static.

## Syntax

```
interface <interface_name>
          {
          // declare constant fields
          // declare methods that abstract
          // by default.
          }
```

## Java Interface Example

Java program to demonstrate working of interface.

```
import java.io.*;

          // A simple interface
          interface In1
          {
              // public, static and final
              final int a = 10;

              // public and abstract
              void display();
          }

          // A class that implements the interface.
          class TestClass implements In1
          {
              // Implementing the capabilities of interface.
              public void display()
              {
                  System.out.println("VIRTUALITY");
              }

              // Driver Code
              public static void main (String[] args)
              {
                  TestClass t = new TestClass();
                  t.display();
                  System.out.println(a);
              }
          }
```

OUTPUT

```
VIRTUALITY
          10
```

---

## Real-world example of Interface

Let's consider the example of vehicles like bicycle, car, bike………, they have common functionalities. So we make an interface and put all these common functionalities. And lets Bicycle, Bike, car ….etc implement all these functionalities in their own class in their own way.

```
import java.io.*;

          interface Vehicle
          {  // all are the abstract methods.
              void changeGear(int a);
              void speedUp(int a);
              void applyBrakes(int a);
          }

          class Bicycle implements Vehicle
          {
              int speed;
              int gear;

              // to change gear override method here
              public void changeGear(int newGear)
              {
                  gear = newGear;
              }

              // to increase speed @Override
              public void speedUp(int increment)
              {
                  speed = speed + increment;
              }

              // to decrease speed @Override
              public void applyBrakes(int decrement)
              {
                  speed = speed - decrement;
              }

              public void printStates()
              {
              System.out.println("speed: " + speed+ " gear: " + gear);
              }
          }

          class Bike implements Vehicle
          {
              int speed;
              int gear;

              // to change gear @Override
              public void changeGear(int newGear)
              {
                  gear = newGear;
              }
```

```
        // to increase speed @Override
        public void speedUp(int increment)
        {
            speed = speed + increment;
        }

        // to decrease speed @Override
        public void applyBrakes(int decrement)
        {
            speed = speed - decrement;
        }
   public void printStates()
        {
        System.out.println("speed: " + speed + " gear: " + gear);
        }
    }
    class GFG {

        public static void main (String[] args)
        {   // creating an inatance of Bicycle
            // doing some operations
            Bicycle bicycle = new Bicycle();
            bicycle.changeGear(2);
            bicycle.speedUp(3);
            bicycle.applyBrakes(1);

            System.out.println("Bicycle present state :");
            bicycle.printStates();

            // creating an instance of bike
            Bike bike = new Bike();
            bike.changeGear(1);
            bike.speedUp(4);
            bike.applyBrakes(3);

            System.out.println("Bike present state :");
            bike.printStates();
        }
    }
```

OUTPUT

```
Bicycle present state :
          speed: 2 gear: 2
          Bike present state :
          speed: 1 gear: 1
```

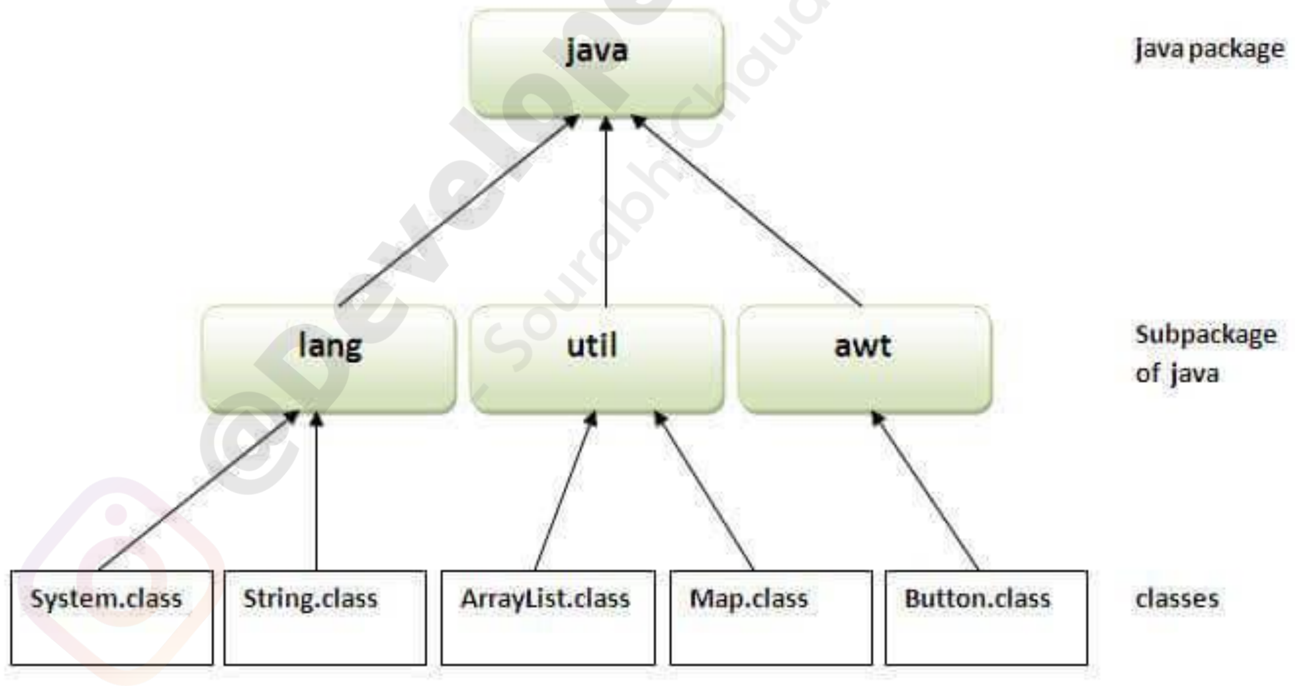
# Learning Encapsulation through Package

## 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.

## Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- 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
```

Example

```
javac -d . Simple.java
```

The -d switch specifies the destination where to put the generated class file. You can use any directory name like /home (in case of Linux), d:/abc (in case of windows) etc. If you want to keep the package within the same directory, you can use . (dot).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 run java package program

```
java pacakagename.javafilename
```

Example

```
java mypack.Simple
```

OUTPUT

```
Welcome to package
```

## How to access package from another package?

There are three ways to access the package from outside the package.

- import package.*;
- import package.classname;
- 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;  //create package here
import pack.*;  //import package

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
{
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
```

# Handling Exceptions in Java

## Exception Handling in Java

- The **Exception Handling in Java** is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.
- In Java, an exception is an event that disrupts the normal flow of the program.
- It is an object which is thrown at runtime.
- **Exception Handling** is a mechanism to handle runtime errors such a ClassNotFoundException, IOException, SQLException, RemoteException, etc.

## Advantage of Exception Handling

- The core advantage of exception handling is to maintain the normal flow of the application.
- An exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take 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 your program and there occurs an exception at statement 5, the rest of the code will not be executed i.e. statement 6 to 10 will not be executed. If we perform exception handling, the rest of the statement will be executed. That is why we use exception handling in Java.

## Types of Java Exceptions

There are mainly two types of exceptions: checked and unchecked. Here, an error is considered as the unchecked exception. According to Oracle, there are three types of exceptions:

- Checked Exception
- Unchecked Exception
- Error

1) Checked Exception

- The classes which directly inherit Throwable class except RuntimeException and Error are known as **checked exceptions**.
- E.g. IOException, SQLException etc. Checked exceptions are checked at compile-time.

2) Unchecked Exception

- The classes which inherit RuntimeException are known as **unchecked exceptions**.
- E.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

3) Error

- Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

---

## Java Exception Keywords

- **try** : The **try** keyword is used to specify a block where we should place exception code. The try block must be followed by either catch or finally. It means, we can't use try block alone.
- **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 important 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 doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

---

## Common Scenarios of Java Exceptions

There are given some scenarios where unchecked exceptions may occur. They are as follows:

1) A scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;  //ArithmeticException
```

2) A scenario where NullPointerException occurs

If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

```
String s=null;
          System.out.println(s.length()); //NullPointerException
```

3) A scenario where NumberFormatException occurs

The wrong formatting of any value may occurNumberFormatException.

```
String s="abc";
          Inti=Integer.parseInt(s);//NumberFormatException
```

## Java Exception Handling Example

Let's see an example of Java Exception Handling where we using a try-catch statement to handle the exception.In this example, 100/0 raises an ArithmeticException which is handled by a try-catch block.

```
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...
```

## Java Multi-catch block

A try block can be followed by one or more catch blocks. Each catch block must contain a different exception handler. So, if you have to perform different tasks at the occurrence of different exceptions, use java multi-catch block.

### Points to remember

- At a time only one exception occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general, i.e. catch for ArithmeticException must come before catch for Exception.

### Example

Let's see a simple example of java multi-catch block.

```java
public class MultipleCatchBlock1
 {
  public static void main(String[] args)
  {
   try{
       int a[]=new int[5];
       a[5]=30/0;
      }
       catch(ArithmeticException e)
           {
            System.out.println("Arithmetic Exception occurs");
           }
       catch(ArrayIndexOutOfBoundsException e)
           {
           System.out.println("ArrayIndexOutOfBounds Exception occurs");
           }
       catch(Exception e)
           {
           System.out.println("Parent Exception occurs");
           }
       System.out.println("rest of the code");
      }
  }
```

### OUTPUT

```
Arithmetic Exception occurs
          rest of the code
```

# Understanding Life cycle of a Thread

## Thread

- Threads allows a program to operate more efficiently by doing multiple things at the same time.
- Threads can be used to perform complicated tasks in the background without interrupting the main program.

---

## How to create thread

There are two ways to create a thread :

- By extending Thread class.
- By implementing Runnable interface.

### Thread class

Thread class provide constructors and methods to create and perform operations on a thread. Thread class extends Object class and implements Runnable interface.

### Runnable interface

The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run(). public void run(): is used to perform action for a thread.

### 1. Create a Thread by Extending a Thread Class

Step 1

You will need to override run( ) method available in Thread class. This method provides an entry point for the thread and you will put your complete business logic inside this method. Following is a simple syntax of **run() method** –

```
public void run( )
```

Step 2

Once Thread object is created, you can start it by calling **start()** method, which executes a call to run( ) method. Following is a simple syntax of **start() method** –

```
void start( );
```

Example:

```
class ThreadDemo extends Thread{
    privateThread t;
    privateStringthreadName;

    ThreadDemo(String name){
    threadName= name;
    System.out.println("Creating "+threadName);
    }

    public void run(){
    System.out.println("Running "+threadName);
    try{
    for(int i =4; i >0; i--){
    System.out.println("Thread: "+threadName+", "+ i);
    // Let the thread sleep for a while.
    Thread.sleep(50);
    }
    }catch(InterruptedException e){
    System.out.println("Thread "+threadName+" interrupted.");
    }
    System.out.println("Thread "+threadName+" exiting.");
    }

    public void start (){
    System.out.println("Starting "+threadName);
    if(t ==null){
             t =newThread(this,threadName);
    t.start();
    }
    }
    }

    public class TestThread
    {
    publicstaticvoid main(Stringargs[]){
    ThreadDemo T1 =newThreadDemo("Thread-1");
    T1.start();

    ThreadDemo T2 =newThreadDemo("Thread-2");
    T2.start();
    }
    }
```

OUTPUT

```
Creating Thread-1
          Starting Thread-1
          Creating Thread-2
          Starting Thread-2
          Running Thread-1
          Thread: Thread-1, 4
          Running Thread-2
          Thread: Thread-2, 4
          Thread: Thread-1, 3
          Thread: Thread-2, 3
          Thread: Thread-1, 2
          Thread: Thread-2, 2
          Thread: Thread-1, 1
          Thread: Thread-2, 1
          Thread Thread-1 exiting.
          Thread Thread-2 exiting.
```

## 2. Create a Thread by Implementing a Runnable Interface

Step 1

you need to implement a run() method provided by a Runnable interface. This method provides an entry point for the thread and you will put your complete logic inside this method. Following is a simple syntax of the **run() method** –

```
public void run( )
```

Step 2

you will instantiate a Thread object using the following constructor –

```
Thread(Runnable threadObj, String threadName);
```

Where, threadObj is an instance of a class that implements the Runnable interface and threadName is the name given to the new thread.

Step 3

Once a Thread object is created, you can start it by calling start() method, which executes a call to run( ) method. Following is a simple syntax of **start() method**–

```
void start();
```

Example:

Here is an example that creates a new thread and starts running it –

```
class Multi3 implements Runnable{
    public void run(){
    System.out.println("thread is running...");
    }

    public static void main(String args[]){
    Multi3 m1=new Multi3();
    Thread t1 =new Thread(m1);
    t1.start();
}
}
```

**OUTPUT**

```
Output:thread is running...
```

**<u>Differences between "extending" and "implementing" Threads</u>**

The major difference is that when a class extends the Thread class, you cannot extend any other class, but by implementing the Runnable interface, it is possible to extend from another class as well.