**[Core Java](https://corejavabyvenkatesh.blogspot.com/2020/04/overloading-vs-overriding-overloading-1_6.html)**

**Overloading Vs Overriding**

**Overloading:**

**1)   If two or more methods with the same name and with different parameters list, then it is said to be method overloading.**

**2)   In method overloading, return type can be same or different.**

**3)   Methods can be overloaded in the same class**

**4)   Methods can be overloaded in inheritance also.**

**5)   In method overloading access modifiers can be same or different**

**6)   Final methods can be overloaded.**

**7)   Static methods can be overloaded.**

**8)   Private methods can be overloaded.**

**Overriding:**

**1)   If two or more methods with the same name and with the same parameters list, then it is said to be method overriding.**

**2)   In method overriding, return type must be same except covariant return type.**

**3)   Methods cannot be overridden in the same class because of ambiguity to call**

**4)   Methods can be overridden in inheritance only.**

**5)   In method overriding, overriding method can have same access modifier or less restrictive access modifier (must not be more restrictive)**

**6)   Final methods cannot be overridden because final keyword used to prevent method overriding.**

**7)   Static methods cannot be overridden because static members are not a part of an object.**

**8)   Private methods cannot be overridden because private members cannot be inherited.**

**Covariant return type:**

**Java permits sub class type as a return type while overriding a method. This is known as covariant return type.**

**Example:**

**class A{}**

**class B extends A{}**

**class C extends B**

**{**

**A get(){} => Overridden method**

**}**

**class D extends C**

**{**

**B get(){} => Overriding method**

**}**

**In the above example overriding method return type is sub class of overridden method return type. This is called covariant return type.**

**Upcasting:**

**Ø Assigning an object or object reference of sub class to super class type is known as upcasting.**

**Ø Upcasting done by system implicitly.**

**Ø Upcasting always valid.**

**Examples:**

**1)   A ob = new B(); (For example A is a super class and B is a sub class)**

**2)   B ob1=new B(); A ob2=ob1;**

**Downcasting:**

**Ø Assigning an object or object reference of super class to sub class typ is known as downcasting.**

**Ø Downcasting must be done by programmer explicitly, otherwise compile time error occurs.**

**Ø Downcasting always needs upcasting**

**Example:**

**1)   A ob1=new B(); => It is upcasting**

**B ob2=(B)ob1; => It is downcasting**

**(In the above example, assume A is a super class and B is a sub class)**

**Abstract Class:**

**Ø A class that is declared with abstract keyword is called as an abstract class.**

**Ø Abstract class can have only abstract methods, only non abstract methods or both abstract and non abstract methods.**

**Abstract Method:**

**Ø A method that has no body is called as an abstract method.**

**Ø Abstract method must be declared with abstract keyword in Java, otherwise compile time error occurs.**

**Example:   abstract void show();**

**Ø Non abstract methods are called concrete methods. A method that has a body is called as concrete method.**

**Example:   void print(){}**

**Ø If the class contains an abstract method, then the class must be declared with abstract keyword, otherwise compile time error occurs.**

**Ø Abstract classes cannot be instantiated.**

**Ø Instantiation means object creation.**

**Ø Abstract class can be inherited by using extends keyword.**

**Ø Whenever abstract class is inherited, then all abstract methods of an abstract class must be overridden in a sub class or sub class must be declared with abstract keyword, otherwise compile time error occurs.**

**Example:**

**abstract class A**

**{**

**abstract void show();**

**void print()**

**{**

**System.out.println(“print() method”);**

**}**

**}**

**class B extends A**

**{**

**void show()**

**{**

**System.out.println(“show() method”);**

**}**

**void display()**

**{**

**System.out.println(“display() method”);**

**}**

**public static void main(String args[])**

**{**

**B ob=new B();**

**ob.show();**

**ob.print();**

**ob.display();**

**}**

**}**

**Ø Abstract methods cannot be final**

**Ø Abstract methods cannot be static**

**Ø Abstract methods cannot be private**

**Ø Abstract class cannot be final**

**Ø Abstract class can have constructors and those constructors are called whenever an object is created to sub class.**

**Ø Abstract class can have static members also.**

**Ø Abstract class can have main method also.**

**Interfaces:**

**Ø** **An interface is a collection of public static final variables and public abstract methods.**

**Ø In interface all variables are implicitly public static final and all methods are implicitly public abstract.**

**Ø An interface itself implicitly abstract**

**Ø Interfaces cannot be instantiated.**

**Ø Interface can be inherited into a class by using implements keyword.**

**Ø Interface can also be inherited by using extends keyword.**

**Ø Whenever interface is inherited then all methods of an interface must be overridden in a sub class or sub class must be declared with abstract keyword, otherwise compile time error occurs.**

**Example:**

**interface A**

**{**

**int x=10;**

**void show();**

**}**

**class B implements A**

**{**

**int y=20;**

**public void show()**

**{**

**System.out.println(“show() method”);**

**}**

**void print()**

**{**

**System.out.println(“print() method”);**

**}**

**public static void main(String args[])**

**{**

**System.out.println(A.x);**

**B ob=new B();**

**System.out.println(ob.y);**

**ob.show();**

**ob.print();**

**}**

**}**

**Note: Class cannot be inherited in interface.**

**Interfaces are introduced in Java to achieve multiple inheritance.**

**Encapsulation:**

**Binding of variables with methods and those methods operating on variables. This is known as an encapsulation.**

**Example:**

**class Emp**

**{**

**private int age;**

**void setAge(int age)**

**{**

**if(age>100)**

**this.age=100;**

**else if(age<0)**

**this.age=0;**

**else**

**this.age=age;**

**}**

**int getAge()**

**{**

**return age;**

**}**

**}**

**class Demo**

**{**

**public static void main(String args[])**

**{**

**Emp e=new Emp();**

**e.setAge(200);**

**int x=e.getAge();**

**System.out.println(x);**

**}**

**}**

**In the above example, age variable bound with setAge() and getAge() methods and these methods are operating on same age variable. This is known as an encapsulation.**

**Inner Classes:**

**A class that is defined in another class is called as an inner class.**

**There are four types of inner classes:**

**1)   Member class**

**2)   Static member class**

**3)   Local class**

**4)   Anonymous class**

**1) Member class:**

**A class that is defined as member of another class is called as member class.**

**Example:**

**class A**

**{**

**class B**

**{**

**void show()**

**{**

**System.out.println(“Welcome”);**

**}**

**}**

**}**

**class Demo**

**{**

**public static void main(String args[])**

**{**

**A a=new A();**

**A.B b=a.new B();**

**b.show();**

**}**

**}**

**Static Member Class:**

**A class that is defined as a static member of another class is called as static member class.**

**Example:**

**class A**

**{**

**static class B**

**{**

**void show()**

**{**

**System.out.println(“Welcome”);**

**}**

**}**

**}**

**class Demo**

**{**

**public static void main(String args[])**

**{**

**A.B b=new A.B();**

**b.show();**

**}**

**}**

**Local Class:**

**A class that is defined in a method is called as local class.**

**Example:**

**class Demo**

**{**

**public static void main(String args[])**

**{**

**class Test**

**{**

**void show()**

**{**

**System.out.println(“Welcome”);**

**}**

**}**

**Test t=new Test();**

**t.show();**

**}**

**}**

**Anonymous Class:**

**It is a one type of local class that has no name. It is always sub class of a class or interface.**

**Example:**

**interface Test**

**{**

**void show();**

**}**

**class Demo**

**{**

**public static void main(String args[])**

**{**

**Test t=new Test()**

**{**

**public void show()**

**{**

**System.out.println(“Welcome”);**

**}**

**};**

**t.show();**

**}**

**}**

**Ø Outer classes cannot be private and cannot be protected.**

**Ø Member class & Static member class can have all access modifiers.**

**Ø Access modifiers cannot be applied to local class and anonymous class.**

**Command Line Arguments:**

**Ø** **The arguments that are passed at the command prompt are called command line arguments.**

**Ø** **Command line arguments are received by main method only.**

**Ø** **The arguments are received string format only.**

**Examples:**

**1)**  **class Demo**

**{**

**public static void main(String args[])**

**{**

**for(String s : args)**

**{**

**System.out.println(s);**

**}**

**}**

**}**

**2)**  **class Demo**

**{**

**public static void main(String args[])**

**{**

**System.out.println(args[0]+args[1]);**

**}**

**}**

**To run the above application:**

**C:\> java Demo Taj Mahal**

**Output: TajMahal**

**C:\> java Demo 10 20**

**Output: 1020 because command line arguments are received as a string format only.**

**To convert string format to other formats we need wrapper classes.**

**Wrapper Classes:**

**Each of Java’s 8 primitive data types has a class and those classes are called wrapper classes because they wrap the data into an object.**

**List of primitive data types:**

**1)**  **byte 2) short 3) int 4) long 5) float 6) double 7) char 8) boolean**

**List of wrapper classes:**

**1)**  **Byte 2) Short 3) Integer 4) Long 5) Float 6) Double 7) Character 8) Boolean**

**Example:**

**class Demo**

**{**

**public static void main(String args[])**

**{**

**int x=Integer.parseInt(args[0]);**

**int y=Integer.parseInt(args[1]);**

**System.out.println(x+y);**

**}**

**}**

**To run the above application:**

**C:\> java Demo 10 20**

**Output: 30**

**To covert primitive type to reference type**

**int a=5;**

**Integer i=new Integer(a);**

**To covert reference type to primitive type:**

**Integer i=new Integer(5);**

**int x=i.intValue();**

**Auto boxing:**

**The process of converting primitive type to the corresponding reference type is known as auto boxing.**

**Example:**

**int x=5;**

**Integer i=x; => It is called as auto boxing**

**Auto unboxing:**

**The process of converting reference type to the corresponding primitive type is known as auto unboxing.**

**Example:**

**Integer i=new Integer(5);**

**int x=i; => it is called as auto unboxing.**

**Both auto boxing and auto unboxing features are introduced in JDK 1.5 verion in 2004.**

**Exception Handling:  
  
  
Errors:**

**A program mistake is said to be an error.**

**There are three types of errors:**

**1)**  **Compile time errors(Syntax errors)**

**2)**  **Runtime errors(Exceptions)**

**3)**  **Logical errors**

**Runtime errors handling mechanism is called as exception handling.**

**In exception handling we use the following keywords.**

**1)**  **try 2) catch 3) throw 4) throws 5) finally**

**The syntax of try and catch blocks:**

**try  
{**

**============= => Task code**

**}catch(ExceptionClassName ObjectReference)**

**{**

**============= => Error Message**

**}**

**try block must be associated with at least one catch block or finally block.**

**All exceptions are classes in Java.**

**Whenever exception occurs in a Java program, then the related exception class object is created by JVM, passed to exception handler(catch block) and exception handler code is executed.**

**There are two types of exceptions:**

**1)**  **Checked Exceptions**

**2)**  **Unchecked Exceptions**

**Checked Exceptions:**

**Ø** **The exception classes  that are derived from java.lang.Exception class are called checked exceptions. Checked exceptions do not include java.lang.RuntimeException class and all its sub classes.**

**Ø** **All checked exceptions must be handled explicitly otherwise compile time error occurs.**

**Ø** **The Java compiler checks for try & catch blocks or throws clause for this kind of exceptions.**

**Ø** **All application specific exceptions are comes under this category.**

**Unchecked Exceptions:**

**Ø** **The exception classes that are derived from java.lang.RuntimeException class are called unchecked exceptions.**

**Ø** **All unchecked exceptions are handled by system implicitly.**

**Ø** **Handling unchecked exceptions are optional by programmer.**

**Ø** **Unchecked exceptions are handled by programmer to display**

**Ø** **user friendly error messages only.**

**Ø** **The Java compiler does not check for try & catch blocks or throws clause for this kind of exceptions.**

**Ø** **All general exceptions comes under this category.**

**Example1:**

**class Demo**

**{**

**public static void main(String args[])**

**{**

**try{**

**int x=Integer.parseInt(args[0]);**

**int y=Integer.parseInt(args[1]);**

**int z=x/y;**

**System.out.println(z);**

**}catch(ArrayIndexOutOfBoundsException ae)**

**{**

**System.err.println(“Please pass two arguments”);**

**}**

**catch(NumberFormatException ne)**

**{**

**System.err.println(“Please pass two numbers only”);**

**}**

**catch(ArithmeticException ae)**

**{**

**System.err.println(“Please pass second                                                                     argument except zero”);**

**}**

**}**

**}**

**Execution:**

**1)**  **C:\> java Demo**

**Please pass two arguments**

**2)**  **C:\> java Demo abc xyz**

**Please pass two numbers only**

**3)**  **C:\> java Demo 10 0**

**Please pass second argument except zero**

**4)**  **C:\> java Demo 10 2**

**5**

**Example2(Creating checked exception and handling):**

**class NegativeNumberException extends Exception**

**{**

**}**

**class Demo**

**{**

**void cube(a) throws NegativeNumberException**

**{**

**if(a>0)**

**System.out.println(a\*a\*a);**

**else**

**throw new NegativeNumberException();**

**}**

**public static void main(String args[])**

**{**

**try{**

**int x=Integer.parseInt(args[0]);**

**Demo d=new Demo();**

**d.cube(x);**

**}catch(NegativeNumberException ne)**

**{**

**System.err.println(ne);**

**}**

**}**

**}**

**Example3(Creating unchecked exception):**

**class NegativeNumberException extends RuntimeException**

**{**

**}**

**class Demo**

**{**

**void cube(a) throws NegativeNumberException**

**{**

**if(a>0)**

**System.out.println(a\*a\*a);**

**else**

**throw new NegativeNumberException();**

**}**

**public static void main(String args[])**

**{**

**int x=Integer.parseInt(args[0]);**

**Demo d=new Demo();**

**d.cube(x);**

**}  
  
Java Streams:  
A stream is a flow of data from source to destination.  
A source can be a keyboard, file, client, server, ... etc.,  
A destination can be a monitor, file, client, server, .. etc.,  
  
In Java, streams are divided into 3 categories:  
1) Console Input/Output Streams  
2) File Input/Output Streams  
3) Network Input/Output Streams  
  
Predefined Streams:  
There are 3 predefined streams  
1) in  
2) out  
3) err  
  
"in" is an object reference of java.io. InputStream class  
"out" & "err" are object reference of java.io.PrintStream   
class.  
  
The above all predefined streams are static members of  
java.lang.System class.  
  
Differences between System.out & System.err  
  
System.out                              System.err  
1) It is used to display           1) It is used to display   
     output messages.                   error messages.  
2) This stream data can         2) This stream data cannot   
     be redirected to a file.            be redirected to a file.  
  
Example:  
class Demo  
{     
          public static void main(String args[])  
          {  
                    System.out.println("Core Java");  
                    System.err.println("Advanced Java");  
          }  
}  
  
C:\> javac Demo.java  
C:\> java Demo  
Output:  
Core Java  
Advanced Java  
  
C:\>java Demo>a.txt  
Output:  
Advanced Java  
C:\>start notepad a.txt  
Core Java appears in a a.txt file because Core Java   
written with System.out   
This stream data can be redirected to a file.  
  
File Streams:  
1) FileInputStream  
2) FileOutputStream  
  
FileInputStream used to read data from file.  
FileOutputStream used to write data to a file.  
  
Program to read data from file:  
class ReadDemo  
        {  
         public static void main(String args[])  
         {  
               try{  
         FileInputStream fis=new FileInputStream(args[0]);  
         int n=fis.available();  
         byte[] b=new byte[n];  
         fis.read(b);  
         String s =new String(b);  
         System.out.println(s);  
               }catch(Exception e)  
               {  
                     System.err.println(e);  
               }   
         }  
}  
  
Program to copy data from one file to another file:  
class CopyDemo  
        {  
         public static void main(String args[])  
         {  
               try{  
         FileInputStream fis=new FileInputStream(args[0]);  
         int n=fis.available();  
         byte[] b=new byte[n];  
         fis.read(b);  
      FileOutputStream fos=new FileOutputStream(args[1]);  
         fout.write(b);  
               }catch(Exception e)  
               {  
                     System.err.println(e);  
               }   
         }  
}  
  
There are two types of streams:  
1) Byte streams  
2) Character streams  
  
Byte streams handle all types of data where as character  
streams handle text only.  
  
List of byte stream classes:  
1) InputStream  
2) PrintStream  
3) FileInputStream  
4) FileOutputStream  
5) ObjectInputStream  
6) ObjectOutputStream  
7) DataInputStream  
8) DataOutputStream  
  
DataInputStream & DataOutputStream classes support  
all primitive data types & strings.  
  
ObjectInputStream & ObjectOutputStream classes   
support objects  
  
List of Character streams:  
1) FileReader  
2) FileWriter  
3) BufferedReader  
4) BufferedWriter  
  
Finally block:  
It is used to perform cleanup activities.  
Clean up activities are closing a file, closing a database  
connection, closing a socket, ..etc.,  
Finally block is executed even exception occurs in a   
program.  
  
Example:  
class ReadDemo  
        {  
         public static void main(String args[])  
         {  
               FileInputStream fis=null;  
               try{  
         fis=new FileInputStream(args[0]);  
         int n=fis.available();  
         byte[] b=new byte[n];  
         fis.read(b);  
         String s =new String(b);  
         System.out.println(s);  
               }catch(Exception e)  
               {  
                     System.err.println(e);  
               }   
               finally  
               {  
                         try{  
                         fis.close();  
                         }catch(Exception e)  
                         {  
                                System.err.println(e);  
                          }  
                }  
         }  
}  
  
Object Streams:  
1) ObjectInputStream:  
It is used to read an object from file  
  
2) ObjectOutputStream:  
It is used to write an object to a file  
  
Serialization:  
It is a process of converting object into a series of bits.  
In Java, object must be serializable to do the   
following operations:  
1) Writing object to a file  
2) Reading object from file  
3) Writing object to a network  
4) Reading object from network  
  
The class must implements java.io.Serializable interface   
to make serializable object   
  
java.io.Serializable interface is a marker interface,   
tag interface or empty interface because no members   
in this interface.  
  
Example:  
import java.io.\*;  
class Emp implements Serializable  
{  
 transient int empNo=101;  
 float salary=5000.00f;   
}  
class Demo  
{  
          public static void main(String args[])  
          {  
 try{  
 Emp e1=new Emp();  
 FileOutputStream fos=  
                                  new FileOutputStream("emp.txt");  
 ObjectOutputStream oos=  
                                           new ObjectOutputStream(fos);  
 oos.writeObject(e1);  
 oos.close();  
 fos.close();  
 FileInputStream fis=  
                                        new FileInputStream("emp.txt");  
 ObjectInputStream ois=new ObjectInputStream(fis);  
 Emp e2=(Emp)ois.readObject();  
 System.out.println(e2.empNo+"\t"+e2.salary);  
 ois.close();  
 fis.close();  
 }catch(Exception e)  
 {  
 System.err.println(e);  
 }  
          }  
  
}**

**transient keyword:  
It is used to prevent serialization. It is used in real   
time applications with passwords, PIN numbers,   
security code, ... etc.,**

**JDK 1.7 Features:  
1) Strings in switch statement  
2) try with resource statement  
3) Handling multiple exceptions with single catch block  
  
4) Generics type inference**

**1) Strings in switch statement:  
This feature allows to write strings in a switch statement.  
  
Example:  
class Demo  
{  
      public static void main(String args[])  
      {  
             switch(args[0])  
             {  
                    case "mon": System.out.println("Monday");  
                                          break;  
                    case "tue": System.out.println("Tuesday");  
                                          break;  
                   case "wed": System.out.println("Wednesday");  
                                          break;  
                    case "thu": System.out.println("Thursday");  
                                          break;  
                    case "fri": System.out.println("Friday");  
                                          break;  
                    case "sat": System.out.println("Saturday");  
                                          break;  
                    case "sun": System.out.println("Sunday");  
                                          break;**

**default: System.out.println("Invalid")**

**}  
      }  
}  
  
2) try with resource statement:  
This feature allows to write a resource with try block.  
A class that implements java.io.Closeable interface   
or java.lang.AutoCloseable interface only can be   
used as a resource.  
  
Syntaxes:  
1) try(resource)  
    {  
              ========  
              ========   
    }  
2) try(resource1; resource2; .......)  
     {  
 ========  
 ========  
     }  
  
The above resources are closed automatically even  
exception occurs in a program.  
This feature is an alternative to a finally block.  
  
Example:  
import java.io.\*;  
class ReadDemo  
{  
          public static void main(String args[])  
          {  
                    try(FileInputStream fis=  
                                     new FileInputStream(args[0])){   
             int n=fis.available();  
             byte[] b=new byte[n];  
             fis.read(b);  
             String s=new String(b);  
             System.out.println(s);  
             }catch(Exception e)  
             {  
               System.err.println(e);  
              }  
          }  
  
}  
  
Handling multiple exceptions with single catch block:  
This feature allows to write more than one exception   
in a catch block to handle.  
  
Example:  
try  
{  
   
}catch(IOException | InterruptedException e)  
{  
  
}  
  
Generics Type Inference:  
This feature allows write generic object in a new way.   
  
Example1:  
ArrayList<Integer> al=new ArrayList<Integer>();  
The above statement can be written from JDK 1.7   
onwards as follows:  
ArrayList<Integer> al=new ArrayList<>();  
  
Example2:  
  
HashMap<Integer, String> hm=  
                                     new HashMap<Integer, String>();  
The above statement can be written from JDK1.7   
onwards as follows:  
HashMap<Integer, String> hm=new HashMap<>();  
  
JDK 1.8 Features:  
Default methods in interface:  
This feature allows to write concrete instance method   
in interface by prefixing default keyword.  
  
Static methods in interface:  
This feature allows to write concrete class method   
in interface.  
  
The above two features are introduced in JDK 1.8   
version in 2014.  
  
Example:  
interface Test  
{  
 default void show()  
 {  
 System.out.println("show() method");  
 }  
 static void print()  
 {  
 System.out.println("print() method");  
 }  
}  
class Demo implements Test  
{  
          public static void main(String args[])  
          {   
               Test.print();   
         Test t=new Demo();  
          t.show();  
          }  
}**