Encapsulation

- binding data and code into single unit

Ex: class

```
class ClassOne{
       int no;
       String name;
       public void display(){
           s.o.p(no+"    "+name);
       }
 }
```

Inheritance

- inheriting the properties of one class to another class
- parent class, old class, super class, base class
- child class, new class, sub class, derived class

Ex:

```
class Employee{
        int empId;
        String empName;
        .....
}
class TeamMember extends Employee{
       String projectName;
       .....
}
class ProjectManager extends Employee{
       int numberOfProjects;
       ......
}
```

Types

* *Single Inheritance*
* *Multilevel Inheritance*
* *Hierarchical Inheritance*

- Single Inheritance

* only one base and one derived class

```
class ClassOne{
      ......
}
class ClassTwo extends ClassOne{
      ......
}
```

- Multilevel Inheritance

* one base class and derived class, derived class will be a base class for another derived class

```
class ClassOne{
      ........
}
class ClassTwo extends ClassOne{
      ..........
}
class ClassThree extends ClassTwo{
      .........
}
```

- Hierarchical Inheritance

* One base class, more than one derived class

```
class ClassOne{
       ......
}
class ClassTwo extends ClassOne{
       ........
}
class ClassThree extends ClassOne{
      ..........
}
```

Polymorphism

poly - many
morphism – forms

Types:

Compile time polymorphism - static binding - Early Binding (Function Overloading)

- compiler able to identify the method to be executed during compilation
- function overloading
- function name will be same but differ with either type of argument or number of arguments

```
class ClassOne{
     public void print(int n){
        .......
     }
     public void display(){
        .....
     }
}
class ClassTwo extends ClassOne{
    public void print(String s){
        ......
    }
    public static void main(String[] args){
        ClassTwo obj=new ClassTwo();
        obj.display();
        obj.print("Hi");
    }
}
```

```
public void print();
public int print();  // changing return type alone will  not be considered as Function
Overloading
```

Run time polymorphism - Dynamic Binding - Late Binging - Function Overriding

- method to be binded is identified during run time
- function overriding - method name will be same and arguments also same

```
class ClassA{
    public void area(){
      ....
    }
}
class ClassB extends ClassA{
    public void area(){
        ......
    }
    public static void main(String[] args){
        //static binding
        ClassB obj=new ClassB();
        obj.area();

        //dynamic binding
        ClassA obj=new ClassB();
        obj.area();
    }
}
```

Abstraction:

- providing essential features by hiding implementation
- what to do and not how to do

Abstract Class:

- defines super class that hold only method declaration, it does not have implementation
- can hold both abstract method and non-abstract method

Ex:

```
public abstract void print();
```

Need:

- subclass will get freedom to have its own implementation
- add new changes

Abstract Class:

- define abstract keyword in class
- abstract class cannot be instantiated
- If a class contain one abstract method, then the class should be mandatorily abstract class
- cannot use constructor as abstract method
- methods declared like private, final cannot be abstract method

Ex:

```
    abstract class Account{

            public void withdraw(){
                s.o.p("Withdraw method in account class");
            }

            public abstract void deposit();
            public abstract void print();
            public abstract void display();
    }
    class SavingsAccount extends Account{
            public void deposit(){
                 s.o.p("deposit in savings account class");
            }
            public void print(){
                   ........
            }
            public void display(){
                   .........
            }
    }
    class Tester{
           public static void main(String args[]){
                SavingsAccount obj=new SavingsAccount();
                Account obj1=new Account();  // not possible to instantiate
                //dynamic binding
                Account obj1=new SavingsAccount();
```

Interface:

- Interface can hold only abstract method
- Interface keyword used instead of class keyword
- tells what to do not how to do?
- inside interface all members are public
- Java8 added default methods and static methods in interface

Rules:

* A Class can implement more than one interfaces
* An Interface can extend more than one Interfaces
* Interfaces cannot be instantiated
* Can create reference but instantiated with child class

Ex:

```
interface InterfaceOne{
    public void display();
}
interface InterfaceTwo{
    public void print();
}
interface InterfaceThree extends InterfaceOne, InterfaceTwo{
    public void show();
}
class ClassOne implements InterfaceOne, InterfaceTwo{
    public void display(){
        ....
    }
    public void print(){
        ......
    }
}
class ClassTwo implements InterfaceThree{
    public void display(){
        ....
    }
    public void print(){
        ......
    }
    public void show(){
        .....
    }
 }
```

Default Methods and Static Methods:

```
  interface InterfaceOne{
        public void display();
        public default void print(){
            .......
        }
        public static void show(){
            .........
        }
    }
    class ClassOne implements InterfaceOne{
        public void display(){
            .......
        }
    }
```

Aggregation and Composition:

- composition - tightly coupled
- aggregation - loosely coupled

Composition:

- Account & Customer

```
    class Account{
        private int accNo;
        private double bal;
        public Account(int accNo, double bal){
           this.accNO=accNO;
           this.bal=bal;
        }
        //getter and setter methods
    }
    class Customer{
        private int custId;
        private String custName;
        private Account account;
        public Customer(int custId, String custName,int accNO, double bal){
           this.custId=custId;
           this.custName=custName;
           account=new Account(accNo, bal);
        }
        //getter and setter method
    }
    class Tester{
        public static void main(String[] args){
           Customer customer=new Customer(1,"ABC",10001,2000.0);
        }
```

Aggregation:

- department and faculty

```
class Faculty{
    priate int fId;
    private String fName;
    public Faculty (int fId, String fName){
        this.fId=fId;
        thid.fName=fName;
    }
    //getter and setter method
}
class Department{
    private int dId;
    private String dName;
    private Faculty faculty;
    public Department(int dId, String dName){
         this.dId=dId;
         this.dName=dName;
    }
    public void setFaculty(Faculty faculty){
         this.faculty=faculty;
    }
 }
 class Tester{
    public static void main(String[] args){
        Faculty f1=new Faculty(101, "ABC");
                       Faculty f1=new Faculty(102, "XYZ");

        Department d1=new Department(1,"CSE");
        Department d1=new Department(2,"EEE");

        d1.setFaculty(f1);
        d2.setFaculty(f2);
    }
 }
```

Static Members:

- static members are associated with the class instead of object
- **Types:**
    * Static Block
    * Static Method
    * Static Variable

Static Block:

- static block gets executed before main method

- static block executes only once
- used to initialize the static variables

```
class ClassOne{
      static int a;
      int y;
      static{
         s.o.p("static block");
         a=20;
         y=30;  // not possible to access non static variable inside static block
      }
      public static void main(String args[]){
          s.o.p("Main method");
          s.o.p(a);
      }
}
```

Static Variable:

- if a variable is declared as static, only one copy of the variable will be created
- static variables are shared among all the objects of the class
- static variable cannot be a local variable

```
class ClassOne{
    static int a=10;
    int b=20;
    public void change(){
        a=a+10;
        b=b+10;
    }
    public static void main(String[] args){
        ClassOne obj1=new ClassOne();
        obj.change();
        s.o.p(a);         //20
        s.o.p(obj1.b);    //30

        ClassOne obj2=new ClassOne();
        obj.change();
        s.o.p(a);         //30
        s.o.p(obj2.b);    //30

        ClassOne obj3=new ClassOne();
        obj.change();
        s.o.p(a);         //40
        s.o.p(obj3.b);    //30
     }
}
```

Static Method:

- static methods are invoked using class name
- static methods access only static variable
- non static method can access both static and non-static variable

```
    class ClassOne{
            int n;
            static int a;

            static void display(){
                 s.o.p("static method");
                 a=20;
                 n=15;    //can not access non static variable inside static method
            }

            public void print(){
                 n=25;
                 a=40;
            }

            public static void main(String[] args){

                ClassOne.display();
                ClassOne obj=new ClassOne();
                obj.print();
            }
     }
```

Final:

- Final is Immutable (cannot be changed during the execution of the program)
- can use final keyword for Class, Methods and Variables

Final Variable:

- Final variable is always be static
- value cannot be changed

```
      class ClassOne{
            static final int a=20;
            public static void change(){
                  a=25;   //can not change the value of final variable
            }
      }
```

Final Method:

- cannot be overridden

```
abstract class ClassOne{
    static final void display(){
        .......
    }
    public abstract void print();
}
class ClassTwo extends ClassOne{
    static void display(){   //can not be overridden
        .....
    }
    public void print(){
        ......
    }
}
```

Final Class:

- cannot be inherited

```
final class ClassOne{
        ......
}
class ClassTwo extends ClassOne{    // not possible to extends final class
        .......
}
```

Access Specifiers:

- protect the members of the class from the outside access
  * private
  * public
  * default
  * protected

	within class	**within package outside the class**	**outside the package but immediate subclass**	**outside the package**
private	yes	no	no	no
default	yes	yes	no	no
protected	yes	yes	yes	no
public	yes	yes	yes	yes

SOLID Principles:

- SRP - Single Responsibility Principle
- OCP - Open Closed Principle
- LSP - Liskov Substitution Principle
- ISP - Interface Seggregation Principle
- DIP - Dependency Inversion Principle

SRP:

- A class should have only one responsibility - should have only one reason

```
public class Employee{  // do not create class for mora than one reason
    public double calculatePay(){
        ........
    }
    public int reportHours(){
        .......
    }
    public void save(){
        .......
    }
}
```

OCP:

- software entities should be open for extension but closed for modification

```
public void area(String shape){
    if(shape.equals("square")){
        .......
    }else if(shape.equals("circle")){
        .......
    }else{
        ......
    }
}

public abstract class shape{
    public abstract double area();
}
public class Circle extends Shape{
    public void area(){
        ......
    }
}
public Sqaure extends Shape{
    public void area(){
        .....
    }
}
     public Triangle extends Shape{
    public void area(){
        .....
    }
}
```

LSP:

- subtypes must be substitutable for their base types

```
public abstract class Account{
     public abstract void deposit(double amount);
     public abstract void withdraw(double amount);
}
public class SavingsAccount extends Account{
     public abstract void deposit(double amount){
        s.o.p("deposit in Savings Account");
     }
     public abstract void withdraw(double amount){
        s.o.p("withdraw in Savings Account");
     }
}
      public class CurrentAccount extends Account{
     public abstract void deposit(double amount){
        s.o.p("deposit in Current Account");
     }
     public abstract void withdraw(double amount){
        s.o.p("withdraw in Current Account");
     }
}
      public class PPFAccount extends Account{
     public abstract void deposit(double amount){
        s.o.p("deposit in PPF Account");
     }
    public abstract void withdraw(double amount){   //Liskov substitution principle is not
    achieved
        s.o.p("withdraw not allowed in PPF Account");
     }
}

interface IWithdraw{
   public abstract void withdraw(double amount);
}
interface IDeposit{
   public abstract void deposit(double amount);
}
public class SavingsAccount implements IWithdraw, IDeposit{
     public abstract void deposit(double amount){
        s.o.p("deposit in Savings Account");
     }
     public abstract void withdraw(double amount){
        s.o.p("withdraw in Savings Account");
     }
}
      public class PPFAccount implements IDeposit{
     public abstract void deposit(double amount){
        s.o.p("deposit in PPF Account");
     }
}
```

ISP:

- the dependency of one class should depend on the small possible interfaces
- instead of using fat interface, can have many small interfaces
- classes should not be forced to implement interfaces that they dont use

DIP:

- depends on abstractions(interfaces) not upon concrete classes

```
enum OutputDevices {  PRINTER, DISK;  }

void copy(OutputDevices dev){
   if(dev==PRINTER){
         write();
   }else{
         read();
   }
}

interface Reader{
   char read();
}
interface Writer{
   char write(char ch);
}
void copy(Reader r, Writer w){
     ............
}
```

Exception Handling:

- uncertain event occur during execution of program that stops the program flow

```
public void print(){
    mark=80;
    avg=0;
    div=0;
    avg=mark/div;
    s.o.p(avg);
}

      public void print(){
    mark=80;
    avg=0;
    div=0;
          try{
       avg=mark/div;
    }catch(Exception e){
       e.printStackTrace();
    }
    s.o.p(avg);
 }
```

Defensive Coding:

```
public void print(){
    mark=80;
    avg=0;
    div=0;
    if(div!=0){
        avg=mark/div;
    }
    s.o.p(avg);
}
```

Exception Handling Keywords:

- **try -**
    - try keyword is used to specify a block where we should place an exception code
    - try must be followed by catch or finally
- **catch -**
    - catch is used to handle the exception
    - it may be followed by finally
- **finally -**
    - used to execute the mandatory code
- **throw -**
    - helps to throw an exception
    - can use one exception
    - it uses object name

- **throws -**
    - specify the chances for the exception to be occurred
    - can use more than one exception
    - it uses class name

Types:
  * Checked Exception
  * Unchecked Exception
  * Error

Checked Exception: Compiletime exception
  - checked exceptions are checked at compile time
  - checked exceptions are mandatory to handle it
  - checked exceptions are directly inherit Throwable class
    Ex: IOException, SQLException, FileNotFoundException

Unchecked Exception: run time exception
  - happens during runtime
  - optional to handle the exception
    Ex: ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException

Checked Exception:

```
public static void readFile() throws FileNotFoundException, IOException{
    try{
        FileReader fileReader=new FileReader("src/cgi/file1.txt");
    }catch(FileNotFoundException e){
        throw e;
    }
}

public static void main(String args[]){
     try{
          readFile();
     }catch(Exception e){
          e.printStackTrace();
     }
}
```

Unchecked Exception:

```
public void divideNumber(){
     Scanner sc=new Scanner(System.in);
     int num1;
     int num2;
     int num3;
```

```
    try{
        num1=sc.nextInt();
        num2=sc.nextInt();
                    s.o.p(num1);
        num3=num1/num2;

    }catch(ArithmeticException|InputMismatchException e){
        ........
    }
}

    try{
       .......
    }catch(ArithmeticException e){
         ........
    }catch(InputMismatchException e){
         ........
    }
```

Finally:

- place the code which needs to be executed mandatorily
- used to close the open resources

Ex:

```
Scanner sc;
try{
    sc=new Scanner(System.in);
    ........
}catch(Exception e){
     .......
}finally{
    sc.close();
}
```

Try with Resources:

- resources that are open in the try block will automatically get closed
- but the resource should implement either Closeable or AutoCloseable interface

```
try(Scanner sc=new Scanner(System.in)){
     ..........
}catch(Exception e){
     .............
}

try(Employee emp=new Employee()){
      .....
}catch(Exception e){
     .....
  }

  public class Employee implements Closeable{
       public static void close(){
           .....
       }
     .......
  }
```

User Defined Exception:

- we can create our own exception for user defined classes
- if student does not exist then can handle by creating StudentNotFoundException

Ex:

```
public class StudentNotFoundException extends Exception{
    public StudentNotFoundException(String msg){
        super(msg);
    }
}

public class Tester{
   public static void searchStudent(int index) throws StudentNotFoundException{
      String names[]={"ABC", "PQR", "XYZ"};
      if(index>2||index<0){
           throw new StudentNotFoundException("Student does not exist");
      }
      s.o.p(names[index]);
   }
   public static void main(String[] args){
       s.o.p("Enter the value between 0 and 2");
       try(Scanner sc=new Scanner(System.in)){
           int pos=sc.nextInt();
           searchStudent(pos);
       }catch(Exception e){
            .......
       }
    }
 }
```

Errors:

- Errors are external to the application program
- errors cannot be recovered
- errors occur in jvm

Ex: OutOfMemoryError, LinkageError, StackOverFlowError

```
public static void main(String[] args){
        String s[]={"aa","bb"};
        main(s);
    }
```

Anonymous Class:

- No name classes
- declaration and instantiation will be in a single expression

Ex:

```
abstract class Employee{
   public abstract void print();
}
class EmployeeDemo extends Employee{
   public void print(){
     s.o.p("print method in implementation class");
   }
   public static void main(String[] args){
      EmployeeDemo obj=new EmployeeDemo();
      obj.print();
   }
}

class EmployeeDemo extends Employee{
    public static void main(String[] args){
       EmployeeDemo obj=new EmployeeDemo(){
          public void print(){
            s.o.p("print inside anonymous class");
          }
         };
       obj.print();
    }
 }
```

Lambda Expression:

```
EmployeeDemo obj=()->s.o.p("print inside lambda expression");
obj.print();
```

IO Streams:

- Input Output Stream
- Streams: Sequence of data - it produces the data or consumes the data
- java.io package contains input and output stream

Types:

- Byte Stream:

- read and write the data in bytes
  - * InputStream

    * OutputStream

- Character Stream:

- read and write the character data
    * Reader
    * Writer

InputStream:

- InputStream is abstract for all the classes represent an input stream of bytes
- InputStream implements AutoCloseable interface
    * ByteArrayInputStream
    * BufferedInputStream
    * FileInputStream
    * ObjectInputStream
    * FilterInputStream
    * DataInputStream
- read() method used to read the data from the stream

program ------> Stream -------------> console

OutputStream:

- represent output stream of bytes
- it is also implements AutoCloseble interface and Flushable interface
    * ByteArrayOutputStream
    * BufferedOutputStream
    * FileOutputStream
    * ObjectOutputStream
    * FilterOutputStream
    * DataOutputStream
- write(), flush() method

Character Stream: Reader

- Reader is the super class for all the classes that reads the character data from stream
    * CharArrayReader
    * BufferedReader
    * FileReader

Character Stream: Writer

- Writer is the super class for all the classes that writes the character to stream
    * CharArrayWriter
    * BufferedWriter
    * FileWriter

Java NIO:

- Non Blocking Input Output - New Input Output
- IO Vs NIO
    * IO is stream oriented   10 20 30 40

* NIO is Buffer oriented
* IO is blocking io
* NIO is non blocking io

NIO:

- Paths
- Files
- Channels
- Selectors
- Buffers
- SocketChannel
- DatagramChannel

Object Class:

- Object class is the root class for all the classes

```
class Employee{      //class Employee extends Object
     ........
}
```

- in package java.lang
- methods of Object class are

equals
finalize()
notify()
notifyAll()
toString()
clone()
wait()

toString():

- if you want to return string representation of an object then we can use toString()

```
public class Employee{
     private int empId;
     private String empName;
     Employee(int empId, String empName){
        this.empId=empId;
        this.empName=empName;
     }
     @Override
     public String toString(){
         return "Employee";
     }
 }
```

Runtime Class:

- Java Runtime class helps to interact with Java Runtime Environment
- provides methods to execute a process, invoke GC, etc

- java.lang.Runtime
- methods

```
exit()
process Exec()
int availableProcessors()
freeMemory()
```

Ex:

```
class ClassOne{
   public static void main(String[] args){
       Runtime.getRuntime().exec("notepad.exe");
   }
}

// shutdown the system
Runtime.getRuntime().exec("c:\\windows\System32\\shutdown -s");
//restart the system
Runtime.getRuntime().exec("c:\\windows\System32\\shutdown -r");
//Memory
Runtime.getRuntime().totalMemory();
Runtime.getRuntime().freeMemory();
```

System classes:

- consists of Standard Input, Standard Output, Standard Error
- in, out, err

Methods:

```
getProperties()
getProperty()
setProperties()
setProperty()
gc()
lineSeparator()
arrayCopy()
```

Ex:

```
String s="Tanuj"+System.lineSeparator()+"kumar";
s.o.p(s);
```

output:

```
Tanuj
kumar
```

```
String name1[]={'k','u','m','a','r'};
String name2[]={'j','a','m','e','s'};
System.arrayCopy(name1,  1,  name2, 1,    1};
              source  pos  dest  pos  length
    output:
      names2  : jumes
```

Process Classes:

- provides control for various processes
- ProcessBuilder.start()
- destroy()
  isAlive()
  waitFor()
  exitValue()

Ex:

```
ProcessBuilder builder=new ProcessBuilder("notepad.exe");
Process p=builder.start();
p.destroy();
```

Nested Class:

- class within another class
- Types:
  * Static Nested Class
  * Non Static Nested Class

Ex:

```
class OuterClass{
    static String name="ABC";

    static class InnerClass{
        public void print(){
            s.o.p(name);
        }
    }
}

 OuterClass.InnerClass obj=new OuterClass.InnerClass();
 obj.print();
```