Classes & Objects

Class Fundamentals:
A class is declared by using class keyword. class is a template for an object, and an object is an instance of a class.
Syntax :

```
class classname {
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}
```

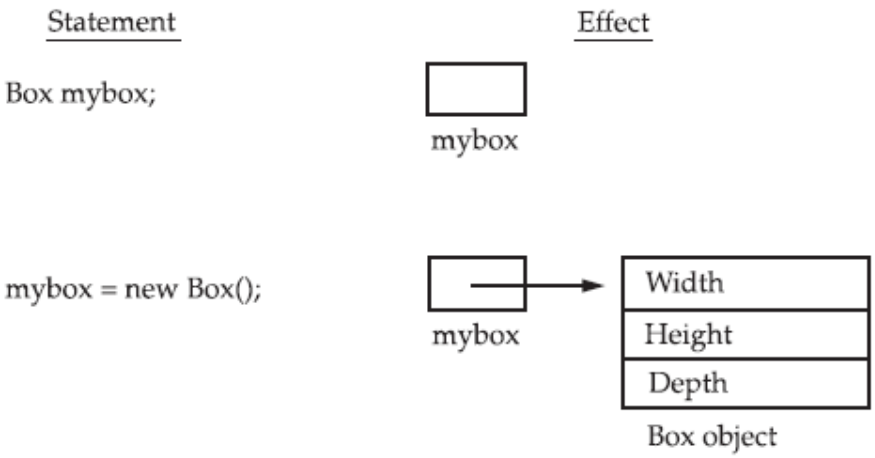
The data, or variables, defined within a class are called instance variables. The code is contained within methods. Collectively, the methods and variables defined within a class are called members of the class.

Declaring Objects/ Instantiating a Class:

Creating objects of a class is a two-step process.

- First, you must declare a variable of the class type which is simply a variable that can refer to an object.
- Second, you must acquire an actual, physical copy of the object and assign it to that variable using the new operator. The new operator dynamically allocates memory for an object and returns a reference to it where the address is stored.

```
Box mybox = new Box();
        OR
Box mybox;                  // declare reference to object
mybox = new Box();          // allocate a Box object
```



Object:

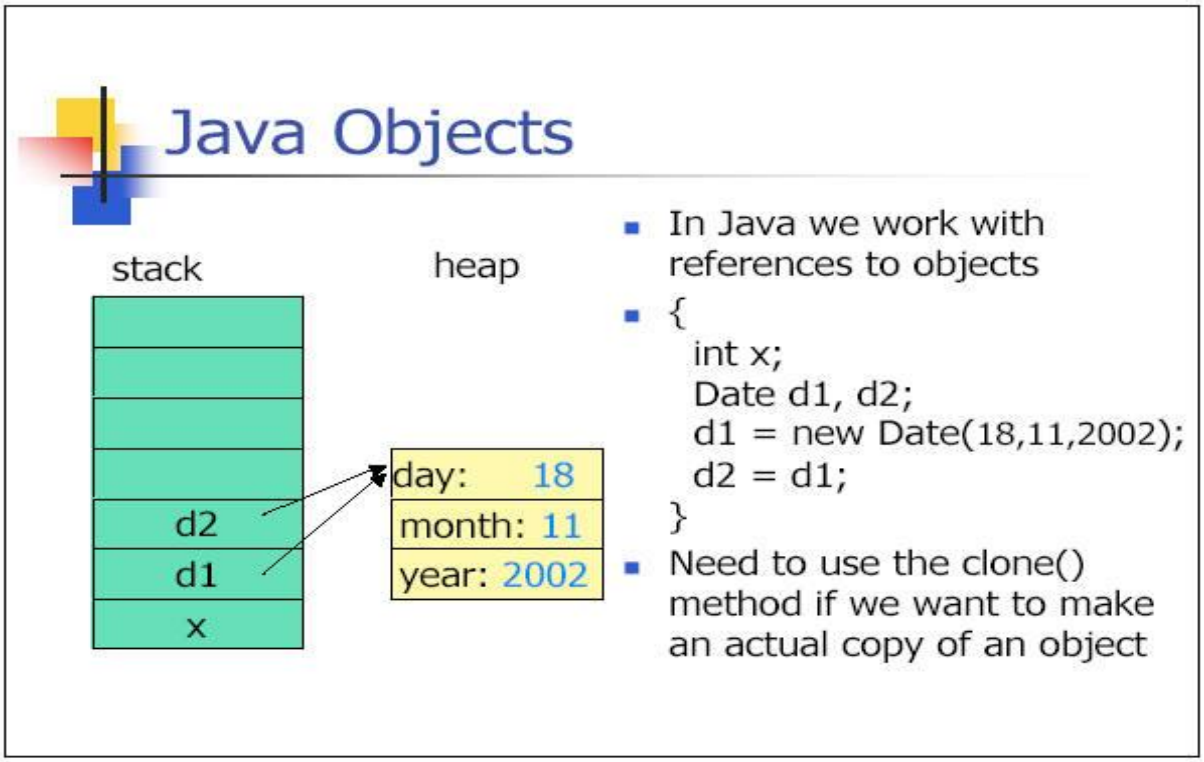
- Object is *a real world entity*.
- Object is *a run time entity*.
- Object is *an entity which has state and behavior*.
- Object is *an instance of a class*.

Java Heap Space

Java Heap space is used by java runtime to allocate memory to Objects and JRE classes. Whenever we create any object, it's always created in the Heap space. Garbage Collection runs on the heap memory to free the memory used by objects that doesn't have any reference.

Java Stack Memory

Java Stack memory is used for execution of a thread. They contain method specific values that are short-lived and references to other objects in the heap that are getting referred from the method. Stack memory is always referenced in LIFO (Last-In-First-Out) order



A Simple Class

```
class Box {
double width;
double height;
double depth;
}
                    // This class declares an object of type Box.
Class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;

mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;

vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

Introducing Methods

This is the general form of a method:

type name(parameter-list) {
// body of method

return value;
}

Methods define the interface to most classes. This allows the class implementer to hide the specific layout of internal data structures behind cleaner method abstractions. Defining methods provide access to data, you can also define methods that are used internally by the class itself.

```
Class Box {
  double width;
  double height;
  double depth;

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

void setDim(double w, double h, double d) {
        width = w;
        height = h;
       depth = d;
```

```
    }
}

class Demo {
public static void main(String args[]) {
Box mybox1 = new Box();
Box mybox2 = new Box();
double vol;

mybox1.setDim(10, 20, 15);
mybox2.setDim(3, 6, 9);

vol = mybox1.volume();
System.out.println("Volume is " + vol);

vol = mybox2.volume();
System.out.println("Volume is " + vol);
}
}
```

Constructors:

A *constructor* initializes an object immediately upon creation. It has the same name as the class in which it resides and is syntactically similar to a method.A constructor doesn't have a return type.The name of the constructor must be the same as the name of the class.Unlike methods, constructors are not considered members of a class.

A constructor is called automatically when a new instance of an object is created.

There are two types of constructors:

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

Default Constructor: It is a constructor which do not take any arguments.If you do not define any constructor in your class, java generates one for you by default.

```
Class Box {
double width;
double height;
double depth;

Box()
{
   System.out.println("Constructing Box");
   width = 10;
   height = 10;
   depth = 10;
}
```

```
double volume() {
return width * height * depth;
}
}
class demo
{
public static void main(String args[]) {
Box mybox1 = new Box();

double vol;

vol = mybox1.volume();
System.out.println("Volume is " + vol);
}
}
```

Parameterized constructor

A constructor that have parameters is known as parameterized constructor.

```
Class Student
{
   int id;
   String name;

   Student(int I,String n)
  {
   id = I;
   name = n;
   }

   void display()
        {
         System.out.println(id+" "+name);
        }
}

Class test{
   public static void main(String args[])
   {
   Student s1 = new Student(111,"Karan");
   Student s2 = new Student(222,"Aryan");
   s1.display();
   s2.display();
  }
}
```

Objects as Parameters

Using Objects as Parameters: So far, we have only been using simple types as parameters to methods. However, it is both correct and common to pass objects to methods. For example, consider the following short program

```
class Test {
int a, b;
Test(int i, int j) {
a = i;
b = j;
}
// return true if o is equal to the invoking object
boolean equalTo(Test o) {
if(o.a == a && o.b == b) return true;
else return false;
}
}
class PassOb {
public static void main(String args[]) {
Test ob1 = new Test(100, 22);
Test ob2 = new Test(100, 22);
Test ob3 = new Test(-1, -1);
System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
}
}
```

Exercise 1:
Write a java program to create 2 objects of complex numbers and pass these objects as parameters to the methods. Perform addition of 2 complex numbers and return the sum as an object.

The this Keyword

Java defines the **this** keyword. **It** can be used inside any method to refer to the *current o*bject.

```
Box(double w, double h, double d)
{
this.width = w;
this.height = h;
this.depth = d;
}
```

- **this** keyword is used to refer to current object.
- **this** is always a reference to the object on which method was invoked.
- **this** can be used to invoke current class constructor.
- **this** can be passed as an argument to another method.

Instance Variable Hiding:

Interestingly, you can have local variables, including formal parameters to methods, which overlap with the names of the class' instance variables. However, when a local variable has the same name as an instance variable, the local variable *hides* the instance variable.

```
class Student
{
int rollno;
String name;
float fee;
Student(int rollno,String name,float fee)
{
this.rollno=rollno;
this.name=name;
this.fee=fee;
}
void display()
{
System.out.println(rollno+" "+name+" "+fee);
}
}

class Test
{
public static void main(String args[])
{
Student s1=new Student(111,"ankit",5000f);
Student s2=new Student(112,"sumit",6000f);
s1.display();
s2.display();
}
}
```

Overloaded Constructors

Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists.The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

```
class Student{
   int id;
   String name;
   int age;
   Student (int i,String n)
  {
         id = i;
        name = n;
   }
   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[])
{
   Student5 s1 = new Student5(111,"Karan");
   Student5 s2 = new Student5(222,"Aryan",25);
   s1.display();
   s2.display();
   }
}
```

Role of this () in constructor overloading

/*this() is used for calling the default constructor from parameterized constructor. It should always be the first statement in constructor body. */

```
public class student
{
  private int rollNum;
  student()
  {
    rollNum =100;
  }
  student(int rnum)
  {
    this();
   rollNum = rollNum+ rnum;
  }
  public int getRollNum() {
          return rollNum;
  }
  public void setRollNum(int rollNum) {
          this.rollNum = rollNum;
  }
}
class TestDemo{
  public static void main(String args[])
  {
     student obj = new student(12);
     System.out.println(obj.getRollNum());
   }
}
```

Garbage Collection:

In some languages, such as C++, dynamically allocated objects must be manually released by use of a **delete** operator. Java takes a different approach; it handles deallocation for you automatically. The technique that accomplishes this is called *garbage collection*. It works like this: when no references to an object exist, that object is assumed to be no longer needed, and the memory occupied by the object can be reclaimed.

Advantage of Garbage Collection

- It makes java **memory efficient** because garbage collector removes the unreferenced objects from heap memory.
- It is **automatically done** by the garbage collector(a part of JVM) so we don't need to make extra efforts.

finalize() method

The finalize() method is invoked each time before the object is garbage collected. This method can be used to perform cleanup processing. This method is defined in Object class as:

```
protected void finalize()
{
//code
}
```

gc() method:

The gc() method is used to invoke the garbage collector to perform cleanup processing. The gc() is found in System and Runtime classes.

```
public class TestGarbage1{
 public void finalize(){System.out.println("object is garbage collected");}
 public static void main(String args[]){
  TestGarbage1 s1=new TestGarbage1();
  TestGarbage1 s2=new TestGarbage1();
  s1=null;
  s2=null;
  System.gc();
 }
}
```

```
object is garbage collected
      object is garbage collected
```

A Stack Class:

```
class Stack
{
int stck[] = new int[10];
int top;
// Initialize top-of-stack
Stack()
 {
top = -1;
```

```
}

void push(int item)
 {
if(top==9)
System.out.println("Stack is full.");
else
stck[++top] = item;
}

int pop()
{
if(top < 0) {
System.out.println("Stack underflow.");
return 0;
}
else
return stck[top--];
}
}

class TestStack
{
public static void main(String args[])
 {
Stack mystack1 = new Stack();
Stack mystack2 = new Stack();

// push some numbers onto the stack
for(int i=0; i<10; i++)
mystack1.push(i);
for(int i=10; i<20; i++)
mystack2.push(i);

System.out.println("Stack in mystack1:");
for(int i=0; i<10; i++)
System.out.println(mystack1.pop());

System.out.println("Stack in mystack2:");
for(int i=0; i<10; i++)
System.out.println(mystack2.pop());
}
}
```

Stack in mystack1:
9
8
7
6
5
4

```
3
2
1
0
Stack in mystack2:
19
18
17
16
15
14
13
12
11
10
```

Overloading Methods

In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. **Method overloading** is also known as **Static Polymorphism**.

Argument lists could differ in –

1. Number of parameters.
2. Data type of parameters.
3. Sequence of Data type of parameters.

When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method.

Advantage of method overloading

1) Method overloading *increases the readability of the program*.

```
class Calculate
{
 void sum (int a, int b)
 {
  System.out.println("sum is"+(a+b)) ;
 }
 void sum (float a, float b)
 {
  System.out.println("sum is"+(a+b));
 }
 Public static void main (String[] args)
 {
  Calculate  cal = new Calculate();
  cal.sum (8,5);      //sum(int a, int b) is method is called.
```

```
  cal.sum (4.6f, 3.8f); //sum(float a, float b) is called.
 }
}
```

```
Sum is 13
Sum is 8.4
```

```
class Overloading3
{
  public void disp(char c, int num)
  {
     System.out.println("c ");
     System.out.println("num ");

  }
  public void disp(int num, char c)
  {
     System.out.println("c ");
     System.out.println("num ");
  }
}
class Sample3
{
  public static void main(String args[])
  {
     Overloading3 obj = new Overloading3();
     obj.disp('x', 51 );
     obj.disp(52, 'y');
  }
}
```

Recursion:

Java supports *recursion*. Recursion is the process of defining something in terms of itself. As it relates to Java programming, recursion is the attribute that allows a method to call itself. A method that calls itself is said to be *recursive*.

```
class Factorial {

int fact(int n) {
int result;
if(n==1)
 return 1;
result = fact(n-1) * n;
return result;
}
}
class Recursion {
public static void main(String args[]) {
Factorial f = new Factorial();
```

```
System.out.println("Factorial of 3 is " + f.fact(3));
System.out.println("Factorial of 4 is " + f.fact(4));
System.out.println("Factorial of 5 is " + f.fact(5));
}
}
```

Advantages of Recursion

1. Reduces time complexity.
2. Performs better in solving problems based on tree structures.

Access Control

The access modifiers in java specifies accessibility (scope) of a data member, method, constructor or class.

There are 4 types of java access modifiers:

1. private
2. default
3. protected
4. public

public:

A class, method, constructor, interface etc declared public can be accessed from any other class. Therefore fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe. It has the widest scope among all other modifiers.

```
//save by A.java

package pack;
public class A
{
public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java

package mypack;
import pack.*;

class B
{
  public static void main(String args[]){
   A obj = new A();
   obj.msg();
  }
}
```

private:

Methods, Variables and Constructors that are declared private can only be accessed within the declared class itself. Private access modifier is the most restrictive access level. Class and interfaces cannot be private. Variables that are declared private can be accessed outside the class if public getter methods are present in the class.

```
class A
{
private int data=40;
private void msg()
{
System.out.println("Hello java");}
}

public class Simple
{
 public static void main(String args[])
{
  A obj=new A();
   System.out.println(obj.data);                //Compile Time Error
     obj.msg();                    //Compile Time Error
}
}
```

protected:

Variables, methods and constructors which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class. The protected access modifier cannot be applied to class and interfaces.

```
//save by A.java
package pack;
public class A{
protected void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B extends A{
  public static void main(String args[]){
   B obj = new B();
   obj.msg();
  }
}
```

Default:

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc. A variable or method declared without any access control modifier is available to any other class in the same package.

The fields in an interface are implicitly public static final and the methods in an interface are by default public.

```
//save by A.java
package pack;
class A{
  void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;
class B{
  public static void main(String args[]){
    A obj = new A();//Compile Time Error
    obj.msg();//Compile Time Error
  }
}
```

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Static:

It is a keyword which is used to define the class members that will be used independent of any object of that class. Static members are initialized for the first time when class is loaded.
The most common example of a **static** member is **main( )**. **main( )** is declared as **static** because it must be called before any objects exist.[i.e., without instantiating the class]

Static Methods
Methods declared as **static** have several restrictions:
• They can only directly call other **static** methods.
• They can only directly access **static** data.
• They cannot refer to **this** or **super** in any way.

Static Blocks:
Static blocks are also called *Static initialization blocks* . A static initialization block is a normal block of code enclosed in braces, { }, and preceded by the static keyword.

```
static {
   // whatever code is needed for initialization goes here
}
```

```
class UseStatic
 {
static int a = 3;
static int b;

static void display (int x)
{
System.out.println("x = " + x);
System.out.println("a = " + a);
System.out.println("b = " + b);
}

static
{
System.out.println("Static block initialized.");
b = a * 4;
}
public static void main(String args[])
 {
display (42);
}
}
```

```
Static block initialized.
x = 42
a = 3
b = 12
```

```
class StaticDemo
 {
       static int a = 42;
       static int b = 99;
 static void callme()
 {
   System.out.println("a = " + a);
  }
}
class StaticByName
{
public static void main(String args[]) {
```

```
StaticDemo.callme();
System.out.println("b = " + StaticDemo.b);
}
}
```

```
a = 42
b = 99
```

Outside of the class in which they are defined, **static** methods and variables can be used independently of any object. To do so, you need only specify the name of their class followed by the dot operator.

Inheritance:

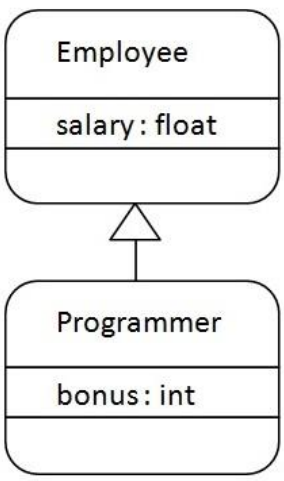
Inheritance in java is a mechanism in which one object acquires all the properties and behaviours of parent object. The idea behind inheritance in java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of parent class, and you can add new methods and fields also.
Inheritance represents the **IS-A relationship**, also known as *parent-child* relationship. The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).
extends is the keyword used to inherit the properties of a class.

Use of inheritance in java

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

Syntax of Java Inheritance

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



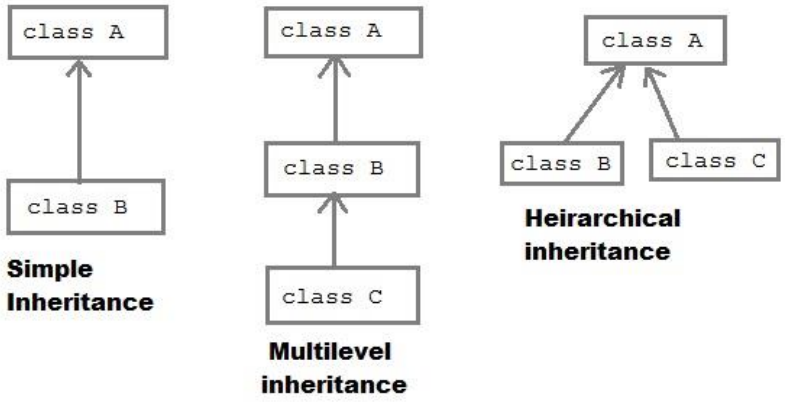
```
class Employee
{
 float salary=40000;
}

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

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

Types of Inheritance

1. Single Inheritance
2. Multilevel Inheritance
3. Hierarchical Inheritance



Single Level Inheritance :

One class extends one class only

```
class Animal
{
void eat()
{
    System.out.println("eating...");
}
}
class Dog extends Animal
{
void bark()
   {
```

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

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

Multilevel Inheritance:

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class.

```
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();
}
}
```

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

```
eating..
```

Hierarchical Inheritance :

In simple terms you can say that Hybrid inheritance is a combination of **Single** and **Multiple inheritance.** In **Hierarchical inheritance** one parent class will be inherited by **many** sub classes.

```
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();//Compile Time.Error
}}
```

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

Polymorphism:

Polymorphism in java is a concept by which we can perform a *single action by different ways*. Polymorphism is derived from 2 greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
There are two types of polymorphism in java:

- **compile time polymorphism and**
- **runtime polymorphism.**

Compile Time polymorphism can be achieved using overloading methods
Run Time Polymorphism can be achieved using overriding methods

Runtime Polymorphism
Runtime polymorphism or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.

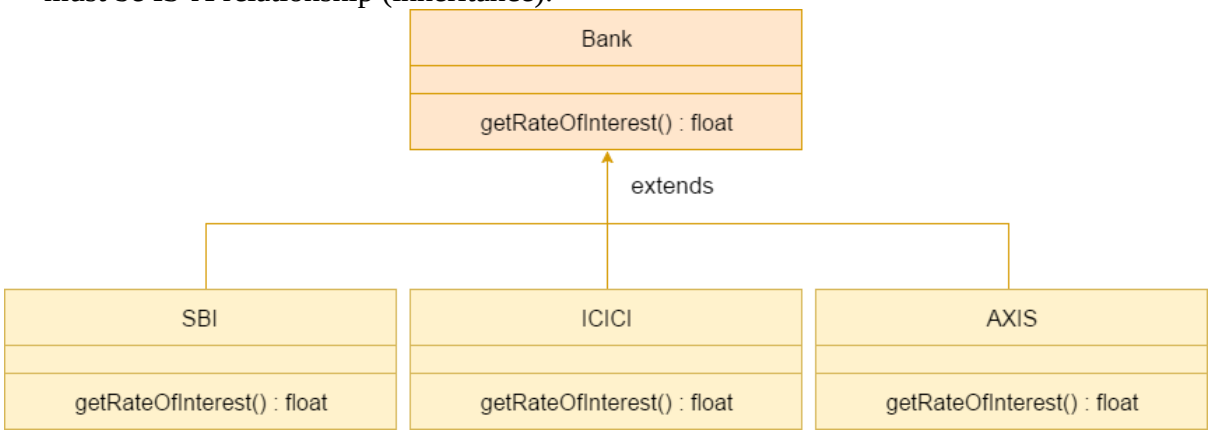
Child class has the same method as of base class. In such cases child class overrides the parent class method without even touching the source code of the base class.

Advantage of Java Method Overriding

- Method Overriding is used to provide specific implementation of a method that is already provided by its super class.
- Method Overriding is used for Runtime Polymorphism

Rules for Method Overriding

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



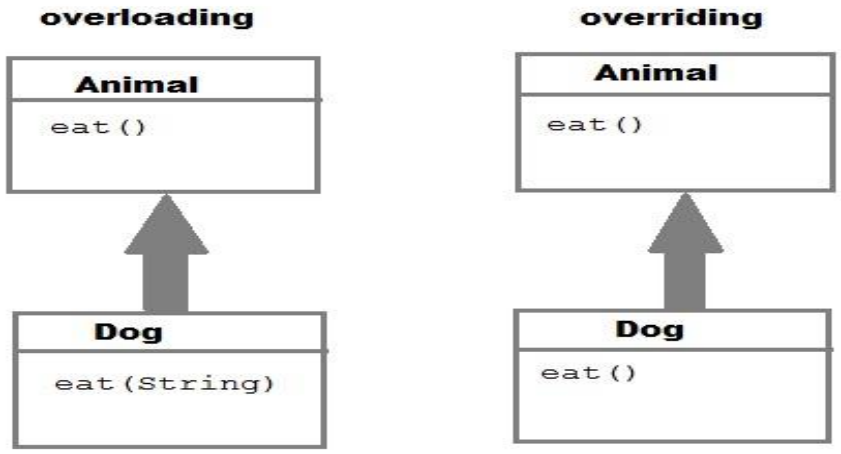

```
class Bank{
float getRateOfInterest(){
return 0;
}
}
class SBI extends Bank{
float getRateOfInterest(){
return 8.4f;
}
}
class ICICI extends Bank{
float getRateOfInterest(){
return 7.3f;
}
}
class AXIS extends Bank{
float getRateOfInterest(){
return 9.7f;
}
}
class TestPolymorphism{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("SBI Rate of Interest: "+b.getRateOfInterest());
b=new ICICI();
System.out.println("ICICI Rate of Interest: "+b.getRateOfInterest());
b=new AXIS();
```

```
System.out.println("AXIS Rate of Interest: "+b.getRateOfInterest());
}
}
```

```
SBI Rate of Interest: 8.4
ICICI Rate of Interest: 7.3
AXIS Rate of Interest: 9.7
```

Difference between Overloading and Overriding

	Overloading	**Overriding**
1	Whenever same method or Constructor is existing multiple times within a class either with different number of parameter or with different type of parameter or with different order of parameter is known as Overloading.	Whenever same method name is existing multiple time in both base and derived class with same number of parameter or same type of parameter or same order of parameters is known as Overriding.
2	Arguments of method must be different at least arguments.	Argument of method must be same including order.
3	Method signature must be different.	Method signature must be same.
4	Private, static and final methods can be overloaded.	Private, static and final methods can not be override.
5	Access modifiers point of view no restriction.	Access modifiers point of view not reduced scope of Access modifiers but increased.
6	Also known as compile time polymorphism or static polymorphism or early binding.	Also known as run time polymorphism or dynamic polymorphism or late binding.
7	Overloading can be exhibited both are method and constructor level.	Overriding can be exhibited only at method label.
8	The scope of overloading is within the class.	The scope of Overriding is base class and derived class.
9	Overloading can be done at both static and non-static methods.	Overriding can be done only at non-static method.
10	For overloading methods return type may or may not be same.	For overriding method return type should be same.



NOTE : Static methods cannot be overridden because, a static method is bounded with class where as instance method is bounded with object.

Super Keyword:

The **super** keyword in java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

1) super is used to refer immediate parent class instance variable.

This scenario occurs when a derived class and base class has same data members. Hence we use super keyword to refer a member of immediate parent class.

```
class Vehicle
{
   int maxSpeed = 120;
}


class Car extends Vehicle
{
   int maxSpeed = 180;

   void display()
   {
       System.out.println("Derived class Speed: " + maxSpeed);
      System.out.println("Base Speed: " + super.maxSpeed);
   }
}
```

```
/* Driver program to test */
class Test
{
   public static void main(String[] args)
   {
      Car small = new Car();
      small.display();
   }
}
```

```
Derived class Speed: 180
Base class Speed: 120
```

2) super can be used to invoke parent class method

The **super keyword** can also be used to invoke or call parent class method. It should be used in case of method overriding. In other word **super keyword** use when base class method name and derived class method name have same name.

```
class Student
{
void message()
{
System.out.println("Good Morning Sir");
}
}

class Faculty extends Student
{
void message()
{
System.out.println("Good Morning Students");
}

void display()
{
message();                    //will invoke or call current class message() method
super.message();              //will invoke or call parent class message() method
}

public static void main(String args[])
{
Student s=new Student();
s.display();
}
}
```

```
Good Morning Students
Good Morning Sir
```

3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke or call the parent class constructor. Constructor are calling from bottom to top and executing from top to bottom.
To establish the connection between base class constructor and derived class constructors JVM provides two implicit methods they are: If super is not used explicitly compiler will automatically add super as the first statement.

- Super()
- Super(...)

Super():

Super() It is used for calling super class default constructor from the context of derived class constructors.

```
class Employee
{
Employee()
{
System.out.println("Employee class Constructor");
}
}
class HR extends Employee
{
HR()
{
super();                    //will invoke or call parent class constructor
System.out.println("HR class Constructor");
}
}
class Supercons
{
public static void main(String[] args)
{
HR obj=new HR();
}
}
```

```
Employee class Constructor
HR class Constructor
```

Super(...)

Super(...) It is used for calling super class parameterize constructor from the context of derived class constructor.

```
class Person
{
int id;
String name;
```

```
Person(int id,String name)
{
this.id=id;
this.name=name;
}
}
class Emp extends Person
{
float salary;
Emp(int id,String name,float salary)
{
        super(id,name);                 //reusing parent constructor
        this.salary=salary;
}
void display()
{
System.out.println(id+" "+name+" "+salary);}
}

class TestSuper5
{
public static void main(String[] args)
{
Emp e1=new Emp(1,"abc",5000f);
e1.display();
}
}
```

```
1 abc 5000
```

Important rules

Rule for default constructor

Whenever the derived class constructor want to call default constructor of base class, in the context of derived class constructors we write super(). It is optional to write because every base class constructor contains single form of default constructor

Rule for Parameterized constructor

Whenever the derived class constructor wants to call parameterized constructor of base class in the context of derived class constructor we must write super(...). which is mandatory to write because a base class may contain multiple forms of parameterized constructors.

Abstract Classes

An *abstract class* is a class that is declared abstract—It can have abstract and non-abstract methods (method with body).Abstract classes cannot be instantiated, but they can be subclassed.

That is we cannot create an object for abstract classes

Abstraction is a process of hiding the implementation details and showing only functionality to the user.

Another way, it shows only important things to the user and hides the internal details for example sending sms, you just type the text and send the message. You don't know the internal processing about the message delivery.

- To use an abstract class, you have to inherit it from another class, provide implementations to the abstract methods in it.
- If you inherit an abstract class, you have to provide implementations to all the abstract methods in it.

Abstract method

Method that is declared without any body within an abstract class is called abstract method. The method body will be defined by its subclass. Abstract method can never be final and static. Any class that extends an abstract class must implement all the abstract methods declared by the super class.

Syntax :

abstract return_type function_name ();

```
abstract class Shape
{
abstract void draw();
}

class Rectangle extends Shape{
void draw(){
System.out.println("drawing rectangle");}
}

class Circle extends Shape{
void draw(){
System.out.println("drawing circle");}
}

class TestAbstraction
{
public static void main(String args[])
{
Rectangle r = new Rectangle();
r.draw();
```

```
Shape s=new Circle();
s.draw();
}
}
```

```
drawing rectangle
drawing circle
```

```
abstract class Bank{
abstract int getRateOfInterest();
}
class SBI extends Bank{
int getRateOfInterest(){return 7;}
}
class PNB extends Bank{
int getRateOfInterest(){return 8;}
}

class TestBank{
public static void main(String args[]){
Bank b;
b=new SBI();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
b=new PNB();
System.out.println("Rate of Interest is: "+b.getRateOfInterest()+" %");
}
}
```

```
Rate of Interest is: 7 %
Rate of Interest is: 8 %
```

Abstract classes with constructors

```
abstract class Bike
{
   Bike(){
    System.out.println("bike is created");
}
   abstract void run();

   void changeGear(){
System.out.println("gear changed");}
 }

 class Honda extends Bike{
 void run(){
System.out.println("running safely..");
}
 }
 class TestAbstraction2{
 public static void main(String args[]){
```

```
  Bike obj = new Honda();
  obj.run();
  obj.changeGear();
 }
}
```

```
 bike is created
    running safely..
    gear changed
```

When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviours at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Packages:

Packages in Java is a mechanism to encapsulate a group of classes, interfaces and sub packages.To create a package is quite easy: simply include a **package** command as the first statement in a Java source file. Any classes declared within that file will belong to the specified package. The **package** statement defines a name space in which classes are stored. If you omit the **package** statement, the class names are put into the default package, which has no name.

This is the general form of the **package** statement:

package *pkg*;

Java uses file system directories to store packages. For example, the **.class** files for any classes you declare to be part of **MyPackage** must be stored in a directory called **MyPackage**.

You can create a hierarchy of packages.

package *pkg1*[*.pkg2*[*.pkg3*]];

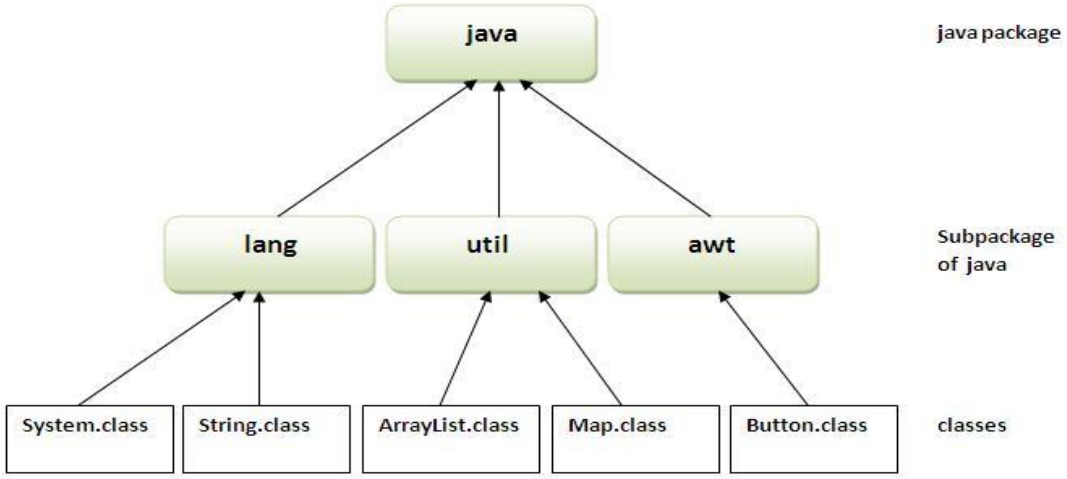
Ex: package java.awt.image;

Advantages of using a package

- ***Reusability:*** Reusability of code is one of the most important requirements in the software industry. Reusability saves time, effort and also ensures consistency. A class once developed can be reused by any number of programs wishing to incorporate the class in that particular program.
- ***Easy*** to locate the files.
- In real life situation there may arise scenarios where we need to define files of the same name. This may lead to "name-space collisions". Packages are a way of avoiding "name-space collisions".

Package are categorized into two forms

- **Built-in Package**:-Existing Java package for example java.lang, java.util etc.
- **User-defined-package:-** Java package created by user to categorized classes and interface



How to compile & Run java package?

If you are not using any IDE, you need to follow this:

Compile :- javac -d . Simple.java

Run :- java mypack.Simple

How to access package from another package?

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

1. import package.*;
2. import package.classname;
3. fully qualified name.

1) Using packagename.*

If you use package.* then all the classes and interfaces of this package will be accessible but not sub packages. The import keyword is used to make the classes and interface of another package accessible to the current package.

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
import pack.*;

class B{
```

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

2) *Using packagename.classname*

If you import package.classname then only declared class of this package will be accessible.

```
//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();
  }
}
```

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.

```
//save by A.java
package pack;
public class A{
  public void msg(){System.out.println("Hello");}
}
```

```
//save by B.java
package mypack;
class B{
  public static void main(String args[]){
   pack.A obj = new pack.A();//using fully qualified name
   obj.msg();
  }
}
```

Access Specifiers

- private: accessible only in the class
- default : so-called "package" access — accessible only in the same package
- protected: accessible (inherited) by subclasses, and accessible by code in same package
- public: accessible anywhere the class is accessible, and inherited by subclasses

Notice that private protected is not syntactically legal.

Access By	private	package	protected	public
the class itself	yes	yes	yes	yes
a subclass in same package	no	yes	yes	yes
non-subclass in same package	no	yes	yes	yes
a subclass in other package	no	no	yes	yes
non-subclass in other package	no	no	no	yes

Exception Handling in Java

An exception (or exceptional event) is a problem that arises during the execution of a program. When an **Exception** occurs the normal flow of the program is disrupted and the program/Application terminates abnormally, which is not recommended, therefore, these exceptions are to be handled.
An exception can occur for many different reasons. Following are some scenarios where an exception occurs.

- A user has entered an invalid data.
- A file that needs to be opened cannot be found.
- A network connection has been lost in the middle of communications or the JVM has run out of memory.

Some of these exceptions are caused by user error, others by programmer error, and others by physical resources that have failed in some manner.

Difference between error and exception

Errors indicate serious problems and abnormal conditions that most applications should not try to handle. Error defines problems that are not expected to be caught under normal circumstances by our program. For example memory error, hardware error, JVM error etc. **Exceptions** are conditions within the code. A developer can handle such conditions and take necessary corrective actions.

Few examples –

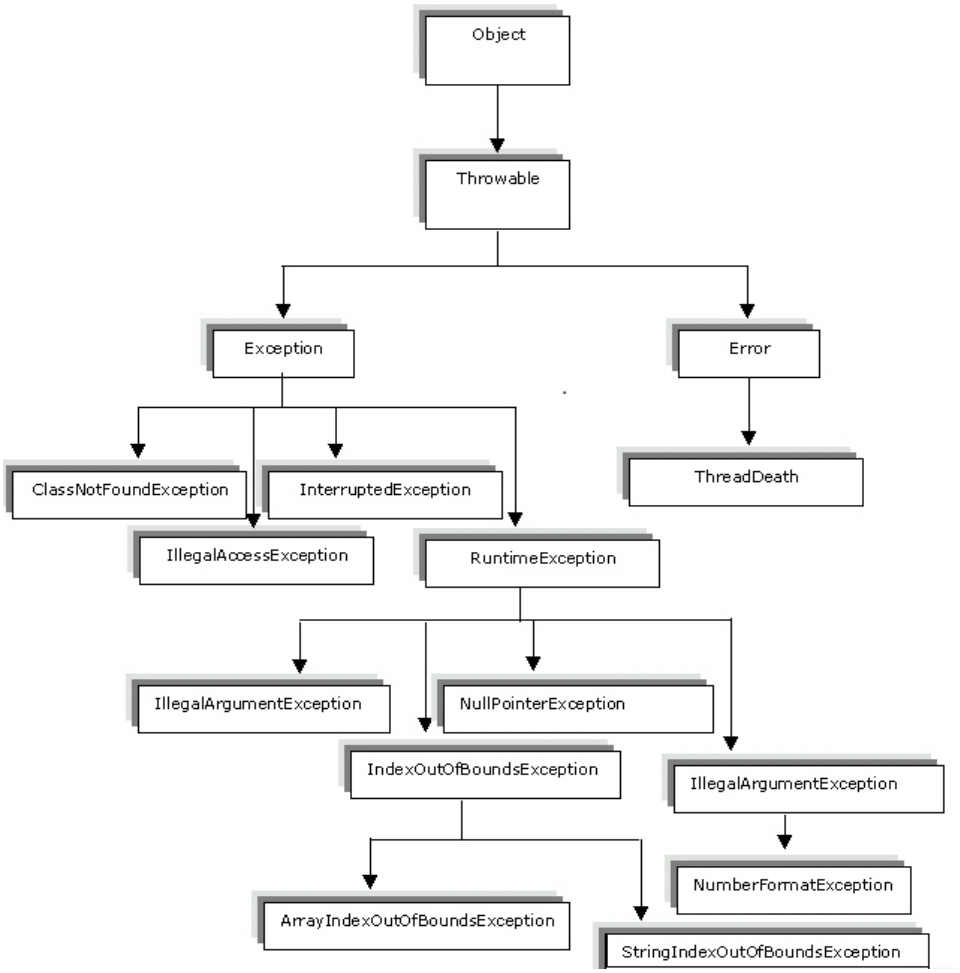
- DivideByZero exception
- NullPointerException
- ArithmeticException
- ArrayIndexOutOfBoundsException

Advantages of Exception Handling

- Exception handling allows us to control the normal flow of the program by using exception handling in program.

- It throws an exception whenever a calling method encounters an error providing that the calling method takes care of that error.
- It also gives us the scope of organizing and differentiating between different error types using a separate block of codes. This is done with the help of try-catch blocks.

Exception hierarchy



Java Exception Handling Keywords

Java provides specific keywords for exception handling purposes,

1. try
2. catch
3. finally
4. throw
5. throws

try-catch –

try is the start of the block and catch is at the end of try block to handle the exceptions. We can have multiple catch blocks with a try and try-catch block can be nested also. catch block requires a parameter that should be of type Exception.

A catch block must be associated with a try block. The corresponding catch block executes if an exception of a particular type occurs within the try block.

```
class Excp
{
 public static void main(String args[])
 {
  int a,b,c;
  try
```

```
import java.io.*;

public class ExcepTest
{
   public static void main(String args[])
 {
      try {
         int a[] = new int[2];
         System.out.println("Access element three :" + a[3]);
      }
    catch(ArrayIndexOutOfBoundsException e)
      {
         System.out.println("Exception thrown  :" + e);
      }
System.out.println("Out of the block");
   }
}
```

For example if an arithmetic exception occurs in try block then the statements enclosed in catch block for arithmetic exception executes.

Syntax of try catch in java

```
try
{
    //statements that may cause an exception
}
catch (exception(type) e(object))
{
    //error handling code
}
```

```
  {
     a=0;
     b=10;
     c=b/a;
      System.out.println("This line will not be executed");
  }
  catch(ArithmeticException e)
  {
   System.out.println("Divided by zero");
  }
  System.out.println("After exception is handled");
 }
}
```

Multiple Catch Blocks

A try block can be followed by multiple catch blocks.
If the try block **throws an exception**, the appropriate catch block (if one exists) will catch it
–catch(ArithmeticException e) is a catch block that can catch ArithmeticException
–catch(NullPointerException e) is a catch block that can catch NullPointerException

All the statements in the catch block will be executed and then the program continues.
If the exception type of exception, matches with the first catch block it gets caught, if not the exception is passed down to the next catch block.

```
class Example2{
  public static void main(String args[]){
    try{
        int a[]=new int[7];
        a[4]=30/0;
        System.out.println("First print statement in try block");
    }
    catch(ArithmeticException e){
       System.out.println("Warning: ArithmeticException");
    }
    catch(ArrayIndexOutOfBoundsException e){
       System.out.println("Warning: ArrayIndexOutOfBoundsException");
    }
    catch(Exception e){
       System.out.println("Warning: Some Other exception");
    }
  System.out.println("Out of try-catch block...");
  }
}
```

```
Warning: ArithmeticException
Out of try-catch block...
```

Java finally block

Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.Java finally block is always executed whether exception is handled or not.

Finally block is optional and can be used only with try-catch block. Since exception halts the process of execution, we might have some resources open that will not get closed, so we can use finally block. finally block gets executed always, whether exception occurred or not

```
class TestFinallyBlock1{
  public static void main(String args[]){
  try{
   int data=25/0;
   System.out.println(data);
  }
  catch(NullPointerException e)
    {
          System.out.println(e);
     }
 finally
    {
      System.out.println("finally block is always executed");
    }
  System.out.println("rest of the code...");
}
}
```

Output:finally block is always executed
Exception in thread main java.lang.ArithmeticException:/ by zero

Java throws keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

throws – When we are throwing any exception in a method and not handling it, then we need to use **throws** keyword in method signature to let caller program know the exceptions that might be thrown by the method. The caller method might handle these exceptions or propagate it to its caller method using throws keyword. We can provide multiple exceptions in the throws clause and it can be used with main() method also.

Syntax of java throws

```
return_type method_name() throws exception_class_name
{
//method code
}
```

throw –

We know that if any exception occurs, an exception object is getting created and then Java runtime starts processing to handle them. Sometime we might want to generate exception explicitly in our code, for example in a user authentication program we should throw exception to client if the password is null. **throw** keyword is used to throw exception to the runtime to handle it.

```
import java.io.*;
class M
{
 void method()throws IOException
{
  throw new IOException("device error");
 }
}
class Testthrows4
{
   public static void main(String args[]) throws IOException
{
     M m=new M();
     m.method();
    System.out.println("normal flow...");
  }
}
```

```
Exception in thread "main" java.io.IOException: device error
```

Interfaces:

An **interface in java** is a blueprint of a class. It has static constants and abstract methods.The interface in java is **a mechanism to achieve abstraction**. There can be only abstract methods in the java interface not method body. It is used to achieve abstraction and multiple inheritance in Java.

Java Interface also **represents IS-A relationship**.It cannot be instantiated just like abstract class.

There are mainly three reasons to use interface. They are given below.

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

Implementing Interfaces

A class uses the **implements** keyword to implement an interface.

Syntax:

```
access_modifier  interface nameofinterface
   {
   Function prototype1;
   Function Prototype 2;
   Type static final variable= value;
   }
```

Rules for using Interface

- Methods inside Interface must not be static, final, native or strictfp.
- All variables declared inside interface are implicitly public static final variables(constants).
- All methods declared inside Java Interfaces are implicitly public and abstract, even if you don't use public or abstract keyword.
- Interface can extend one or more other interface.
- Interface cannot implement a class.
- Interface can be nested inside another interface.

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

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

	abstract Classes	**Interfaces**
1	abstract class can extend only one class or one abstract class at a time	interface can extend any number of interfaces at a time
2	abstract class can extend from a class or from an abstract class	interface can extend only from an interface

3	abstract class can have both abstract and concrete methods	interface can have only abstract methods
4	A class can extend only one abstract class	A class can implement any number of interfaces
5	In abstract class keyword 'abstract' is mandatory to declare a method as an abstract	In an interface keyword 'abstract' is optional to declare a method as an abstract
6	abstract class can have protected , public and public abstract methods	Interface can have only public abstract methods i.e. by default
7	abstract class can have static, final or static final variable with any access specifier	interface can have only static final (constant) variable i.e. by default

```
abstract class Shape {

  abstract double area();

}
public interface Drawable {
    public abstract void draw();

}

public class Circle extends Shape implements Drawable {

    double radius;

    Circle(double aRadius){

    radius= aRadius;
}


   double area(){

       return Math.PI*radius*radius;
}

    public void draw(){
       System.out.println("This is a circle");
   }

}
Class test
{
Public static void main(String args[])
```

```
{
Shape obj= new Rectangle();
Shape obj1= new Circle();

obj.draw();
obj1.draw();
}
}
```

Final Class, Final methods & Final variables:

Final methods:

While method overriding is one of Java's most powerful features, there will be times when you will want to prevent it from occurring. To disallow a method from being overridden, specify **final** as a modifier at the start of its declaration. Methods declared as **final** cannot be overridden.

The following fragment illustrates **final**:

```
class Bike
{
  final void run()
{
System.out.println("running");
}
}

class Honda extends Bike
{
   void run()
{
    System.out.println("running safely with 100kmph");
}
 public static void main(String args[]){
   Honda honda= new Honda();
   honda.run();
    }
}
```

Complie Time Error as Final methods cannot be overridden

Final Class:

Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final. Declaring a class as final implicitly declares all of its methods as final, too. As you might expect, it is illegal to declare a class as both abstract and final since an abstract class is incomplete by itself and relies upon its subclasses to provide complete implementations.

Here is an example of a final class:

```
final class Bike{}

class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}

  public static void main(String args[]){
  Honda1 honda= new Honda();
  honda.run();
  }
}
```

```
Output:Compile Time Error
```

Final variable

If you make any variable as final, you cannot change the value of final variable (It will be constant).

```
class Bike
{
 final int speedlimit=90; //final variable
 void run()
{
  speedlimit=400;
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();  //Compile Time Error
 }
}
```

```
Compile Time Error
```