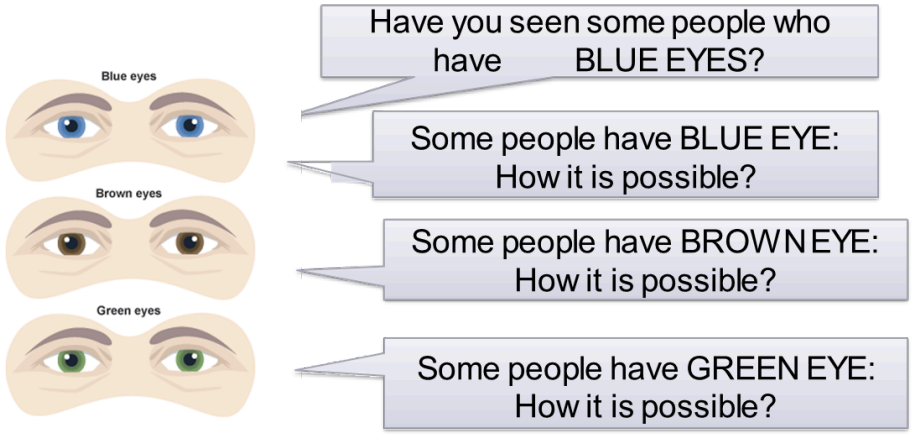
Java Day4

Inheritance in real world



Inheritance

- Inheritance is one of the cornerstones of OOP because it allows for the creation of hierarchical classifications
- Using inheritance, you can create a general class at the top
- This class may then be inherited by other, more specific classes
- Each of these classes will add only those attributes and behaviors that are unique to it

Generalization/ Specialization

- In keeping with Java terminology, a class that is inherited is referred to as a superclass
- The class that does the inheriting is referred to as the subclass
- Each instance of a subclass includes all the members of the superclass
- The subclass inherits all the properties of its superclass

Association, Aggregation, Composition

- These terms are used to signify the relationship between classes
- They are the basic building blocks of OOPS

Association

- Association is a relationship between two objects
- The association between objects could be
  - one-to-one
  - one-to-many
  - many-to-one
  - many-to-many
- Types of Association
  - Aggregation
  - Composition

- ***Example:*** A Student and a Faculty are having an association

Aggregation

- Aggregation is a special case of association
- A directional association between objects
- When an object 'has-a' another object, then you have got an aggregation between them
- Aggregation is also called a "Has-a" relationship.
- Example: College has a Student Object

Composition

- Composition is a special case of aggregation
- In a more specific manner, a restricted aggregation is called composition
- When an object contains the other object, if the contained object cannot exist without the existence of container object, then it is called composition
- ***Example:*** A class contains students. A student cannot exist without a class. There exists composition between class and students

IS-A relationship: Manager IS-A Employee

Their Manager

HAS-A relationship

- ***HAS-A*** relationship is expressed with containership
- Containership simply means using instance variables that refer to other objects
- Example:
- The class House will have an instance variable which refers to a Kitchen object
  - It means that, House HAS-A Kitchen
  - Note that, something like Kitchen HAS-A House is not valid in this context

HAS-A relationship (Contd.),

- Let us take one personal computer.
- It has a monitor, CPUbox, keyboard and mouse, etc.
- Technically we can say that,
  - Personal Computer class HAS-A monitor.
  - Personal Computer class HAS-A CPUbox
  - Personal Computer class HAS-A keyboard.
  - Personal Computer class HAS-A mouse.
  - The most important point is : the 4 independent components like monitor, keyboard, CPUbox and mouse cannot function separately on its own.
  - But, by combining them, we are creating a new type of useful class called Personal Computer.

Java's Inheritance Model

- Java uses the single inheritance model
- In single inheritance, a subclass can inherit from **one (and only one)** superclass

Code Syntax for Inheritance:

```
class derived-class-name extends base-class-name
 {
// code goes here
}
```

Inheritance – A Simple Example

```
class A{
  int m, n;
  void display1( ){
    System.out.println("m and n are:"+m+" "+n);
  }
}
class B extends A{
  int c;
  void display2( ){
    System.out.println("c :" + c);
  }
  void sum(){
    System.out.println("m+n+c = " + (m+n+c));
  }
}
```

Inheritance – A Simple Example (Contd.).

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

    A s1 = new A();    // creating objects
    B s2 = new B();
    s1.m = 10; s1.n = 20;

    System.out.println("State of object A:");
    s1.display1();
    s2.m = 7; s2.n = 8; s2.c = 9;
    System.out.println("State of object B:");
    s2.display1();
    s2.display2();
    System.out.println("sum of m, n and c in object B is:");
    s2.sum();
  }
}
```

Accessing Superclass Members from a Subclass Object

- A subclass includes all of the members of its superclass
- But, it cannot directly access those members of the super class that have been declared as **private**.

```
class A{
  int money;
  private int pocketMoney;

  void fill (int money, int pocketMoney)
  {
    this.money = money;
    this.pocketMoney = pocketMoney;
  }
}
```

Accessing Superclass Members from a Subclass Object (Contd.).

Will this compile now?

```
class B extends A{
  int total;
  void sum( ){
    total = money + pocketMoney;
  }  }
class AccessDemo
{
  public static void main(String args[ ])
  {
    B subob = new B();
    subob.fill(10,12);
    subob.sum();
    System.out.println("Total: " + subob.total);
  }
}
```

A Possible Solution To The Program

```
class  A{
  int money;
  private int pocketMoney;
  void fill(int money, int pocketMoney)
  {
    this.money = money;
    this.pocketMoney = pocketMoney;
  }
  public int getPocketMoney(){
    return pocketMoney;
  }
}
```

A Possible Solution To The Program (Contd.).

```
  class B extends A{
    int total;
    void sum( ) {
      total = money + getPocketMoney();     }
  }
  class AccessDemo {
    public static void main(String args[ ]) {
          B subob = new B();
          subob.fill(10,12);
          subob.sum();
          System.out.println("Total: " + subob.total);
    }
  }
```

Will this compile now?

Using super

- The creation and initialization of the superclass object is a prerequisite to the creation of the subclass object.
- When a subclass object is created,
  - It creates the superclass object
  - Invokes the relevant superclass constructor.
    - The initialized superclass attributes are then inherited by the subclass object
  - finally followed by the creation of the subclass object
    - initialization of its own attributes through a relevant constructor subclass

Using super (Contd.).

- The constructors of the superclass are never inherited by the subclass
- This is the only exception to the rule that a subclass inherits all the properties of its superclass

A Practical Example

Employee class should be put inside the mypack directory..

```
package mypack;
 class Employee {
  int Employeeno;     String Empname;
    Employee()        {
         System.out.println(" Employee No-arg Constructor Begins");
         Employeeno =0; Empname= null ;
         // the above assignments are unnecessary .. Why?
         System.out.println(" Employee No-arg Constructor Ends");
    }
    Employee(int Employeeno)       {
    System.out.println(" Employee 1-arg Constructor Begins");
    this.Employeeno=Employeeno;
    this.Empname= "UNKNOWN";
    System.out.println(" Employee 1-arg Constructor Ends");
    }
```

A Practical Example (Contd.).

```
    Employee(int Employeeno, String s) {
         System.out.println(" Employee 2-arg Constructor Begins");
         this.Employeeno = Employeeno;
         this.Empname  = s;
         System.out.println(" Employee 2-arg Constructor Ends");
    }
    void display()        {
         System.out.println(" Employee Number = "+Employeeno);
         System.out.println(" Employee Name = "+Empname);
    }

  } // End of the Employee class
```

A Practical Example (Contd.).

```
class Manager extends Employee
{
     String deptname;
     Manager(int Employeeno, String name, String deptname )
     {
          super(Employeeno, name);
          // parent class 2-arg constructor is called

          System.out.println(" Manager 3-arg Constructor Begins");
          this.deptname = deptname;
          System.out.println(" Manager 3-arg Constructor Ends");
        }
```

A Practical Example (Contd.).

```
  void display() {
         super.display();
         // parent class display() function is called
         System.out.println(" Deptname = "+deptname);
  }
public static void main( String a[])  {
       System.out.println(" [Main function Begins----------------------] ");
       System.out.println(" Creating an object for manager class
       "); Manager mm = new Manager(10,"Gandhi","Banking");
       System.out.println(" Printint the manager details .... : ");
       mm.display();
        System.out.println(" [Main function Ends----------------------] ");
       }
  }
```

Using super to Call Superclass Constructors

- super() if present, must always be the first statement executed inside a subclass constructor.
- It clearly tells you the order of invocation of constructors in a class hierarchy.
- Constructors are invoked in the order of their derivation

Constructors – Order of Invocation

Constructors in a class hierarchy are invoked in the order of their derivation.

```
class X  {
  X( ){
    System.out.println("Inside X's Constructor");  }  }

class Y  extends X {
  Y( ) {
    System.out.println("Inside Y's Constructor");  }  }

class Z extends Y {
  Z(){
    System.out.println("Inside Z's Constructor");  }  }

class OrderOfConstructorCallDemo{
  public static void main(String args[]){
              Z z = new Z();
  }
}
```

You can easily find the output of this program..

Constructors – Order of Invocation (Contd.),

- When we invoke a super() statement from within a subclass constructor, we are invoking the immediate super class' constructor
- This holds good even in a multi level hierarchy
- Remember, **super( )** can only be given as the first statement within a constructor

Using this() in a constructor

- this(argument list) statement invokes the **constructor** of the **same class**
- first line of a constructor must EITHER be a super *(call on the super class constructor)* OR a this *(call on the constructor of same class)*
- If the first statement within a constructor is NEITHER super() NOR this(), then the compiler will automatically insert a super(). (That is, invocation to the super class' no argument constructor)

Types of Inheritance

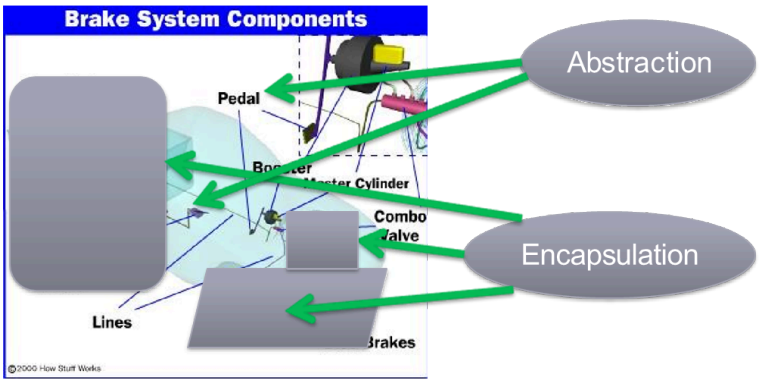
```
package com.Day4.Inheritance;


class GrandFather{
```

```
    public void land(){
        System.out.println("10 Acres of Land....");
    }
}
class Father extends GrandFather{
    public void house(){
        System.out.println("3BHK House....");
    }
}
class Son extends Father{
    public void car(){
        System.out.println("7 Seater Car...");
    }
}
class Daughter extends Father{
    public void house2(){
        System.out.println("Owned another big house...");
    }
}

public class InheritanceDemo {
    public static void main(String[] args) {
        Father father=new Father();
        father.house();
        System.out.println("--------------------------");
        Son son=new Son();
        son.car();
        son.house();
    }
}
```

Encapsulation and Abstraction



Encapsulation is hiding the implementation level details

Abstraction is exposing only the interface

Defining a Sample point Class

```
class Point {
   int x;    int y;
   void setX( int x){
    x = (x > 79 ? 79 : (x < 0 ? 0 :x)); }
  void setY (int y){
    y = (y > 24 ? 24 : (y < 0 ? 0 : y)); }
  int getX( ){ return x; }
  int getY( ){ return y;}
}
```

Class Declaration for Point

```
class Point{
  private int x;
  private int y;
  public void setX( int x){
    x= (x > 79 ? 79 : (x < 0 ? 0 :x));
  }
  public void setY (int y){
    y= (y > 24 ? 24 : (y < 0 ? 0 : y));
  }
  public int getX( ){
    return x;
  }
public int getY( ){
    return y;
  }
}
```

Class Declaration for Point (Contd.).

```
class PointDemo {
  public static void main(String args[ ] ){
    int a, b;
    Point p1 = new Point( );
    p1.setX(22);
    p1.setY(44);
    a = p1.getX( );
    System.out.println("The value of a is "+a);
    b = p1.getY( );
    System.out.println("The value of b is "+b);
  }
}
```

Expected Output :
The value of a is 22
The value of b is 24

Actual Output :
The value of a is 0
The value of b is 0

?

Class Declaration for Point - modified

```
class Point{
  private int x;
  private int y;
  public void setX( int x){
    this.x= (x > 79 ? 79 : (x < 0 ? 0 :x));
  }
  public void setY (int y){
    this.y= (y > 24 ? 24 : (y < 0 ? 0 : y));
  }
  public int getX( ){
    return x;
  }
public int getY( ){
    return y;
  }
}
```

Class Declaration for Point - modified (Contd.).

```
class PointDemo {
  public static void main(String args[ ] ){
    int a, b;
    Point p1 = new Point( );
    p1.setX(22);
    p1.setY(44);
    a = p1.getX( );
    System.out.println("The value of a is "+a);
    b = p1.getY( );
    System.out.println("The value of b is "+b);
  }
}
```

Output :
The value of a is 22
The value of b is 24

Encapsulation

```
package com.Day4.Encapsulation;
class Student{
    private int rno;
    private String name;
    private int marks;
    public Student(int rno,String name,int marks){
        this.rno=rno;
        this.name=name;
        this.marks=marks;
    }
    public int getRno(){
       return rno;
    }
    public String getName(){
        return name;
    }
    public int getMarks(){
        return marks;
    }
    public void setRno(int rno){
        this.rno=rno;
    }
    public void setName(String name){
        this.name=name;
    }
    public void setMarks(int marks){
        this.marks=marks;
    }

}
public class EncapsulationDemo {
    public static void main(String[] args) {
        int[] nums=new int[10];
        Student[] students=new Student[3];
        students[0]=new Student(101,"Jannath",85);
        students[1]=new Student(102,"Ashwin",93);
```

```
        students[2]=new Student(103,"Dheena",56);

        int total=0;
        for(Student temp:students){
            System.out.println("Name: "+temp.getName()+"\nRno: "+temp.getR
no()+"\nMarks: "+temp.getMarks());
            total+=temp.getMarks();
        }
        double average=(double)total/ students.length;
        System.out.println("Average Mark: "+average);
    }
}
```

Polymorphism

Method Overriding

- When a method in a subclass has the same prototype as a method in the superclass, then the method in the subclass is said to override the method in the superclass
- When an overridden method is called from an object of the subclass, it will always refer to the version defined by the subclass
- The version of the method defined by the superclass is hidden or overridden
- A method in a subclass has the same prototype as a method in its superclass if it has the same name, type signature (the same type, sequence and number of parameters), and the same return type as the method in its superclass. In such a case, a method in a subclass is said to override a method in its superclass.

Method Overriding(Contd.).

```
class A{
  int a,b;
  A(int m, int n){
    a = m;
    b = n;
  }
  void display(){
    System.out.println("a and b are :" + a +" " +      b);
  }
}
```

Method Overriding (Contd.).

```
class B extends A{
  int c;
  B(int m, int n, int o){
    super(m,n);
    c = o;
  }
  void display() {
    System.out.println("c :" + c);
  }
}
```

Method Overriding (Contd.).

```
class OverrideDemo{
  public static void main(String args[]){
    B subOb = new B(4,5,6);
    subOb.display();
  }
}
```

The output produced by this program is shown below:
c: 6
When display() is invoked on an object of class B, the version of display() defined within B is invoked. In other words, the version of display() inside B **overrides** the version of display declared in its superclass, i.e., class A.

Using super to Call an Overridden Method

```
class A{
  int a,b;
  A(int m, int n){
    a = m;
    b = n;
  }
  void display(){
    System.out.println("a and b are :" + a +" " +     b);
  }
}
class B extends A{
  int c;
```

Using super to Call an Overridden Method (Contd.).

```
  B(int m, int n, int o){
    super(m,n);
    c = o;
  }
  void display() {
    super.display();
    System.out.println("c :" + c);
  }
}
class OverrideDemo{
  public static void main(String args[]){
    B subOb = new B(4,5,6);
    subOb.display();
  }
}
```

Using super to Call an Overridden Method (Contd.).

- The following is the output of the aforesaid program:
  a and b: 4 5
  c: 6

- Method overriding occurs only when the names and the type signatures of two methods across at least two classes (i.e., a superclass and a subclass) in a class hierarchy are identical.
- If they are not, then the two methods are simply overloaded.
- Elaborating this theme further, if the method in a subclass has a different signature or return type than the method in the superclass, then the subclass will have two forms of the same method.

Superclass Reference Variable

- A reference variable of type superclass can be assigned a reference to any subclass object derived from that superclass.

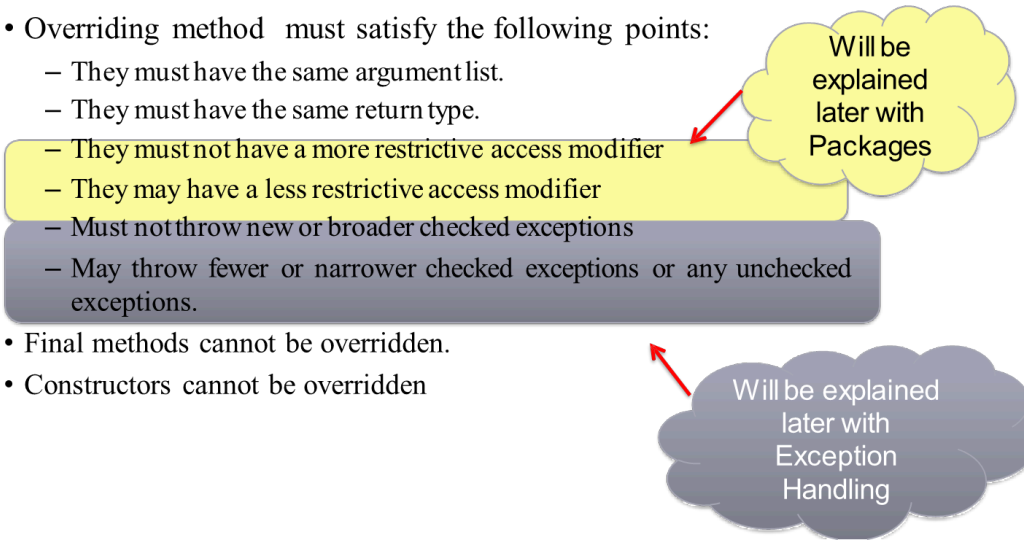
```
class  A1 {
}
class A2 extends A1 {
}
class A3 {
    public static void main(String[] args) {
        A1 x;
        A2 z = new A2();
        x = new A2();//valid
        z = new A1();//invalid
    }
```

A Superclass Reference Variable Can Reference a Subclass Object

- Method calls in Java are resolved dynamically at runtime
- In Java all variables know their dynamic type
- Messages (method calls) are always bound to methods on the basis of the dynamic type of the receiver
- This method resolution is done dynamically at runtime

Rules for method overriding

- Overriding method must satisfy the following points:
  - They must have the same argument list.
  - They must have the same return type.
  - They must not have a more restrictive access modifier
  - They may have a less restrictive access modifier
  - Must not throw new or broader checked exceptions
  - May throw fewer or narrower checked exceptions or any unchecked exceptions.
- Final methods cannot be overridden.
- Constructors cannot be overridden



Why Overridden Methods? A Design Perspective

- Overridden methods in a class hierarchy is one of the ways that Java implements the "**single interface, multiple implementations**" aspect of polymorphism
- Part of the key to successfully applying polymorphism is understanding the fact that the super classes and subclasses form a hierarchy which moves from lesser to greater specialization
- The superclass provides all elements that a subclass can use directly
- It also declares those methods that the subclass must implement on its own

- This allows the subclass the flexibility to define its own method implementations, yet still enforce a consistent interface

- In other words, the subclass will override the method in the superclass

Dynamic Method Dispatch or Runtime Polymorphism

- Method overriding forms the basis of one of Java's most powerful concepts: **dynamic method dispatch**

- Dynamic method dispatch occurs when the Java language resolves a call to an overridden method at runtime, and, in turn, implements runtime polymorphism

- Java makes runtime polymorphism possible in a class hierarchy with the help of two of its features:
  - superclass reference variables
  - overridden methods

- A superclass reference variable can hold a reference to a subclass object

- Java uses this fact to resolve calls to overridden methods at runtime

- When an overridden method is called through a superclass reference, Java determines which version of the method to call based upon the type of the object being referred to at the time the call occurs

Overridden Methods and Runtime Polymorphism - An Example

```
class Figure {
  double dimension1;
  double dimension2;
 Figure(double x, double y){
  dimension1 = x;
  dimension2 = y;
 }

 double area() {
  System.out.println("Area of Figure is undefined");
  return 0;
 }
}
```

Overridden Methods and Runtime Polymorphism – An Example (Contd.).

```
class Rectangle extends Figure {
     Rectangle(double x, double y) {            super(x,y);  }
      double area() //method overriding
      {
        System.out.print("Area of rectangle is :");
        return dimension1 * dimension2;
     }
}
class Triangle extends Figure {
  Triangle(double x, double y) {   super(x,y);  }
double area()     //method overriding {
     System.out.print("Area for triangle is :");
     return dimension1 * dimension2 / 2;
  }
}
```

Overridden Methods and Runtime Polymorphism - An Example (Contd.).

```
class FindArea {
  public static void main(String args[]){
    Figure f           = new Figure(10,10);
    Rectangle r        = new Rectangle(9,5);
    Triangle t         = new Triangle(10,8);
    Figure fig;    //reference variable

    fig = r;
    System.out.println("Area of rectangle is :" + fig.area());
    fig = t;
    System.out.println("Area of triangle is :" + fig.area());
    fig = f;
    System.out.println(fig.area());
  }
}
```

Runtime Polymorphism – Another Example

```
class BigB {
public void role() {
  System.out.println(" My name is BigB");
}
}

class FatherRole extends BigB
{
  // child class is overriding the role() method

public void role(){
    System.out.println("My role is Father when I am with my son
  !");
}
}


class DriverRole extends BigB{
  //child class is overriding the name() method
public void role(){
  System.out.println(" My role is Driver when I am driving a car!");
}
}

class CEORole extends BigB{
  //child class is overriding the name() method
public void role(){
  System.out.println(" My role is CEO when I am inside my own company
  ");
}
}


public  class Dyanmic_dispatch  {
public static void main(String ss[])  {

  System.out.println(" To demonstrate Runtime Polymorphism: ");

   BigB v;
// Parent class reference variable can point to
// any of its CHILD class objects....

  v = new BigB();          v.role();

  v= new FatherRole();     v.role();

  v= new DriverRole();     v.role();

  v= new CEORole();        v.role();
}
}
```

Use of instance of operator

- The instanceof operator in java allows you determine the type of an object
- The instanceof operator compares an object with a specified type
- It takes an object and a type and returns true if object belongs to that type. It returns false otherwise.
- We use instanceof operator to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements an interface

instanceof operator example

```
class A {
    int i, j;
}
class B extends A {
    int a, b;
}
class C extends A {
    int m, n;
}
```

instanceof operator example1

```
class InstanceOfImpl{
    public static void main(String args[ ]) {
        A a = new A( );
        B b = new B( );
        C c = new C( );
        A ob=b;
        if (ob instanceof B)
            System.out.println("ob now refers to B");
        else
            System.out.println("Ob is not instance of B");
        if (ob instanceof A)
            System.out.println("ob is also instance of A");
        else
            System.out.println("Ob is not instance of A");
        if (ob instanceof C)
            System.out.println("ob now refers to C");
        else
            System.out.println("Ob is not instance of C");
    }
}
```

ob now refers to B
ob is also instance of A
Ob is not instance of C

The Cosmic Class – The Object Class

- Java defines a special class called **Object.** It is available in java.lang package
- All other classes are subclasses of **Object**
- **Object** is a superclass of all other classes; i.e., Java's own classes, as well as user-defined classes
- This means that a reference variable of type **Object** can refer to an object of any other class
- Object defines the following methods, which means that they are available in every object

Method	Explanation
Object clone()	Create a new object that is the same as the object being cloned.
boolean equals(Object object)	Determines whether one object is equal to another.
void finalize()	Called before an unused object is reclaimed from the heap by the garbage collector
final Class getClass()	Obtains the class of an object at runtime
int hashCode	Returns the hash code associated with the invoking object
final void notify()	Resumes execution of a thread waiting on the invoking object
final void notifyAll()	Resumes execution of all waiting threads on the invoking object.
String toString()	Returns a string that describes the object.
final void wait final void wait(long milliseconds) final void wait(long milliseconds, long nanoseconds)	Waits on another thread of execution.

Method overloading

```
package com.Day4.PolyMorphism;
class MathUtility{
    public int add(){
        int num1=10;
        int num2=20;
        return num1+num2;
    }
    public int add(int num1,int num2){
        return num1+num2;
    }
    public double add(int num1,double num2){
        return num1+num2;
    }
    public int add(int num1,int num2,int num3){
```

```
        return num1+num2+num3;
    }
    public int max(int num1,int num2){
        return Math.max(num1,num2);
    }
    public int max(int num1,int num2,int num3){
        return Math.max(Math.max(num1,num2),num3);
    }
}
public class MethodOverloading {
    public static void main(String[] args) {
        MathUtility calc=new MathUtility();
        calc.add(10,50.25);
        System.out.println(calc.max(85,65,100));
    }
}
```

Method overriding

```
package com.Day4.PolyMorphism;

class Animal{
    void eat(){
        System.out.println("Animals can eat");
    }
}
class Cat extends Animal{
    void eat(){
        System.out.println("Cat can eat");
    }
}
class Dog extends Animal{
    void eat(){
        System.out.println("Dog can eat");
    }
}
public class MethodOverriding {
    public static void main(String[] args) {
```

```
        Animal animal=new Animal();
        animal.eat();
        Animal dog=new Dog();
        dog.eat();
        Animal cat=new Cat();
        cat.eat();
    }
}
```