Inheritance

Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object. It is an important part of OOPs (Object Oriented programming system).

The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.

Why use inheritance in java

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

Terms used in Inheritance

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

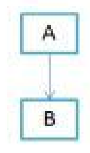
Extends:

The extends keyword extends a class (indicates that a class is inherited from another class). When a class needs to inherit the properties and behavior from another class, it has to use the extends keyword. In this way, the class being extended becomes the Parent class and the class that extends the Parent class becomes the Child class.

Types of Inheritance

Single Inheritance

When a class inherits another class, it is known as a *single inheritance*. In the example given below, Dog class inherits the Animal class, so there is the single inheritance.



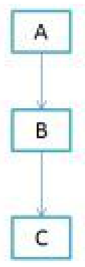
(a) Single Inheritance

```
class Animal{
String name="dog";
    void eat()
    {
    System.out.println("eating...");}
    }
    class Dog extends Animal{
    void bark()
    {
    System.out.println("I am"+name);
    System.out.println("barking...");}
    }
    class TestInheritance{
    public static void main(String args[]){
    Dog d=new Dog();
    d.bark();
    d.eat();
    }}
```

Multilevel Inheritance Example

When there is a chain of inheritance, it is known as *multilevel inheritance*. As you can see

the example given below, BabyDog class inherits the Dog class which again inherits the Animal class, so there is a multilevel inheritance.



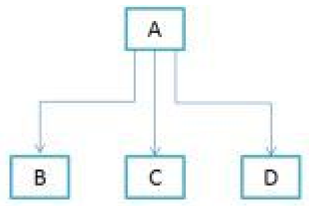
(d) Multilevel Inheritance

File: TestInheritance2.java

```
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();
}}
```

Hierarchical Inheritance Example

When two or more classes inherits a single class, it is known as *hierarchical inheritance*. In the example given below, Dog and Cat classes inherits the Animal class, so there is hierarchical inheritance.



(c) Hierarchical Inheritance

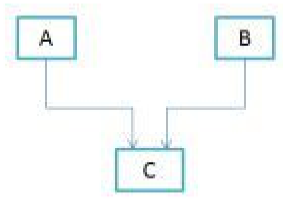
File: TestInheritance3.java

```
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();
}}
```

Multiple Inheritance

Multiple Inheritance” refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with “multiple inheritance” is that the derived class will have to manage the dependency on two base classes.



(b) Multiple Inheritance

- Note 1: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

- Note 2: Most of the new OO languages like Small Talk, Java, C# do not support Multiple inheritance. Multiple Inheritance is supported in C++.

```
interface AnimalEat {
   void eat();
}
interface AnimalTravel {
   void travel();
}
class Animal implements AnimalEat, AnimalTravel {
   public void eat() {
      System.out.println("Animal is eating");
   }
   public void travel() {
      System.out.println("Animal is travelling");
   }
}
public class Demo {
   public static void main(String args[]) {
      Animal a = new Animal();
      a.eat();
      a.travel();
   }
}
```

Super Keyword in Java

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

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

Output:

```
black
white
```

}

In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword.

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

We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.

```
class Animal{
String color="white";
}
class Dog extends Animal{
String color="black";
void printColor(){
System.out.println(color);//prints color of Dog class
System.out.println(super.color);//prints color of Animal class
}
}
class TestSuper1{
public static void main(String args[]){
Dog d=new Dog();
d.printColor();
}}
```

2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

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

```
void eat(){System.out.println("eating bread...");}
void bark(){System.out.println("barking...");}
void work(){
super.eat();
bark();
}
}
class TestSuper2{
public static void main(String args[]){
Dog d=new Dog();
d.work();  }}
```

super() can be used to invoke immediate parent class constructor.

```
class Animal{
Animal(){System.out.println("animal is created");}
}
class Dog extends Animal{
Dog(){
super();
System.out.println("dog is created");
}
}
class TestSuper3{
public static void main(String args[]){
Dog d=new Dog();
}}
```

Constant

- A constant is a variable whose value cannot change once it has been assigned. Java doesn't have built-in support for constants.
- A constant can make our program more easily read and understood by others. In addition, a constant is cached by the JVM as well as our application, so using a constant can improve performance.
- To define a variable as a constant, we just need to add the keyword “final” in front of the variable declaration.
- Syntax

```
final float pi = 3.14f;
```

Java Final Keyword

In Java, the final keyword is used to denote constants. It can be used with variables, methods, and classes.

Once any entity (variable, method or class) is declared final, it can be assigned only once. That is,

- the final variable cannot be reinitialized with another value
- the final method cannot be overridden
- the final class cannot be extended

1) Java final variable

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

```
class Bike9{
 final int sl=90;//final variable
 void run(){
  sl=400;
System.out.println("Speed limit="+sl);
 }
 public static void main(String args[]){
 Bike9 obj=new  Bike9();
 obj.run();
 }
}
```

```
Output:Compile Time Error
```

2) Java final method

If you make any method as final, you cannot override it.

Example of final method

```
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();
   }
}
```

```
Output:Compile Time Error
```

3) Java final class

If you make any class as final, you cannot extend it.

Example of final class

```
final class Bike{
void abc()
{
out.println("running safely with 100kmph");
}}
class Honda1 extends Bike{
  void run(){System.out.println("running safely with 100kmph");}
  public static void main(String args[]){
  Honda1 honda= new Honda1();
  honda.run();  }
  }
}
```

```
Output:Compile Time Error
```

Method Overloading in Java

If a class has multiple methods having same name but different in parameters, it is known as **Method Overloading**.

Advantage of method overloading

Method overloading *increases the readability of the program*.

Different ways to overload the method

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1.) Method Overloading: changing no. of arguments

In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
void  add(int a,int b)
{
int s=a+b;
System.out.println(s);
}
void add(int a,int b,int c)
{
int s= a+b+c;
System.out.println(s);
}
public static void main(String[] args){
Adder ad=new Adder();
ad.add(1,2);
ad.add(1,2,3)
}}
```

2) Method Overloading: changing data type of arguments

In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

```
class Adder{
 void add(int a, int b)
{
Int s= a+b;
System.out.println(s);
}
 Void add(double a, double b)
{
double s=a+b;
System.out.println(s);

}
public static void main(String[] args){
Adder ad=new Adder();
ad.add(1,2);
ad.add(1.5,2.5)
}}
```

3. Changing the Order of the Parameters of Methods

```
class Test3{

static double add(int a,double b)

{

return (a+b);

}

static double add(double a,int b)
{
return (a+b);
}
public static void main(String args[])
{
System.out.println(add(1,1.2));
System.out.println(add(1.2,1));
}}
```

Method Overriding in Java

If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java**.

In other words, If a subclass provides the specific implementation of the method that has been declared by one of its parent class, it is known as method overriding.

Usage of Java Method Overriding

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

Example:

```
class Vehicle{
  void run(){System.out.println("Vehicle is running");}
}
class Bike2 extends Vehicle{
 void run(){System.out.println("Bike is running safely");}
  public static void main(String args[]){
  Bike2 obj = new Bike2();//creating object
  obj.run();//calling method
  }
}
```

Output:

```
Bike is running safely
```

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

Points to Remember

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.(we cannot create object of abstract class)
- It can have constructors and static methods also.
- It can have final methods which will force the subclass not to change the body of the method.(abstract class ma nabhayeko methods harulai child class ma use garna sakidaina)

Abstract methods

Abstract methods are those types of methods that don't require implementation for its declaration. These methods don't have a body which means no implementation. A few properties of an abstract method are:

- An **abstract method in Java** is declared through the keyword "abstract".
- While the declaration of the abstract method, the abstract keyword has to be placed before the name of the method.
- There is no body in an abstract method, only the signature of the method is present.
- An **abstract method in Java** doesn't have curly braces, but the end of the method will have a semicolon (;)

```
abstract class Animal {
  abstract void makeSound();
  public void eat() {
    System.out.println("I can eat.");
  }}
class Dog extends Animal {
  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}
class Main {
  public static void main(String[] args) {
    // create an object of Dog class
    Dog d1 = new Dog();
    d1.makeSound();
    d1.eat();
  }
}
```

Java Interface

An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

We use the `interface` keyword to create an interface in Java. For example,

```
interface Language {

  public void getType();

  public void getVersion();
}
```

Implementing an Interface

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the `implements` keyword to implement an interface.

Like abstract classes, we cannot create objects of interfaces.

To use an interface, other classes must implement it. We use the `implements` keyword to implement an interface.

Example 1: Java Interface

```
interface Polygon {
  void getArea(int length, int breadth);
}
// implement the Polygon interface
class Rectangle implements Polygon {

  // implementation of abstract method
  public void getArea(int length, int breadth) {
    System.out.println("The area of the rectangle is " + (length * breadth));
}
}
class Main {
  public static void main(String[] args) {
    Rectangle r1 = new Rectangle();
    r1.getArea(5, 6);
  }
}
```

Advantages of interface

- Similar to abstract classes, interfaces help us to achieve abstraction in Java.

Here, we know getArea() calculates the area of polygons but the way area is calculated is different for different polygons. Hence, the implementation of getArea() is independent of one another.

- Interfaces provide specifications that a class (which implements it) must follow.

  In our previous example, we have used getArea() as a specification inside the interface Polygon. This is like setting a rule that we should be able to get the area of every polygon.

  Now any class that implements the Polygon interface must provide an implementation for the getArea() method.
- Interfaces are also used to achieve multiple inheritance in Java. For example,

```
interface Line {
}
interface Polygon {
}
class Rectangle implements Line, Polygon {
}
```

Here, the class Rectangle is implementing two different interfaces. This is how we achieve multiple inheritance in Java.

Note: All the methods inside an interface are implicitly public and all fields are implicitly public static final

Implementing multiple inheritance in java using interface

```
interface AnimalEat {
  void eat();
}
interface AnimalTravel {
  void travel();
}
```

```
class Animal implements AnimalEat, AnimalTravel {
   public void eat() {
      System.out.println("Animal is eating");
   }
   public void travel() {
      System.out.println("Animal is travelling");
   }
}
public class Demo {
   public static void main(String args[]) {
       Animal a = new Animal();
      a.eat();
      a.travel();
   }
}
```

Java Package

A **java package** is a group of similar types of classes, interfaces and sub-packages.
Package in java can be categorized in two form, built-in package and user-defined package.
There are many built-in packages such as java, lang, awt, javax, swing, net, io, util, sql etc.
Here, we will have the detailed learning of creating and using user-defined packages.

Advantage of Java Package

1) Java package is used to categorize the classes and interfaces so that they can be easily maintained.
2) Java package provides access protection.
3) Java package removes naming collision.

Upcasting and Downcasting in Java

A process of converting one data type to another is known as **Typecasting** and **Upcasting** and **Downcasting** is the type of object typecasting. In Java, the object can also be typecasted like the datatypes. **Parent** and **Child** objects are two types of objects. So, there are two types of typecasting possible for an object, i.e., **Parent to Child** and **Child to Parent** or can say **Upcasting** and **Downcasting**.

1) Upcasting

Upcasting is a type of object typecasting in which a **child object** is typecasted to a **parent class object**. By using the Upcasting, we can easily access the variables and methods of the parent class to the child class. Here, we don't access all the variables and the method. We access only some specified variables and methods of the child class. **Upcasting** is also known as **Generalization** and **Widening**.

UpcastingExample.java

```
class  Parent{
  void PrintData() {
    System.out.println("method of parent class");
  }  }
class Child extends Parent {
  void PrintData() {
    System.out.println("method of child class");
  }  }
class UpcastingExample{
  public static void main(String args[]) {
   Parent obj1 = (Parent) new Child();
    obj1.PrintData();
      } }
```

Downcasting

Upcasting is another type of object typecasting. In Upcasting, we assign a parent class reference object to the child class. In Java, we cannot assign a parent class reference object to the child class, but if we perform downcasting, we will not get any compile-time error. However, when we run it, it throws the **"ClassCastException"**. Now the point is if downcasting is not possible in Java, then why is it allowed by the compiler? In Java, some scenarios allow us to perform downcasting. Here, the subclass object is referred by the parent class.

Below is an example of downcasting in which both the valid and the invalid scenarios are explained:

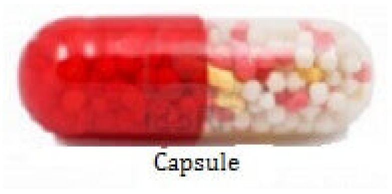
DowncastingExample.java

```
//Parent class
class Parent {
   String name;
      void showMessage()
   {
      System.out.println("Parent method is called");
   }
}

class Child extends Parent {
   int age;

   @Override
   void showMessage()
   {
      System.out.println("Child method is called");
   }
}

public class Downcasting{

   public static void main(String[] args)
   {
      Parent p = new Child();
      p.name = "Shubham";



      Child c = (Child)p;
```

```
    c.age = 18;
    System.out.println(c.name);
    System.out.println(c.age);
    c.showMessage();
  }
}
```

Encapsulation in Java

Encapsulation in Java is a *process of wrapping code and data together into a single unit*, for example, a capsule which is mixed of several me0dicines.



We can create a fully encapsulated class in Java by making all the data members of the class private. Now we can use setter and getter methods to set and get the data in it.

```
class Person {

 // private field
 private int age;


 // setter method
 public void setAge() {
   age=20;
 }
// getter method
 public int getAge() {
    return age;
  }

}

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

    // create an object of Person
    Person p1 = new Person();

    // change age using setter
    p1.setAge();

    // access age using getter
    System.out.println("My age is " + p1.getAge());
  }
}
```

In the above example, we have a `private` field `age`. Since it is `private`, it cannot be accessed from outside the class.

In order to access `age`, we have used `public` methods: `getAge()` and `setAge()`. These methods are called getter and setter methods.

Making `age` private allowed us to restrict unauthorized access from outside the class. This is **data hiding**.