Chapter 4.1

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.

Inheritance represents the IS-A relationship which is also known as a parent-child relationship

Advantages:

- *Code 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.
- *Extendability*: child classes can be added per requirement

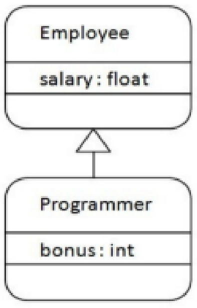
The extends keyword is used to inherit the features of parent class.

The class that inherits the features is called sub class or derived class or child class whereas the class from which the features are inherited is called super class or base class or parent class.

Syntax of java inheritance:

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

Java Inheritance Example



As displayed in the above figure, Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS- A Employee.** It means that Programmer is a type of Employee.

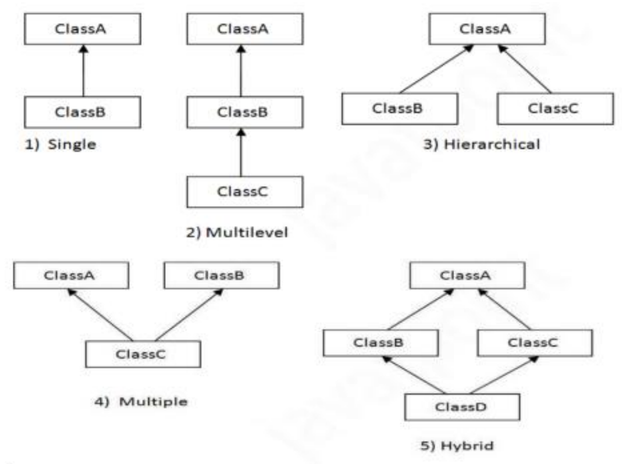
```
class Employee {
float salary=40000;
}
class Programmer extends Employee {
int bonus=10000;
public static void main(String[] args) {
Programmer p=new Programmer();
System.out.println("Programmer salary is:"+p.salary);
System.out.println("Bonus of Programmer is:"+p.bonus);
}
}
```

Output:

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

Types of Inheritance in Java

On the basis of class, there can be three types of inheritance in java: single, multilevel and hierarchical. In java programming, multiple and hybrid inheritance is supported through interface only. When one class inherits multiple classes, it is known as multiple inheritance.



Examples:

i) Single level inheritance:

```
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();

}}
```

Output:

```
barking...

eating...
```

Why is multiple inheritance not supported in java?

To reduce the complexity and simplify the language, multiple inheritance is not supported in java.

Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.

Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

When Constructors Are Called (order of constructor call)

When a class hierarchy is created, in what order are the constructors for the classes that make up the hierarchy called? For example, given a subclass called B and a superclass called A, is A's constructor called before B's, or vice versa? The answer is that in a class hierarchy, constructors are called in order of derivation, from superclass to subclass. Further, since super() must be the first statement executed in a subclass' constructor, this order is the same whether or not super() is used. If super() is not used, then the default or parameter less constructor of each superclass will be executed. The following program illustrates when constructors are executed:

```
// Demonstrate when constructors are called.
// Create a super class.
class A {
A() { System.out.println("Inside A's constructor."); }
}
// Create a subclass by extending class A.
class B extends A{
B() {
System.out.println("Inside B's constructor.");
}
}
// Create another subclass by extending B.
class C extends B {
C() {
System.out.println("Inside C's constructor."); }
}
class CallingCons {
public static void main(String[] args) {
C c=new C();
}
}
```

The output from tis program is shown here:

```
Inside A’s constructor
Inside B’s constructor
Inside C’s constructor
```

A superclass variable can refer to a subclass object:

In java a superclass reference variable can refer to the object of the subclass which has inherited features from it;i.e, superclass and subclass are type compatible.

```
class Al{
public int a;
public int b=1;
}
class B1 extends Al
{
public int b=2;
}
class Demol{
public static void main(String args[])
{
A1 obj=new B1(); //also called upcasting
System.out.println(((B1) obj).b);
}
}
```

Using super:

Whenever a subclass needs to refer to its immediate superclass, it can do so by use of the keyword **super**. Super has two general uses:

- The first calls the super class constructor.
- The second is used to access a member of the superclass that has been hidden by a member of a subclass.

Example: Using super to access the superclass member from sub class.

```
class Al{
public int b=1;
}
class B1 extends Al
{
public int b=2;
public int getSuperB ()
{
return super.b; //using super to access the b of the class Al
}
}
class Demo1(
public static void main(String args[])
{
B1 obj=new B1();
System.out.println(obj.b);
System.out.println(obj.getSuperB());
}
}
```

Example: Using super to call the superclass constructor

```
class Al{
public int b;
public Al (int b)
{
this.b=b;
}
}
{
class B1 extends Al
{
public Bl (int value)
{
super (value);
}
public int b=2;
public int getSuperB ()
{
return super.b;
}
}
class Demol{
public static void main(String args[])
{
B1 obj=new B1 (1);
System.out.println(obj.b);
System.out.println(obj.getSuperB ());
}
}
```

Method Overriding

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method.

Let's take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat (). Boy class is giving its own implementation to the eat () method or in other words it is overriding the eat () method.

```
class Human{
//Overridden method
public void eat() {
System.out.println("Human is eating"); }
}
class Boy extends Human {
//Overriding method
public void eat() {
System.out.println("Boy is eating");
}
public static void main(String args[]) {
Boy obj = new Boy();
//This will call the child class version of eat()
obj.eat();
}
}
```

Output:

Boy is eating

Rules of method overriding in Java

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method (the method of parent class). The data types of the arguments and their sequence should exactly match.

2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier, because all of these three access modifiers are more restrictive than public.

Another Examples:

Animal.java

```
public class Animal {
public void sound() {
System.out.println("Animal is making a sound");
}
}
```

Horse.java

```
class Horse extends Animal {
@Override
public void sound(){
System.out.println("Neigh");
}
public static void main(String args[]) {
Animal obj= new Horse();
obj.sound();
}
}
```

Output:

```
Neigh
```

Dynamic Method Dispatch

Method overriding forms the basis for one of Java's most powerful concepts: dynamic method dispatch. Dynamic method dispatch is the mechanism by which a call to an overridden method is resolved at run time, rather than compile time. Dynamic method dispatch is important because this is how Java implements run-time polymorphism

Example:

```
class Vehicle {
private int a;
public void run() {
System.out.println("Running......");
}
}
class Splender extends Vehicle {
public void run()
{
System.out.println("running in 50...");
}
}
class Dio extends Vehicle
{
public void run() {
System.out.println("running in 35....");
}
}
class RuntimepolyDemo{
public static void main(String args[])
{
Vehicle vehicle;
vehicle=new Splender();
vehicle.run();
vehicle=new Dio();
vehicle.run();
}
}
```

Output:

```
running in 50…
running in 35…
```

Note: In above program which run to call is decided in runtime.

Abstract classes:

A class which is declared with the abstract keyword is known as an abstract class in Java. It can have abstract and non-abstract methods (method with the body).

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

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

Abstraction lets us focus on what the object does instead of how it does it.

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)

2. Interface (100%)

Note:

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods. Abstract method are those methods which doesn't have method body.
- It cannot be instantiated.(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.

Example:

```
abstract class Shape1{
     abstract void draw();//abstract method which doesn’t have body.
}
//In real scenario, implementation is provided by others i.e. unknown by end
user
class Rectangle1 extends Shape1{
    void draw(){System.out.println("drawing rectangle");}
}
class Circle1 extends Shape1{
    void draw(){System.out.println("drawing circle");}
}
//In real scenario, method is called by programmer or user
class TestAbstraction1{
    public static void main(String args[]){
        Shape1 s=new Circle1();//In a real scenario, object is provided through
method, e.g., getShape() method
        s.draw();
    }
}
```

Output:

```
drawing circle
```

Using final to Prevent Overriding

```
class A {


    final void meth() {
        System.out.println("This is a final method.");
    }
}
class B extends A {
    void meth() { // ERROR! Can't override.
    System.out.println("Illegal!");
    }
}
```

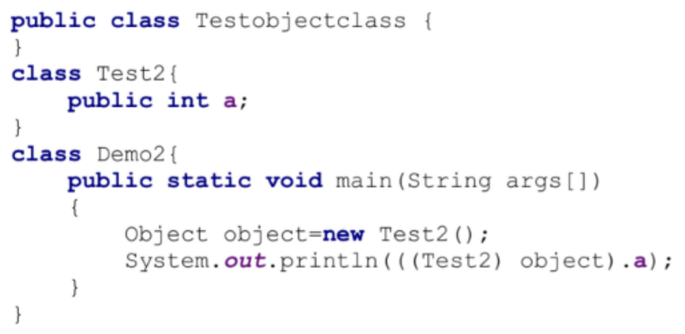
Using final to Prevent Inheritance

Example:

```
final class A {
    //...
}
// The following class is illegal.
class B extends A {
    // ERROR! Can't subclass A // ...
}
```

The Object Class

There is one special class, Object, defined by Java. All other classes are subclasses of Object. That is, Object is a superclass of all other classes. This means that a reference variable of type Object can refer to an object of any other class. Also, since arrays are implemented as classes, a variable of type Object can also refer to any array.



```
public class Testobjectclass {
}
class Test2{
    public int a;
}
class Demo2{
    public static void main(String args[])
    {
        Object object=new Test2();
        System.out.println(((Test2) object).a);
    }
}
```