UNIT :- 3

Java Object Creation of Inherited Class

In inheritance, subclass acquires super class properties. An important point to note is, when subclass object is created, a separate object of super class object will not be created. Only a subclass object object is created that has super class variables.

This situation is different from a normal assumption that a constructor call means an object of the class is created, so we can’t blindly say that whenever a class constructor is executed, object of that class is created or not.

```
// A Java program to demonstrate that both super class
// and subclass constructors refer to same object

// super class
classFruit
{
    publicFruit()
    {
        System.out.println("Super class constructor");
        System.out.println("Super class object hashcode :"+
                        this.hashCode());
        System.out.println(this.getClass().getName());
    }
}

// sub class
classApple extendsFruit
{
    publicApple()
    {
        System.out.println("Subclass constructor invoked");
        System.out.println("Sub class object hashcode :"+  this.hashCode());
        System.out.println(this.hashCode() + "    "+ super.hashCode());
        System.out.println(this.getClass().getName() + "  "+ super.getClass().getName());
    }
}

// driver class
publicclassTest
{
    publicstaticvoidmain(String[] args)
    {
        Apple myApple = newApple();
    }
}
```

Output:

```
super class constructor

super class object hashcode :366712642

Apple

sub class constructor
```

```
sub class object hashcode :366712642

366712642   366712642

Apple  Apple
```

Controlling Access to Members of a Class

Access level modifiers determine whether other classes can use a particular field or invoke a particular method. There are two levels of access control:

- At the top level—public, or *package-private* (no explicit modifier).
- At the member level—public, private, protected, or *package-private* (no explicit modifier).

A class may be declared with the modifier public, in which case that class is visible to all classes everywhere. If a class has no modifier (the default, also known as *package-private*), it is visible only within its own package (packages are named groups of related classes — you will learn about them in a later lesson.)

At the member level, you can also use the public modifier or no modifier (*package-private*) just as with top-level classes, and with the same meaning. For members, there are two additional access modifiers: private and protected. The private modifier specifies that the member can only be accessed in its own class. The protected modifier specifies that the member can only be accessed within its own package (as with *package-private*) and, in addition, by a subclass of its class in another package.

The following table shows the access to members permitted by each modifier.

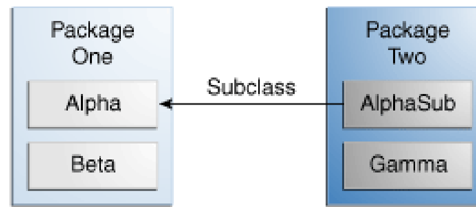
Access Levels

Modifier	**Class**	**Package**	**Subclass**	**World**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

The first data column indicates whether the class itself has access to the member defined by the access level. As you can see, a class always has access to its own members. The second column indicates whether classes in the same package as the class (regardless of their parentage) have access to the member. The third column indicates whether subclasses of the class declared outside this package have access to the member. The fourth column indicates whether all classes have access to the member.

Access levels affect you in two ways. First, when you use classes that come from another source, such as the classes in the Java platform, access levels determine which members of those classes your own classes can use. Second, when you write a class, you need to decide what access level every member variable and every method in your class should have.

Let's look at a collection of classes and see how access levels affect visibility. The following figure shows the four classes in this example and how they are related.



Classes and Packages of the Example Used to Illustrate Access Levels

The following table shows where the members of the Alpha class are visible for each of the access modifiers that can be applied to them.

Visibility				
Modifier	**Alpha**	**Beta**	**Alphasub**	**Gamma**
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Default Base Class Constructors Are Used with Inheritance

A derived Java class does not inherit a constructor from its base class. If a base class has a default constructor, i.e., a constructor with no arguments, then that constructor is automatically called when a derived class is instantiated if the derived class has its own default constructor. A default constructor is automatically provided by the compiler if no constructors are present in the Java source code.

Inheritance and constructors in Java

In Java, constructor of base class with no argument gets automatically called in derived class constructor. For example, output of following program is:

Base Class Constructor Called
Derived Class Constructor Called
// filename: Main.java

classBase {

```
  Base() {

    System.out.println("Base Class Constructor Called ");

  }

}

classDerived extendsBase {

  Derived() {

    System.out.println("Derived Class Constructor Called ");

  }

}

publicclassMain {

  publicstaticvoidmain(String[] args) {

    Derived d = newDerived();

  }

}
```

But, if we want to call parameterized contructor of base class, then we can call it using super(). The point to note is **base class constructor call must be the first line in derived class constructor**. For example, in the following program, super(_x) is first line derived class constructor.

```
// filename: Main.java

classBase {

  intx;

  Base(int_x) {

    x = _x;

  }

}



classDerived extendsBase {

  inty;

  Derived(int_x, int_y) {

    super(_x);
```

```
    y = _y;
  }
  voidDisplay() {
    System.out.println("x = "+x+", y = "+y);
  }
}


publicclassMain {
  publicstaticvoidmain(String[] args) {
    Derived d = newDerived(10, 20);
    d.Display();
  }
}
```

Output:
x = 10, y = 20

Super Keyword in Java

The **super** keyword in java is a reference variable that is used to refer parent class objects. The keyword "super" came into the picture with the concept of Inheritance. It is majorly used in the following contexts:

1. Use of super with variables: This scenario occurs when a derived class and base class has same data members. In that case there is a possibility of ambiguity for the JVM. We can understand it more clearly using this code snippet:

```
/* Base class vehicle */

classVehicle

{

    intmaxSpeed = 120;

}



/* sub class Car extending vehicle */

classCar extendsVehicle
```

```
{
    intmaxSpeed = 180;


    voiddisplay()
    {
        /* print maxSpeed of base class (vehicle) */
        System.out.println("Maximum Speed: "+ super.maxSpeed);
    }
}
  /* Driver program to test */
classTest
{
    publicstaticvoidmain(String[] args)
    {
        Car small = newCar();
        small.display();
    }
}
```

Output:

```
Maximum Speed: 120
```

2. Use of super with methods: This is used when we want to call parent class method. So whenever a parent and child class have same named methods then to resolve ambiguity we use super keyword. This code snippet helps to understand the said usage of super keyword.

```
/* Base class Person */
classPerson
{
    voidmessage()
    {
        System.out.println("This is person class");
    }
}

/* Subclass Student */
classStudent extendsPerson
{
```

```
    voidmessage()
    {
        System.out.println("This is student class");
    }

    // Note that display() is only in Student class
    voiddisplay()
    {
        // will invoke or call current class message() method
        message();

        // will invoke or call parent class message() method
        super.message();
    }
}

/* Driver program to test */
classTest
{
    publicstaticvoidmain(String args[])
    {
        Student s = newStudent();

        // calling display() of Student
        s.display();
    }
}
```

Output:

```
This is student class
This is person class
```

3. **Use of super with constructors:** super keyword can also be used to access the parent class constructor. One more important thing is that, ‘’super’ can call both parametric as well as non parametric constructors depending upon the situation. Following is the code snippet to explain the above concept:

```
/* superclass Person */
classPerson
{
    Person()
    {
        System.out.println("Person class Constructor");
    }
}

/* subclass Student extending the Person class */
classStudent extendsPerson
{
    Student()
    {
        // invoke or call parent class constructor
        super();
```

```
        System.out.println("Student class Constructor");
    }
}

/* Driver program to test*/
classTest
{
    publicstaticvoidmain(String[] args)
    {
        Student s = newStudent();
    }
}
```

Output:

```
Person class Constructor
Student class Constructor
```

Difference between abstract class and interface

Abstract class and interface both are used to achieve abstraction where we can declare the abstract methods. Abstract class and interface both can't be instantiated.

But there are many differences between abstract class and interface that are given below.

Abstract class	Interface
1) Abstract class can **have abstract and non-abstract**methods.	Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also.
2) Abstract class **doesn't support multiple inheritance**.	Interface **supports multiple inheritance**.
3) Abstract class **can have final, non-final, static and non-static variables**.	Interface has **only static and final variables**.
4) Abstract class **can provide the implementation of interface**.	Interface **can't provide the implementation of abstract class**.

5) The **abstract keyword** is used to declare abstract class.	The **interface keyword** is used to declare interface.
6) An **abstract class**can extend another Java class and implement multiple Java interfaces.	An **interface** can extend another Java interface only.
7) An **abstract class**can be extended using keyword "extends".	An **interface class**can be implemented using keyword "implements".
8) A Java**abstractclass**can have class members like private, protected, etc.	Members of a Java interface are public by default.

9)**Example:**
public abstract class Shape{
public abstract void draw();
}
	Example:
	public interface Drawable{
	void draw();
	}

Simply, abstract class achieves partial abstraction (0 to 100%) whereas interface achieves fully abstraction (100%).

Example of abstract class and interface in Java

Let's see a simple example where we are using interface and abstract class both.

```
1. //Creating interface that has 4 methods
2. interface A{
3. void a();//bydefault, public and abstract
4. void b();
5. void c();
6. void d();
7. }
8.
9. //Creating abstract class that provides the implementation of one method of A interface
10. abstract class B implements A{
11. public void c(){System.out.println("I am C");}
12. }
13.
14. //Creating subclass of abstract class, now we need to provide the implementation of rest of the m
    ethods
15. class M extends B{
16. public void a(){System.out.println("I am a");}
17. public void b(){System.out.println("I am b");}
```

```
18. public void d(){System.out.println("I am d");}
19. }
20.
21. //Creating a test class that calls the methods of A interface
22. class Test5{
23. public static void main(String args[]){
24. A a=new M();
25. a.a();
26. a.b();
27. a.c();
28. a.d();
29. }}
```

Output:

```
I am a
I am b
I am c
I am d
```

Java - Abstraction

As per dictionary, **abstraction** is the quality of dealing with ideas rather than events. For example, when you consider the case of e-mail, complex details such as what happens as soon as you send an e-mail, the protocol your e-mail server uses are hidden from the user. Therefore, to send an e-mail you just need to type the content, mention the address of the receiver, and click send.

Likewise in Object-oriented programming, abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

In Java, abstraction is achieved using Abstract classes and interfaces.

Abstract Class

A class which contains the **abstract** keyword in its declaration is known as abstract class.

- Abstract classes may or may not contain *abstract methods*, i.e., methods without body (public void get();)
- But, if a class has at least one abstract method, then the class **must**be declared abstract.
- If a class is declared abstract, it cannot be instantiated.
- 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.

Example

This section provides you an example of the abstract class. To create an abstract class, just use the **abstract** keyword before the class keyword, in the class declaration.

```
/* File name : Employee.java */
publicabstractclassEmployee{
privateString name;
privateString address;
privateint number;

publicEmployee(String name,String address,int number){
System.out.println("Constructing an Employee");
this.name = name;
this.address= address;
this.number= number;
}

publicdoublecomputePay(){
System.out.println("Inside Employee computePay");
return0.0;
}

publicvoidmailCheck(){
System.out.println("Mailing a check to "+this.name +" "+this.address);
}

publicStringtoString(){
return name +" "+ address +" "+ number;
}
```

```
publicStringgetName(){
return name;
}

publicStringgetAddress(){
return address;
}
publicvoidsetAddress(StringnewAddress){
address=newAddress;
}

publicintgetNumber(){
return number;
}
}
```

You can observe that except abstract methods the Employee class is same as normal class in Java. The class is now abstract, but it still has three fields, seven methods, and one constructor.

Now you can try to instantiate the Employee class in the following way −

```
/* File name : AbstractDemo.java */
publicclassAbstractDemo{

publicstaticvoid main(String[]args){
/* Following is not allowed and would raise error */
Employee e =newEmployee("George W.","Houston, TX",43);
System.out.println("\n Call mailCheck using Employee reference--");
e.mailCheck();
}
}
```

When you compile the above class, it gives you the following error −

```
Employee.java:46: Employee is abstract; cannot be instantiated
      Employee e = new Employee("George W.", "Houston, TX", 43);
                   ^
1 error
```

Inheriting the Abstract Class

We can inherit the properties of Employee class just like concrete class in the following way −

Example

```
/* File name : Salary.java */

publicclassSalaryextendsEmployee{

privatedouble salary;// Annual salary


publicSalary(String name,String address,int number,double salary){

super(name, address, number);

setSalary(salary);

}

publicvoidmailCheck(){

System.out.println("Within mailCheck of Salary class ");

System.out.println("Mailing check to "+getName()+" with salary "+ salary);

}


publicdoublegetSalary(){

return salary;

}


publicvoidsetSalary(doublenewSalary){

if(newSalary>=0.0){

salary=newSalary;

}

}
```

```
publicdoublecomputePay(){

System.out.println("Computing salary pay for "+getName());

return salary/52;

}

}
```

Here, you cannot instantiate the Employee class, but you can instantiate the Salary Class, and using this instance you can access all the three fields and seven methods of Employee class as shown below.

```
/* File name : AbstractDemo.java */

publicclassAbstractDemo{


publicstaticvoid main(String[]args){

Salary s =newSalary("MohdMohtashim","Ambehta, UP",3,3600.00);

Employee e =newSalary("John Adams","Boston, MA",2,2400.00);

System.out.println("Call mailCheck using Salary reference --");

s.mailCheck();

System.out.println("\n Call mailCheck using Employee reference--");

e.mailCheck();

}

}
```

This produces the following result −

Output

```
Constructing an Employee
Constructing an Employee
Call mailCheck using Salary reference --
Within mailCheck of Salary class
Mailing check to MohdMohtashim with salary 3600.0

 Call mailCheck using Employee reference--
Within mailCheck of Salary class
Mailing check to John Adams with salary 2400.0
```

Abstract Methods

If you want a class to contain a particular method but you want the actual implementation of that method to be determined by child classes, you can declare the method in the parent class as an abstract.

- **abstract** keyword is used to declare the method as abstract.
- You have to place the **abstract** keyword before the method name in the method declaration.
- An abstract method contains a method signature, but no method body.
- Instead of curly braces, an abstract method will have a semoi colon (;) at the end.

Following is an example of the abstract method.

Example

```
publicabstractclassEmployee{
privateString name;
privateString address;
privateint number;

publicabstractdoublecomputePay();
// Remainder of class definition
}
```

Declaring a method as abstract has two consequences −

- The class containing it must be declared as abstract.
- Any class inheriting the current class must either override the abstract method or declare itself as abstract.

Note − Eventually, a descendant class has to implement the abstract method; otherwise, you would have a hierarchy of abstract classes that cannot be instantiated.

Suppose Salary class inherits the Employee class, then it should implement the **computePay()** method as shown below −

```
/* File name : Salary.java */
publicclassSalaryextendsEmployee{
privatedouble salary;// Annual salary
```

```
publicdoublecomputePay(){
System.out.println("Computing salary pay for "+getName());
return salary/52;
}
// Remainder of class definition
}
```

Java - Interfaces

An interface is a reference type in Java. It is similar to class. It is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface.

Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods.

Writing an interface is similar to writing a class. But a class describes the attributes and behaviors of an object. And an interface contains behaviors that a class implements.

Unless the class that implements the interface is abstract, all the methods of the interface need to be defined in the class.

An interface is similar to a class in the following ways −

- An interface can contain any number of methods.
- An interface is written in a file with a **.java** extension, with the name of the interface matching the name of the file.
- The byte code of an interface appears in a **.class** file.
- Interfaces appear in packages, and their corresponding bytecode file must be in a directory structure that matches the package name.

However, an interface is different from a class in several ways, including −

- You cannot instantiate an interface.
- An interface does not contain any constructors.
- All of the methods in an interface are abstract.
- An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
- An interface is not extended by a class; it is implemented by a class.
- An interface can extend multiple interfaces.

-

Declaring Interfaces

The **interface** keyword is used to declare an interface. Here is a simple example to declare an interface −

Example

Following is an example of an interface −

```
/* File name : NameOfInterface.java */
importjava.lang.*;
// Any number of import statements

publicinterfaceNameOfInterface{
// Any number of final, static fields
// Any number of abstract method declarations\
}
```

Interfaces have the following properties −

- An interface is implicitly abstract. You do not need to use the **abstract**keyword while declaring an interface.
- Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
- Methods in an interface are implicitly public.

Example

```
/* File name : Animal.java */
interfaceAnimal{
publicvoid eat();
publicvoid travel();
}
```

Implementing Interfaces

When a class implements an interface, you can think of the class as signing a contract, agreeing to perform the specific behaviors of the interface. If a class does not perform all the behaviors of the interface, the class must declare itself as abstract.

A class uses the **implements** keyword to implement an interface. The implements keyword appears in the class declaration following the extends portion of the declaration.

Example

```
/* File name : MammalInt.java */
publicclassMammalIntimplementsAnimal{

publicvoid eat(){
System.out.println("Mammal eats");
}

publicvoid travel(){
System.out.println("Mammal travels");
}

publicintnoOfLegs(){
return0;
}

publicstaticvoid main(Stringargs[]){
MammalInt m =newMammalInt();
m.eat();
m.travel();
}
}
```

This will produce the following result −

Output

```
Mammal eats
Mammal travels
```

When overriding methods defined in interfaces, there are several rules to be followed −

- Checked exceptions should not be declared on implementation methods other than the ones declared by the interface method or subclasses of those declared by the interface method.
- The signature of the interface method and the same return type or subtype should be maintained when overriding the methods.
- An implementation class itself can be abstract and if so, interface methods need not be implemented.

When implementation interfaces, there are several rules −

- A class can implement more than one interface at a time.
- A class can extend only one class, but implement many interfaces.
- An interface can extend another interface, in a similar way as a class can extend another class.

Extending Interfaces

An interface can extend another interface in the same way that a class can extend another class. The **extends** keyword is used to extend an interface, and the child interface inherits the methods of the parent interface.

The following Sports interface is extended by Hockey and Football interfaces.

Example

```
// Filename: Sports.java
publicinterfaceSports{
publicvoidsetHomeTeam(String name);
publicvoidsetVisitingTeam(String name);
}


// Filename: Football.java
publicinterfaceFootballextendsSports{
publicvoidhomeTeamScored(int points);
publicvoidvisitingTeamScored(int points);
publicvoidendOfQuarter(int quarter);
}
```

```
// Filename: Hockey.java
publicinterfaceHockeyextendsSports{
publicvoidhomeGoalScored();
publicvoidvisitingGoalScored();
publicvoidendOfPeriod(int period);
publicvoidovertimePeriod(intot);
}
```

The Hockey interface has four methods, but it inherits two from Sports; thus, a class that implements Hockey needs to implement all six methods. Similarly, a class that implements Football needs to define the three methods from Football and the two methods from Sports.

Extending Multiple Interfaces

A Java class can only extend one parent class. Multiple inheritance is not allowed. Interfaces are not classes, however, and an interface can extend more than one parent interface.

The extends keyword is used once, and the parent interfaces are declared in a comma-separated list.

For example, if the Hockey interface extended both Sports and Event, it would be declared as −

Example

```
publicinterfaceHockeyextendsSports,Event
```

Tagging Interfaces

The most common use of extending interfaces occurs when the parent interface does not contain any methods. For example, the MouseListener interface in the java.awt.event package extended java.util.EventListener, which is defined as −

Example

```
packagejava.util;
publicinterfaceEventListener
{}
```

An interface with no methods in it is referred to as a **tagging** interface. There are two basic design purposes of tagging interfaces −

Creates a common parent − As with the EventListener interface, which is extended by dozens of other interfaces in the Java API, you can use a tagging interface to create a common parent

among a group of interfaces. For example, when an interface extends EventListener, the JVM knows that this particular interface is going to be used in an event delegation scenario.

Adds a data type to a class − This situation is where the term, tagging comes from. A class that implements a tagging interface does not need to define any methods (since the interface does not have any), but the class becomes an interface type through polymorphism.

CLASS	INTERFACE
Supports only multilevel and hierarchical inheritances but not multiple inheritance	Supports all types of inheritance – multilevel, hierarchical and multiple
"extends" keyword should be used to inherit	"implements" keyword should be used to inherit
Should contain only concrete methods (methods with body)	Should contain only abstract methods (methods without body)
The methods can be of any access specifier (all the four types)	The access specifier must be public only
Methods can be final and static	Methods should not be final and static
Variables can be private	Variables should be public only
Can have constructors	Cannot have constructors
Can have main() method	Cannot have main() method as main() is a concrete method

Java - Packages

Packages are used in Java in order to prevent naming conflicts, to control access, to make searching/locating and usage of classes, interfaces, enumerations and annotations easier, etc.

A **Package** can be defined as a grouping of related types (classes, interfaces, enumerations and annotations) providing access protection and namespace management.

Some of the existing packages in Java are −

- **java.lang** − bundles the fundamental classes
- **java.io** − classes for input , output functions are bundled in this package

Programmers can define their own packages to bundle group of classes/interfaces, etc. It is a good practice to group related classes implemented by you so that a programmer can easily determine that the classes, interfaces, enumerations, and annotations are related.

Since the package creates a new namespace there won't be any name conflicts with names in other packages. Using packages, it is easier to provide access control and it is also easier to locate the related classes.

Creating a Package

While creating a package, you should choose a name for the package and include a **package** statement along with that name at the top of every source file that contains the classes, interfaces, enumerations, and annotation types that you want to include in the package.

The package statement should be the first line in the source file. There can be only one package statement in each source file, and it applies to all types in the file.

If a package statement is not used then the class, interfaces, enumerations, and annotation types will be placed in the current default package.

To compile the Java programs with package statements, you have to use -d option as shown below.

```
javac -d Destination_folder file_name.java
```

Then a folder with the given package name is created in the specified destination, and the compiled class files will be placed in that folder.

Example

Let us look at an example that creates a package called **animals**. It is a good practice to use names of packages with lower case letters to avoid any conflicts with the names of classes and interfaces.

Following package example contains interface named *animals* −

```
/* File name : Animal.java */
package animals;


interfaceAnimal{
publicvoid eat();
publicvoid travel();
}
```

Now, let us implement the above interface in the same package *animals* −

```
package animals;
/* File name : MammalInt.java */
```

```
publicclassMammalIntimplementsAnimal{


publicvoid eat(){

System.out.println("Mammal eats");

}


publicvoid travel(){

System.out.println("Mammal travels");

}


publicintnoOfLegs(){

return0;

}

publicstaticvoid main(Stringargs[]){

MammalInt m =newMammalInt();

m.eat();

m.travel();

}

}
```

Now compile the java files as shown below −

```
$ javac -d . Animal.java
$ javac -d . MammalInt.java
```