



Inheritance



OBJECTIVES OF THIS CHAPTER

- 9.1 Inheritance and its types
- 9.2 Interfaces
- 9.3 Use of super method
- 9.4 Abstract Classes and Methods
- 9.5 Method Overriding
- 9.6 Final Class, Method and Variables

9.1 INTRODUCTION TO INHERITANCE

Inheritance is a feature or a process in which new classes are created from the existing classes. Here, Existing class is known as super class while new class is known as sub class. It refers to the ability of an object to take on one or more characteristics from other classes of objects. This process is same as a child inherits the traits of his/her parents. The purpose of inheritance is to consolidate and reuse an existing code. For example, if the objects "student", "teacher" and "office staff" are subclasses of Person super class. Code applying to all of them can be consolidated into a Person super class.

The characteristics inherited are usually instance variables or methods. The new class created is also called "derived class" or "child class" and the existing class is known as the "base class" or "parent class". The sub (derived) class now is said to be inherited from the super (base) class. We can understand inheritance in graphical form as shown below :

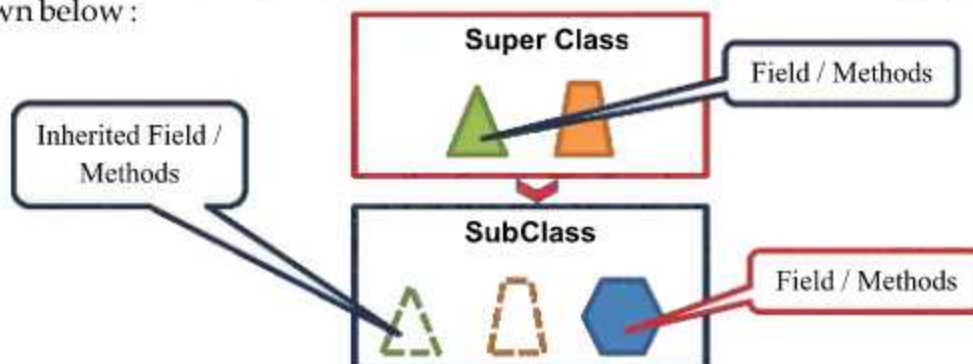


Fig. 9.1: Concept of Inheritance

Terminologies related to the Inheritance:

❖ **Class** : It is defined as a collection of objects sharing common properties. It is a kind of blueprint of objects created in a Java program.

❖ **Sub Class:** It is also known as derived class. This category of classes inherits the options from another class. Sub Class can include the inheritable fields and behaviors of its Super class as well as generated within its own.

❖ **Super Class:** The super class is also known as the parent class. It represents the category of those classes whose fields or methods are inheritable by a subclass.

❖ **Reusability:** As the name is defined, it is a technique of reusing members of the existing class in the new created class as it is. It can reuse the fields or methods of existing class. In short, we can say that it permits reusing the code.

These all terminologies are the basics of inheritance in Java. Following is the syntax and example of Inheritance in java :

Syntax

```
class derived_class extends base_class
{
//fields
//methods
}
```

Note : extends keyword is used to inherit the members of a super class in java.

Following program shows the implementation of inheritance in JAVA

```
class SupClass
{
void showStart()
{
System.out.println("Welcome to Computer Application Subject");
}
}
class SubClass extends SupClass
{
void showEnd()
{
System.out.println("Bye! Hope to see you again..");
}
}
class CAProg13
{
    public static void main(String arg[])
    {
        SubClass obj=new SubClass();
        obj.showStart();
        obj.showEnd();
    }
}
```

Compilation, Execution and Output of Program 9.1 (CAProg13.java)

```
D:\JavaProg>javac CAProg13.java
D:\JavaProg>java CAProg13
Welcome to Computer Application Subject
Bye! Hope to see you again..
D:\JavaProg>_
```

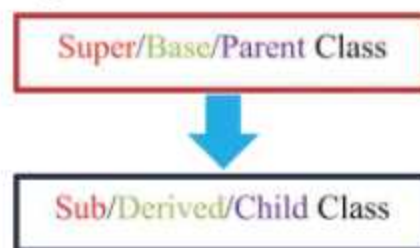
9.1.1 TYPES OF INHERITANCE

Java supports different types of inheritance. Some most widely used types of inheritance in java are as follows:

- ❖ Single Inheritance
- ❖ Multi-level Inheritance
- ❖ Hierarchical Inheritance
- ❖ Hybrid Inheritance

❖ Single Inheritance:

This is a simplest type of inheritance in java. In single inheritance, a sub-class is derived from only one super class. It inherits the properties and behavior of a single-parent class. Sometimes it is also known as simple inheritance. We can have following graphical representation of Single Inheritance in Java.



Single Inheritance in Java

As we can see in the above diagram, parent class is inherited in child class. Here, Child class will include all the properties of the base class also. But, no property or behavior of Child class will be inherited by Parent Class. Following example shows the mechanism of single inheritance in a Java :

Following program shows the implementation of Single inheritance in JAVA

```
class Person
{
int age;
String sname;
}
```



```

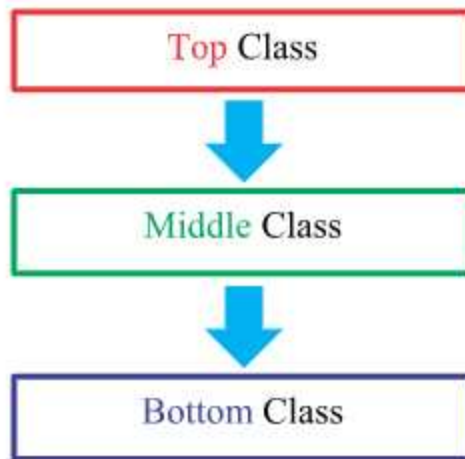
class Student extends Person
{
int sclass,rollno;
void setData(int age_Input,String sname_Input,int sclass_Input,int
rollno_Input)
{
age=age_Input;
sname=sname_Input;
sclass=sclass_Input;
rollno=rollno_Input;
}
void showRecord()
{
System.out.println("Age:"+age);
System.out.println("Student Name: "+sname);
System.out.println("Student Class:"+sclass);
System.out.println("Roll No:"+rollno);
}
}
class CProgl4
{
    public static void main(String arg[])
    {
        Student obj=new Student();
        obj.setData(16,"Navleen",7,1001);
        obj.showRecord();
    }
}

```

Compilation, Execution and Output of Program 9.2 (CProgl4.java)

❖ Multi-level Inheritance

This type of inheritance can be viewed as a chain of single inheritance. In multi-level inheritance, a class is derived from a class which is further derived from another class. In simple words, we can say that a class that has more than one parent class, but at different levels, is called multi-level inheritance. A common sub class can not have multiple super classes at a same time in this type of inheritance. Following graphical representation shows the mechanism of multilevel Inheritance in Java.



Multilevel Inheritance in Java

As we can see in the above diagram, Top class is inherited by Middle class which is further inherited by Bottom Class. Here, Bottom class will include all the properties of Top and Middle class. But, No property or behavior of Middle class will be inherited by Top Class or Bottom class by Top/Middle class. Following example shows the mechanism of Multilevel inheritance in a Java.

Following program shows the implementation of Multilevel inheritance in JAVA

```
class Top
{
void show()
{
System.out.println("This is a method of TOP Class");
}
}
class Middle extends Top
{
void display()
{
System.out.println("This is a method of Middle Class");
}
}
class Bottom extends Middle
{
void print()
{
System.out.println("This is a method of Bottom Class");
}
}
```

```

class CAProg16
{
    public static void main(String arg[])
    {
        Bottom obj=new Bottom();
        obj.show();
        obj.display();
        obj.print();
    }
}

```

Compilation, Execution and Output of Program 9.3 (CAProg16.java)

```

D:\JavaProg>javac CAProg16.java

D:\JavaProg>java CAProg16
This is a method of TOP Class
This is a method of Middle Class
This is a method of Bottom Class

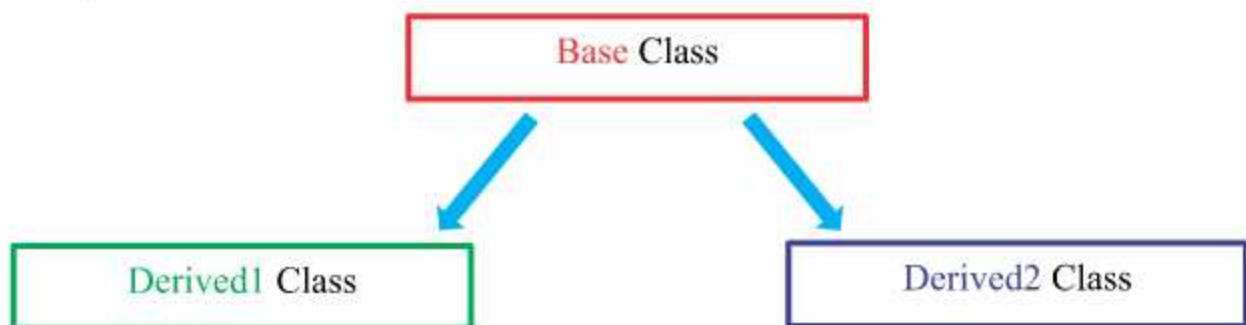
D:\JavaProg>

```

Here, we can see that the method of Top class [show()] and Middle class [display()] are inherited in Bottom class. It allows us to call all the methods [show(), display() and print()] with the object of Bottom class.

❖ Hierarchical Inheritance (Tree Inheritance):

When two or more classes inherits a single class, it is known as hierarchical inheritance. This type of inheritance is useful in the case when we want to share the methods or properties of one common base class into multiple derived classes. As we can see in the diagram given below, Derived1 and Derived2 classes inherits the Base class. This kind of layout forms hierarchical inheritance.



Hierarchical Inheritance in Java

Following example shows the mechanism of Hierarchical inheritance in a Java.

Following program shows the implementation of Hierarchical inheritance in JAVA

```
class Base
{
void show()
{
System.out.println("Base Class");
}
}
class Derived1 extends Base
{
void display()
{
System.out.println("Derived 1 Class");
}
}
class Derived2 extends Base
{
void print()
{
System.out.println("Derived 2 Class");
}
}
class CAProg17
{
    public static void main(String arg[])
    {
        Derived1 obj1=new Derived1();
        obj1.show();
        obj1.display();
        Derived2 obj2=new Derived2();
        obj2.show();
        obj2.print();
    }
}
```


Compilation, Execution and Output of Program 9.4 (CAProg17.java)

```
D:\JavaProg>javac CAProg17.java

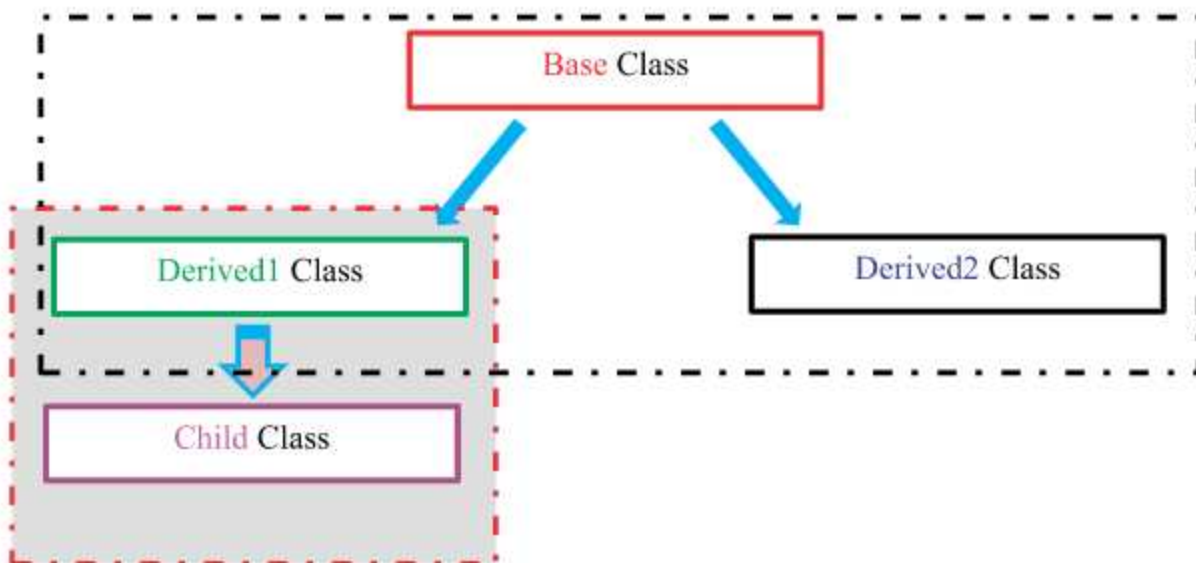
D:\JavaProg>java CAProg17
Base Class
Derived 1 Class
Base Class
Derived 2 Class

D:\JavaProg>
```

Here, we can see that the method of Base class [show()] is inherited in Both Derived1 and Derived2 class. It allows us to call the method of Base class with the object of either Derived1 class or Derived2 class.

❖ Hybrid Inheritance:

Hybrid means consisting of more than one form of inheritance within one inheritance type. Hybrid inheritance is the combination of any two or more types of inheritance in Java. We can have an example of simple hybrid inheritance in the form of a diagram as shown below :



In this diagram, we use Hierarchical and single Inheritance to form Hybrid Inheritance. Following example shows the implementation of Hybrid inheritance a Java:

Following program shows the implementation of Hybrid inheritance in JAVA

```
class Base
{
void show()
{
System.out.println("Base Class");
}
}
class Derived1 extends Base
{
void display()
{
System.out.println("Derived 1 Class");
}
}
class Derived2 extends Base
{
void print()
{
System.out.println("Derived 2 Class");
}
}
class Child extends Derived1
{
void send()
{
System.out.println("This is Child Class here");
}
}
class CProgl8
{
    public static void main(String arg[])
    {
        Child obj1=new Child();
        obj1.show();
        obj1.display();
        obj1.send();
        Derived2 obj2=new Derived2();
        obj2.show();
        obj2.print();
    }
}
```

Compilation, Execution and Output of Program 9.5 (CAProg18.java)

```

D:\JavaProg>javac CAProg18.java

D:\JavaProg>java CAProg18
Base Class
Derived 1 Class
This is Child Class here
Base Class
Derived 2 Class

D:\JavaProg>_
```

Some other types of inheritance are also available in OOPs. Java does not allow multiple inheritance among classes. We shall discuss multiple inheritance after understanding the concept of Interfaces.

9.2 INTERFACES IN JAVA

The interface in Java is a mechanism to achieve abstraction. There can be only abstract methods in the Java interface, not the method body. A class implements an interface and inherit the abstract methods of the interface. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. In the new versions of Java (Java 8 onwards), Method bodies can also be existed within an interface, but 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. When a class implements the interface, all the methods of the interface need to be defined in the class. In other words, you can say that interfaces can have abstract methods and variables. It cannot have a method body.

Note : Abstract methods do not have body, they only have declaration but no definition.

Similarities between class and interface:

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

Difference between class and interface

- ❖ We cannot instantiate an interface.
- ❖ An interface does not contain any constructors.
- ❖ All of the methods in an interface are abstract (only default or static methods can have body).
- ❖ An interface can contain only static and final fields.

- ❖ An interface is not extended by a class; it is implemented by a class.
- ❖ An interface can extend multiple interfaces.

9.2.1 DECLARING INTERFACES

The interface keyword is used to declare an interface. Following is the syntax and example of interface:

Syntax:

```
interface InterfaceName
{
    // Any number of final, static fields
    // Any number of abstract method declarations
}
```

We can declare the interface using above mentioned syntax as per requirement of our program to use abstract methods. Lets have an example to understand the use of interface in detail.

Following program shows the implementation of interface in JAVA

```
interface Student
{
    final int MinAge=18;
    public void show();
}
class CAProg19 implements Student
{
    public void show()
    {
        System.out.println("Welcome to Student Mangement");
        System.out.println("Minimum age of Student is:"+MinAge);
    }
    public static void main(String arg[])
    {
        CAProg19 obj=new CAProg19( );
        obj.show();
    }
}
```

Compilation, Execution and Output of Program 9.6 (CAProg19.java)

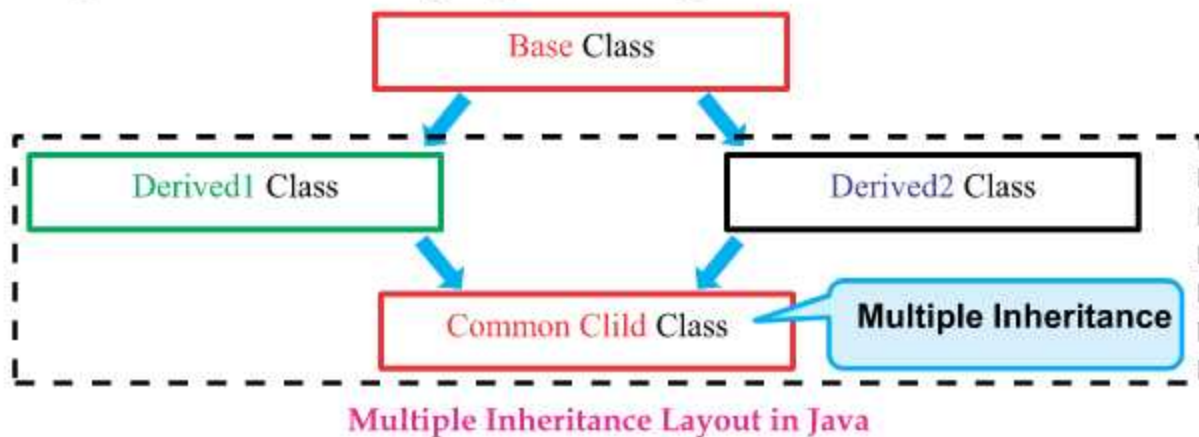
```
D:\JavaProg>javac CAProg19.java

D:\JavaProg>java CAProg19
Welcome to Student Mangement
Minimum age of Student is:18

D:\JavaProg>
```

❖ Multiple Inheritance in Java

Java does not support multiple inheritances among classes due to ambiguity. For example, consider the following diagram of multiple inheritance.



The above Diagram shows the mechanism of multiple inheritance in OOPs application. If this layout is used in java classes, it gives error because the compiler cannot decide which method of base class is to be invoked. It is so because methods of base class are derived through two classes named Derived1 and Derived2. Due to this reason, Java does not support multiple inheritances at the class level but can be achieved through an interface.

Multiple Inheritance among interfaces can be shown as follows :

Following program shows the implementation of multiple inheritance in JAVA

```
interface Base
{
    public void show();
}
interface Derived1 extends Base {
    public void display();
}
interface Derived2 extends Base {
```

```

public void print();
}
class CommonChild implements Derived1,Derived2 // Multiple Inheritance
{
    public void show()
    {
        System.out.println("Base Class");
    }
    public void display()
    {
        System.out.println("Derived 1 Class");
    }
    public void print()
    {
        System.out.println("Derived 2 Class");
    }
}
class CAProg20
{
    public static void main(String arg[])
    {
        CommonChild obj=new CommonChild();
        obj.show();
        obj.display();
        obj.print();
    }
}

```

Compilation, Execution and Output of Program 9.7 (CAProg20.java)



```

D:\JavaProg>javac CAProg20.java

D:\JavaProg>java CAProg20
Base Class
Derived 1 Class
Derived 2 Class

D:\JavaProg>

```

Here we can see, we have extended one common base interface into two different Derived Interfaces which are further inherited in one common class. Multiple Inheritance is possible in java using interface only.

9.3 SUPER KEYWORD IN JAVA

Super keyword allows us to access the members of superclass. Common use of the super keyword is to eliminate the confusion between superclasses and subclasses that have methods with the same name. We can use the concept of super in two different forms.

1. **super keyword:** This scenario occurs when a derived class and base class has same members. In that case there is a possibility of ambiguity for the JVM. super keyword, in such a case, is used to define which member is exactly needed to be invoked. Some highlights about super keywords are as under:

- ❖ To call methods of the superclass that is overridden in the subclass.
- ❖ To access attributes (fields) of the superclass if both superclass and subclass have attributes with the same name.

We can understand the concept of super keyword using following example

Following program shows the use of super keyword in JAVA

```
class Base
{
    void show()
    {
        System.out.println("Base class show Method");
    }
}
class CAProg21 extends Base
{
    void show()
    {
        super.show(); //Accessing Base class method using super
        System.out.println("Derived class show Method");
    }
    public static void main(String arg[])
    {
        CAProg21 obj=new CAProg21();
        obj.show();
    }
}
```

Compilation, Execution and Output of Program 9.8 (CAProg21.java)



```
D:\JavaProg>javac CAProg21.java
D:\JavaProg>java CAProg21
Base class show Method
Derived class show Method
D:\JavaProg>
```

2. Super method : super keyword can also be used as a method to access the parent class constructor. We can call parameterized as well as non-parameterized constructors using super method. This method plays a very significant role as we have already studied that if a class defines a constructor with parameters then its default constructor is not automatically generated. So, when we use a constructor in inheritance and our base class is having a parameterized constructor then it becomes compulsory to call the base class constructor in the derived class and pass the values to the parameters. A solution to this issue is that, we can use the super method to explicitly invoke the constructor of the base class in the derived class. There are a few important points about the super method:

- ❖ super() method must be the first statement in the derived class constructor; otherwise, a compilation error would be displayed.
- ❖ When we explicitly place super in the derived class constructor, the Java compiler doesn't call the default constructor of the parent class (base class).

We have understood the concept of inheritance so far. Let's have a closer look at the super method in the form of an example:

Following program shows the use of super method in JAVA

```
class Base
{
    Base(int id) //base class constructor with parameters
    {
        System.out.println("Student id is: "+id);
    }
}
class Derived extends Base
{
    Derived(int sid) //derived class constructor with parameters
    {
        super(sid); //Passing parameters to base class constructor
    }
    void show(String sname)
    {
        System.out.println("Student name is: "+sname);
    }
}

class CAProg22
{
    public static void main(String arg[])
    {
        Derived obj=new Derived(1001);
        obj.show("Shivpreet");
    }
}
```

Compilation, Execution and Output of Program 9.9 (CAProg22.java)

```
D:\JavaProg>javac CAProg22.java

D:\JavaProg>java CAProg22
Student id is: 1001
Student name is: Shivpreet

D:\JavaProg>_
```

We can use super keyword or super method similarly in any type of inheritance for any of the element like fields, methods etc.

9.4 ABSTRACTION IN JAVA

Abstraction is a process of hiding the implementation details and showing only functionality to the user. In other words we can say, it shows only essential things to the user and hides the internal complexities. For example, deriving a car is just about using the controls like steering, accelerator, brakes, gears or other electronic control. Internal functioning of engine remain hidden from the user.

Abstraction can be achieved using either abstract classes or interfaces in Java. The abstract keyword is a non-access modifier, used for classes and methods: We can classify abstraction in following types:

- ❖ **Abstract class:** It is a restricted class that cannot be used to create objects. To access the members of this class, it must be inherited from another class.
- ❖ **Abstract methods:** These methods can only be used in an abstract class, and does not have a body. The body is provided by the subclass. (We have already introduced this concept in interface section of this chapter)

Lets understand the concept of abstract class as an example

Following program shows the use of abstract class and abstract method in JAVA

```
abstract class Base
{
    abstract void show(); //abstract method
    void display()
    {
        System.out.println("This is not an abstract function");
    }
}

class CAProg23
{
    public static void main(String arg[])
    {
        Base obj=new Base();
        obj.display();
    }
}
```


Compilation, Execution and Output of Program 9.10 (CAProg23.java)

```
Command Prompt
D:\JavaProg>javac CAProg23.java
CAProg23.java:13: Base is abstract; cannot be instantiated
    Base obj=new Base();
                ^
1 error
D:\JavaProg>
```

As we can see, this program can not be executed because abstract class can not be instantiated. We can use the same program in correct way as given below:

Following program shows the correct use of abstract class and abstract method in JAVA

```
Class Base
{
    abstract void show(); //abstract method
    void display()
    {
        System.out.println("This is not an abstract function");
    }
}
class CAProg23 extends Base
{
    void show() //defination of abstract method
    {
        System.out.println("This is an abstract function");
    }
    public static void main(String arg[])
    {
        CAProg23 obj=new CAProg23();
        obj.show();
        obj.display();
    }
}
```

Compilation, Execution and Output of Program 9.11 (CAProg23.java)

```
D:\JavaProg>javac CAProg23.java
D:\JavaProg>java CAProg23
This is an abstract function
This is not an abstract function
D:\JavaProg>
```

In the above given example, we can see the correct way to declare and access the abstract class and abstract method in a class.

9.5 METHOD OVERRIDING IN JAVA

We have already studied about method overloading where multiple methods are declared with same name but having different number of arguments or different data type of arguments within a same class. But, If subclass 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. Some basic rules for method overriding are as follows :

Rules for Java Method Overriding:

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

Lets understand method overriding in JAVA in the form of an example in detail :

Following program shows the implementation of method overriding in JAVA

```
class Base
{
    void display()
    {
        System.out.println("Base class is here");
    }
}
class Derived extends Base
{
    void display() //Method overriding
    {
        System.out.println("Derived class is here");
    }
}
```

```

class CAProg24
{
    public static void main(String arg[])
    {
        Derived obj=new Derived();
        obj.display();
    }
}

```

Compilation, Execution and Output of Program 9.12 (CAProg24.java)

```

D:\JavaProg>javac CAProg24.java
D:\JavaProg>java CAProg24
Derived class is here
D:\JavaProg>

```

This example explain the use of method overriding in JAVA. We can not override static methods. The main method can also not be overridden.

Difference between method overloading and method overriding:

There are many differences between method overloading and method overriding in java as given below:

No.	Method Overloading	Method Overriding
1)	Method overloading is used to increase the readability of the program.	Method overriding is used to provide the specific implementation of the method that is already provided by its super class.
2)	Method overloading is performed within same class.	Method overriding occurs in two classes that have IS-A (inheritance) relationship.
3)	In case of method overloading, parameters must be different.	In case of method overriding, parameters must be same.

4)	Method overloading is the example of compile time polymorphism.	Method overriding is the example of run time polymorphism.
5)	In java, method overloading can't be performed by changing return type of the method only.	Return type must be same in method overriding.

9.6 FINAL CLASS, METHOD AND VARIABLES IN JAVA

final keyword is one of the most important keywords in Java that can be used with entities in Java to restrict their use. We can use it with class, methods, variables. Java final keyword is a non-access specifiers that is used to impose some restriction on class, variable, and method. If we initialize a variable with the final keyword, then we cannot modify its value. If we declare a method as final, then it cannot be overridden by any subclasses. And, if we declare a class as final, we restrict the class from being inherited. We can graphically represent the use of final keyword as follows

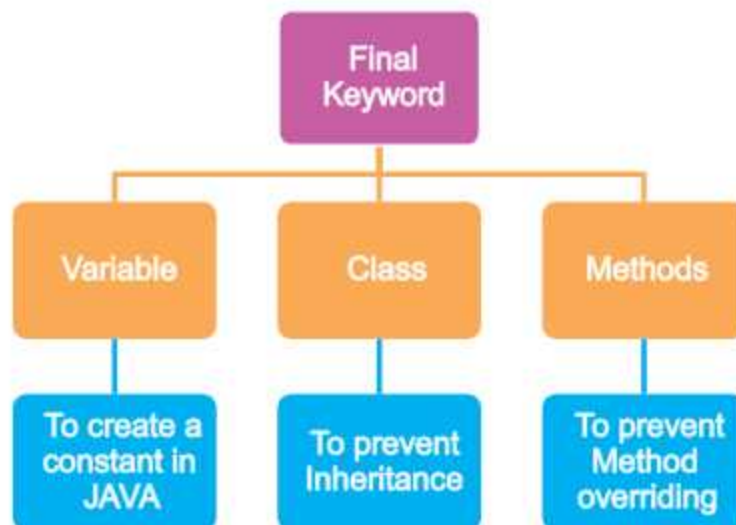


Fig : Different uses of final Keyword

Lets discuss the use of this final keyword with different elements in JAVA.

❖ final variable :

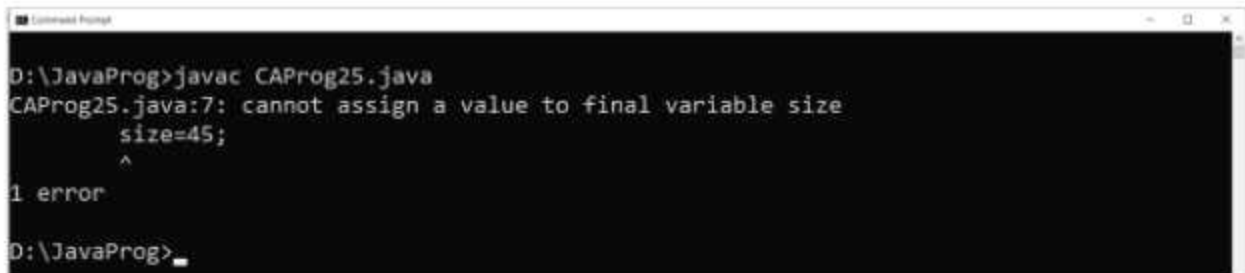
If you declare any variable as final, its value can't be modified. This variable would become a constant. This also means that we must initialize a final variable while declaration because after that no changes are allowed in final variable. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can

be changed i.e. we can add or remove elements from the final array or final collection. It is a good practice to represent final variables in all uppercase, using underscore to separate words. Lets understand the use of final variable in the form of an example:

Following program shows the use of final variable in JAVA

```
class CAProg25
{
    public static void main(String arg[])
    {
        final int size=40; //final variable
        System.out.println("Size of class is: "+size);
        size=45;
    }
}
```

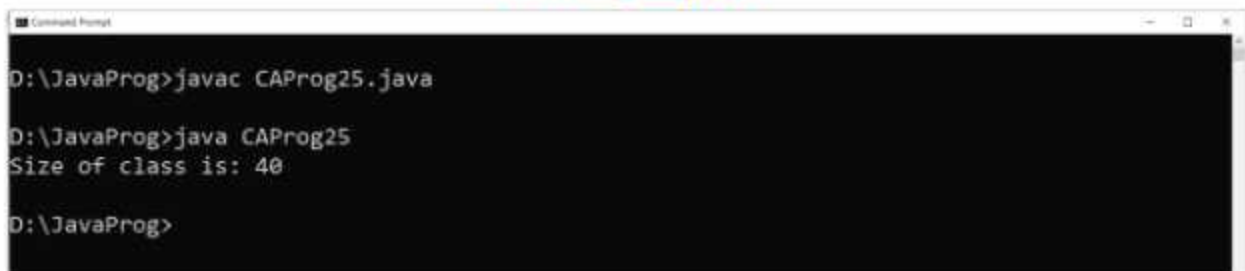
Compilation, Execution and Output of Program 9.13 (CAProg25.java)



```
D:\JavaProg>javac CAProg25.java
CAProg25.java:7: cannot assign a value to final variable size
    size=45;
    ^
1 error
D:\JavaProg>
```

Here we can clearly see that no change is allowed in final variable after declaration. If we delete the statement `size=45;` then the program will be executed correctly and the output would be as given bellow:

After Correction, Compilation, Execution and Output of Program 9.13 (CAProg25.java)



```
D:\JavaProg>javac CAProg25.java
D:\JavaProg>java CAProg25
Size of class is: 40
D:\JavaProg>
```

Here we can see the output of the program after removing the said statement.

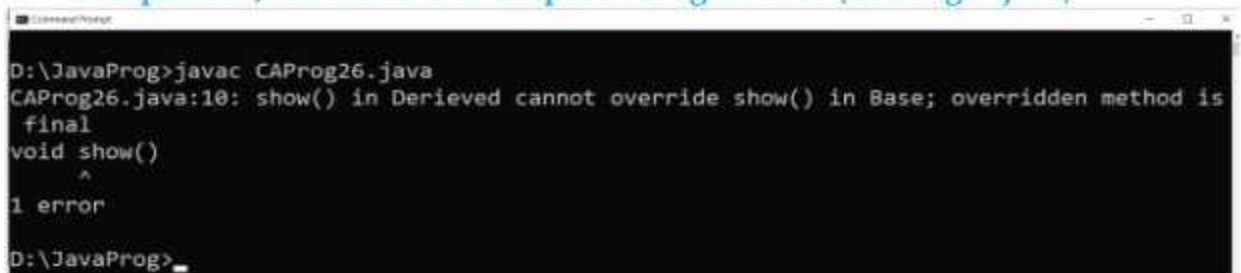
❖ Final Methods:

When a method is declared final keyword, it is called a final method. A final method cannot be overridden. We must declare methods with the final keyword for which we are required to follow the same implementation throughout all the derived classes. It is a requirement of several applications where any of the change is essentially required to be restricted. We can have an example of final method as follows:

Following program shows the use of final method in JAVA

```
class Base
{
    final void show() //final method
    {
        System.out.println("Base Function");
    }
}
class Derived extends Base
{
    void show() //overriding final method which shows error
    {
        System.out.println("Derived Function");
    }
}
class CAProg26
{
    public static void main(String arg[])
    {
        Derived obj=new Derived();
        obj.show();
    }
}
```

Compilation, Execution and Output of Program 9.14 (CAProg26.java)



```
D:\JavaProg>javac CAProg26.java
CAProg26.java:10: show() in Derived cannot override show() in Base; overridden method is
    final
void show()
    ^
1 error
D:\JavaProg>
```


As we can see in the program output, no final method can be overridden in any of the derived class.

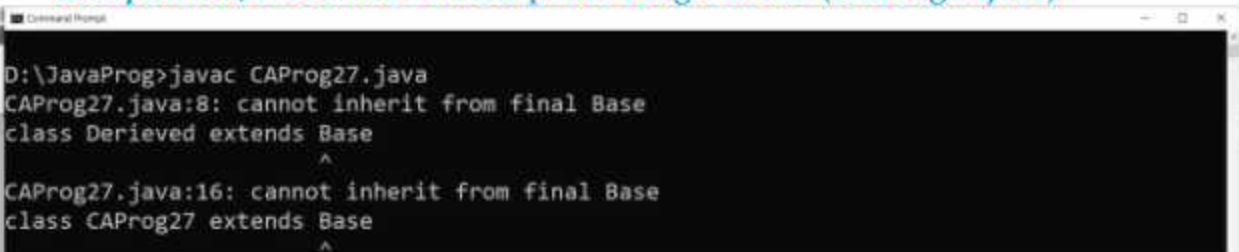
❖ Final classes:

When a class is declared with final using final keyword, this class is restricted to be extended in any of the sub class. These type of classes are useful to prevent the overriding of class methods in derived classes. We can have an example of final class as follows:

Following program shows the use of final class in JAVA

```
final class Base //final class
{
void show1()
{
System.out.println("Base Function");
}
}
class Derieved extends Base //shows error as final class cannot be inherited
{
void show2()
{
System.out.println("Derived Function");
}
}
class CAProg27
{
    public static void main(String arg[])
    {
        Derived obj=new Derived();
        obj.show2();
    }
}
```

Compilation, Execution and Output of Program 9.15 (CAProg27.java)



```
D:\JavaProg>javac CAProg27.java
CAProg27.java:8: cannot inherit from final Base
class Derieved extends Base
    ^
CAProg27.java:16: cannot inherit from final Base
class CAProg27 extends Base
    ^
```

Here we can see that final class is not allowed to be inherited. If we try to do so, an error message stating the same is being displayed.



Points to Remember

1. Inheritance is a feature or a process in which new classes are created from the existing classes.
2. Class is defined as a collection of objects sharing common properties. It is a kind of blueprint of objects created in a Java program.
3. Sub Class inherits the options from another class including inheritable fields and behaviors of its Super class.
4. Sub Class is also known as derived class.
5. Super Class represents fields or methods which are inheritable by a subclass.
6. Reusability is a technique of reusing methods of the existing class in the new created class.
7. Single Inheritance, Multi-level Inheritance, Hierarchical Inheritance, Hybrid Inheritance are some of the types of Inheritance.
8. Hybrid means consisting of more than one form of inheritance within one inheritance type.
9. Java does not allow multiple inheritance among classes.
10. The interface in Java is a mechanism to achieve abstraction.
11. Method bodies can also be existed within an interface, but only for default methods and static methods.
12. We cannot instantiate an interface and it does not contain any constructors.
13. Multiple inheritance can be achieved interfaces in JAVA.
14. Super keyword allows us to access the members of superclass.
15. Super method can be used to access the constructor of parent class.
16. Abstraction is a process of hiding the implementation details and showing only functionality to the user.
17. If we declare a method as final, then it cannot be overridden by any subclasses.

- iii. Explain super keyword.
- iv. What is method overriding?
- v. Write a short note on final variable?

Que:4 Long Answer type Questions:

- i. What is inheritance? Explain any three types of inheritance
- ii. What do you mean by abstraction? Explain abstract elements in Java.
- iii. What do mean by final keyword? Explain Final class and Method with example.
- iv. Explain multiple inheritance with suitable example?