Inheritance

Inheritance is the mechanism of deriving new class from old one, old class is knows as superclass and new class is known as subclass. The subclass inherits all of its instances variables and methods defined by the superclass and it also adds its own unique elements. Thus we can say that subclass are specialized version of superclass.

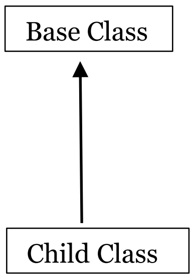
Benefits of Java's Inheritance

1. Reusability of code
2. Code Sharing
3. Consistency in using an interface

Classes

Superclass(Base Class)	Subclass(Child Class)
It is a class from which other classes can be derived.	It is a class that inherits some or all members from superclass.

Types of Inheritance in Java



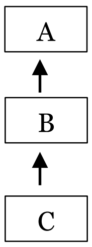
1. Single Inheritance - one class extends one class only

```
class inherit1
{
    static int i=10;
    inherit1()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    inherit2()
    {
        super();
        System.out.println("Value of i in Child class is"+i);
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
    }
}
```

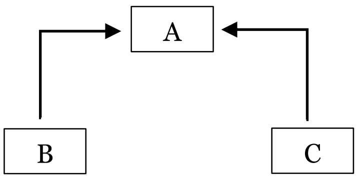
Base Class

Child Class inherited from Base Class. Note: "***extends***" keyword is used to inherit a sub class from superclass.

2. Multilevel Inheritance – It is a ladder or hierarchy of single level inheritance. It means if ***Class A is extended by Class B and then further Class C extends Class B*** then the whole structure is termed as Multilevel Inheritance. Multiple classes are involved in inheritance, but one class extends only one. The lowermost subclass can make use of all its super classes' members.



3. **Hierarchical Inheritance** - one class is extended by many subclasses. It is **one-to-many** relationship.



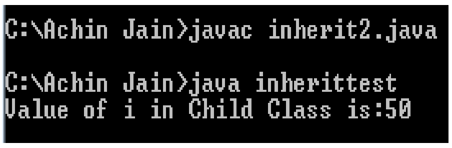
Example of Member Access and Inheritance

Case 1: Member Variables have no access modifier (default access modifier)

```
class inherit1
{
    int i=10;
    void meth0()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        System.out.println("Value of i in Child Class is:"+(i*5));
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

Variable "i" with no access modifier and we are using it in child class. Program runs with no error in this case as members with default access modifier can be used in child class

Output



```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Value of i in Child Class is:50
```

Case 2: Member Variables have public/protected access modifier. If a member of class is declared as either public or protected than it can be accessed from child or inherited class.

```
class inherit1
{
    public int i=10;
    protected int j=5;
    void meth0()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        j=j+i;
        System.out.println("Value of j in Child Class is:"+j);
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

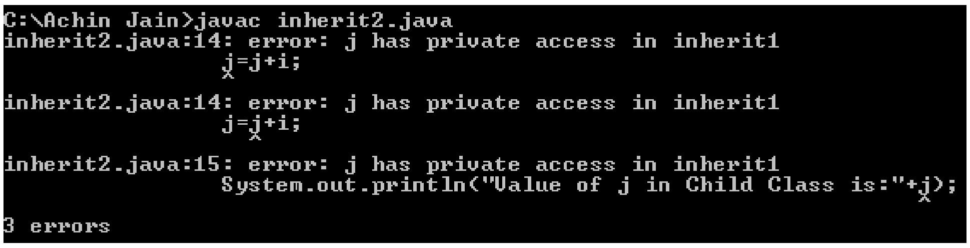
Case 3: Member Variables have private access modifier. If a member of class is declared as private than it cannot be accessed outside the class not even in the inherited class.

```
class inherit1
{
    public int i=10;
    private int j=5;
    void meth0()
    {
        System.out.println("Value of i in Base Class is:"+i);
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        j=j+i;
        System.out.println("Value of j in Child Class is:"+j);
    }
}
```

Variable 'j' is declared as Private in class inherit1 which means it cannot be accessed outside the class scope.

In the inherited class an attempt to modify the value of a private variable is made which results in compile time error

Output:



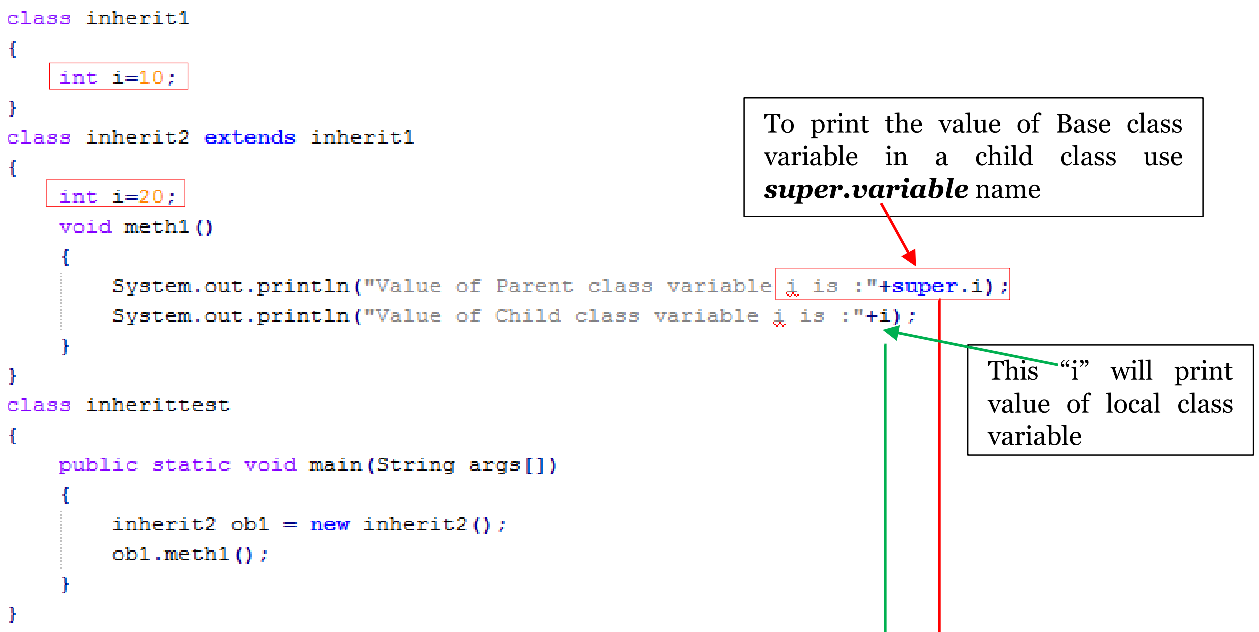
```
C:\Achin Jain>javac inherit2.java
inherit2.java:14: error: j has private access in inherit1
                j=j+i;
                ^
inherit2.java:14: error: j has private access in inherit1
                j=j+i;
                  ^
inherit2.java:15: error: j has private access in inherit1
                System.out.println("Value of j in Child Class is:"+j);
                                                                   ^
3 errors
```

super Keyword:

super is a reference variable that is used to refer immediate parent class object. Uses of super keyword are as follows:

1. super() is used to invoke immediate parent class constructors
2. super is used to invoke immediate parent class method
3. super is used to refer immediate parent class variable

Example:

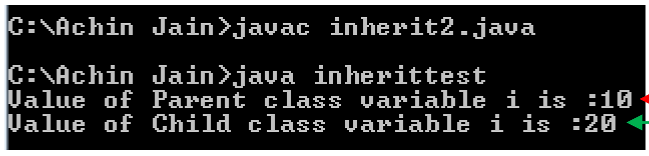


```
class inherit1
{
    int i=10;
}
class inherit2 extends inherit1
{
    int i=20;
    void meth1()
    {
        System.out.println("Value of Parent class variable i is :"+super.i);
        System.out.println("Value of Child class variable i is :"+i);
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

To print the value of Base class variable in a child class use ***super.variable*** name

This "i" will print value of local class variable

Output:

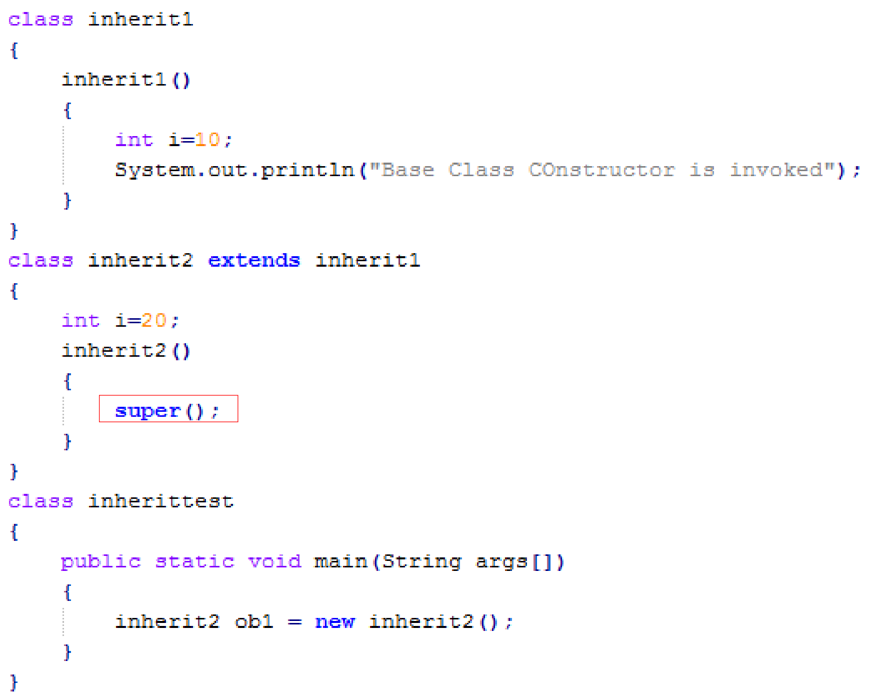


```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Value of Parent class variable i is :10
Value of Child class variable i is :20
```

Example to call Immediate Parent Class Constructor using super Keyword

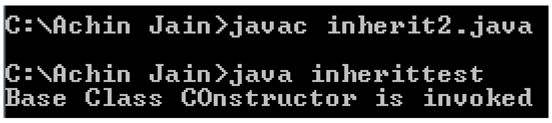
The super() keyword can be used to invoke the Parent class constructor as shown in the example below.

Note: super() is added in each class constructor automatically by compiler. As default constructor is provided by compiler automatically but it also adds super() for the first statement. If you are creating your own constructor and you don't have super() as the first statement, compiler will provide super() as the first statement of the constructor.



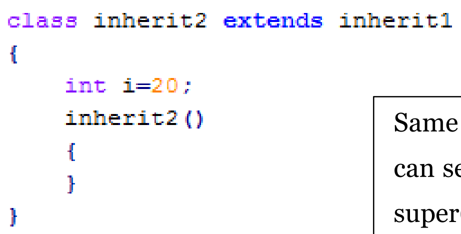
```
class inherit1
{
    inherit1()
    {
        int i=10;
        System.out.println("Base Class COnstructor is invoked");
    }
}
class inherit2 extends inherit1
{
    int i=20;
    inherit2()
    {
        super();
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
    }
}
```

Output:



```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Base Class COnstructor is invoked
```

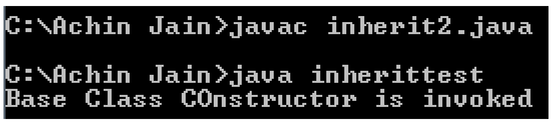
Example where super() is provided by compiler



```
class inherit2 extends inherit1
{
    int i=20;
    inherit2()
    {
    }
}
```

Same program as above but no super keyword is used. You can see in the o/p below that compiler implicitly adds the super() keyword and same output is seen

Output:



```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Base Class COnstructor is invoked
```

Method Overriding

If a class inherits a method from its super class, then there is a chance to override the method provided that it is not marked final. The benefit of overriding is: ability to define a behavior that's specific to the subclass type which means a subclass can implement a parent class method based on its requirement. In object-oriented terms, overriding means to override the functionality of an existing method.

```
class inherit1
{
    void meth1()
    {
        System.out.println("Base Method");
    }
}
class inherit2 extends inherit1
{
    void meth1()
    {
        System.out.println("Child Method");
    }
}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

meth1() method is overridden in the child class. Now when meth1() method is invoked from object of child class then compiler will first search for method definition in class from which it is invoked. If method definition is not found then compiler will look for method definition in the parent class.

Ouput:

```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Child Method
```

Case – When there is no method definition in child class

```
class inherit1
{
    void meth1()
    {
        System.out.println("Base Method");
    }
}
class inherit2 extends inherit1
{

}
class inherittest
{
    public static void main(String args[])
    {
        inherit2 ob1 = new inherit2();
        ob1.meth1();
    }
}
```

Method definition is not found in child class, so compiler will now search in Parent Class.

Output:

```
C:\Achin Jain>javac inherit2.java

C:\Achin Jain>java inherittest
Base Method
```

Final Keyword

Final keyword can be used in the following ways:

1. **Final Variable :** Once a variable is declared as final, its value cannot be changed during the scope of the program
2. **Final Method :** Method declared as final cannot be overridden
3. **Final Class :** A final class cannot be inherited

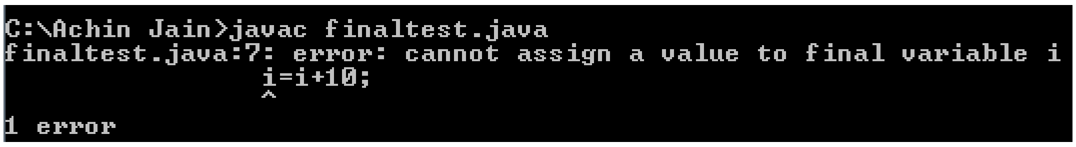
Example of Final Variable

```
class finalvar
{
    final int i=10;
    finalvar()
    {
        System.out.println("Value of Final Variable i is :"+i);
        i=i+10;
        System.out.println("Value of Final Variable i after change is :"+i);
    }
}
class finaltest
{
    public static void main(String args[])
    {
        finalvar obj1 = new finalvar();
    }
}
```

Variable "i" is declared as final

In this code we are trying to modify the value of a Final Variable which will result in error as shown in the output

Output:



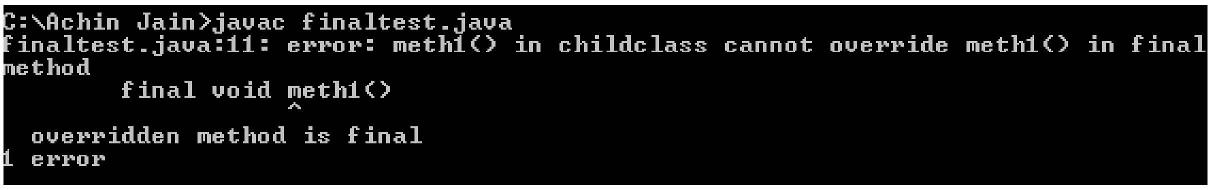
```
C:\Achin Jain>javac finaltest.java
finaltest.java:7: error: cannot assign a value to final variable i
                i=i+10;
                ^
1 error
```

Example of Final Method

```
class finalmethod
{
    int i=10;
    final void meth1()
    {
        System.out.println("Value of Final Variable i is :"+i);
    }
}
class childclass extends finalmethod
{
    final void meth1()
    {
        System.out.println("Value of Final Variable i is :"+i);
    }
}
class finaltest
{
    public static void main(String args[])
    {
        childclass obj1 = new childclass();
        obj1.meth1();
    }
}
```

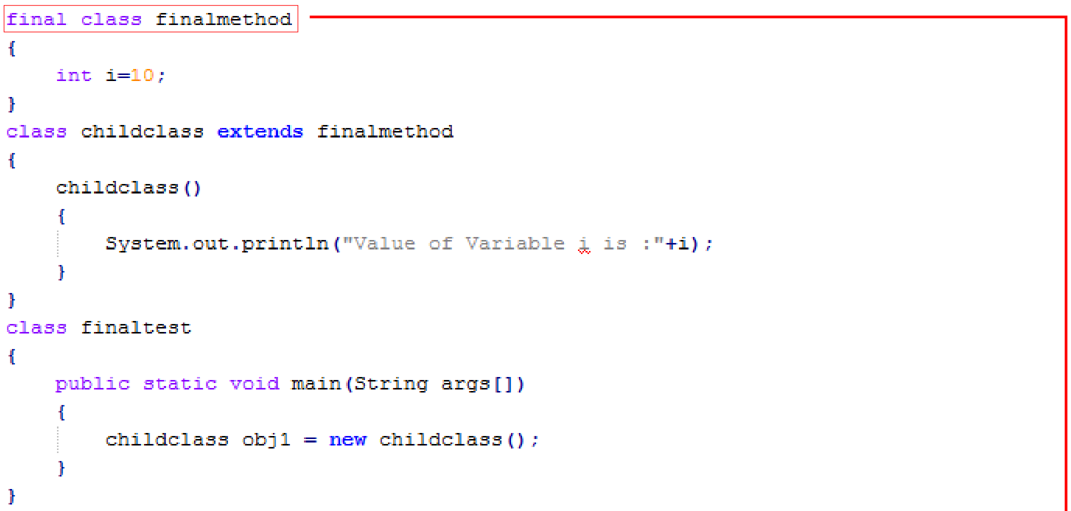
In this example an attempt to override a final method (meth1()) is made which results in error

Output



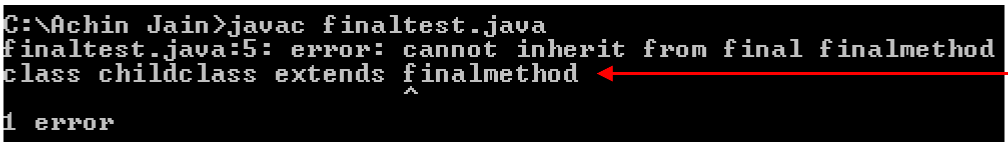
```
C:\Achin Jain>javac finaltest.java
finaltest.java:11: error: meth1() in childclass cannot override meth1() in final
method
        final void meth1()
                   ^
  overridden method is final
1 error
```

Example of Final Class



```
final class finalmethod
{
    int i=10;
}
class childclass extends finalmethod
{
    childclass()
    {
        System.out.println("Value of Variable i is :"+i);
    }
}
class finaltest
{
    public static void main(String args[])
    {
        childclass obj1 = new childclass();
    }
}
```

Output

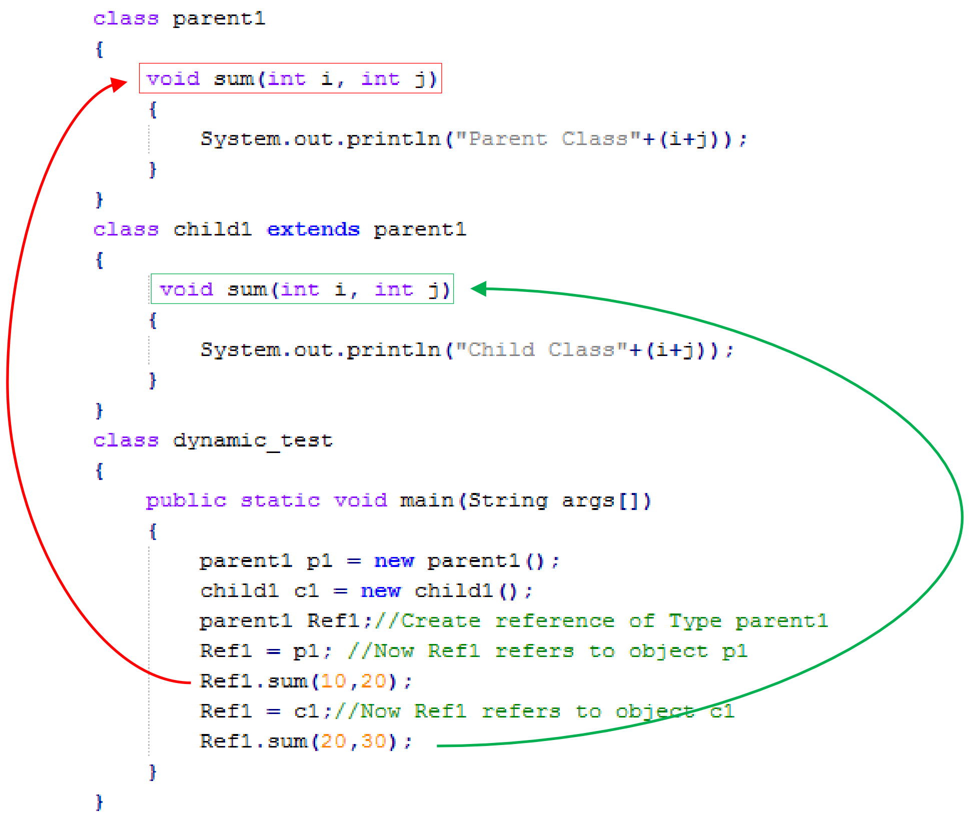


```
C:\Achin Jain>javac finaltest.java
finaltest.java:5: error: cannot inherit from final finalmethod
class childclass extends finalmethod
                         ^
1 error
```

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. Method to execution based upon the type of the object being referred to at the time the call occurs. Thus, this determination is made at run time. In other words, it is the type of the object being referred to (not the type of the reference variable) that determines which version of an overridden method will be executed.

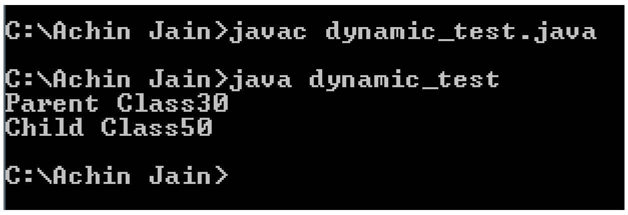
Example



```
class parent1
{
    void sum(int i, int j)
    {
        System.out.println("Parent Class"+(i+j));
    }
}
class child1 extends parent1
{
    void sum(int i, int j)
    {
        System.out.println("Child Class"+(i+j));
    }
}
class dynamic_test
{
    public static void main(String args[])
    {
        parent1 p1 = new parent1();
        child1 c1 = new child1();
        parent1 Ref1;//Create reference of Type parent1
        Ref1 = p1; //Now Ref1 refers to object p1
        Ref1.sum(10,20);
        Ref1 = c1;//Now Ref1 refers to object c1
        Ref1.sum(20,30);
    }
}
```

In this example a reference is created "***Ref1***" of type parent1. To call an overridden method of any class this reference variable is assigned an object of that class. For ex. To call sum() method of child class Ref1 is assigned object c1 of child class.

Output



```
C:\Achin Jain>javac dynamic_test.java

C:\Achin Jain>java dynamic_test
Parent Class30
Child Class50

C:\Achin Jain>
```

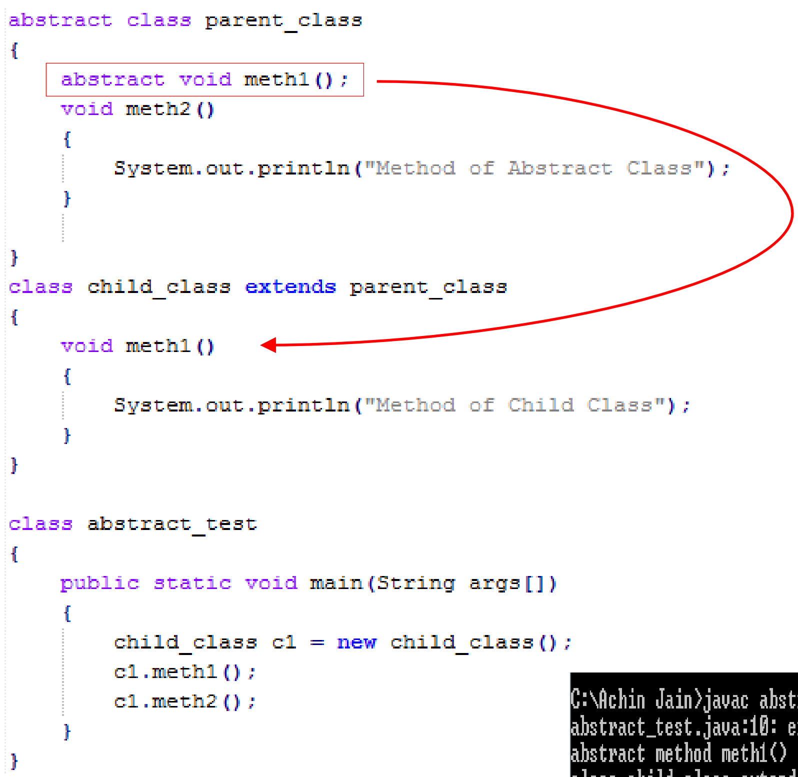
Abstract Classes

When the keyword abstract appears in a class definition, it means that zero or more of its methods are abstract.

- An abstract method has no body.
- Some of the subclass has to override it and provide the implementation.
- Objects cannot be created out of abstract class.
- Abstract classes basically provide a guideline for the properties and methods of an object.

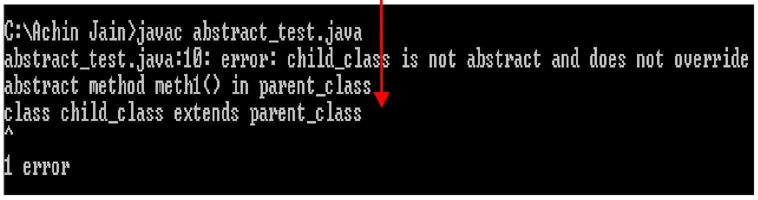
- In order to use abstract classes, they have to be subclassed.
- There are situations in which you want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.

Example



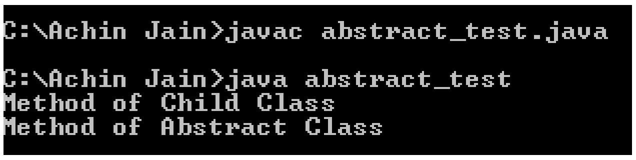
```
abstract class parent_class
{
    abstract void meth1();
    void meth2()
    {
        System.out.println("Method of Abstract Class");
    }

}
class child_class extends parent_class
{
    void meth1()
    {
        System.out.println("Method of Child Class");
    }
}

class abstract_test
{
    public static void main(String args[])
    {
        child_class c1 = new child_class();
        c1.meth1();
        c1.meth2();
    }
}
```

Since Method meth1() is declared as abstract it needs to be override in the inherited class. However if the method is not overridden compile time error will be generated as shown below.



```
C:\Achin Jain>javac abstract_test.java
abstract_test.java:10: error: child_class is not abstract and does not override
abstract method meth1() in parent_class
class child_class extends parent_class
^
1 error
```

Output



```
C:\Achin Jain>javac abstract_test.java

C:\Achin Jain>java abstract_test
Method of Child Class
Method of Abstract Class
```

Interface

An interface is a collection of abstract methods. A class implements an interface, thereby inheriting the abstract methods of the interface. An interface is not a class. Writing an interface is similar to writing a class, but they are two different concepts. A class describes the attributes and behaviours of an object. An interface contains behaviours 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.

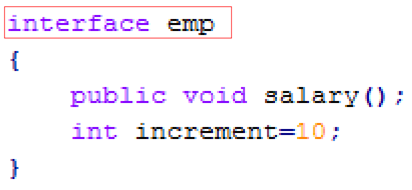
An interface is similar to a class in the following ways:

- 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 bytecode 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.

Example to Create Interface

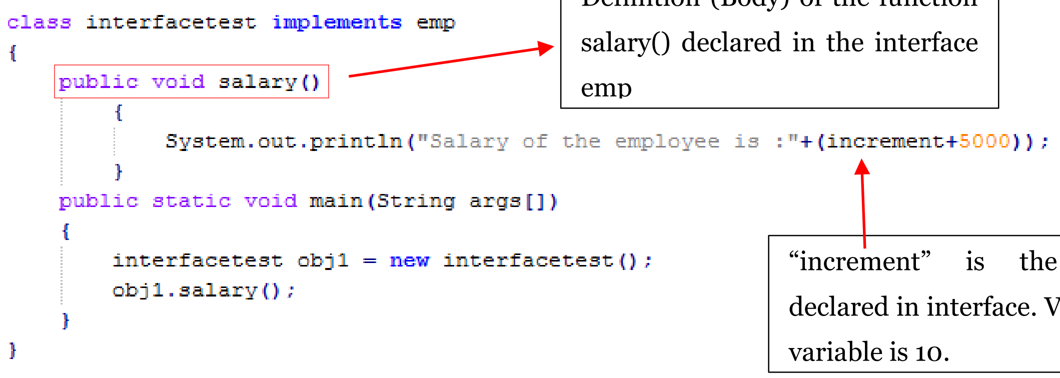


```
interface emp
{
    public void salary();
    int increment=10;
}
```

As you can see in the above example method is not declared as abstract explicitly and similarly variable is not declared as static final.

But when Java compiles the code, method is declared as abstract implicitly by compiler and variables are declared as static final.

Example to implement interface

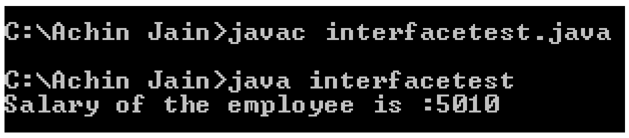


```
class interfacetest implements emp
{
    public void salary()
        {
            System.out.println("Salary of the employee is :"+(increment+5000));
        }
    public static void main(String args[])
    {
        interfacetest obj1 = new interfacetest();
        obj1.salary();
    }
}
```

Definition (Body) of the function salary() declared in the interface emp

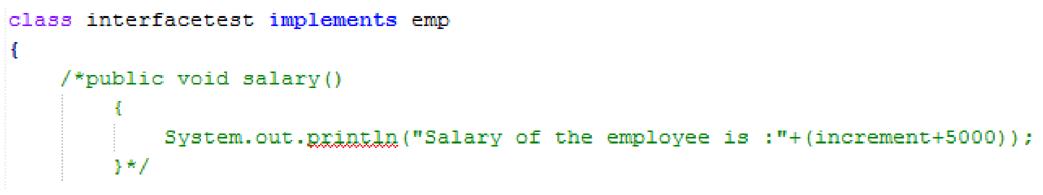
"increment" is the variable declared in interface. Value of the variable is 10.

Output



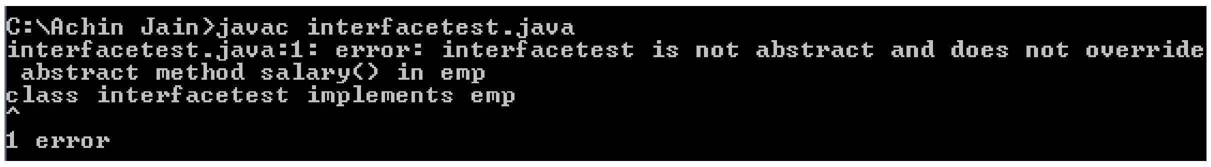
```
C:\Achin Jain>javac interfacetest.java

C:\Achin Jain>java interfacetest
Salary of the employee is :5010
```

Case: When definition of the method declared in interface is not provided in the inherited class. In this case an error will be shown during compile time of the program.



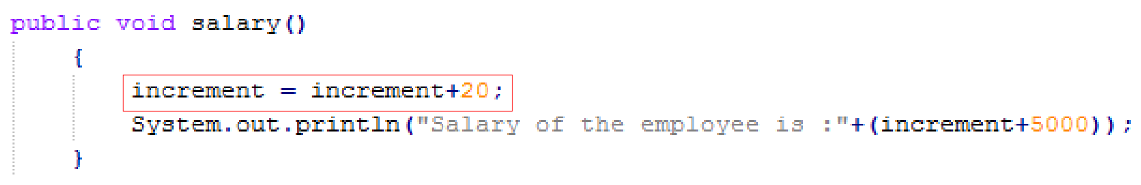
```
class interfacetest implements emp
{
    /*public void salary()
        {
            System.out.println("Salary of the employee is :"+(increment+5000));
        }*/
```

Output:



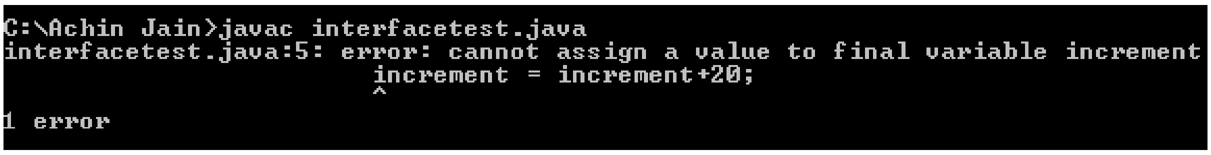
```
C:\Achin Jain>javac interfacetest.java
interfacetest.java:1: error: interfacetest is not abstract and does not override
 abstract method salary() in emp
class interfacetest implements emp
^
1 error
```

<u>Case:</u> Try to modify the value of variable declared in interface



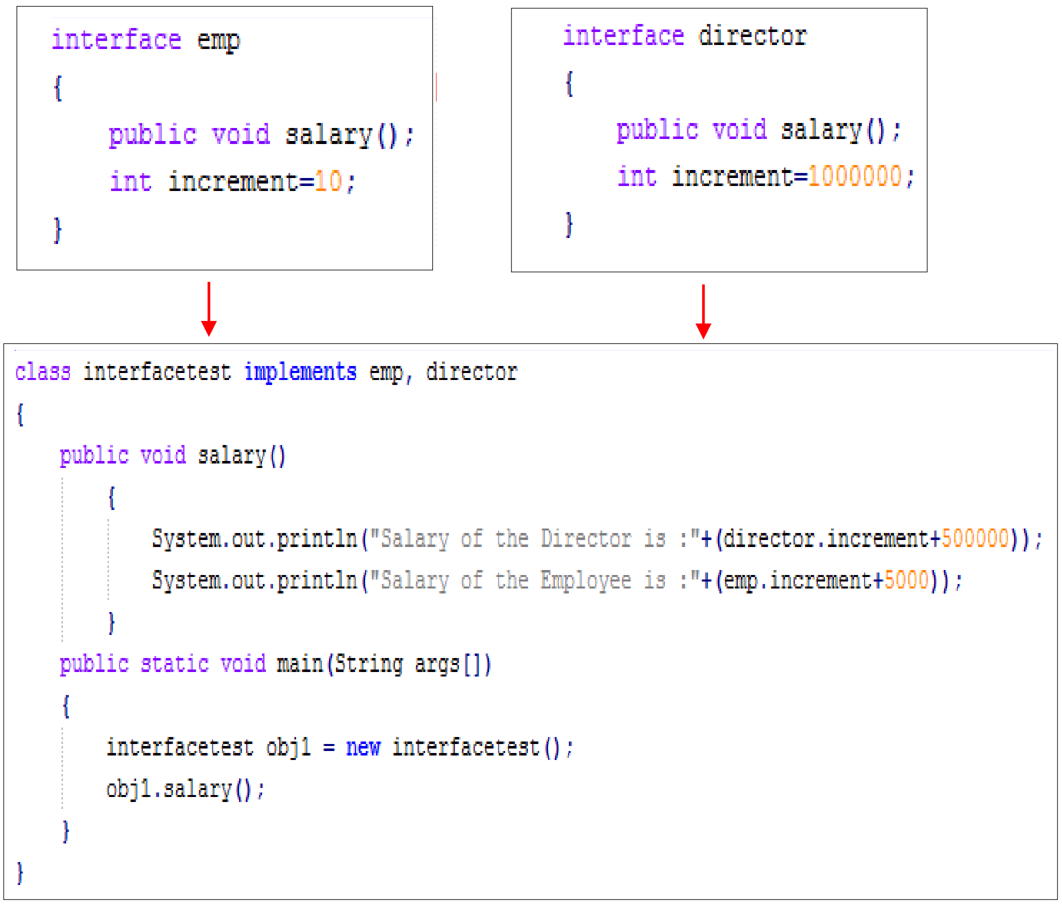
```java
public void salary()
    {
        increment = increment+20;
        System.out.println("Salary of the employee is :"+(increment+5000));
    }
```

<u>Output</u>



```
C:\Achin Jain>javac interfacetest.java
interfacetest.java:5: error: cannot assign a value to final variable increment
                        increment = increment+20;
                        ^
1 error
```

<u>How Interface provide Multiple Inheritance in Java</u>



```java
interface emp
{
    public void salary();
    int increment=10;
}
```

```java
interface director
{
    public void salary();
    int increment=1000000;
}
```

```java
class interfacetest implements emp, director
{
    public void salary()
        {
            System.out.println("Salary of the Director is :"+(director.increment+500000));
            System.out.println("Salary of the Employee is :"+(emp.increment+5000));
        }
    public static void main(String args[])
    {
        interfacetest obj1 = new interfacetest();
        obj1.salary();
    }
}
```

In the above example, two interfaces are created with names "emp" and "director" and both interfaces contain same method salary(). In the class interfacetest we have implemented both the interfaces. Now as per the definition of interface a method declared must be overridden in the class that implements the interface. Although in the above example there are two salary methods declared in different interfaces, but in the child class there is only single instance of the method. So whenever a call is made to the salary method it is always the overridden method that gets invoked, leaving no room for ambiguity.

Package

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 name space 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, 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.

Example to Create Package

When creating a package, you should choose a name for the package and put a package statement with that name at the top of every source file that contains the classes, interfaces that you want to include in the package. The package statement should be the first line in the source file.
In the example a package with name “***achin_java***” is created with statement “*package achin_java;*”.

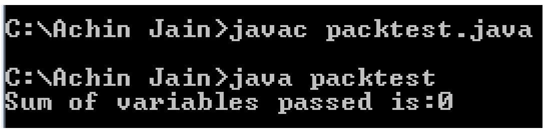
```
package achin_java;

public class add
    {
        int a;
        public int b;
        private int c;
        protected int d;
        public void add_meth()
        {
            System.out.println("Sum of variables passed is:"+(a+b+c+d));
        }
    }
```

Importing the Package in other Class

To import a package use keyword "***import***" followed by package name and class name. If you want to import all classes use .* instead of single class name.

```
import achin_java.*;

class packtest
{
    public static void main(String args[])
    {
        add obj = new add();
        obj.add_meth();
    }
}
```

Output:



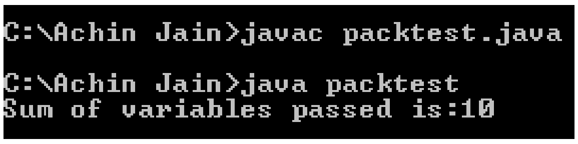
```
C:\Achin Jain>javac packtest.java

C:\Achin Jain>java packtest
Sum of variables passed is:0
```

Case 1: Trying to access a public variable outside the package.

In the code of package "achin_java" four different types of variables are declared. Variable 'b' is declared as public. See the following code in which we have assigned a value to 'b' in class packtest.

```
add obj = new add();
obj.b=10;
obj.add_meth();
```

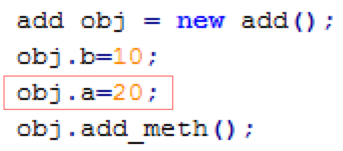
Output



```
C:\Achin Jain>javac packtest.java

C:\Achin Jain>java packtest
Sum of variables passed is:10
```

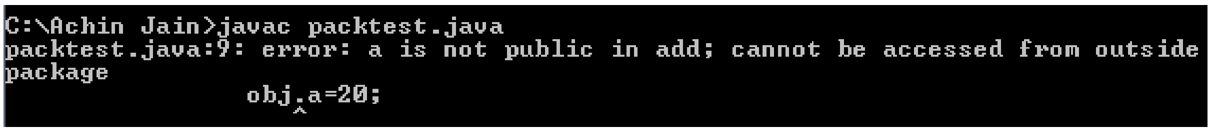
Case 2: Trying to access a variable with default access modifier outside the package.

Variable 'a' is declared as public. See the following code in which we have assigned a value to 'a' in class packtest. You will see an error as shown in the output



```
add obj = new add();
obj.b=10;
obj.a=20;
obj.add_meth();
```

Output:



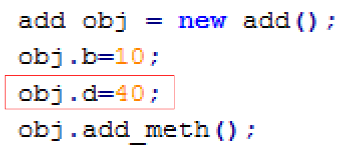
```
C:\Achin Jain>javac packtest.java
packtest.java:9: error: a is not public in add; cannot be accessed from outside
package
                obj.a=20;
                   ^
```

Case 3: Trying to access a variable with private access modifier outside the package. This is clear case in which you don't have permission to access private variable outside its scope.

Case 4: Trying to access a variable with protected access modifier outside the package.

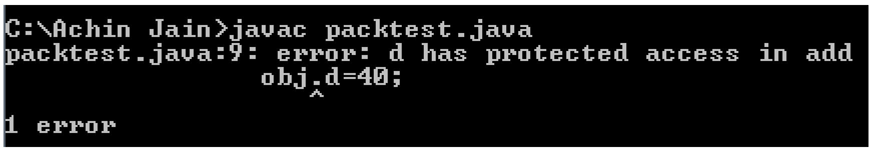
In this there are two different cases. One is trying to access the protected variable outside the package in the same way as described above for other cases. Other case if we are inheriting a class defined in a package and then trying to use the protected declared member.

When class is not inherited



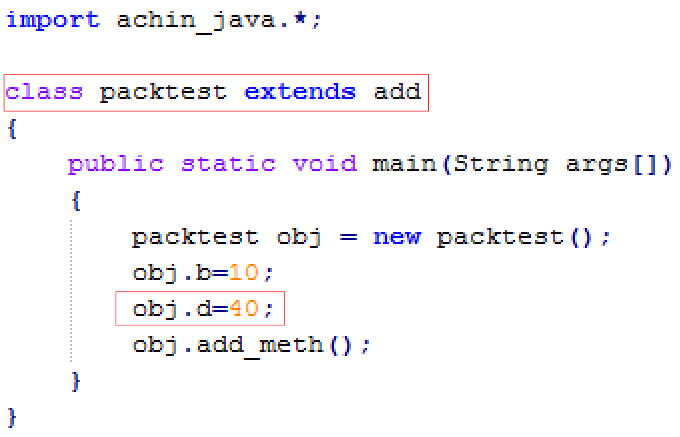
```
add obj = new add();
obj.b=10;
obj.d=40;
obj.add_meth();
```

Output:



```
C:\Achin Jain>javac packtest.java
packtest.java:9: error: d has protected access in add
                obj.d=40;
                   ^
1 error
```

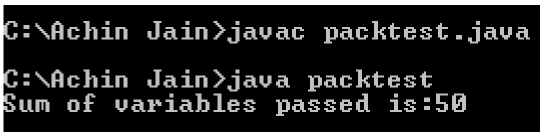
Case when class is inherited



```
import achin_java.*;

class packtest extends add
{
    public static void main(String args[])
    {
        packtest obj = new packtest();
        obj.b=10;
        obj.d=40;
        obj.add_meth();
    }
}
```

We can access a protected member in sub class of different package. As in this example packtest class inherits class add and now we are accessing protected variable '***d***' which results in no error and program executes as expected.

Output



```
C:\Achin Jain>javac packtest.java

C:\Achin Jain>java packtest
Sum of variables passed is:50
```

Access Protection in Packages

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non subclass	No	No	No	Yes