Java - Inheritance

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance the information is made manageable in a hierarchical order.

The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

extends Keyword

extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax

```
class Super {
   .....
   .....
}
class Sub extends Super {
   .....
   .....
}
```

Sample Code

Following is an example demonstrating Java inheritance. In this example, you can observe two classes namely Calculation and My_Calculation.

Using extends keyword, the My_Calculation inherits the methods addition() and Subtraction() of Calculation class.

Copy and paste the following program in a file with name My_Calculation.java

Example

```
class Calculation {
   int z;

   public void addition(int x, int y) {
      z = x + y;
      System.out.println("The sum of the given numbers:"+z);
   }

   public void Subtraction(int x, int y) {
      z = x - y;
      System.out.println("The difference between the given numbers:"+z);
   }
}

public class My_Calculation extends Calculation {
  public void multiplication(int x, int y) {
    z = x * y;
    System.out.println("The product of the given numbers:"+z);
  }

  public static void main(String args[]) {
    int a = 20, b = 10;
    My_Calculation demo = new My_Calculation();
    demo.addition(a, b);
    demo.Subtraction(a, b);
    demo.multiplication(a, b);
  }
}
```

Compile and execute the above code as shown below.

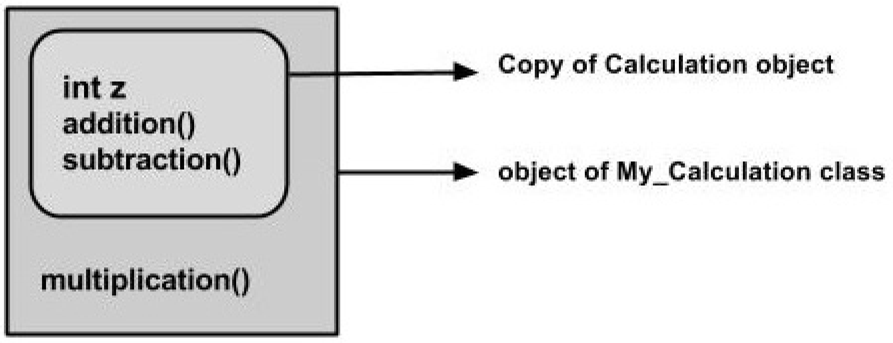
```
javac My_Calculation.java
java My_Calculation
```

After executing the program, it will produce the following result –

Output

```
The sum of the given numbers:30
The difference between the given numbers:10
The product of the given numbers:200
```

In the given program, when an object to **My_Calculation** class is created, a copy of the contents of the superclass is made within it. That is why, using the object of the subclass you can access the members of a superclass.



The Superclass reference variable can hold the subclass object, but using that variable you can access only the members of the superclass, so to access the members of both classes it is recommended to always create reference variable to the subclass.

If you consider the above program, you can instantiate the class as given below. But using the superclass reference variable (**cal** in this case) you cannot call the method **multiplication()**, which belongs to the subclass My_Calculation.

```
Calculation demo = new My_Calculation();
```

```
demo.addition(a, b);
demo.Subtraction(a, b);
```

Note – A subclass inherits all the members (fields, methods, and nested classes) from its superclass. Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

The super keyword

The **super** keyword is similar to **this** keyword. Following are the scenarios where the super keyword is used.

- It is used to **differentiate the members** of superclass from the members of subclass, if they have same names.
- It is used to **invoke the superclass** constructor from subclass.

Differentiating the Members

If a class is inheriting the properties of another class. And if the members of the superclass have the names same as the sub class, to differentiate these variables we use super keyword as shown below.

```
super.variable
super.method();
```

Sample Code

This section provides you a program that demonstrates the usage of the **super** keyword.

In the given program, you have two classes namely *Sub_class* and *Super_class*, both have a method named display() with different implementations, and a variable named num with different values. We are invoking display() method of both classes and printing the value of the variable num of both classes. Here you can observe that we have used super keyword to differentiate the members of superclass from subclass.

Copy and paste the program in a file with name Sub_class.java.

Example

```
class Super_class {
   int num = 20;

   // display method of superclass
   public void display() {
      System.out.println("This is the display method of superclass");
   }
}

public class Sub_class extends Super_class {
   int num = 10;
```

```
  // display method of sub class
  public void display() {
    System.out.println("This is the display method of subclass");
  }

  public void my_method() {
    // Instantiating subclass
    Sub_class sub = new Sub_class();

    // Invoking the display() method of sub class
    sub.display();

    // Invoking the display() method of superclass
    super. display();

    // printing the value of variable num of subclass
    System.out.println("value of the variable named num in sub class:"+ sub.num);

    // printing the value of variable num of superclass
    System.out.println("value of the variable named num in super class:"+ super.num);
  }

  public static void main(String args[]) {
    Sub_class obj = new Sub_class();
    obj.my_method();
  }
}
```

Compile and execute the above code using the following syntax.

```
javac Super_Demo
java Super
```

On executing the program, you will get the following result –

Output

```
This is the display method of subclass
This is the display method of superclass
value of the variable named num in sub class:10
value of the variable named num in super class:20
```

Invoking Superclass Constructor

If a class is inheriting the properties of another class, the subclass automatically acquires the default constructor of the superclass. But if you want to call a parameterized constructor of the superclass, you need to use the super keyword as shown below.

```
super(values);
```

Sample Code

The program given in this section demonstrates how to use the super keyword to invoke the parametrized constructor of the superclass. This program contains a superclass and a subclass, where the superclass contains a parameterized constructor which accepts a integer value, and we used the super keyword to invoke the parameterized constructor of the superclass.

Copy and paste the following program in a file with the name Subclass.java

Example

```
class Superclass {
   int age;

   Superclass(int age) {
      this.age = age;
   }

   public void getAge() {
      System.out.println("The value of the variable named age in super class is: " +age);
   }
}

public class Subclass extends Superclass {
   Subclass(int age) {
      super(age);
   }

   public static void main(String args[]) {
      Subclass s = new Subclass(24);
      s.getAge();
   }
}
```

Compile and execute the above code using the following syntax.

```
javac Subclass
java Subclass
```

On executing the program, you will get the following result –

Output

```
The value of the variable named age in super class is: 24
```

IS-A Relationship

Programmer is the subclass and Employee is the superclass. The relationship between the two classes is **Programmer IS-A Employee**. It means that Programmer is a type of Employee.

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

HAS-A relationship

Aggregation in Java

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship.

Consider a situation, Employee object contains many informations such as id, name, emailId etc. It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

```
class Employee
{
int id;
String name;
Address address;//Address is a class
...
}
```

In such case, Employee has an entity reference address, so relationship is Employee HAS-A address.

Why use Aggregation?

- For Code Reusability.

When use Aggregation?

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

example of Aggregation

In this example, Employee has an object of Address, address object contains its own information's such as city, state, country etc. In such case relationship is Employee HAS-A address.

Address.java

```
public class Address {
String city,state,country;

public Address(String city, String state, String country) {
   this.city = city;
   this.state = state;
   this.country = country;
}

}
```

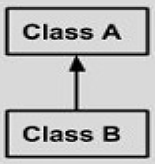
Emp.java

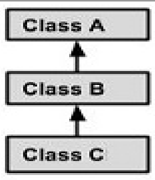
```
public class Emp {
int id;
String name;
Address address;

public Emp(int id, String name,Address address) {
   this.id = id;
   this.name = name;
   this.address=address;
}

void display(){
System.out.println(id+" "+name);
System.out.println(address.city+" "+address.state+" "+address.country);
}

public static void main(String[] args) {
Address address1=new Address("gzb","UP","india");
Address address2=new Address("gno","UP","india");
Emp e=new Emp(111,"varun",address1);
Emp e2=new Emp(112,"arun",address2);

e.display();
e2.display();
}
}
```

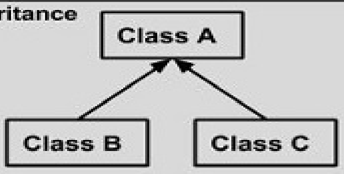
```
Output:111 varun
       gzb UP india
       112 arun
       gno UP india
```

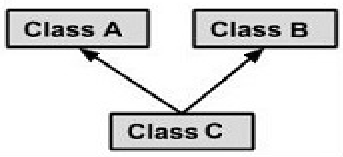
Types of Inheritance

There are various types of inheritance as demonstrated below.

Single Inheritance 	public class A {
	……..
	}
	public class B **extends** A {
	………
	}

Multi Level Inheritance 	public class A { ………………..}
	public class B **extends** A {………………..}
	public class C **extends** B {………………… }

Hierarchical Inheritance 	public class A { ………………..}
	public class B **extends** A {………………..}
	public class C **extends** A {………………… }

Multiple Inheritance 	public class A { ………………..}
	public class B {………………..}
	public class C **extends** A,B {
	…………………
	} // Java does not support mutiple Inheritance

A very important fact to remember is that Java does not support multiple inheritance. This means that a class cannot extend more than one class. Therefore following is illegal –

Example

```
public class extends Animal, Mammal{}
```

However, a class can implement one or more interfaces, which has helped Java get rid of the impossibility of multiple inheritance.

Java - Access Modifiers

Java provides a number of access modifiers to set access levels for classes, variables, methods, and constructors. The four access levels are –

- Visible to the package, the default. No modifiers are needed.
- Visible to the class only (private).
- Visible to the world (public).
- Visible to the package and all subclasses (protected).

Default Access Modifier - No Keyword

Default access modifier means we do not explicitly declare an access modifier for a class, field, method, etc.

A variable or method declared without any access control modifier is available to any other class in the same package. The fields in an interface are implicitly public static final and the methods in an interface are by default public.

Example

Variables and methods can be declared without any modifiers, as in the following examples –

```
String version = "1.5.1";

boolean processOrder() {


   return true;
}
```

Private Access Modifier - Private

Methods, variables, and constructors that are declared private can only be accessed within the declared class itself.

Private access modifier is the most restrictive access level. Class and interfaces cannot be private.

Variables that are declared private can be accessed outside the class, if public getter methods are present in the class.

Using the private modifier is the main way that an object encapsulates itself and hides data from the outside world.

Example

The following class uses private access control –

```
public class Logger {
   private String format;

   public String getFormat() {
       return this.format;
   }

   public void setFormat(String format) {
       this.format = format;
   }
}
```

Here, the *format* variable of the Logger class is private, so there's no way for other classes to retrieve or set its value directly.

So, to make this variable available to the outside world, we defined two public methods: *getFormat()*, which returns the value of format, and *setFormat(String)*, which sets its value.

Public Access Modifier - Public

A class, method, constructor, interface, etc. declared public can be accessed from any other class. Therefore, fields, methods, blocks declared inside a public class can be accessed from any class belonging to the Java Universe.

However, if the public class we are trying to access is in a different package, then the public class still needs to be imported. Because of class inheritance, all public methods and variables of a class are inherited by its subclasses.

Example

The following function uses public access control –

```
public static void main(String[] arguments) {
    // ...
}
```

The main() method of an application has to be public. Otherwise, it could not be called by a Java interpreter (such as java) to run the class.

Protected Access Modifier - Protected

Variables, methods, and constructors, which are declared protected in a superclass can be accessed only by the subclasses in other package or any class within the package of the protected members' class.

The protected access modifier cannot be applied to class and interfaces. Methods, fields can be declared protected, however methods and fields in a interface cannot be declared protected.

Protected access gives the subclass a chance to use the helper method or variable, while preventing a nonrelated class from trying to use it.

Example

The following parent class uses protected access control, to allow its child class override *openSpeaker()* method –

```
class AudioPlayer {
  protected boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}

class StreamingAudioPlayer extends AudioPlayer {
  boolean openSpeaker(Speaker sp) {
    // implementation details
  }
}
```

Here, if we define openSpeaker() method as private, then it would not be accessible from any other class other than *AudioPlayer*. If we define it as public, then it would become accessible to all the outside world. But our intention is to expose this method to its subclass only, that's why we have used protected modifier.

Access Control and Inheritance

The following rules for inherited methods are enforced –

- Methods declared public in a superclass also must be public in all subclasses.
- Methods declared protected in a superclass must either be protected or public in subclasses; they cannot be private.
- Methods declared private are not inherited at all, so there is no rule for them.

Java Method Overriding

we learned about inheritance. Inheritance is an OOP property that allows us to derive a new class (subclass) from an existing class (superclass). The subclass inherits the attributes and methods of the superclass.

Now, if the same method is defined in both the superclass and the subclass, then the method of the subclass class overrides the method of the superclass. This is known as method overriding.

Example 1: Method Overriding

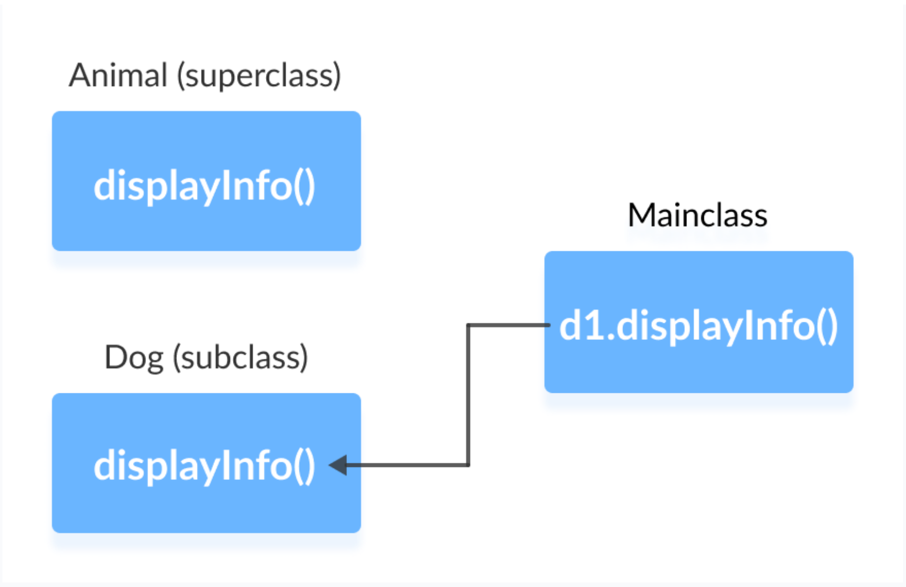
```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   @Override
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:

```
I am a dog.
```

In the above program, the `displayInfo()` method is present in both the `Animal` superclass and the `Dog` subclass.

When we call `displayInfo()` using the `d1` object (object of the subclass), the method inside the subclass `Dog` is called. The `displayInfo()` method of the subclass overrides the same method of the superclass.



Notice the use of the `@Override` annotation in our example. In Java, annotations are the metadata that we used to provide information to the compiler. Here, the `@Override` annotation specifies the compiler that the method after this annotation overrides the method of the superclass.

It is not mandatory to use `@Override`. However, when we use this, the method should follow all the rules of overriding. Otherwise, the compiler will generate an error.

Java Overriding Rules

- Both the superclass and the subclass must have the same method name, the same return type and the same parameter list.
- We cannot override the method declared as `final` and `static`.
- We should always override abstract methods of the superclass.

super Keyword in Java Overriding

A common question that arises while performing overriding in Java is:

Can we access the method of the superclass after overriding?

Well, the answer is **Yes**. To access the method of the superclass from the subclass, we use the `super` keyword.

Example 2: Use of super Keyword

```
class Animal {
   public void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   public void displayInfo() {
      super.displayInfo();
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:

```
I am an animal.
I am a dog.
```

In the above example, the subclass `Dog` overrides the method `displayInfo()` of the superclass `Animal`.

When we call the method `displayInfo()` using the `d1` object of the `Dog` subclass, the method inside the `Dog` subclass is called; the method inside the superclass is not called.

Inside `displayInfo()` of the `Dog` subclass, we have used `super.displayInfo()` to call `displayInfo()` of the superclass.

It is important to note that constructors in Java are not inherited. Hence, there is no such thing as constructor overriding in Java.

However, we can call the constructor of the superclass from its subclasses. For that, we use `super()`.

Access Specifiers in Method Overriding

The same method declared in the superclass and its subclasses can have different access specifiers. However, there is a restriction.

We can only use those access specifiers in subclasses that provide larger access than the access specifier of the superclass. For example,

Suppose, a method `myClass()` in the superclass is declared `protected`. Then, the same method `myClass()` in the subclass can be either `public` or `protected`, but not `private`.

Example 3: Access Specifier in Overriding

```
class Animal {
   protected void displayInfo() {
      System.out.println("I am an animal.");
   }
}

class Dog extends Animal {
   public void displayInfo() {
      System.out.println("I am a dog.");
   }
}

class Main {
   public static void main(String[] args) {
      Dog d1 = new Dog();
      d1.displayInfo();
   }
}
```

Output:

```
I am a dog.
```

In the above example, the subclass `Dog` overrides the method `displayInfo()` of the superclass `Animal`.

Whenever we call `displayInfo()` using the `d1` (object of the subclass), the method inside the subclass is called.

Notice that, the `displayInfo()` is declared `protected` in the `Animal` superclass. The same method has the `public` access specifier in the `Dog` subclass. This is possible because the `public` provides larger access than the `protected`.

Java Abstract Class and Abstract Methods

Java Abstract Class

The abstract class in Java cannot be instantiated (we cannot create objects of abstract classes). We use the `abstract` keyword to declare an abstract class. For example,

```
// create an abstract class
abstract class Language {
  // fields and methods
}
...

// try to create an object Language
// throws an error
Language obj = new Language();
```

An abstract class can have both the regular methods and abstract methods. For example,

```
abstract class Language {

  // abstract method
  abstract void method1();

  // regular method
  void method2() {
    System.out.println("This is regular method");
  }
}
```

Java Abstract Method

A method that doesn't have its body is known as an abstract method. We use the same `abstract` keyword to create abstract methods. For example,

```
abstract void display();
```

Here, `display()` is an abstract method. The body of `display()` is replaced by `;`.
If a class contains an abstract method, then the class should be declared abstract. Otherwise, it will generate an error. For example,

```
// error
// class should be abstract
class Language {

  // abstract method
  abstract void method1();
}
```

Example: Java Abstract Class and Method

Though abstract classes cannot be instantiated, we can create subclasses from it. We can then access members of the abstract class using the object of the subclass. For example,

```
abstract class Language {

  // method of abstract class
  public void display() {
    System.out.println("This is Java Programming");
  }
}

class Main extends Language {

  public static void main(String[] args) {

    // create an object of Main
    Main obj = new Main();

    // access method of abstract class
    // using object of Main class
    obj.display();
  }
}
```

Output

```
This is Java programming
```

In the above example, we have created an abstract class named `Language`. The class contains a regular method `display()`.

We have created the Main class that inherits the abstract class. Notice the statement,

```
obj.display();
```

Here, `obj` is the object of the child class `Main`. We are calling the method of the abstract class using the object `obj`.

Implementing Abstract Methods

If the abstract class includes any abstract method, then all the child classes inherited from the abstract superclass must provide the implementation of the abstract method. For example,

```
abstract class Animal {
  abstract void makeSound();

  public void eat() {
    System.out.println("I can eat.");
  }
}

class Dog extends Animal {

  // provide implementation of abstract method
  public void makeSound() {
    System.out.println("Bark bark");
  }
}

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

    // create an object of Dog class
    Dog d1 = new Dog();

    d1.makeSound();
    d1.eat();
  }
}
```

Output

```
Bark bark
I can eat.
```

In the above example, we have created an abstract class `Animal`. The class contains an abstract method `makeSound()` and a non-abstract method `eat()`.
We have inherited a subclass `Dog` from the superclass `Animal`. Here, the subclass `Dog` provides the implementation for the abstract method `makeSound()`.
We then used the object `d1` of the `Dog` class to call methods `makeSound()` and `eat()`.

Note: If the `Dog` class doesn't provide the implementation of the abstract method `makeSound()`, `Dog` should also be declared as abstract. This is because the subclass `Dog` inherits `makeSound()` from `Animal`.

Accesses Constructor of Abstract Classes

An abstract class can have constructors like the regular class. And, we can access the constructor of an abstract class from the subclass using the `super` keyword. For example,

```
abstract class Animal {
   Animal() {
      ….
   }
}

class Dog extends Animal {
   Dog() {
      super();
      ...
   }
}
```

Here, we have used the `super()` inside the constructor of `Dog` to access the constructor of the `Animal`.

Java final keyword

In Java, the `final` keyword is used to denote constants. It can be used with variables, methods, and classes.
Once any entity (variable, method or class) is declared `final`, it can be assigned only once. That is,

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

1. Java final Variable

In Java, we cannot change the value of a final variable. For example,

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

    // create a final variable
    final int AGE = 32;

    // try to change the final variable
    AGE = 45;
    System.out.println("Age: " + AGE);
  }
}
```

In the above program, we have created a final variable named `age`. And we have tried to change the value of the final variable.

When we run the program, we will get a compilation error with the following message.

```
cannot assign a value to final variable AGE
    AGE = 45;
    ^
```

Note: It is recommended to use uppercase to declare final variables in Java.

2. Java final Method

In Java, the `final` method cannot be overridden by the child class. For example,

```
class FinalDemo {
    // create a final method
    public final void display() {
      System.out.println("This is a final method.");
    }
}

class Main extends FinalDemo {
  // try to override final method
  public final void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

In the above example, we have created a final method named `display()` inside the `FinalDemo` class. Here, the `Main` class inherits the `FinalDemo` class.

We have tried to override the final method in the `Main` class. When we run the program, we will get a compilation error with the following message.

```
display() in Main cannot override display() in FinalDemo
  public final void display() {
                    ^
  overridden method is final
```

3. Java final Class

In Java, the final class cannot be inherited by another class. For example,

```
// create a final class
final class FinalClass {
  public void display() {
    System.out.println("This is a final method.");
  }
}

// try to extend the final class
class Main extends FinalClass {
  public  void display() {
    System.out.println("The final method is overridden.");
  }

  public static void main(String[] args) {
    Main obj = new Main();
    obj.display();
  }
}
```

In the above example, we have created a final class named `FinalClass`. Here, we have tried to inherit the final class by the `Main` class.

When we run the program, we will get a compilation error with the following message.

```
cannot inherit from final FinalClass
class Main extends FinalClass {
                   ^
```

Dynamic method dispatch or Runtime polymorphism in Java

Runtime Polymorphism in Java is achieved by Method overriding in which a child class overrides a method in its parent. An overridden method is essentially hidden in the parent class, and is not invoked unless the child class uses the super keyword within the overriding method. This method call resolution happens at runtime and is termed as Dynamic method dispatch mechanism.

Example

Let us look at an example.

```
class Animal {
  public void move() {
    System.out.println("Animals can move");
  }
}

class Dog extends Animal {
  public void move() {
    System.out.println("Dogs can walk and run");
  }
}

public class TestDog {

  public static void main(String args[]) {

    Animal a = new Animal(); // Animal reference and object
    Animal b = new Dog(); // Animal reference but Dog object

    a.move(); // runs the method in Animal class
    b.move(); // runs the method in Dog class
  }
}
```

This will produce the following result –

Output

```
Animals can move
Dogs can walk and run
```

In the above example, you can see that even though **b** is a type of Animal it runs the move method in the Dog class. The reason for this is: In compile time, the check is made on the reference type. However, in the runtime, JVM figures out the object type and would run the method that belongs to that particular object.

Therefore, in the above example, the program will compile properly since Animal class has the method move. Then, at the runtime, it runs the method specific for that object.