# Method Overloading in Java

* If a [class](https://www.javatpoint.com/object-and-class-in-java) has multiple methods having same name but different in parameters, it is known as **Method Overloading**.
* If we have to perform only one operation, having same name of the methods increases the readability of the [program](https://www.javatpoint.com/java-programs).
* Suppose you have to perform addition of the given numbers but there can be any number of arguments, if you write the method such as a(int,int) for two parameters, and b(int,int,int) for three parameters then it may be difficult for you as well as other programmers to understand the behavior of the method because its name differs.

## Advantage of method overloading

Method overloading *increases the readability of the program*.

### Different ways to overload the method

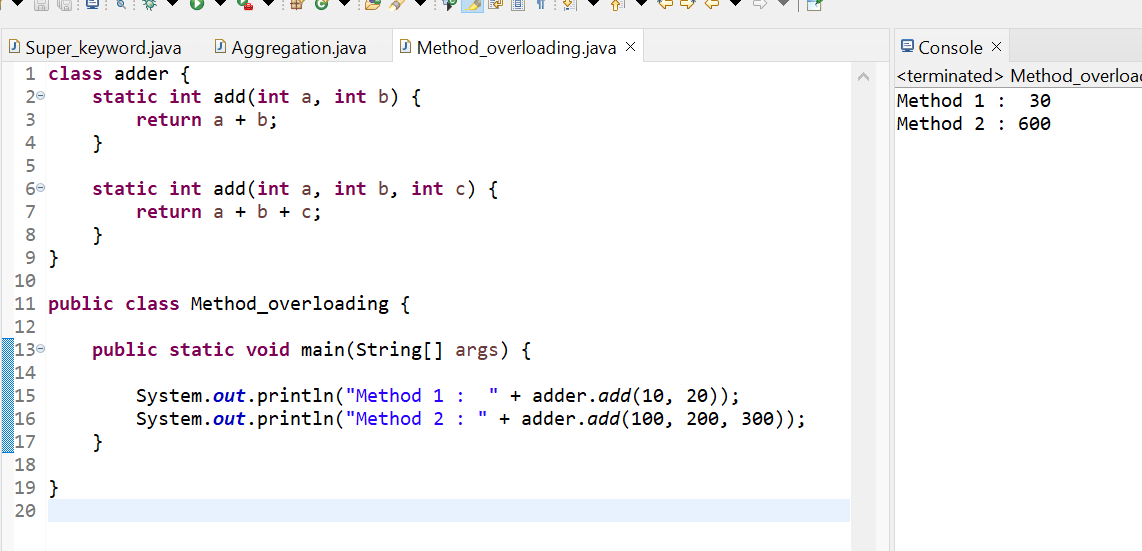
There are two ways to overload the method in java

* By changing number of arguments
* By changing the data type

**NOTE : In Java, Method Overloading is not possible by changing the return type of the method only**

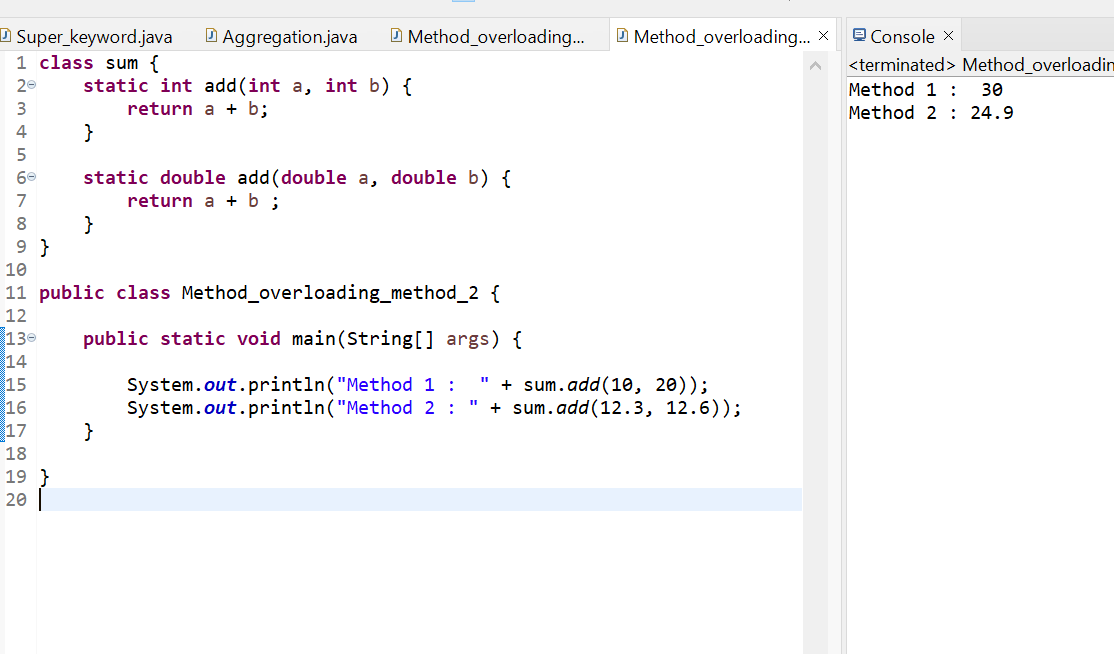
### 1) Method Overloading: changing no. of arguments

* In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.
* In this example, we are creating [static methods](https://www.javatpoint.com/static-keyword-in-java) so that we don't need to create instance for calling methods.



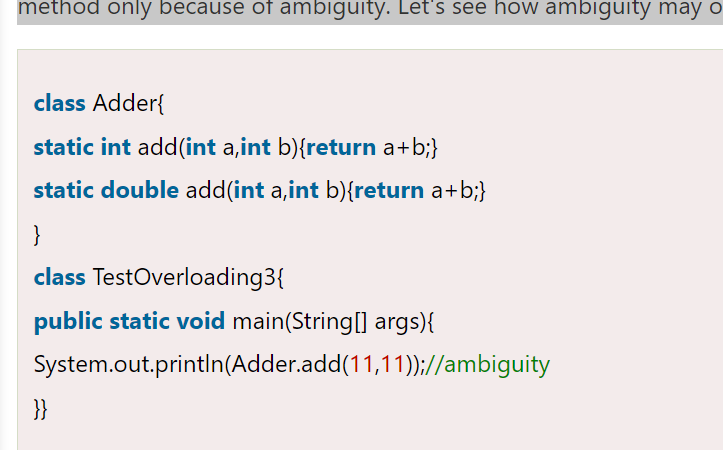
### 2) Method Overloading: changing data type of arguments

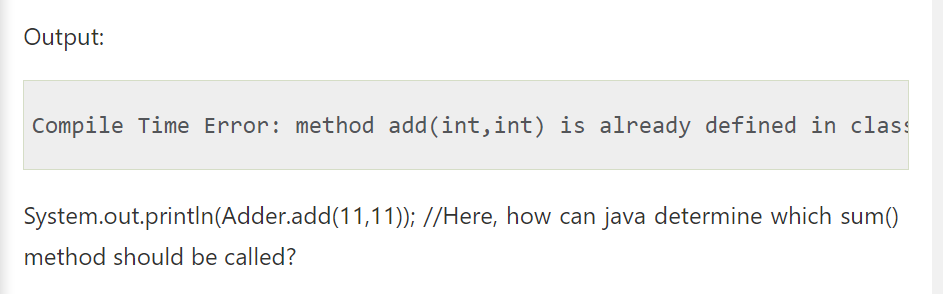
* In this example, we have created two methods that differs in [data type](https://www.javatpoint.com/java-data-types). The first add method receives two integer arguments and second add method receives two double arguments.



### Q) Why Method Overloading is not possible by changing the return type of method only?

In java, method overloading is not possible by changing the return type of the method only because of ambiguity. Let's see how ambiguity may occur:

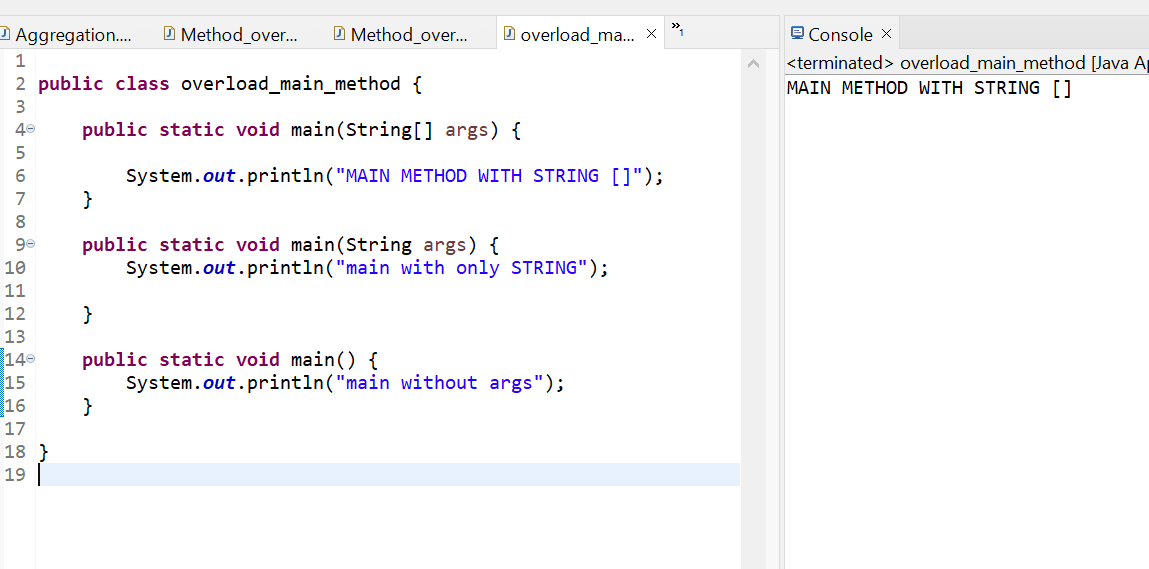




**NOTE : Compile Time Error is better than Run Time Error. So, java compiler renders compiler time error if you declare the same method having same parameters.**

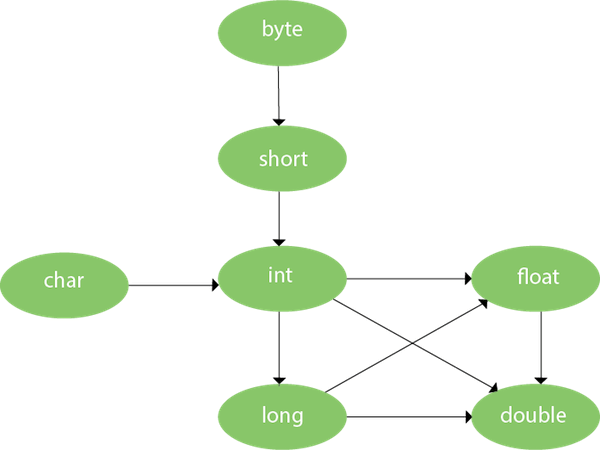
### Can we overload java main() method?

Yes, by method overloading. You can have any number of main methods in a class by method overloading. But [JVM](https://www.javatpoint.com/jvm-java-virtual-machine) calls main() method which receives string array as arguments only. Let's see the simple example:

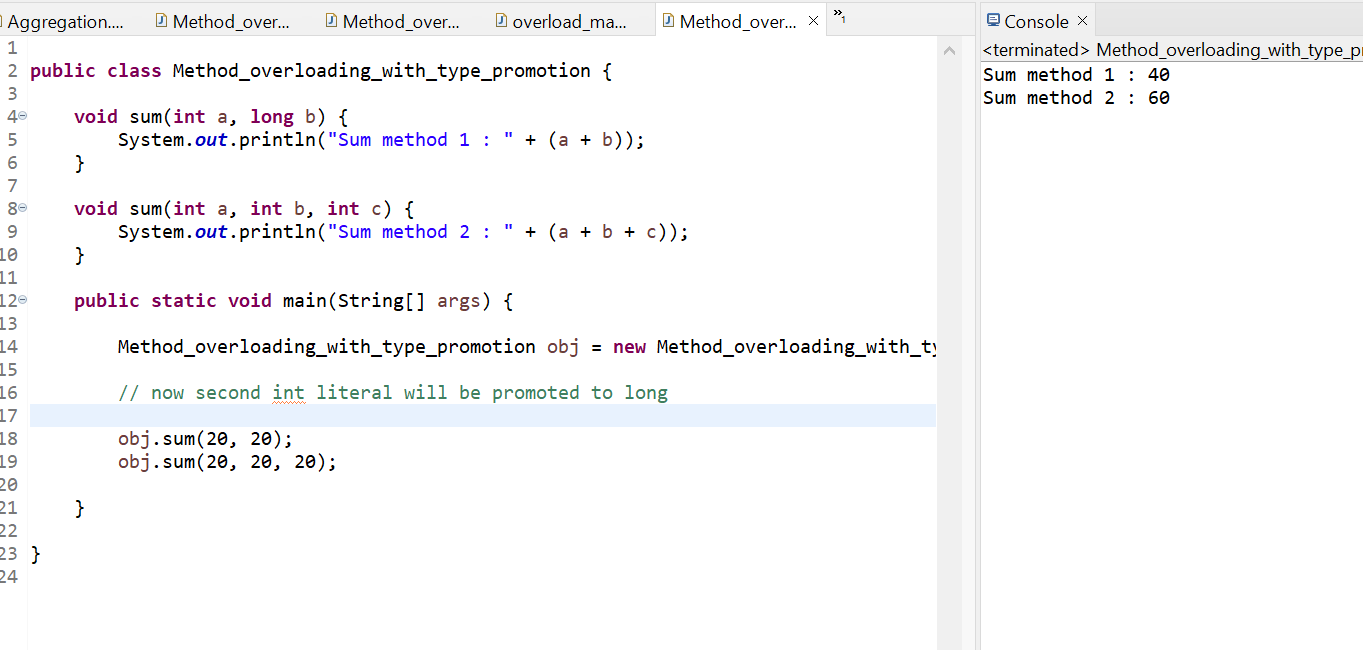


## Method Overloading and Type Promotion

One type is promoted to another implicitly if no matching datatype is found. Let's understand the concept by the figure given below:



As displayed in the above diagram, byte can be promoted to short, int, long, float or double. The short datatype can be promoted to int, long, float or double. The char datatype can be promoted to int,long,float or double and so on.



**NOTE :** One type is not de-promoted implicitly for example double cannot be depromoted to any type implicitly.

# Method Overriding in Java

* If subclass (child class) 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

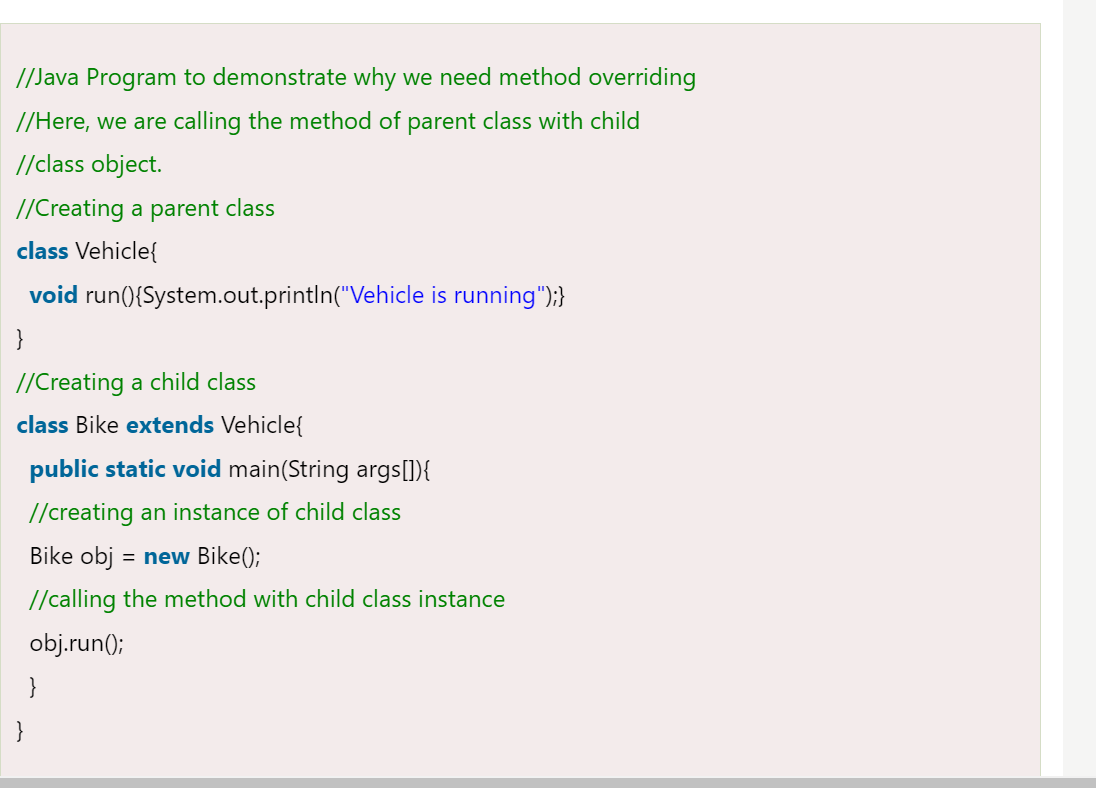
### Usage of Java Method Overriding

* Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
* Method overriding is used for runtime polymorphism

#### Rules for Java Method Overriding

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

### Understanding the problem without method overriding



OUTPUT :

Vehicle is running

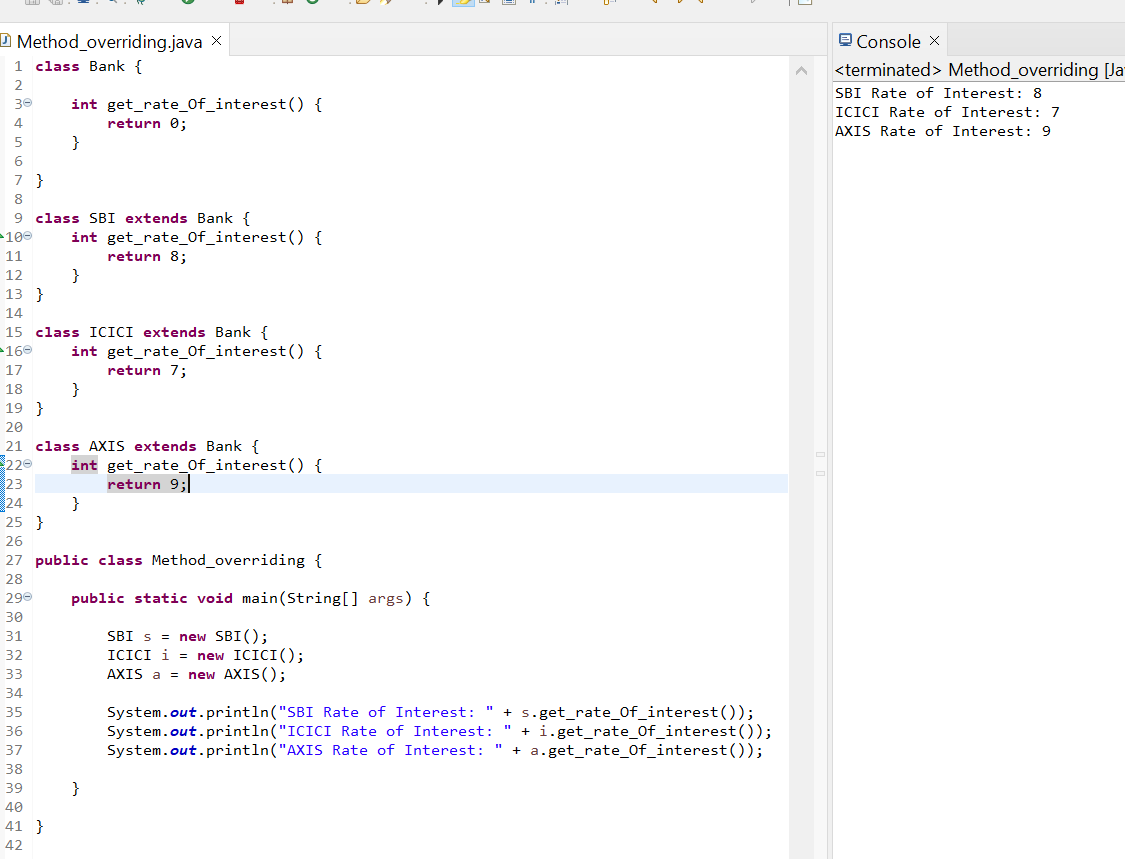
Problem is that I have to provide a specific implementation of run() method in subclass that is why we use method overriding.

### A real example of Java Method Overriding

Consider a scenario where Bank is a class that provides functionality to get the rate of interest. However, the rate of interest varies according to banks. For example, SBI, ICICI and AXIS banks could provide 8%, 7%, and 9% rate of interest.

# **Java method overriding example of bank**

**Java method overriding is mostly used in Runtime Polymorphism**



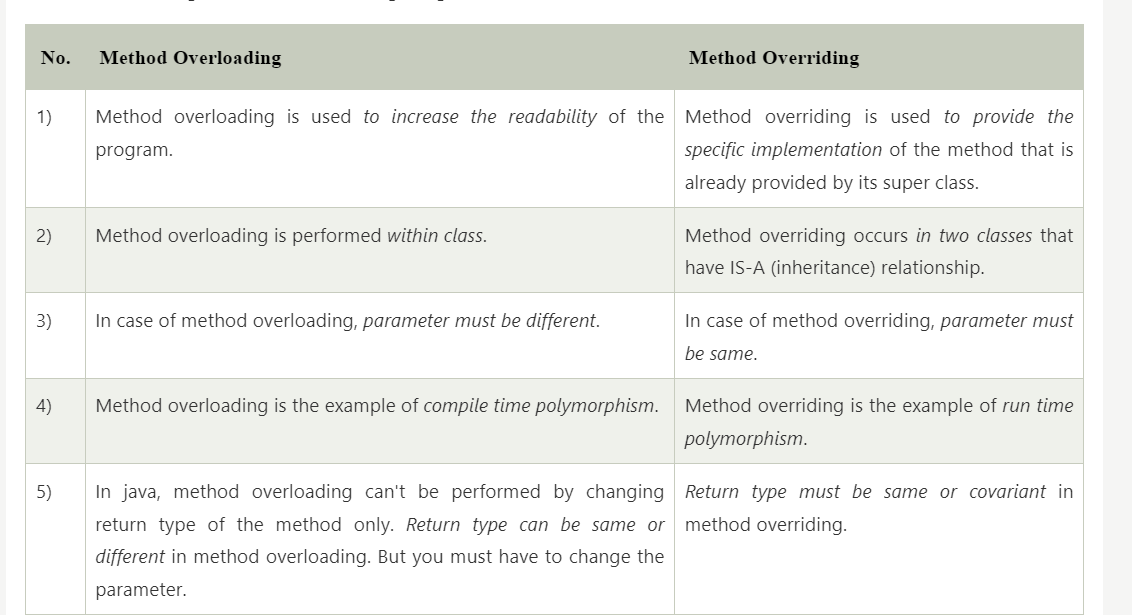
### Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

### Can we override java main method?

No, because the main is a static method.

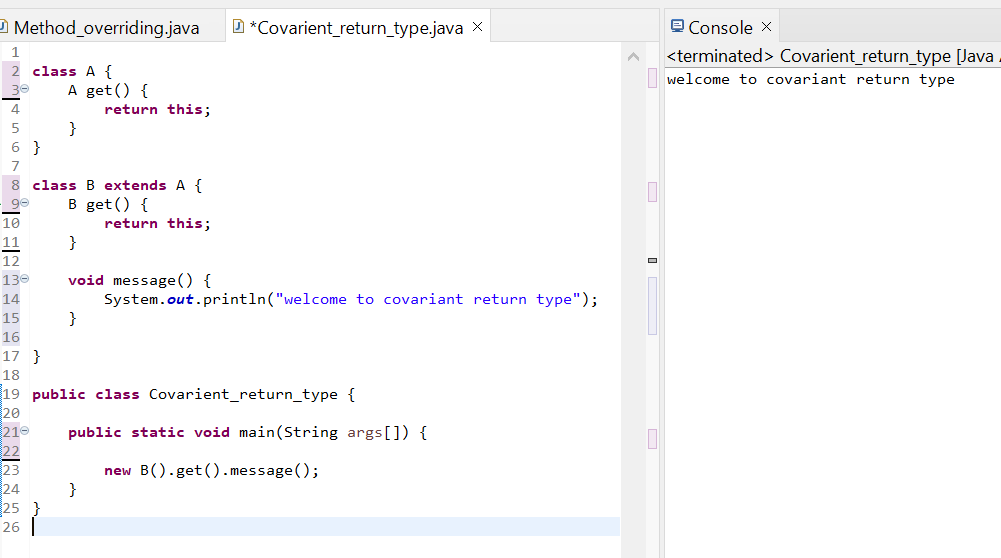
## Difference between method Overloading and Method Overriding in java





# Covariant Return Type

* The covariant return type specifies that the return type may vary in the same direction as the subclass.
* Before Java5, it was not possible to override any method by changing the return type. But now, since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive but it changes its return type to subclass type. Let's take a simple example:



As you can see in the above example, the return type of the get() method of A class is A but the return type of the get() method of B class is B. Both methods have different return type but it is method overriding. This is known as covariant return type.

## Advantages of Covariant Return Type

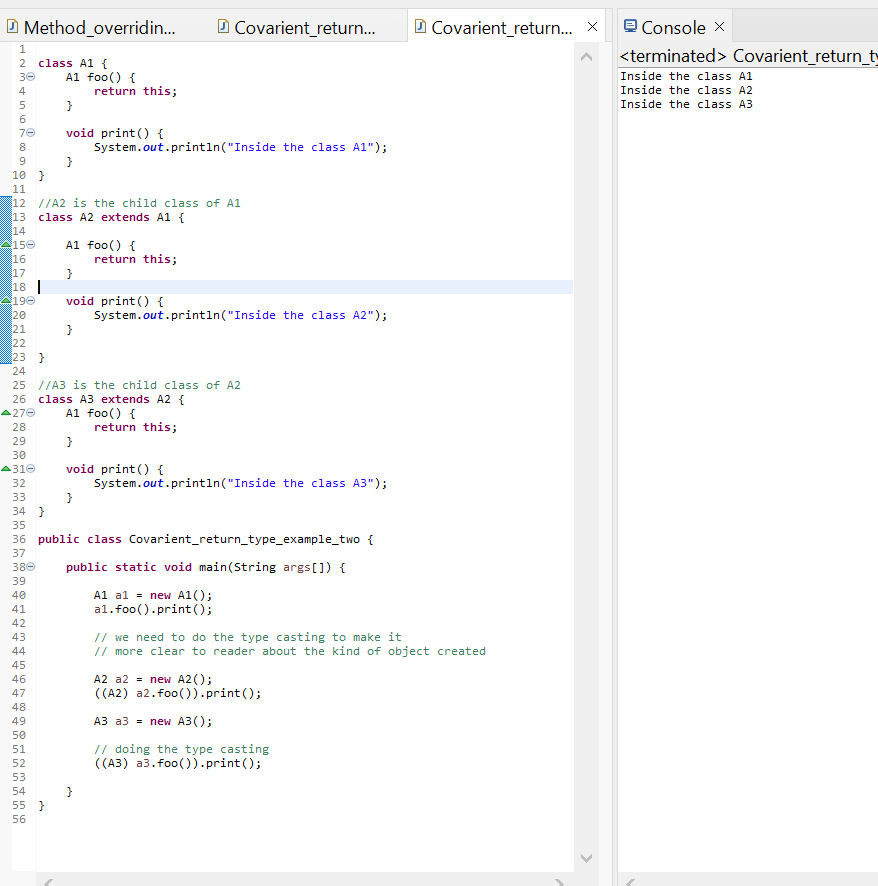
Following are the advantages of the covariant return type.

1) Covariant return type assists to stay away from the confusing type casts in the class hierarchy and makes the code more usable, readable, and maintainable.

2) In the method overriding, the covariant return type provides the liberty to have more to the point return types.

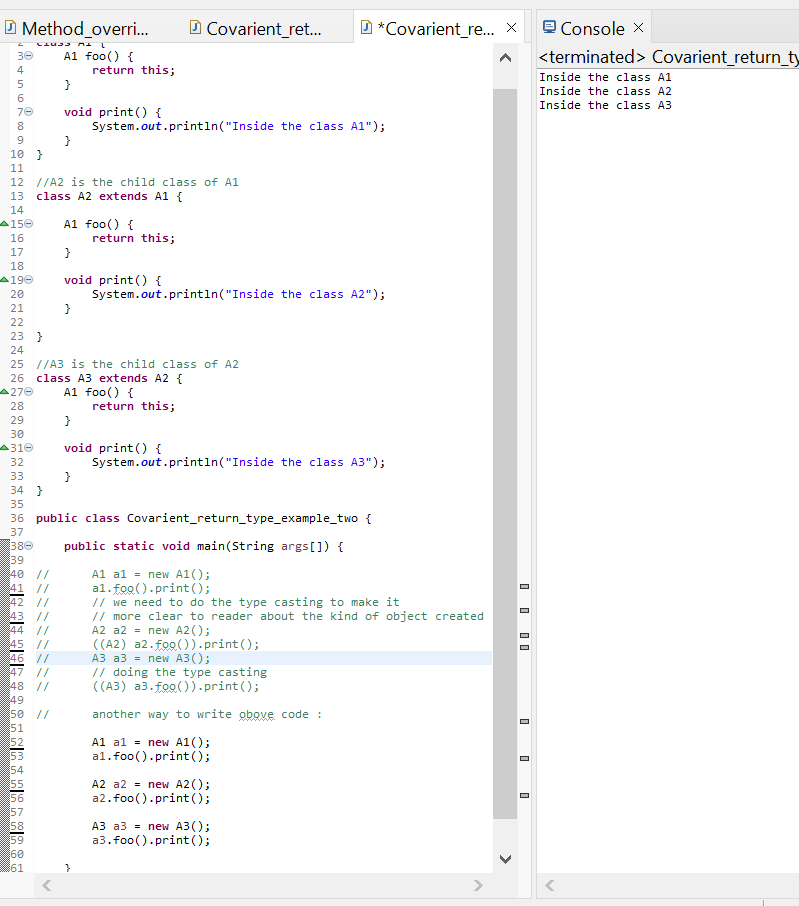
3) Covariant return type helps in preventing the run-time *ClassCastExceptions* on returns.

Let's take an example to understand the advantages of the covariant return type.



**Explanation:** In the above program, class A3 inherits class A2, and class A2 inherits class A1. Thus, A1 is the parent of classes A2 and A3. Hence, any object of classes A2 and A3 is also of type A1. As the return type of the method *foo()* is the same in every class, we do not know the exact type of object the method is actually returning. We can only deduce that returned object will be of type A1, which is the most generic class. We can not say for sure that returned object will be of A2 or A3. It is where we need to do the typecasting to find out the specific type of object returned from the method *foo()*. It not only makes the code verbose; it also requires precision from the programmer to ensure that typecasting is done properly; otherwise, there are fair chances of getting the *ClassCastException*. To exacerbate it, think of a situation where the hierarchical structure goes down to 10 - 15 classes or even more, and in each class, the method *foo()* has the same return type. That is enough to give a nightmare to the reader and writer of the code.

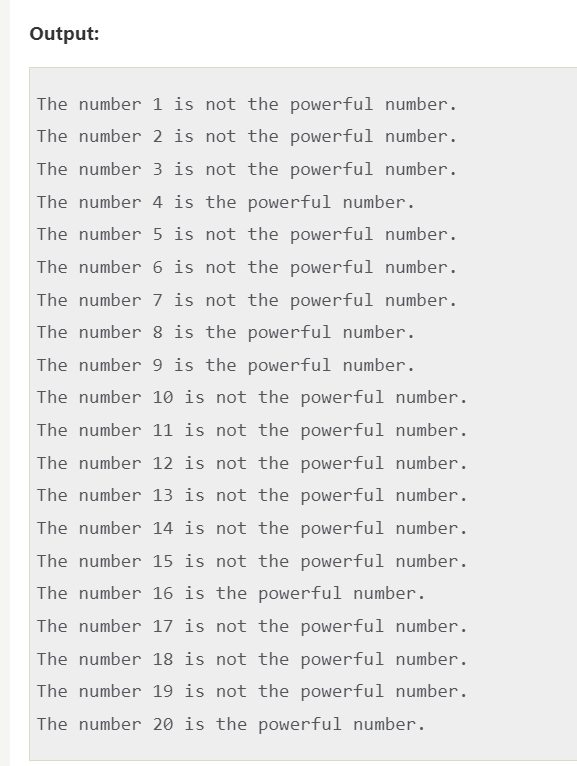
The better way to write the above is:



**Explanation:** In the above program, no typecasting is needed as the return type is specific. Hence, there is no confusion about knowing the type of object getting returned from the method *foo()*. Also, even if we write the code for the 10 - 15 classes, there would be no confusion regarding the return types of the methods. All this is possible because of the covariant return type.

### How is Covariant return types implemented?

Java doesn't allow the return type-based overloading, but JVM always allows return type-based overloading. JVM uses the full signature of a method for lookup/resolution. Full signature means it includes return type in addition to argument types. i.e., a class can have two or more methods differing only by return type. javac uses this fact to implement covariant return types.



**Explanation:** For every number from 1 to 20, the method *isPowerfulNo()* is invoked with the help of for-loop. For every number, a vector *primeFactors* is created for storing its prime divisors. Then, we check whether square of every number present in the vector *primeFactors* divides the number or not. If all square of all the number present in the vector *primeFactors* divides the number completely, the number is a powerful number; otherwise, not.

# Super Keyword in Java

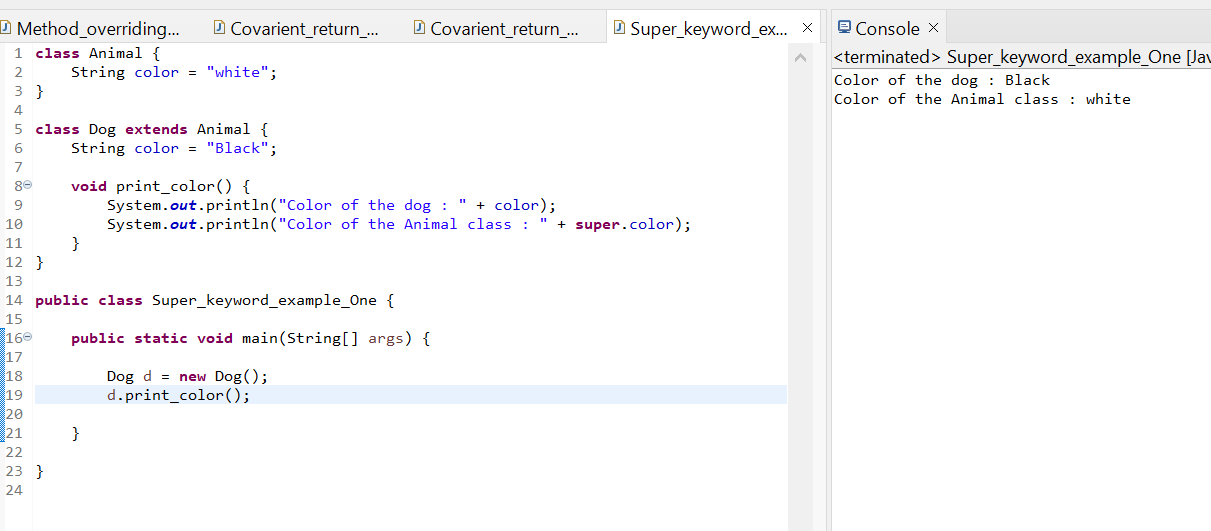
* The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.
* Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

## Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

## 1) super is used to refer immediate parent class instance variable

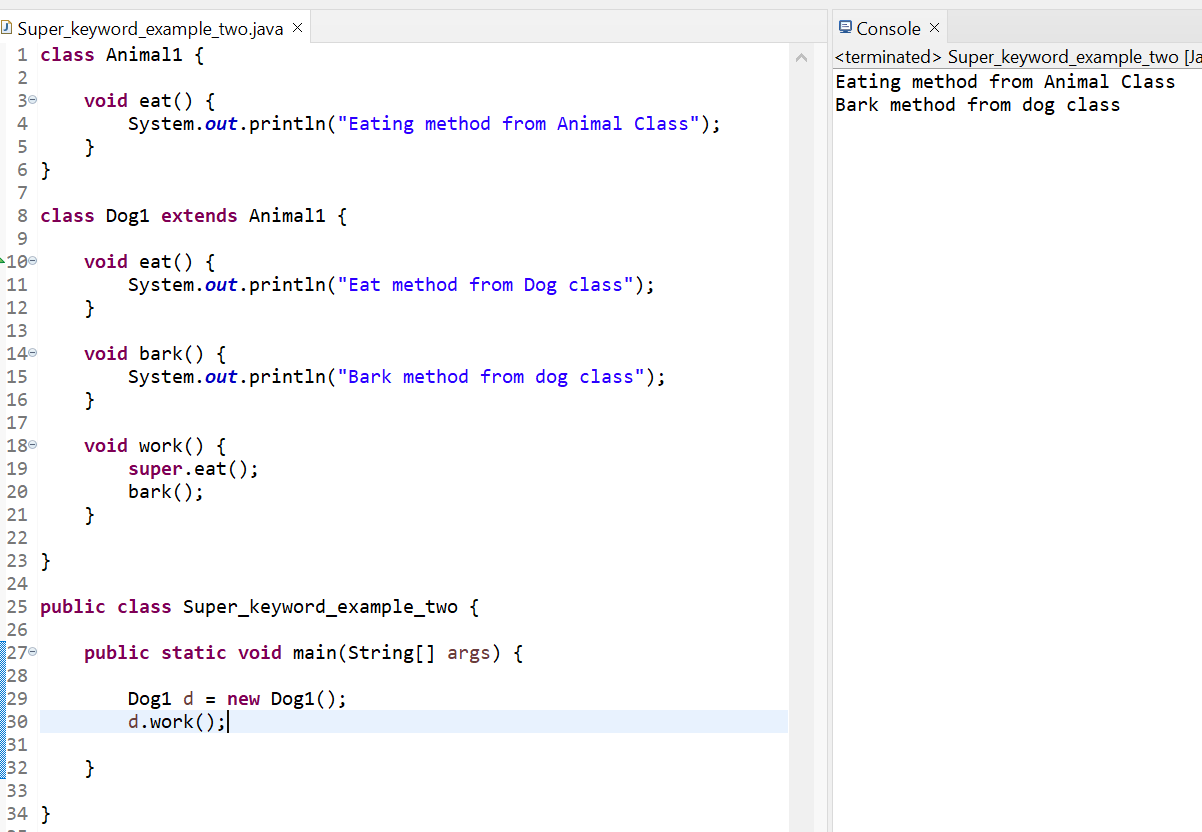
We can use super keyword to access the data member or field of parent class. It is used if parent class and child class have same fields.



In the above example, Animal and Dog both classes have a common property color. If we print color property, it will print the color of current class by default. To access the parent property, we need to use super keyword

## 2) super can be used to invoke parent class method

The super keyword can also be used to invoke parent class method. It should be used if subclass contains the same method as parent class. In other words, it is used if method is overridden.

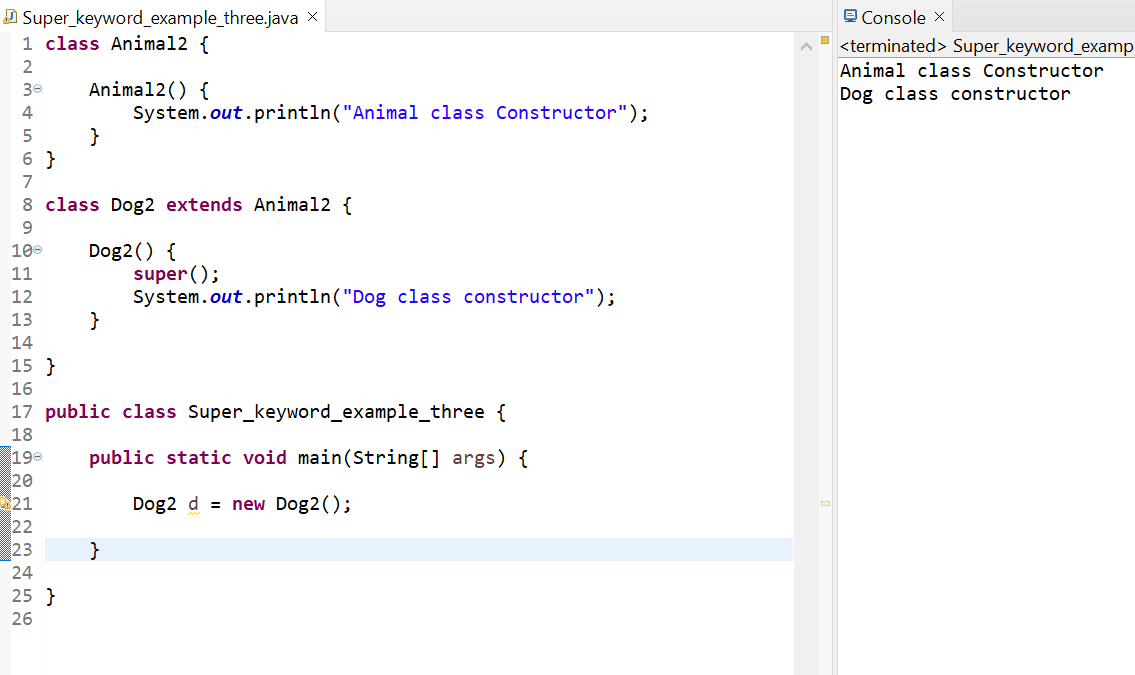


In the above example Animal and Dog both classes have eat() method if we call eat() method from Dog class, it will call the eat() method of Dog class by default because priority is given to local.

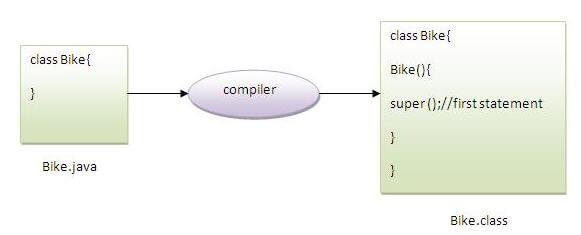
To call the parent class method, we need to use super keyword.

## 3) super is used to invoke parent class constructor.

The super keyword can also be used to invoke the parent class constructor. Let's see a simple example:

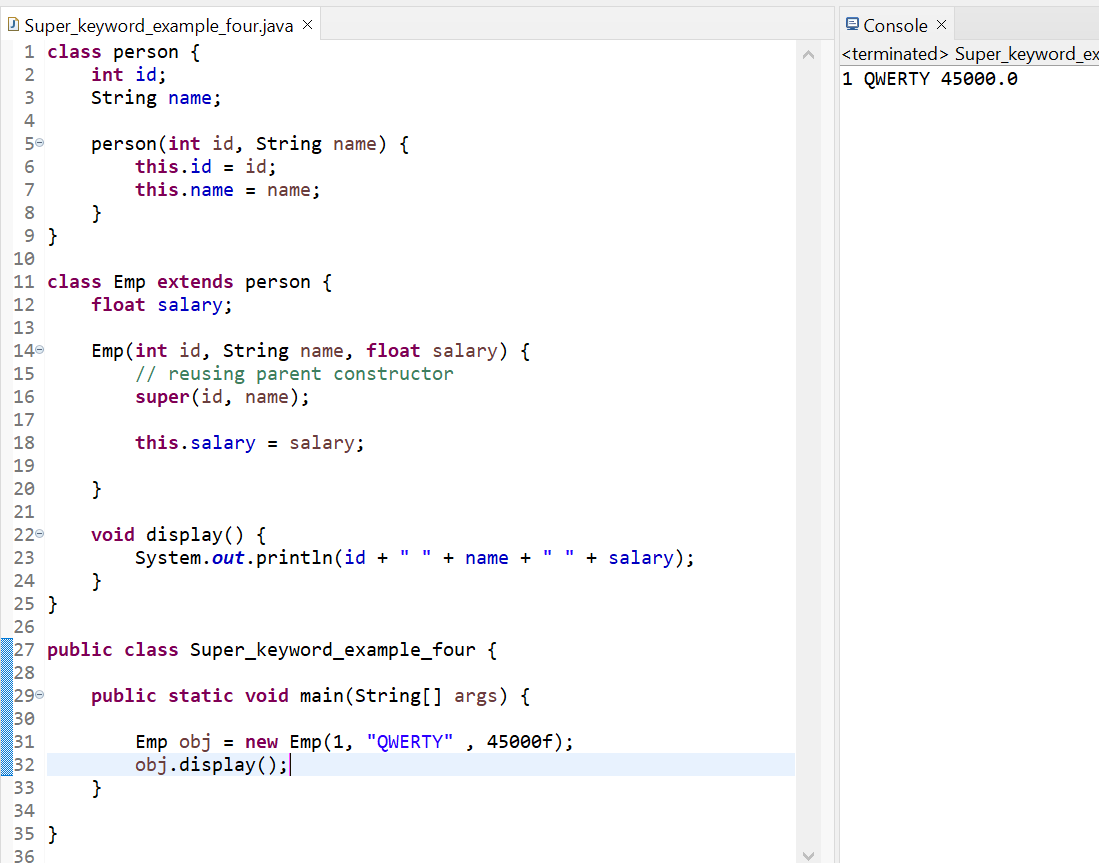


**Note: super() is added in each class constructor automatically by compiler if there is no super() or this().**

****

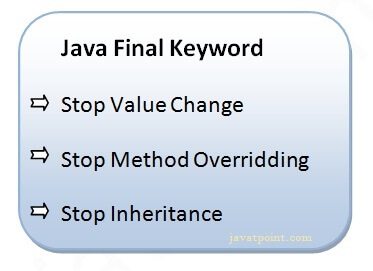
**As we know well that default constructor is provided by compiler automatically if there is no constructor. But, it also adds super() as the first statement.**

## super example: real use

****

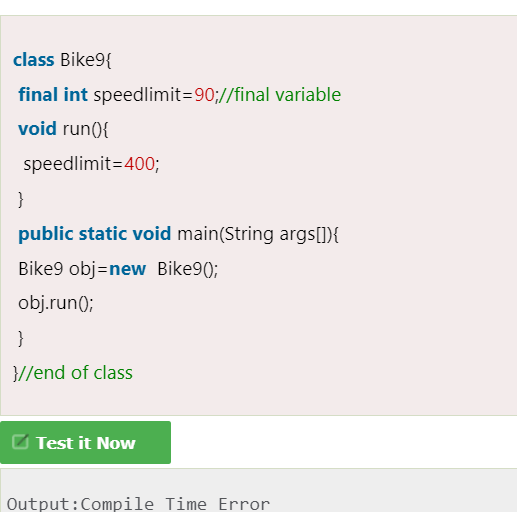
# Final Keyword In Java

* The **final keyword** in java is used to restrict the user. The java final keyword can be used in many context. Final can be:
  + variable
  + method
  + Class
* The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.



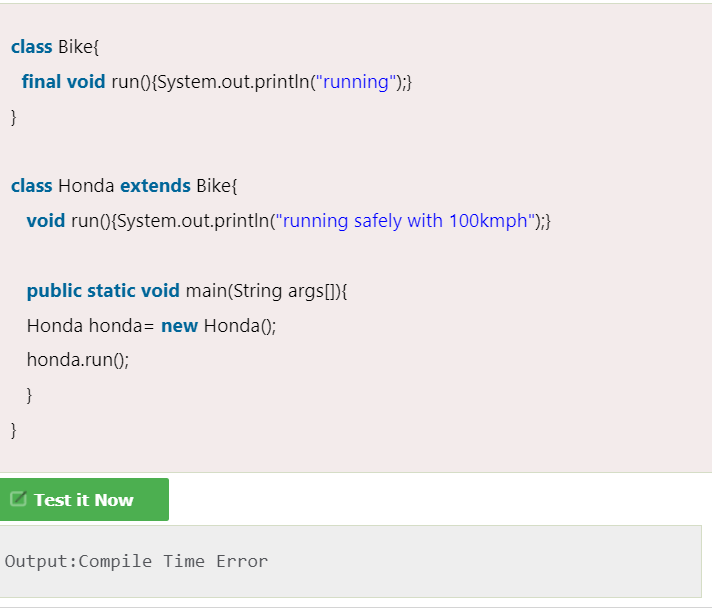
## 1) Java final variable

* If you make any variable as final, you cannot change the value of final variable(It will be constant).
* **Example of final variable**
* There is a final variable speedlimit, we are going to change the value of this variable, but It can't be changed because final variable once assigned a value can never be changed.



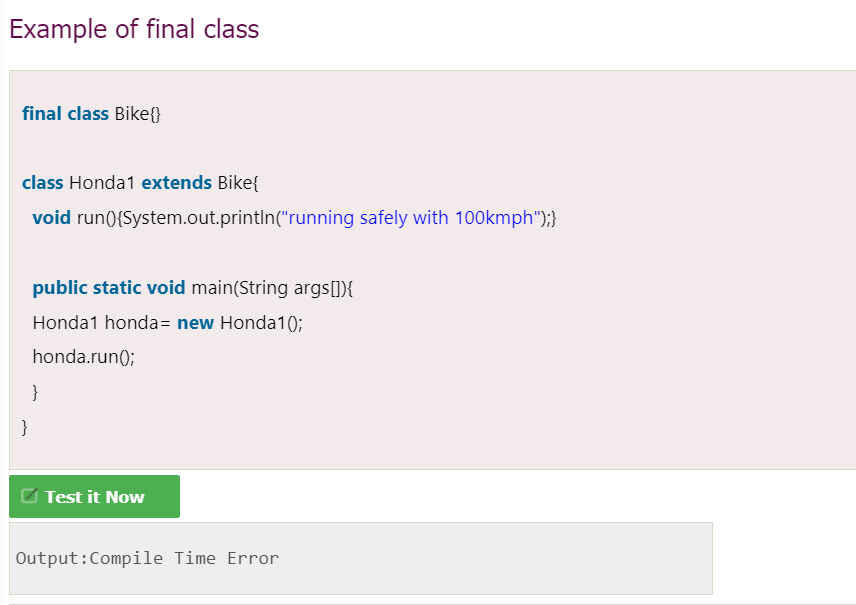
## 2) Java final method

If you make any method as final, you cannot override it.



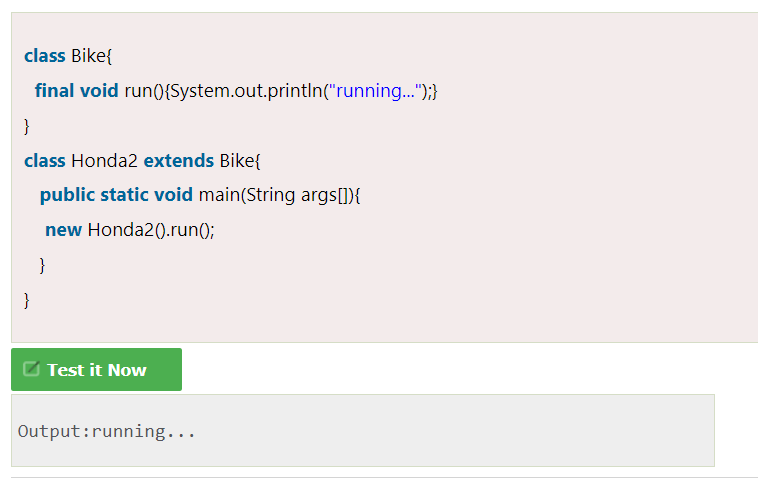
## 3) Java final class

If you make any class as final, you cannot extend it.



### Q) Is final method inherited?

Ans) Yes, final method is inherited but you cannot override it. For Example:

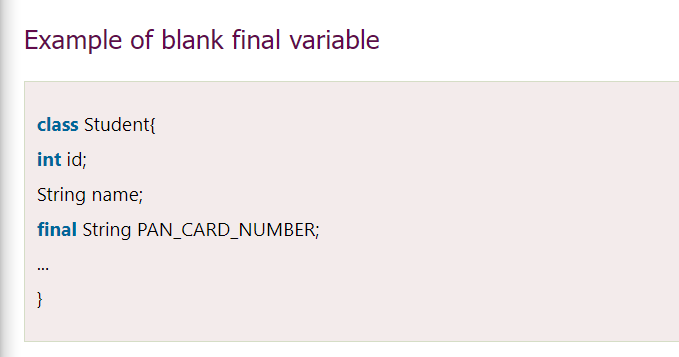


### Q) What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank final variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed, it is useful. For example PAN CARD number of an employee.

It can be initialized only in constructor.



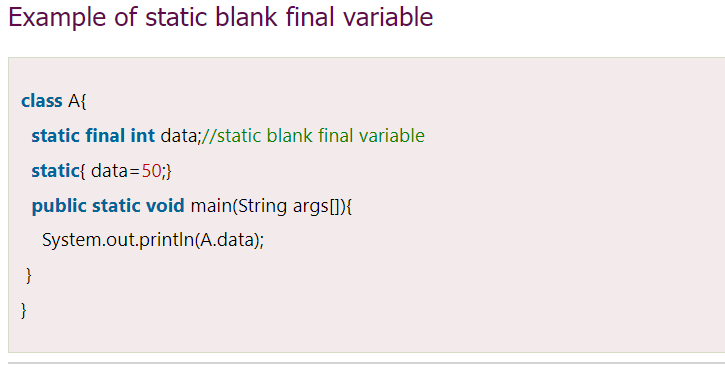
### Que) Can we initialize blank final variable?

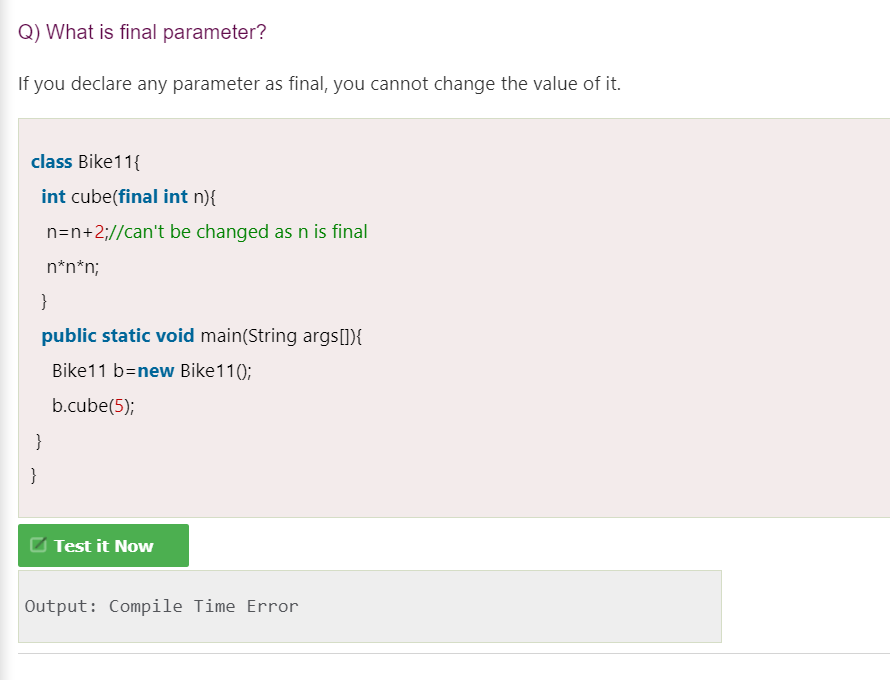
Yes, but only in constructor. For example:



### static blank final variable

A static final variable that is not initialized at the time of declaration is known as static blank final variable. It can be initialized only in static block.





### Q) Can we declare a constructor final?

No, because constructor is never inherited.

# **Java - Polymorphism**

* Polymorphism is the ability of an object to take on many forms.
* Polymorphism is an important feature of [Java OOPs concept](https://www.tutorialspoint.com/java/java_oops_concepts.htm) and it allows us to perform multiple operations by using the single name of any method (interface).
* Any Java object that can pass more than one IS-A test is considered to be polymorphic.
* In Java, all [Java objects](https://www.tutorialspoint.com/java/java_object_classes.htm) are polymorphic since any object will pass the IS-A test for its own type and for the class Object.
* **Polymorphism in Java** is a concept by which we can perform a *single action in different ways*.
* Polymorphism is derived from 2 Greek words: poly and morphs. The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
* **There are two types of polymorphism in Java: compile-time polymorphism and runtime polymorphism.** We can perform polymorphism in java by method overloading and method overriding.
* **If you overload a static method in Java, it is the example of compile time polymorphism.**

###### **Use of Polymorphism in Java**

* The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object.
* It is important to know that the only possible way to access an object is through a [reference variable](https://www.tutorialspoint.com/java/java_variable_types.htm). A reference variable can be of only one type. Once declared, the type of a reference variable cannot be changed.
* The reference variable can be reassigned to other objects provided that it is not declared final. The type of the reference variable would determine the methods that it can invoke on the object.
* A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

## Runtime Polymorphism in Java

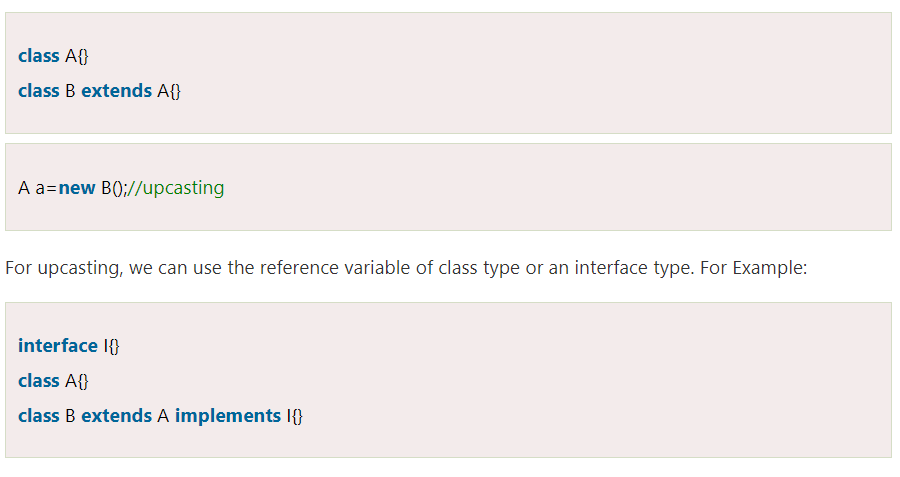
* **Runtime polymorphism** or **Dynamic Method Dispatch** is a process in which a call to an overridden method is resolved at runtime rather than compile-time.
* In this process, an overridden method is called through the reference variable of a superclass.
* The determination of the method to be called is based on the object being referred to by the reference variable.

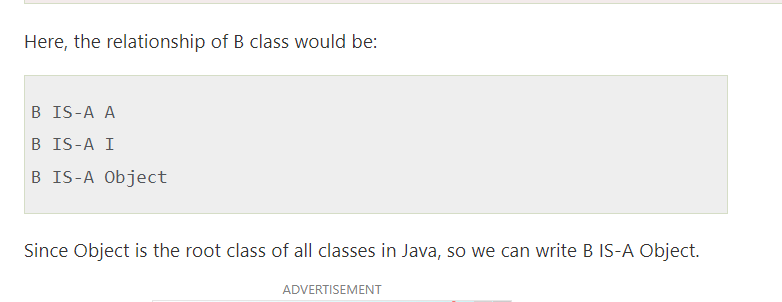
Let's first understand the upcasting before Runtime Polymorphism.

### Upcasting

If the reference variable of Parent class refers to the object of Child class, it is known as upcasting. For example:



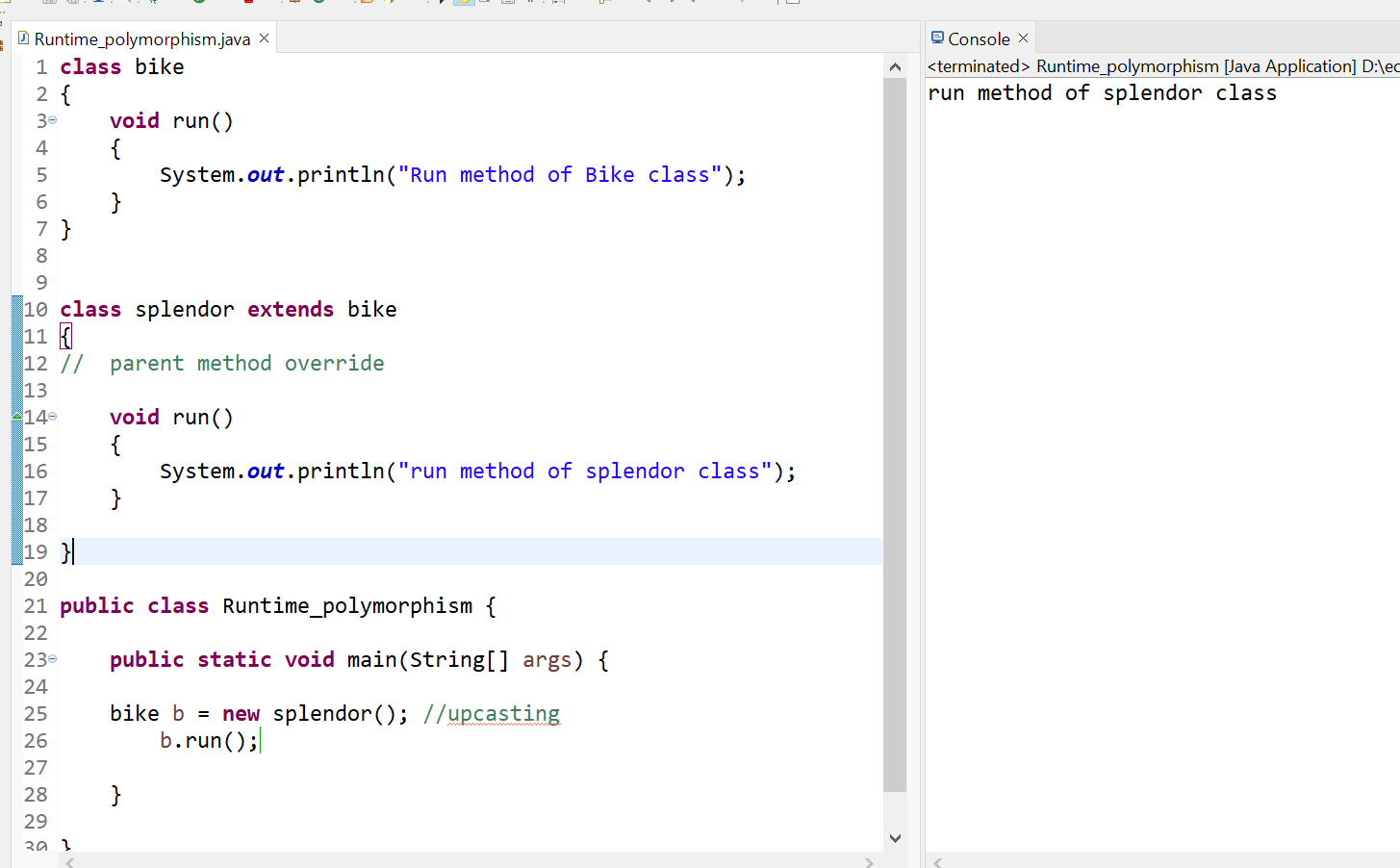




### Example of Java Runtime Polymorphism

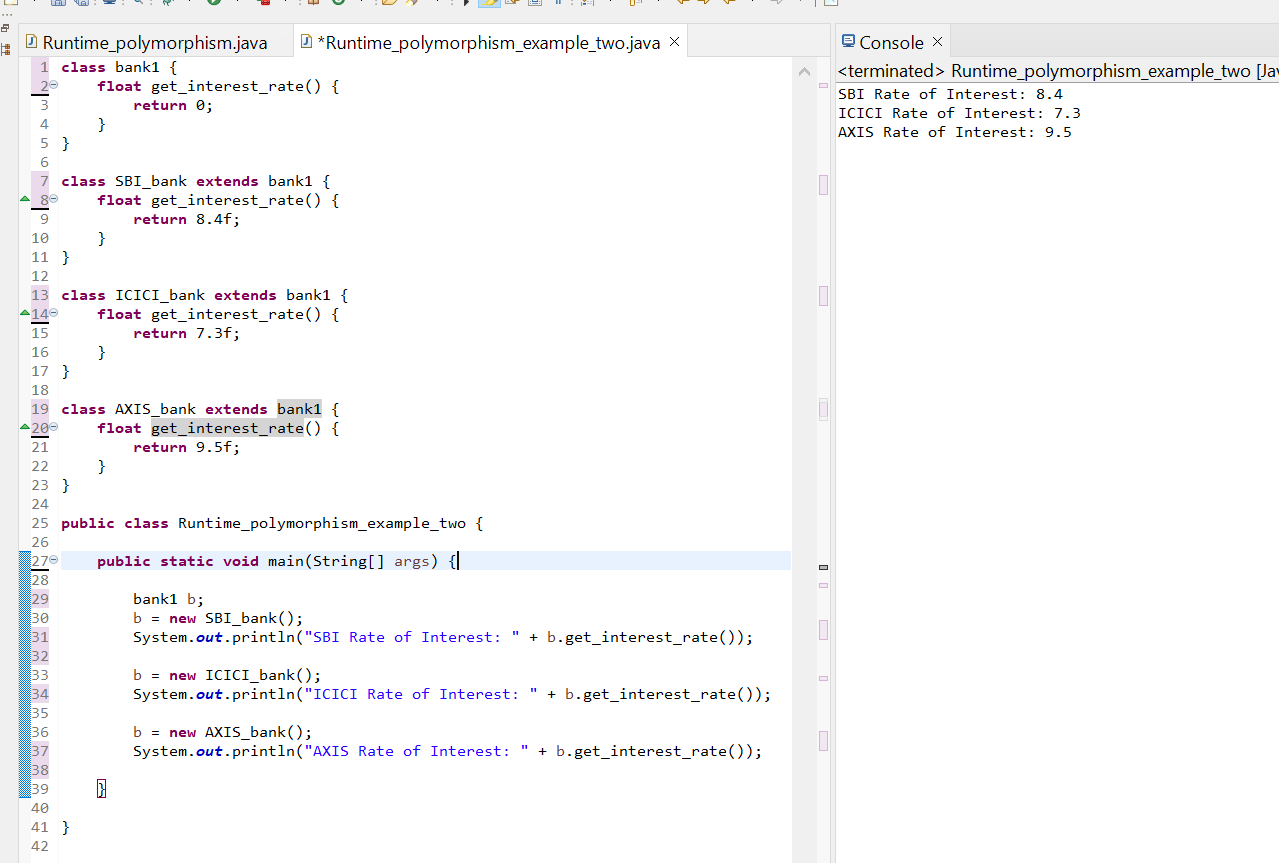
In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run() method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.



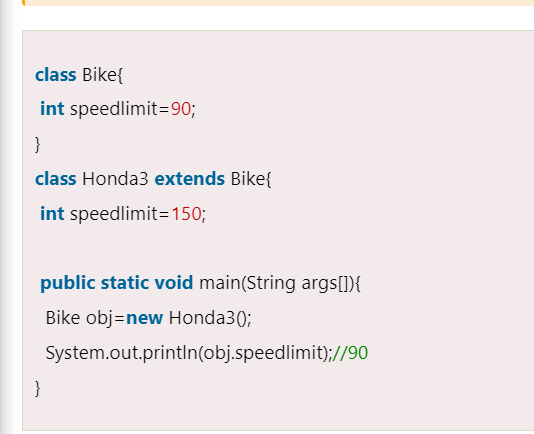
## Java Runtime Polymorphism Example: Bank

Consider a scenario where Bank is a class that provides a method to get the rate of interest. However, the rate of interest may differ according to banks. For example, SBI, ICICI, and AXIS banks are providing 8.4%, 7.3%, and 9.7% rate of interest.



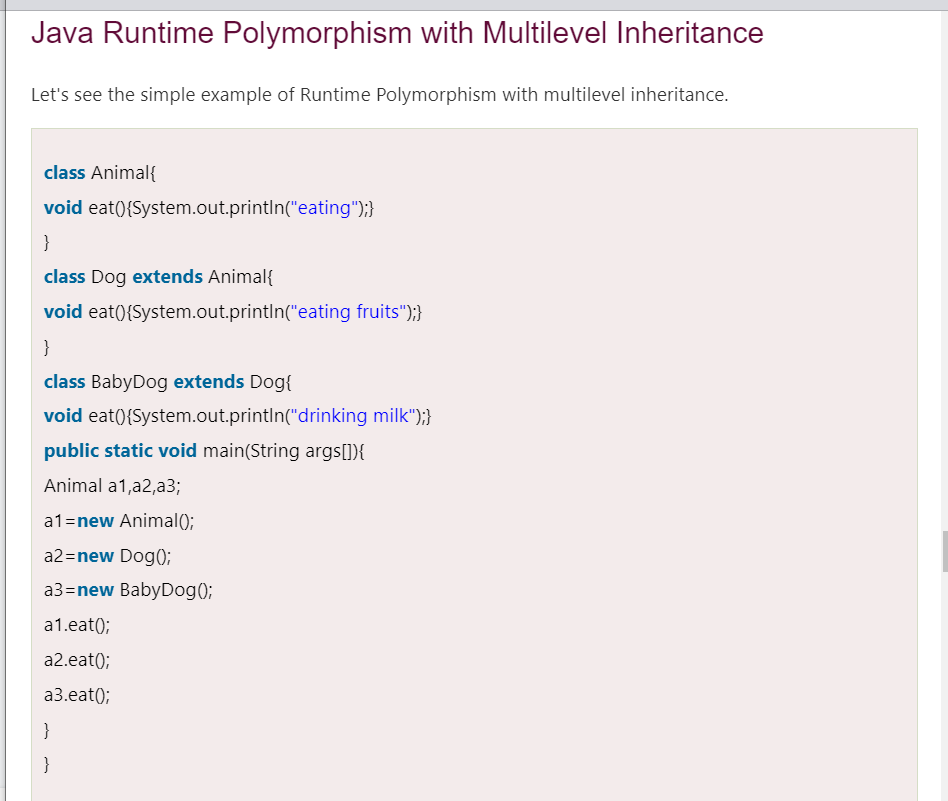
## Java Runtime Polymorphism with Data Member

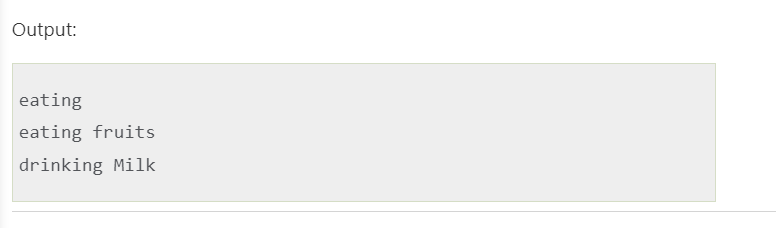
* A method is overridden, not the data members, so runtime polymorphism can't be achieved by data members.
* In the example given below, both the classes have a data member speedlimit. We are accessing the data member by the reference variable of Parent class which refers to the subclass object. Since we are accessing the data member which is not overridden, hence it will access the data member of the Parent class always.
* **NOTE :** Rule: Runtime polymorphism can't be achieved by data members.

****

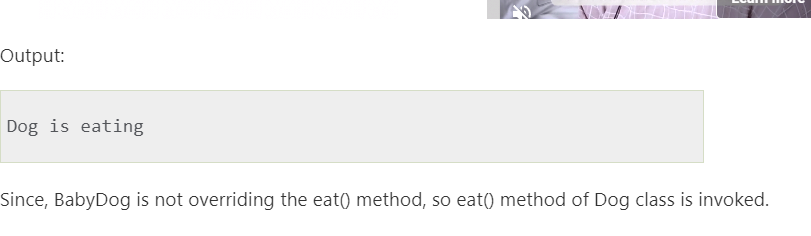
**OUTPUT :**

**90**

****

****

****

****

# Static Binding and Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

1. Static Binding (also known as Early Binding).
2. Dynamic Binding (also known as Late Binding).

### Understanding Type

Let's understand the type of instance.

#### 1) variables have a type

Each variable has a type, it may be primitive and non-primitive.

1. **int** data=30;

Here data variable is a type of int.

#### 2) References have a type

1. **class** Dog{
2. **public** **static** **void** main(String args[]){
3. Dog d1;//Here d1 is a type of Dog
4. }
5. }

#### 3) Objects have a type

| An object is an instance of particular java class,but it is also an instance of its superclass. |
| --- |

1. **class** Animal{}
3. **class** Dog **extends** Animal{
4. **public** **static** **void** main(String args[]){
5. Dog d1=**new** Dog();
6. }
7. }

| Here d1 is an instance of Dog class, but it is also an instance of Animal. static binding When type of the object is determined at compiled time(by the compiler), it is known as static binding.  If there is any private, final or static method in a class, there is static binding.   Dynamic binding When type of the object is determined at run-time, it is known as dynamic binding.    In the above example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal.So compiler doesn't know its type, only its base type.  **Static binding vs Dynamic binding**     Java instanceof  * The **java instanceof operator** is used to test whether the object is an instance of the specified type (class or subclass or interface). * The instanceof in java is also known as type *comparison operator* because it compares the instance with type. It returns either true or false. If we apply the instanceof operator with any variable that has null value, it returns false.     An object of subclass type is also a type of parent class. For example, if Dog extends Animal then object of Dog can be referred by either Dog or Animal class.     Downcasting with java instanceof operator When Subclass type refers to the object of Parent class, it is known as downcasting. If we perform it directly, compiler gives Compilation error. If you perform it by typecasting, ClassCastException is thrown at runtime. But if we use instanceof operator, downcasting is possible.   1. Dog d=**new** Animal();//Compilation error   If we perform downcasting by typecasting, ClassCastException is thrown at runtime.   1. Dog d=(Dog)**new** Animal(); 2. //Compiles successfully but ClassCastException is thrown at runtime |
| --- |