ADVANCE PROGRAMMING Overloading and Overriding in Java

By
Dr. Amal Sufuih Ajrash

Java Inheritance

Inheritance is one of the key features of OOP that allows us to create a new class from an existing class. The new class that is created is known as **subclass** (child or derived class) and the existing class from where the child class is derived is known as **superclass** (parent or base class).

The `extends` keyword is used to perform inheritance in Java. For example,

```
class Animal {   // methods and fields }
// use of extends keyword to perform inheritance
class Dog extends Animal {   // methods and fields of Animal  // methods and fields of Dog}
```

In the above example, the `Dog` class is created by inheriting the methods and fields from the `Animal` class. Here, `Dog` is the subclass and `Animal` is the superclass.

Example 1: Java Inheritance

```
class Animal {
  String name;
  public void eat() {
    System.out.println("I can eat");
  }  }

class Dog extends Animal {
  public void display() {
    System.out.println("My name is " + name);
 } }

class Main {
  public static void main(String[] args) {
    Dog labrador = new Dog();
    labrador.name = "Rohu";
    labrador.display();
    labrador.eat();
  }
}
```

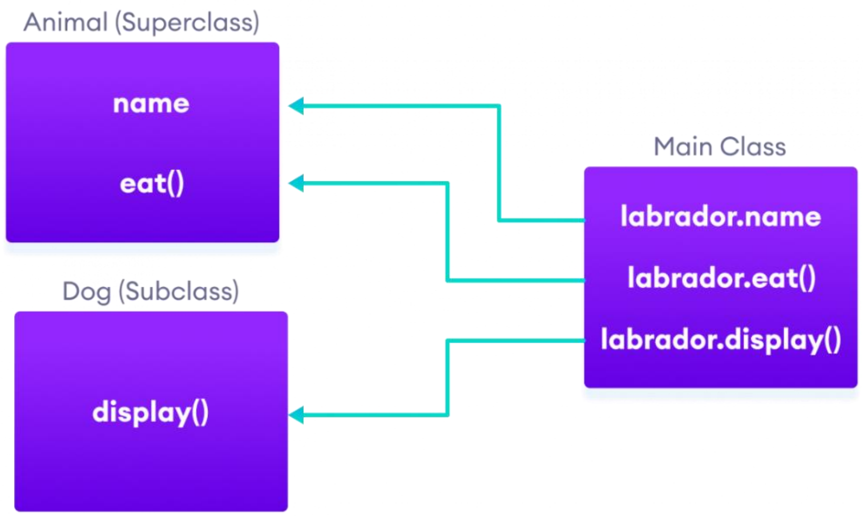
Output

```
My name is Rohu
I can eat
```

In the above example, we have derived a subclass Dog from superclass Animal. Notice the statements,

```
labrador.name = "Rohu";
labrador.eat();
```

Here, labrador is an object of Dog. However, name and eat() are the members of the Animal class. Since Dog inherits the field and method from Animal, we are able to access the field and method using the object of the Dog.



Java Inheritance Implementation

is-a relationship: In Java, inheritance is an **is-a** relationship. That is, we use inheritance only if there exists an is-a relationship between two classes. For example,

- **Car** is a **Vehicle**
- **Orange** is a **Fruit**
- **Surgeon** is a **Doctor**
- **Dog** is an **Animal**

Here, **Car** can inherit from **Vehicle**, **Orange** can inherit from **Fruit**, and so on.

Method Overloading in Java

Method Overloading is a feature that allows a class to have more than one method having the same name, if their argument lists are different. It is similar to constructor overloading in Java, that allows a class to have more than one constructor having different argument lists.

let’s get back to the point, when I say argument list it means the parameters that a method has: For example the argument list of a method add(int a, int b) having two parameters is different from the argument list of the method add(int a, int b, int c) having three parameters.

مفهوم الـ Overloading

Overloading: تعني تعريف أكثر من دالة، لهم نفس الإسم ، مختلفون في عدد أو نوع البارامترات.

الفكرة من الـ **Overloading**، هي تجهيز عدة دوال لهم نفس الإسم، هذه الدوال تكون متشابهة من حيث الوظيفة، مختلفة قليلاً في الأداء.فعلياً، تكون كل دالة تحتوي على ميزات إضافية عن الدالة التي أنشأت قبلها.

Three ways to overload a method

In order to overload a method, the argument lists of the methods must differ in either of these:
1. Number of parameters.
For example: This is a valid case of overloading

```
add(int, int)
add(int, int, int)
```

2. Data type of parameters.
For example:

```
add(int, int)
add(int, float)
```

3. Sequence of Data type of parameters.
For example:

```
add(int, float)
add(float, int)
```

Invalid case of method overloading

When I say argument list, I am not talking about return type of the method, for example if two methods have same name, same parameters and have different return type, then this is not a valid method overloading example. This will throw compilation error.

```
int add(int, int)
float add(int, int)
```

Method Overloading example

Example 1: Overloading – Different Number of parameters in argument list

```
class DisplayOverloading
{
    public void disp(char c)
    {
         System.out.println(c);
    }
    public void disp(char c, int num)
    {
         System.out.println(c + " "+num);
    }
}
public class Main {
   public static void main(String args[])
   {
       DisplayOverloading obj = new DisplayOverloading();
       obj.disp('a');
       obj.disp('a',10);
   }
}
```

Output:

```
a
a 10
```

In the above example – method disp() is overloaded based on the number of parameters – We have two methods with the name disp but the parameters they have are different. Both are having different number of parameters.

Example 2: Overloading – Difference in data type of parameters

```
class DisplayOverloading2
{
   public void disp(char c)
   {
      System.out.println(c);
   }
   public void disp(int c)
   {
     System.out.println(c );
   }
}

public class Main {
   public static void main(String args[])
   {
      DisplayOverloading2 obj = new DisplayOverloading2();
      obj.disp('a');
      obj.disp(5);
   }
}
```

Output:

```
a
5
```

Example3: Overloading – Sequence of data type of arguments

Here method disp() is overloaded based on sequence of data type of parameters – Both the methods have different sequence of data type in argument list. First method is having argument list as (char, int) and second is having (int, char). Since the sequence is different, the method can be overloaded without any issues.

```
class DisplayOverloading3
{
  public void disp(char c, int num)
  {
     System.out.println("I’m the first definition of method disp");
  }
  public void disp(int num, char c)
```

```
   {
      System.out.println("I’m the second definition of method disp" );
   }
}
public class Main {
{
  public static void main(String args[])
   {
      DisplayOverloading3 obj = new DisplayOverloading3();
      obj.disp('x', 51 );
      obj.disp(52, 'y');
   }
}
```

Output:

```
I’m the first definition of method disp
I’m the second definition of method disp
```

Method Overloading and Type Promotion

When a data type of smaller size is promoted to the data type of bigger size than this is called type promotion, for example: byte data type can be promoted to short, a short data type can be promoted to int, long, double etc. **What it has to do with method overloading?** Well, it is very important to understand type promotion else you will think that the program will throw compilation error but in fact that program will run fine because of type promotion.

```
class DisplayOverloading4

{

   public void disp(char c)

   { System.out.println(c);    }

   public void disp(int c)

   { System.out.println(c ); }

}

public class Main {
   public static void main(String args[])
  {
      DisplayOverloading4 obj = new DisplayOverloading4();
      obj.disp('a');
      obj.disp(1.2);  }}
```

Output:

```
Main.java:18: error: no suitable method found for disp(double)

          obj.disp(1.2);
```

also, if the disp(double c) , it can call using the object   obj.disp(10)

```
class Demo{
   void disp(int a, double b){
          System.out.println("Method A");
   }
   void disp(int a, double b, double c){
          System.out.println("Method B");
   }
   void disp(int a, float b){
          System.out.println("Method C");
   }
   public static void main(String args[]){
          Demo obj = new Demo();
          /* This time promotion won't happen as there is
           * a method with arg list as (int, float)
           */
          obj.disp(100, 20.67f);
   }
}
```

Output: Method C

As you see that this time type promotion didn’t happen because there was a method with matching argument type.

Type Promotion table:

The data type on the left side can be promoted to the any of the data type present in the right side of it.

byte → short → int → long
short → int → long
int → long → float → double
float → double
long → float → double

Let's see few Valid/invalid cases of method overloading

Case 1:

int mymethod(int a, int b, float c)
int mymethod(int var1, int var2, float var3)
Result: Compile time error. Argument lists are exactly same. Both methods are having same number, data types and same sequence of data types.

Case 2:

int mymethod(int a, int b)
int mymethod(float var1, float var2)
Result: Perfectly fine. Valid case of overloading. Here data types of arguments are different.

Case 3:

int mymethod(int a, int b)
int mymethod(int num)
Result: Perfectly fine. Valid case of overloading. Here number of arguments are different.

Case 4:

float mymethod(int a, float b)
float mymethod(float var1, int var2)
Result: Perfectly fine. Valid case of overloading. Sequence of the data types of parameters are different, first method is having (int, float) and second is having (float, int).

Case 5:

int mymethod(int a, int b)

```
float mymethod(int var1, int var2)
```

Result: Compile time error. Argument lists are exactly same. Even though return type of methods are different, it is not a valid case. Since return type of method doesn’t matter while overloading a method.

Question 1 – return type, method name and argument list same.

```
class Demo1
{
  public int myMethod(int num1, int num2)
  {
     System.out.println("First myMethod of class Demo");
     return num1+num2;
  }
  public int myMethod(int var1, int var2)
  {
     System.out.println("Second myMethod of class Demo");
     return var1-var2;
  }
}
public class Main
{
  public static void main(String args[])
  {
     Demo obj1= new Demo();
     obj1.myMethod(10,10);
     obj1.myMethod(20,12);
  }
}
```

Answer:
It will throw a compilation error: More than one method with same name and argument list cannot be defined in a same class.

Question 2 – return type is different. Method name & argument list same.

```
class Demo2
```

```
{
  public double myMethod(int num1, int num2)
  {
    System.out.println("First myMethod of class Demo");
    return num1+num2;
  }
  public int myMethod(int var1, int var2)
  {
    System.out.println("Second myMethod of class Demo");
    return var1-var2;
  }
}
public class Main
{
  public static void main(String args[])
  {
    Demo2 obj2= new Demo2();
    obj2.myMethod(10,10);
    obj2.myMethod(20,12);
  }
}
```

Answer:
It will throw a compilation error: More than one method with same name and argument list cannot be given in a class even though their return type is different. **Method return type doesn't matter in case of overloading.**

Method Overriding in Java

Declaring a method in **sub class** which is already present in **parent class** is known as method overriding. Overriding is done so that a child class can give its own implementation to a method which is already provided by the parent class. In this case the method in parent class is called overridden method and the method in child class is called overriding method.

مفهوم الـ **Overriding**
Override : تعني تعريف الدالة التي ورثها الـ Subclass من الـ Superclass من جديد، هذه الدالة الجديدة تكون مشابهة للدالة الموروثة من حيث الشكل فقط، أي لها نفس الإسم و النوع و عدد البارامترات، لكن محتواها مختلف.
الهدف الحقيقي من الـ **Overriding** هو إتاحة الفرصة للـ Subclass ليعرف الدوال حسب حاجته.

Method Overriding Example

Lets take a simple example to understand this. We have two classes: A child class Boy and a parent class Human. The Boy class extends Human class. Both the classes have a common method void eat(). Boy class is giving its own implementation to the eat() method or in other words it is overriding the eat() method. The purpose of Method Overriding is clear here. Child class wants to give its own implementation so that when it calls this method, it prints Boy is eating instead of Human is eating.

```
class Human{
  //Overridden method
  public void eat()
  { System.out.println ("Human is eating"); }
}
class Boy extends Human{
  //Overriding method
  public void eat()
   { System.out.println("Boy is eating");  }
  public static void main( String args[]) {
    Boy obj = new Boy();
    //This will call the child class version of eat()
    obj.eat();
  }
}
```

Output: Boy is eating

Advantage of method overriding

The main advantage of method overriding is that the class can give its own specific implementation to a inherited method **without even modifying the parent class code**. This is helpful when a class has several child classes, so if a child class needs to use the parent class method, it can use it and the other classes that want to have different implementation can use overriding feature to make changes without touching the parent class code.

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 (will be discussed in later tutorials).

Method Overriding and Dynamic Method Dispatch

Method Overriding is an example of runtime polymorphism. When a parent class reference points to the child class object then the call to the overridden method is determined at runtime, because during method call which method (parent class or child class) is to be executed is determined by the type of object. This process in which call to the overridden method is resolved at runtime is known as dynamic method dispatch. Lets see an example to understand this:

```
class ABC{
   //Overridden method
   public void disp()
   {
        System.out.println("disp() method of parent class");
   }
}
class Demo extends ABC{
   //Overriding method
   public void disp(){
        System.out.println("disp() method of Child class");
   }
   public void newMethod(){
        System.out.println("new method of child class");
   }
   public static void main( String args[]) {
        /* When Parent class reference refers to the parent class object then in this case
overridden method (the method of parent class)
        ABC obj = new ABC();
        obj.disp();

        /* When parent class reference refers to the child class object then the overriding
method (method of child class) is called dynamic method dispatch and runtime polymorphism
         */
        ABC obj2 = new Demo();
        obj2.disp();
   }
}
```

Output:

```
disp() method of parent class
disp() method of Child class
```

In the above example the call to the disp() method using second object (obj2) is runtime polymorphism (or dynamic method dispatch).
Note: In dynamic method dispatch the object can call the overriding methods of child class and all the non-overridden methods of base class but it cannot call the methods which are newly declared in the child class. In the above example the object obj2 is calling the disp(). However if you try to call the newMethod() method (which has been newly declared in Demo class) using obj2 then you would give compilation error with the following message:

```
Exception in thread "main" java.lang. Error: Unresolved compilation
problem: The method xyz() is undefined for the type ABC
```

Rules of method overriding in Java

1. Argument list: The argument list of overriding method (method of child class) must match the Overridden method (the method of parent class). The data types of the arguments and their sequence should exactly match.
2. Access Modifier of the overriding method (method of subclass) cannot be more restrictive than the overridden method of parent class. For e.g. if the Access Modifier of parent class method is public then the overriding method (child class method) cannot have private, protected and default Access modifier, because all of these three access modifiers are more restrictive than public.
   For e.g. This is **not allowed** as child class disp method is more restrictive(protected) than base class(public)

```
class MyBaseClass{
  public void disp()
  {
     System.out.println("Parent class method");
  }
}
class MyChildClass extends MyBaseClass{
  protected void disp(){
     System.out.println("Child class method");
  }
 public static void main( String args[]) {
     MyChildClass obj = new MyChildClass();
     obj.disp();
  }
}
```

Output:

Exception in thread "main" java.lang.Error: Unresolved compilation problem: Cannot reduce the visibility of the inherited method from MyBaseClass, However this is perfectly valid scenario as public is less restrictive than protected. Same access modifier is also a valid one.

```
class MyBaseClass{
  protected void disp()
  {
     System.out.println("Parent class method");
  }
}
class MyChildClass extends MyBaseClass{
  public void disp(){
    System.out.println("Child class method");
  }
  public static void main( String args[]) {
    MyChildClass obj = new MyChildClass();
    obj.disp();
  }
}
```

Output: Child class method

3. private, static and final methods cannot be overridden as they are local to the class. However static methods can be re-declared in the sub class, in this case the sub-class method would act differently and will have nothing to do with the same static method of parent class.
4. Overriding method (method of child class) can throw unchecked exceptions, regardless of whether the overridden method(method of parent class) throws any exception or not. However the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. We will discuss this in detail with example in the upcoming tutorial.
5. Binding of overridden methods happen at runtime which is known as dynamic binding.
6. If a class is extending an abstract class or implementing an interface then it has to override all the abstract methods unless the class itself is a abstract class.

Difference between method Overloading and Overriding in Java

Overloading vs Overriding in Java

1. Overloading happens at compile-time while Overriding happens at runtime: The binding of overloaded method call to its definition has happens at compile-time however binding of overridden method call to its definition happens at runtime.
2. Static methods can be overloaded which means a class can have more than one static method of same name. Static methods cannot be overridden, even if you declare a same static method in child class it has nothing to do with the same method of parent class.
3. The most basic difference is that overloading is being done in the same class while for overriding base and child classes are required. Overriding is all about giving a specific implementation to the inherited method of parent class.
4. Static binding is being used for overloaded methods and dynamic binding is being used for overridden/overriding methods.
5. Performance: Overloading gives better performance compared to overriding. The reason is that the binding of overridden methods is being done at runtime.
6. private and final methods can be overloaded but they cannot be overridden. It means a class can have more than one private/final methods of same name but a child class cannot override the private/final methods of their base class.
7. Return type of method does not matter in case of method overloading, it can be same or different. However in case of method overriding the overriding method can have more specific return type (refer this).
8. Argument list should be different while doing method overloading. Argument list should be same in method Overriding.

Overloading example

```
//A class for adding upto 5 numbers
class Sum
{
   int add(int n1, int n2)
   {
      return n1+n2;
   }
   int add(int n1, int n2, int n3)
   {
      return n1+n2+n3;
   }
   int add(int n1, int n2, int n3, int n4)
   {
      return n1+n2+n3+n4;
```

```
    }
    int add(int n1, int n2, int n3, int n4, int n5)
    {
        return n1+n2+n3+n4+n5;
    }
    public static void main(String args[])
    {
            Sum obj = new Sum();
            System.out.println("Sum of two numbers: "+obj.add(20, 21));
            System.out.println("Sum of three numbers: "+obj.add(20, 21, 22));
            System.out.println("Sum of four numbers: "+obj.add(20, 21, 22, 23));
            System.out.println("Sum of five numbers: "+obj.add(20, 21, 22, 23, 24));
    }
}
```

Output:

```
Sum of two numbers: 41
Sum of three numbers: 63
Sum of four numbers: 86
Sum of five numbers: 110
```

Here we have 4 versions of same method add. We are overloading the method add() here.

Overriding example

```
package beginnersbook.com;
class CarClass
{
    public int speedLimit()
    {
        return 100;
    }
}
class Ford extends CarClass
{
    public int speedLimit()
    {
        return 150;
    }
    public static void main(String args[])
    {
            CarClass obj = new Ford();
            int num= obj.speedLimit();
            System.out.println("Speed Limit is: "+num);
    }
}
```

Output:

```
Speed Limit is: 150
```

Here speedLimit() method of class Ford is overriding the speedLimit() method of class CarClass.