March 24, 2013 NO COMMENTS

Compile Time Polymorphism in Java

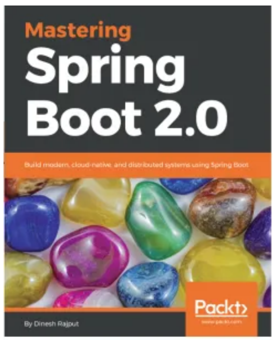
Advertisements

Compile time polymorphism or static method dispatch is a process in which a call to an overloading method is resolved at compile time rather than at run time. In this process, we done overloading of methods is called through the reference variable of a class here no need to superclass.

Method Overloading in Java:
If a class have multiple methods by same name but different parameters, it is known as Method Overloading.

If we have to perform only one operation, having the same name of the methods increases the readability of the program. 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 sum(int,int) for two parameters, and sum2(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. So, we perform method overloading to figure out the program quickly.

Advantage of method overloading?



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

1. Example of By changing number of arguments:

```
class Calculation{
  void sum(int a,int b)
  {
     System.out.println(a+b);
  }
  void sum(int a,int b,int c){
     System.out.println(a+b+c);
   }

  public static void main(String args[]){
  Calculation obj=new Calculation();
  obj.sum(10,10,10);
  obj.sum(20,20);

  }
}
```

output:

```
30
40
```

2. Example of By changing the data type:

In this example, we have created two overloaded methods that differs in data type. The first sum method receives two integer arguments and second sum method receives two double arguments.

```
class Calculation{
  void sum(int a,int b){
    System.out.println(a+b);
   }
  void sum(double a,double b){
    System.out.println(a+b);
```

```
    }
  }
```

output:
23.0
50

Why Method Overloaing is not possible by changing the return type of method?

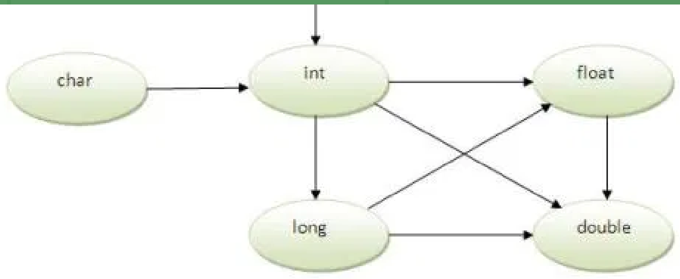
In java, method overloading is not possible by changing the return type of the method because there may occur ambiguity. Let's see how ambiguity may occur:

```
class Calculation{
  int sum(int a,int b)
  {
    System.out.println(a+b);
  }
  double sum(int a,int b){
    System.out.println(a+b);
   }

  public static void main(String args[]){
  Calculation obj=new Calculation();
  int result=obj.sum(30,20); //Compile Ti
  }
}
```

Method Overloading and TypePromotion:

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.

Example of Method Overloading with TypePromotion:

```
class Calculation{
  void sum(int a,long b){System.out.print
  void sum(int a,int b,int c){System.out.

  public static void main(String args[]){
  Calculation obj=new Calculation();
  obj.sum(20,20);//now second int literal
  obj.sum(20,20,20);

  }
}
```

Output:

```
40
60
```

Example of Method Overloading with TypePromotion if matching found:

If there are matching type arguments in the method, type promotion is not performed.

```
    Calculation obj=new Calculation();
    obj.sum(20,20);//now int arg sum() meth
    }
  }
```

Example of Method Overloading with TypePromotion in case ambiguity:

If there are no matching type arguments in the method, and each method promotes similar number of arguments, there will be ambiguity.

```
  class Calculation{
    void sum(int a,long b){System.out.print
    void sum(long a,int b){System.out.print

    public static void main(String args[]){
    Calculation obj=new Calculation();
    obj.sum(20,20);//now ambiguity
    }
  }
```

Output:

Compile Time Error

Using null to overload methods in Java [duplicate]:

The following code compiles and goes fine.

```
  public class Main
  {
      public void temp(Object o)
      {
          System.out.println("The method wi
      }

      public void temp(String s)
      {
        }

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

In this code, the method to be invoked is the one that accepts the parameter of type String

Output:

The method with the receiving parameter of type String has been invoked.

If more than one member method is both accessible and applicable to a method invocation, it is necessary to choose one to provide the descriptor for the run-time method dispatch. The Java programming language uses the rule that the most specific method is chosen.

Where you pass null as argument for an overloaded method, the method chosen is the method with the most specialized type, so in this case: String is chosen rather than the most tolerant: Object.

Among Object/String/int the choice is clear for the compiler: you will get the String's one cause an int cannot be null and so its corresponding method is not eligible to be called in this case.

But if you change int for Integer, compiler will be confuse because both methods taking String is as accurate as Integer's one (orthogonal in hierarchy).

<<Previous << || Index || >>Next >>
« Previous Next »