Tutorial Link https://course.testpad.chitkara.edu.in/tutorials/Java : Function Overloading/5b1e56467becc0459deae0e8

TUTORIAL

Java : Function Overloading

In Java it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. Method overloading is one of the ways that Java implements polymorphism. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call. Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call. Here is a simple example that illustrates method overloading:

Java
```java
// Demonstrate method overloading.
class DemoOverload
{
  void test()
  {
    System.out.println("No parameters");
  }
  // Overload test for one integer parameter.
  void test(int a)
  {
    System.out.println("a =  " + a);
  }
  // Overload test for two integer parameters.
  void test(int a, int b)
  {
    System.out.println("a and b = " + a + " " + b);
  }
  // overload test for a double parameter
  double test(double a)
  {
    System.out.println("double a = " + a);
    return a*a;
  }
}
class Main {
  public static void main(String args[]) {
    DemoOverload ob = new DemoOverload();
    double result;

    ob.test();
    ob.test(5);
    ob.test(5, 55);
    result = ob.test(12.34);
    System.out.println("Result of ob.test(12.34) = " +
result);
  }
}

```

This program generates the following output:

```
No parameters
a = 5
a and b = 5 55
double a: 12.34
Result of ob.test(12.34): 152.2756
```

As you can see, test( ) is overloaded four times. The first version takes no parameters, the second takes one integer parameter, the third takes two integer parameters, and the fourth takes one double parameter. The fact that the fourth version of test( ) also returns a value is of no consequence relative to overloading, since return types do not play a role in overload resolution.

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method's parameters. However, this match need not always be exact. In some cases Java's automatic type conversions can play a role in overload resolution. For example, consider the following program:

```java
// Automatic type conversions apply to overloading.
class DemoOverload
{
  void test()
  {
    System.out.println("No parameters");
  }
  // Overload test for two integer parameters.
  void test(int a, int b)
  {
    System.out.println("a and b: " + a + " " + b);
  }
  // overload test for a double parameter
  void test(double a)
  {
    System.out.println("Inside test(double) a: " + a);
  }
}
class Main {
  public static void main(String args[]) {
    DemoOverload ob = new DemoOverload();
    int i = 5;
    ob.test();
    ob.test(5, 55);
    ob.test(i); // this will invoke test(double)
    ob.test(12.3); // this will invoke test(double)
  }
}

```

This program generates the following output:

```
No parameters
a and b: 5 55
```

```
Inside test(double) a: 5
Inside test(double) a: 12.3
```

As you can see, this version of OverloadDemo does not define test(int). Therefore, when test( ) is called with an integer argument inside Overload, no matching method is found. However, Java can automatically convert an integer into a double, and this conversion can be used to resolve the call. Therefore, after test(int) is not found, Java elevates i to double and then calls test(double). Of course, if test(int) had been defined, it would have been called instead. Java will employ its automatic type conversions only if no exact match is found.

Also in the following program:

```java
class Sum
{
  // Overloaded sum(). This sum takes three int parameters
  public int sum(int x, int y, int z) {
    return (x + y + z);
  }
  // Overloaded sum(). This sum takes two double parameters
  public double sum(double x, double y) {
    return (x + y);
  }
  // Overloaded sum(). This sum takes three double parameters
  public double sum(double x, double y, double z) {
    return (x + y + z);
  }
}
class Main {
  public static void main(String args[]) {
    Sum s = new Sum();
    System.out.println(s.sum(10, 20));
    System.out.println(s.sum(10, 20, 30));
    System.out.println(s.sum(10.5, 20.5));
  }
}
```

The call with two integer arguments will call the method with double arguments, whereas the call with three integer arguments will call the integer arguments functions not the double one as it is defined.

Method overloading supports polymorphism because it is one way that Java implements the "one interface, multiple methods" paradigm. To understand how, consider the following. In languages that do not support method overloading, each method must be given a unique name. However, frequently you will want to implement essentially the same method for different types of data. Consider the absolute value function. In languages that do not support overloading, there are usually three or more versions of this function, each with a slightly different name. For instance, in C, the function abs( ) returns the absolute value of an integer, labs( ) returns the absolute value of a long integer, and fabs( ) returns the absolute value of a floating-point value. Since C does not support overloading, each function has to have its own name, even though all three functions do essentially the same thing. This makes the situation more complex, conceptually, than it actually is. Although the underlying concept of each function is the same, you still have three names to remember. This situation does not occur in Java, because each absolute value method can use the same name. Indeed, Java's standard class library includes an absolute value method, called abs( ). This method is overloaded by Java's Math class to handle all numeric types. Java determines which version of abs( ) to call based upon the type of argument.

The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the name abs represents the general action which is being performed. It is left to the compiler to choose the right specific version for a particular circumstance. You, the programmer, need only remember the general operation being performed. Through the application of polymorphism, several names have been reduced to one. Although this example is fairly simple, if you expand the concept, you can see how overloading can help you manage greater complexity. When you overload a method, each version of that method can perform any activity you desire.

There is no rule stating that overloaded methods must relate to one another.