Method Overloading

Objectives:

1. overload a method
2. overload constructors
3. use constructor arguments

Contents

Overloading Methods

How to define overloaded methods

You can define more than one method with the same name. This is called method overloading.

Both class and instance methods can be overloaded.

Overloading poses a problem. When you invoke a method, you use its name.

If two or more methods have the same name, how would Java know which one it should use?
This problem is solved by making each overloaded method have its own signature.

A method signature is determined by:

- the number of arguments passed to it
- the types of the arguments

Here is a method named multiply. It accepts two arguments, both double. The portion within the ( ), marked in yellow, is the signature of the method.

```
public static double multiply(double x, double y)
    {
        return (x * y);
    }
```

Here is another method, also named multiply, but its signature is different.

```
public static double multiply(double x, double y, double z)
    {
        return (x * y * z);
    }
```

Java can tell these two method apart by their signatures. They are different, in this case, because the number of arguments is different (two versus three).

Methods with different types on the arguments also have different signatures. Consider this example.
Both have two arguments, but the types are different.

```
public static  long add(long x, double y)
    {
        return (x * y);
    }

public static long add(int x, long y)
    {
        return (x * y );
    }
```

How to use overloaded methods

Use an overloaded method just like you would any method. Java will determine which overload to use based on the signature of the method. Using the above example, with the two add methods, Java will choose the second add method because add was invoked with int and a long arguments.

```
public static void main(String[] args)
```

```
{
      int a = 5;
      long b = 55;
      long result = 0;

      result = add(a, b);             //int and long sent as arguments
}
```

In this example, Java chooses the first add method.

```
public static void main(String[] args)
{
      double a = 5;
      long b = 55;
      long result = 0;

      result = add(b, a);             //long and double sent as 
arguments
}
```

The next example would result in an error, because there is no overloaded method that matches the call.

```
public static void main(String[] args)
{
      double a = 5;
      double b = 55;
      long result = 0;

      result = add(b, a);             //double and double sent;  causes 
error
}
```

This example would also cause an error, because there is no matching method:

```
public static void main(String[] args)
{
      double a = 5;
      double b = 55;
      int x = 66;
      long result = 0;

      result = add(x, b, a);             //causes error
}
```

Note: All of these example used static methods, but instance methods can also be overloaded in the same way.

Overloading Constructors

How to define overloaded constructors

You can define more than one constructor with the same name.

If there are more than one constructors in a class, Java uses the signature to determine which one to use.
The signatures of the methods are indicated, one in green, the other in yellow.

Here is a class with two constructors:

```
public class Employee
    {
        private String lastName;
        private String firstName;
        private int age;

        public Employee(String l, String f, int a)
        {
            lastName = l;
            firstName = f;
            age = a;
         }

        public Employee(String l, String f)
        {
            lastName = l;
            firstName = f;
        }

    }
```

How to use overloaded constructors

Use an overloaded constructor just like you would any constructor. Java will determine which
overload to use based on the signature of the constructor.

In the following example:

- for the emp object, Java uses the second constructor in the Employee class.
- for the emp2 object, Java uses the first constructor in the Employee class.

```
public static void main(String[] args)
{
      Employee emp = new Employee("Jones",  "Ed");

      Employee emp2 = new Employee("Nguyen", "Cathy", 29);

}
```

It is an error if you try to create an object and there is no matching constructor, as in this example:

```
public static void main(String[] args)
{
   Employee emp = new Employee("Jones");    //error; no matching
constructor


}
```

---

this reference

The this reference points to the object being used. It is not normally needed, but when using accessor and mutator methods, it can be used.

Consider this example. It is the Employee class with accessor and mutator methods for the lastName variable. Notice the setLastName method. It is sent a String which we call s. s is assigned to the lastName variable. Java can understand this.

```
public class Employee
{

   private String lastName;
   private String firstName;
   private int age;

   //mutator method for lastName variable
   public void setLastName(String s)
   {
        lastName = s;
   }


   //accessor method for lastName variable
   public String getLastName()
   {
        return lastName;
   }


}
```

But what if we named the variable `lastName` instead of `s`. As in this example:

```
public class Employee
{

   private String lastName;
   private String firstName;
   private int age;

   //mutator method for lastName variable
   public void setLastName(String lastName)
   {
        lastName = lastName;
   }


   //accessor method for lastName variable
   public String getLastName()
   {
        return lastName;
   }
}
```

Now Java would be confused on the assignment statement.

```
      lastName = lastName;
```

To eliminate the confusion, you could use the this reference.

```
      this.lastName = lastName;
```

Now the complete example would look like this:

```
public class Employee
{

   private String lastName;
   private String firstName;
   private int age;

   //mutator method for lastName variable
   public void setLastName(String lastName)
   {
        this.lastName = lastName;
   }


   //accessor method for lastName variable
   public String getLastName()
   {
        return lastName;
   }


}
```

The this reference means "the current object". So this.lastName means the lastName variable of the current object.