Passing Information to a Method or a Constructor

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor. For example, the following is a method that computes the monthly payments for a home loan, based on the amount of the loan, the interest rate, the length of the loan (the number of periods), and the future value of the loan:

```
public double computePayment(
                 double loanAmt,
                 double rate,
                 double futureValue,
                 int numPeriods) {
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest),
                   - numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator)
                   - ((futureValue * partial1) / denominator);
    return answer;
}
```

This method has four parameters: the loan amount, the interest rate, the future value and the number of periods. The first three are double-precision floating point numbers, and the fourth is an integer. The parameters are used in the method body and at runtime will take on the values of the arguments that are passed in.

Note: *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.

Parameter Types

You can use any data type for a parameter of a method or a constructor. This includes primitive data types, such as doubles, floats, and integers, as you saw in the `computePayment` method, and reference data types, such as objects and arrays.

Here's an example of a method that accepts an array as an argument. In this example, the method creates a new `Polygon` object and initializes it from an array of `Point` objects (assume that `Point` is a class that represents an x, y coordinate):

```
public Polygon polygonFrom(Point[] corners) {
    // method body goes here
}
```

Note: If you want to pass a method into a method, then use a lambda expression or a method reference.

Arbitrary Number of Arguments

You can use a construct called *varargs* to pass an arbitrary number of values to a method. You use varargs when you don't know how many of a particular type of argument will be passed to the method. It's a shortcut to creating an array manually (the previous method could have used varargs rather than an array).

To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name. The method can then be called with any number of that parameter, including none.

```
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
                    * (corners[1].x - corners[0].x)
                    + (corners[1].y - corners[0].y)
                    * (corners[1].y - corners[0].y);
    lengthOfSide1 = Math.sqrt(squareOfSide1);

    // more method body code follows that creates and returns a
    // polygon connecting the Points
}
```

You can see that, inside the method, `corners` is treated like an array. The method can be called either with an array or with a sequence of arguments. The code in the method body will treat the parameter as an array in either case.

You will most commonly see varargs with the printing methods; for example, this `printf` method:

```
public PrintStream printf(String format, Object... args)
```

allows you to print an arbitrary number of objects. It can be called like this:

```
System.out.printf("%s: %d, %s%n", name, idnum, address);
```

or like this

```
System.out.printf("%s: %d, %s, %s, %s%n", name, idnum, address, phone, email);
```

or with yet a different number of arguments.

Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope. It cannot be the same as the name of another parameter for the same method or constructor, and it cannot be the name of a local variable within the method or constructor.

A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to *shadow* the field. Shadowing fields can make your code difficult to read and is conventionally used only within constructors and methods that set a particular field. For example, consider the following `Circle` class and its `setOrigin` method:

```
public class Circle {
    private int x, y, radius;
    public void setOrigin(int x, int y) {
        ...
    }
}
```

The `Circle` class has three fields: `x`, `y`, and `radius`. The `setOrigin` method has two parameters, each of which has the same name as one of the fields. Each method parameter shadows the field that shares its name. So using the simple names `x` or `y` within the body of the method refers to the parameter, *not* to the field. To access the field, you must use a qualified name. This will be discussed later in this lesson in the section titled "Using the `this` Keyword."

Passing Primitive Data Type Arguments

Primitive arguments, such as an `int` or a `double`, are passed into methods *by value*. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost. Here is an example:

```
public class PassPrimitiveByValue {

    public static void main(String[] args) {

        int x = 3;

        // invoke passMethod() with
        // x as argument
        passMethod(x);

        // print x to see if its
        // value has changed
        System.out.println("After invoking passMethod, x = " + x);

    }

    // change parameter in passMethod()
    public static void passMethod(int p) {
        p = 10;
    }
}
```

When you run this program, the output is:

```
After invoking passMethod, x = 3
```

Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods *by value*. This means that when the method returns, the passed-in reference still references the same object as before. *However*, the values of the object's fields *can* be changed in the method, if they have the proper access level.

For example, consider a method in an arbitrary class that moves `Circle` objects:

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

```
moveCircle(myCircle, 23, 56)
```

Inside the method, `circle` initially refers to `myCircle`. The method changes the `x` and `y` coordinates of the object that `circle` references (i.e., `myCircle`) by 23 and 56, respectively. These changes will persist when the method returns. Then `circle` is assigned a reference to a new `Circle` object with `x = y = 0`. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by `circle` has changed, but, when the method returns, `myCircle` still references the same `Circle` object as before the method was called.

« Previous • Trail • Next »