Java 101 primer: Fields and methods in Java

Concepts and techniques for Java programming with fields and methods

"Java 101: Classes and objects in Java" (JavaWorld, September 2015) covered the fundamentals of object-based programming, including an overview of fields and methods. If you want to become proficient at building object-based Java applications, however, you'll need to deepen your understanding of all of these topics. This **Java 101 primer** starts you out with a look at seven fundamental concepts and techniques for programming with fields and methods.

1. Field constants

You can create a read-only field by including keyword `final` in its declaration. The result is known as a *constant*. For example, `final int DAYS_IN_MONTH = 30;` and `final static double NORMAL_BODY_TEMP = 98.6;` declare constants `DAYS_IN_MONTH` and `NORMAL_BODY_TEMP`. By convention, a constant's name is expressed in capital letters.

Instance and class constants are different. Each object can see a different value for an instance constant, but a class constant presents the same value to all objects. See the difference in the listing below.

Listing 1. Instance vs class constants

```
class Month
{
   final static int NUM_MONTHS = 12;
```

```
    final int DAYS_IN_MONTH;

    Month(int days_in_month)
    {
       DAYS_IN_MONTH = days_in_month;
    }

    public static void main(String[] args)
    {
       System.out.println(Month.NUM_MONTHS);
       Month feb = new Month(28);
       System.out.println(feb.DAYS_IN_MONTH);
       Month jul = new Month(31);
       System.out.println(jul.DAYS_IN_MONTH);
    }
}
```

`Month.java` first declares a `NUM_MONTHS` class constant that's initialized to `12`. All objects created from the `Month` class will see the same value for this constant. A class constant must be assigned a value when the constant is declared and cannot be subsequently modified.

Next, the class declares a `DAYS_IN_MONTH` instance constant. This constant is not initialized as part of its declaration, although it could be. Instead, initialization is deferred to the constructor, which is the only other place where it could be initialized. Such a constant is known as a *blank final*. An instance constant cannot be subsequently modified.

Compile the code as follows:

```
javac Month.java
```

Then run the resulting application:

```
java Month
```

You should observe the following output:

```
12
28
31
```

2. Field-access rules

Fields are accessed in different ways depending on the kind of field (instance or class) and context (from within a class or from code external to the class). These four rules will help you avoid mistakes when accessing different kinds of fields in different contexts:

1. Specify an instance field name without a prefix when accessing this field from another instance field, constructor, or instance method in the same class. Example: `author`.
2. Specify a class field name without a prefix when accessing this field from another instance or class field, constructor, or instance or class method in the same class. Example: `counter`.
3. Specify an object reference followed by the member access operator followed by the instance field name when accessing this field (provided that it is accessible) from outside of its class or from a class method in the same class. Example: `book.title`.
4. Specify a class name followed by the member access operator followed by the class field name when accessing this field (provided that it is accessible) from outside of its class. Example: `Book.counter`.

In some cases shadowing (wherein a parameter or local variable hides or masks an instance field) is an issue. You can resolve it by

prepending `this.` to an instance field name or the class name and member-access operator to a class-field name. For example, if you had to assign a parameter value to a same-named instance field, you would also prepend `this.` to the field name.

3. Chaining instance method calls

Two or more instance method calls can be chained together via the member access operator, which results in more compact code. To accomplish instance method call chaining, you need to re-architect your instance methods. The key is to design your methods to return a reference to the current object, which is indicated via the `this` keyword.

Below you can see that I've changed the return types of the methods being chained together to the *class type*. I've also used return this; to ensure that each method will always return the current object reference.

Listing 2. Setting up an instance method call chain

```
public class TG
{
   public static void main(String[] args)
   {
      Turtle turtle = new Turtle().penDown();
      turtle.move(10).turnLeft().move(10).turnRight().
move(10).penUp();
   }
}

class Turtle
{
   Turtle penUp()
```

```
    {
       System.out.println("pen up");
       return this;
    }

    Turtle penDown()
    {
       System.out.println("pen down");
       return this;
    }

    Turtle turnLeft()
    {
       System.out.println("turn left");
       return this;
    }

    Turtle turnRight()
    {
       System.out.println("turn right");
       return this;
    }

    Turtle move(int numUnits)
    {
       System.out.println("moving " + numUnits + "
 units");
       return this;
    }
 }
```

The source code in Listing 2 is from a Turtle Graphics, an application I created that demonstrates instance method call chaining. This application consists of a `TG` main class and a `turtle` helper class.

The main class instantiates `Turtle` and chains various instance method calls to this reference.

You will notice that `Turtle` doesn't include a constructor. I chose to not declare a constructor because there was nothing to initialize. When no constructors are declared, the compiler generates a default no-argument constructor that does nothing. The compiler doesn't generate this constructor when at least one constructor is declared.

4. Pass-by-value arguments

A method or constructor call includes zero or more arguments that are passed to the method or constructor. Java passes arguments to methods and constructors via *pass-by-value*, which passes the value of a variable or the value of another expression to that element. A pass-by-value argument is demonstrated below.

```
Library library = new Library();
Book book = new Book("Moby Dick", 1851);
library.add(book);
```

With pass-by-value, a called method or constructor cannot change its argument(s). For example, there is no way for `Library's void add(Book book)` method to change the argument that was passed to the book parameter. So you couldn't do this:

```
class Library
{
   void add(Book book)
   {
      book = new Book("...", 2015);
      // ...
   }
```

```
}
```

and expect the value in the caller's book local variable (of the previous `Book book = new Book("Moby Dick", 1851);` expression) to change. If you did successfully change the argument, the JVM would probably crash the first time that it attempted to assign a new value to `null` in a `library.add(null);` method call.

5. Recursion and the method-call stack

A method normally executes statements that may include calls to other methods. We saw this in the "Return statements" section of the main article, where a `copy()` method called `System.in.read()` and `System.out.println()`. However, it's often useful to have a method call itself. This programming technique is known as *recursion*.

For example, suppose you needed to write a method to return a *factorial*, which is the product of all the positive integers up to and including a specific integer. Knowing that ! is the mathematical symbol for factorial, you can guess that 4! equals 4x3x2x1, or 24. A first approach to writing this method could consist of the code presented below:

```
static int factorial(int n)
{
   int product = 1;
   for (int i = 2; i <= n; i++)
      product *= i;
   return product;
}
```

Although this code accomplishes its task via iteration, `factorial()` could be written more compactly by adopting a recursive style:

```
static int factorial(int n)
{
   if (n == 1)
      return 1; // base problem
   else
      return n * factorial(n - 1);
}
```

The recursive approach expresses a problem in simpler terms of itself. According to this example, the simplest problem, which is also known as the *base problem*, is 1! `(1)`. When an argument greater than 1 is passed to `factorial()`, this method divides the problem into a simpler problem by calling itself with the next smaller argument value. Eventually, the base problem will be reached. For example, calling `factorial(4)` results in the following stack of expressions:

```
4 * factorial(3)
3 * factorial(2)
2 * factorial(1)
```

This last expression is at the top of the stack. When `factorial(1)` returns 1, these expressions are evaluated as the stack begins to unwind, in the following order:

1. `2 * factorial(1)` now becomes 2*1 (2)
2. `3 * factorial(2)` now becomes 3*2 (6)
3. `4 * factorial(3)` now becomes 4*6 (24)

Recursion provides an elegant way to express many problems. Additional examples include searching tree-based data structures for specific values and, in a hierarchical file system, finding and outputting the names of all files that contain specific text.

Unlimited recursion and stack space exhaustion

Recursion consumes stack space, so make sure that your recursion eventually ends in a base problem; otherwise, you will run out of stack space and your application will be forced to terminate.

The method-call stack

Method calls require a *method-call stack* to keep track of the statements to which execution must return. Furthermore, the stack keeps track of parameters and local variables on a per-method-call basis. Think of the method-call stack as a pile of clean trays in a cafeteria — you pop a clean tray from the top of the pile and the dishwasher will push the next clean tray onto the top of the pile. When a method is called, the JVM pushes the called method onto the method-call stack, along with its arguments and the address of the first statement to execute on that method. The JVM also allocates stack space for the method's parameters and/or local variables. When the method returns, the JVM removes the parameter/local variable space, pops the address and arguments off of the stack, and transfers execution to the statement at the given address.

Think of the method-call stack as a pile of clean trays in a cafeteria — you pop a clean tray from the top of the pile and the dishwasher will push the next clean tray onto the top of the pile.

6. Rules for calling methods

Methods are called in different ways depending on the kind of method (instance or class) and context (from within a class or from code external to the class). Here are four rules for calling different kinds of methods in various contexts:

1. Specify an instance method name without a prefix when calling the method from another instance method or constructor in the same class. Example: `add(book)`.
2. Specify a class method name without a prefix when calling it from another instance or class method, or from a constructor in the same class. Example: `search(values, value)`.
3. Specify an object reference followed by the member access operator followed by the instance method name when calling a method from outside of its class or from a class method in the same class (provided that it is accessible). Example: `book.getTitle()`.
4. Specify a class name followed by the member access operator followed by the class method name when calling a method from outside of its class (provided that it is accessible). Example: `Book.showCount()`.

Don't forget to make sure that the number of arguments passed to a method — along with the order in which they are passed and the types of these arguments — agree with their parameter counterparts in the method being called. Otherwise, the compiler will report an error.

7. Utility classes

A *utility class* consists of static fields and/or static methods. The standard class library contains examples of utility classes, including `Math`. Here's another example of a utility class:

Listing 3. A utility class in Java

```
class Utilities
{
   // Prevent Utilities from being instantiated by
declaring a
```

```
   // private noargument constructor.

   private Utilities()
   {
   }

   static double average(double[] values)
   {
      double sum = 0.0;
      for (int i = 0; i < values.length; i++)
         sum += values[i];
      return sum / values.length;
   }

   static void copy() throws java.io.IOException // I'll
discuss throws and
   {                                              //
exceptions in a future
      while (true)                                //
article.
      {
         int _byte = System.in.read();
         if (_byte == -1)
            return;
         System.out.print((char) _byte);
      }
   }

   static int factorial(int n)
   {
      if (n == 1)
         return 1; // base problem
      else
         return n * factorial(n - 1);
```

```
    }

    static int search(int[] values, int srchValue)
    {
       for (int i = 0; i < values.length; i++)
          if (values[i] == srchValue)
             return i; // return index of found value
       return -1; // -1 is an invalid index, so it's use-
ful for indicating
                  // "value not found".
    }
}
```

The `Utilities` class declared in Listing 3 serves as a placeholder for most of the class methods that you saw in the main article. To prevent `Utilities` from being instantiated, I've declared a private, no-argument (and empty) constructor.

Here's a small class that demonstrates a number of `Utilities` methods.

Listing 4. Utilities methods

```
class UtilDemo
{
   public static void main(String[] args)
   {
      double[] values = { 10.0, 20.0, 30.0, 40.0 };
      System.out.println(Utilities.average(values));
      System.out.println(Utilities.factorial(5));
      int[] numbers = { 25, 49, 33, 89 };
      System.out.println(Utilities.search(numbers, 33));
      System.out.println(Utilities.search(numbers, 34));
   }
```

```
}
```

If you compile Listing 6 as follows:

```
javac UtilDemo.java
```

and run the resulting application:

```
java UtilDemo
```

you should observe the following output:

```
25.0
120
2
-1
```

Conclusion

In "Java 101: Classes and objects in Java" you were presented with a minimal set of features for working with classes and objects. This primer built on that article with additional concepts and techniques related to fields and methods — namely field constants, field-access rules, chaining instance method calls, pass-by-value arguments, recursion and the method-call stack, rules for calling methods, and utility classes. Continue learning about this topic by downloading and experimenting with the provided source code. And be sure to look for the next article in this series, where I will introduce inheritance, the game-changer that differentiates *object-based programming* from *object-oriented programming*.