[Previous Section | Chapter Index | Main Index]

Section 4.7

The Truth About Declarations

Subsections

NAMES ARE FUNDAMENTAL TO PROGRAMMING, as I said a few chapters ago. There are a lot of details involved in declaring and using names. I have been avoiding some of those details. In this section, I'll reveal most of the truth (although still not the full truth) about declaring and using variables in Java. The material in the subsections "Initialization in Declarations" and "Named Constants" is particularly important, since I will be using it regularly from now on.

4.7.1 Initialization in Declarations

When a variable declaration is executed, memory is allocated for the variable. This memory must be initialized to contain some definite value before the variable can be used in an expression. In the case of a local variable, the declaration is often followed closely by an assignment statement that does the initialization. For example,

```
int count;    // Declare a variable named count.
count = 0;    // Give count its initial value.
```

However, the truth about declaration statements is that it is legal to include the initialization of the variable in the declaration statement. The two statements above can therefore be abbreviated as

```
int count = 0;  // Declare count and give it an initial value.
```

The computer still executes this statement in two steps: Declare the variable `count`, then assign the value 0 to the newly created variable. The initial value does not have to be a constant. It can be any expression. It is legal to initialize several variables in one declaration statement. For example,

```
char firstInitial = 'D', secondInitial = 'E';

int x, y = 1;   // OK, but only y has been initialized!

int N = 3, M = N+2;  // OK, N is initialized
                     //        before its value is used.
```

This feature is especially common in `for` loops, since it makes it possible to declare a loop control variable at the same point in the loop where it is initialized. Since the loop control variable generally has nothing to do with the rest of the program outside the loop, it's reasonable to have its declaration in the part of the program where it's actually used. For example:

```
for ( int i = 0;  i < 10;  i++ ) {
   System.out.println(i);
}
```

You should remember that this is simply an abbreviation for the following, where I've added an extra pair of braces to show that `i` is considered to be local to the `for` statement and no longer exists after the `for` loop ends:

```
{
   int i;
```

```
   for ( i = 0;  i < 10;  i++ ) {
      System.out.println(i);
   }
}
```

A member variable can also be initialized at the point where it is declared, just as for a local variable. For example:

```
public class Bank {
   private static double interestRate = 0.05;
   private static int maxWithdrawal = 200;
     .
     .  // More variables and subroutines.
     .
}
```

A static member variable is created as soon as the class is loaded by the Java interpreter, and the initialization is also done at that time. In the case of member variables, this is not simply an abbreviation for a declaration followed by an assignment statement. Declaration statements are the only type of statement that can occur outside of a subroutine. Assignment statements cannot, so the following is illegal:

```
public class Bank {
   private static double interestRate;
   interestRate = 0.05;  // ILLEGAL:
   .                     //    Can't be outside a subroutine!:
   .
   .
```

Because of this, declarations of member variables often include initial values. In fact, as mentioned in Subsection 4.2.4, if no initial value is provided for a member variable, then a default initial value is used. For example, when declaring an integer member variable, `count`, "`static int count;`" is equivalent to "`static int count = 0;`".

Even array variables can be initialized. An array contains several elements, not just a single value. To initialize an array variable, you can provide a list of values, separated by commas, and enclosed between a pair of braces. For example:

```
int[] smallPrimes = { 2, 3, 5, 7, 11, 13, 17, 23, 29 };
```

In this statement, an array of int of length 9 is created and filled with the values in the list. The length of the array is determined by the number of items in the list.

Note that this syntax for initializing arrays **cannot** be used in assignment statements. It can only be used in a declaration statement at the time when the array variable is declared.

It is also possible to initialize an array variable with an array created using the `new` operator (which **can** also be used in assignment statements). For example:

```
String[] namelist = new String[100];
```

but in that case, of course, all the array elements will have their default value.

4.7.2 Named Constants

Sometimes, the value of a variable is not supposed to change after it is initialized. For example, in the above example where `interestRate` is initialized to the value `0.05`, it's quite possible that 0.05 is

meant to be the value throughout the entire program. In that case, the programmer is probably defining the variable, `interestRate`, to give a meaningful name to the otherwise meaningless number, `0.05`. It's easier to understand what's going on when a program says "`principal += principal*interestRate;`" rather than "`principal += principal*0.05;`".

In Java, the modifier "`final`" can be applied to a variable declaration to ensure that the value stored in the variable cannot be changed after the variable has been initialized. For example, if the member variable `interestRate` is declared with

```
public final static double interestRate = 0.05;
```

then it would be impossible for the value of `interestRate` to change anywhere else in the program. Any assignment statement that tries to assign a value to `interestRate` will be rejected by the computer as a syntax error when the program is compiled. (A "final" modifier on a public interest rate makes a lot of sense -- a bank might want to publish its interest rate, but it certainly wouldn't want to let random people make changes to it!)

It is legal to apply the `final` modifier to local variables and even to formal parameters, but it is most useful for member variables. I will often refer to a static member variable that is declared to be `final` as a named constant, since its value remains constant for the whole time that the program is running. The readability of a program can be greatly enhanced by using named constants to give meaningful names to important quantities in the program. A recommended style rule for named constants is to give them names that consist entirely of upper case letters, with underscore characters to separate words if necessary. For example, the preferred style for the interest rate constant would be

```
public final static double INTEREST_RATE = 0.05;
```

This is the style that is generally used in Java's standard classes, which define many named constants. For example, we have already seen that the Math class contains a variable `Math.PI`. This variable is declared in the Math class as a "public final static" variable of type double. Similarly, the `Color` class contains named constants such as `Color.RED` and `Color.YELLOW` which are public final static variables of type `Color`. Many named constants are created just to give meaningful names to be used as parameters in subroutine calls. For example, the standard class named `Font` contains named constants `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC`. These constants are used for specifying different styles of text when calling various subroutines in the `Font` class.

Enumerated type constants (see Subsection 2.3.3) are also examples of named constants. The enumerated type definition

```
enum Alignment { LEFT, RIGHT, CENTER }
```

defines the constants `Alignment.LEFT`, `Alignment.RIGHT`, and `Alignment.CENTER`. Technically, Alignment is a class, and the three constants are public final static members of that class. Defining the enumerated type is similar to defining three constants of type, say, int:

```
public static final int ALIGNMENT_LEFT = 0;
public static final int ALIGNMNENT_RIGHT = 1;
public static final int ALIGNMENT_CENTER = 2;
```

In fact, this is how things were generally done before the introduction of enumerated types, and it is what is done with the constants `Font.PLAIN`, `Font.BOLD`, and `Font.ITALIC` mentioned above. Using the integer constants, you could define a variable of type int and assign it the values `ALIGNMENT_LEFT`, `ALIGNMENT_RIGHT`, or `ALIGNMENT_CENTER` to represent different types of alignment. The only problem with this is that the computer has no way of knowing that you intend the value of the variable to represent an alignment, and it will not raise any objection if the value that is assigned to the variable is not one of the three valid alignment values. With the enumerated type, on the other hand, the only

values that can be assigned to a variable of type Alignment are the constant values that are listed in the definition of the enumerated type. Any attempt to assign an invalid value to the variable is a syntax error which the computer will detect when the program is compiled. This extra safety is one of the major advantages of enumerated types.

Curiously enough, one of the major reasons to use named constants is that it's easy to change the value of a named constant. Of course, the value can't change while the program is running. But between runs of the program, it's easy to change the value in the source code and recompile the program. Consider the interest rate example. It's quite possible that the value of the interest rate is used many times throughout the program. Suppose that the bank changes the interest rate and the program has to be modified. If the literal number 0.05 were used throughout the program, the programmer would have to track down each place where the interest rate is used in the program and change the rate to the new value. (This is made even harder by the fact that the number 0.05 might occur in the program with other meanings besides the interest rate, as well as by the fact that someone might have, say, used 0.025 to represent half the interest rate.) On the other hand, if the named constant `INTEREST_RATE` is declared and used consistently throughout the program, then only the single line where the constant is initialized needs to be changed.

As an extended example, I will give a new version of the `RandomMosaicWalk` program from the previous section. This version uses named constants to represent the number of rows in the mosaic, the number of columns, and the size of each little square. The three constants are declared as `final static` member variables with the lines:

```
final static int ROWS = 20;        // Number of rows in mosaic.
final static int COLUMNS = 30;     // Number of columns in mosaic.
final static int SQUARE_SIZE = 15; // Size of each square in mosaic.
```

The rest of the program is carefully modified to use the named constants. For example, in the new version of the program, the Mosaic window is opened with the statement

```
Mosaic.open(ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE);
```

Sometimes, it's not easy to find all the places where a named constant needs to be used. If you don't use the named constant consistently, you've more or less defeated the purpose. It's always a good idea to run a program using several different values for any named constant, to test that it works properly in all cases.

Here is the complete new program, `RandomMosaicWalk2`, with all modifications from the previous version shown in red. I've left out some of the comments to save space.

```
public class RandomMosaicWalk2 {

    final static int ROWS = 20;        // Number of rows in mosaic.
    final static int COLUMNS = 30;     // Number of columns in mosaic.
    final static int SQUARE_SIZE = 15; // Size of each square in mosaic.

    static int currentRow;    // Row currently containing the disturbance.
    static int currentColumn; // Column currently containing the disturbance.

    public static void main(String[] args) {
        Mosaic.open( ROWS, COLUMNS, SQUARE_SIZE, SQUARE_SIZE );
        fillWithRandomColors();
        currentRow = ROWS / 2;   // start at center of window
        currentColumn = COLUMNS / 2;
        while (Mosaic.isOpen()) {
            changeToRandomColor(currentRow, currentColumn);
            randomMove();
```