THROWING EXCEPTIONS AND GLOBAL AND LOCAL VARIABLES

Throwing Exceptions

I have been talking about the "contract" of a subroutine. The contract says what the subroutine will do, provided that the caller of the subroutine provides acceptable values for subroutine's parameters. The question arises, though, what should the subroutine do when the caller violates the contract by providing bad parameter values?

We've already seen that some subroutines respond to bad parameter values by throwing exceptions. (See Section 3.7.) For example, the contract of the built-in subroutine `Double.parseDouble` says that the parameter should be a string representation of a number of type double; if this is true, then the subroutine will convert the string into the equivalent numeric value. If the caller violates the contract by passing an invalid string as the actual parameter, the subroutine responds by throwing an exception of type *NumberFormatException*.

Many subroutines throw *IllegalArgumentExceptions* in response to bad parameter values. You might want to take this response in your own subroutines. This can be done with a throw statement. An exception is an object, and in order to throw an exception, you must create an exception object. You won't officially learn how to do this until Chapter 5, but for now, you can use the following syntax for a `throw` statement that throws an *IllegalArgumentException*:

```
throw  new  IllegalArgumentException( error-message );
```

where **error-message** is a string that describes the error that has been detected. (The word "new" in this statement is what creates the object.) To use this statement in a subroutine, you would check whether the values of the parameters are legal. If not, you would throw the exception. For example, consider the `print3NSequence` subroutine from the beginning of this section. The parameter of `print3NSequence` is supposed to be a positive integer. We can modify the subroutine definition to make it throw an exception when this condition is violated:

```
static void print3NSequence(int startingValue) {

   if (startingValue <= 0)  // The contract is violated!
      throw new IllegalArgumentException( "Starting value
must be positive." );
   .
   .   // (The rest of the subroutine is the same as
before.)
   .
```

If the start value is bad, the computer executes the `throw` statement. This will immediately terminate the subroutine, without executing the rest of the body of the subroutine. Furthermore, the program as a whole will crash unless the exception is "caught" and handled elsewhere in the program by a `try..catch` statement, as discussed in Section 3.7.

Global and Local Variables

I'll finish this section on parameters by noting that we now have three different sorts of variables that can be used inside a subroutine: local variables declared in

the subroutine, formal parameter names, and static member variables that are declared outside the subroutine but inside the same class as the subroutine.

Local variables have no connection to the outside world; they are purely part of the internal working of the subroutine. Parameters are used to "drop" values into the subroutine when it is called, but once the subroutine starts executing, parameters act much like local variables. Changes made inside a subroutine to a formal parameter have no effect on the rest of the program (at least if the type of the parameter is one of the primitive types -- things are more complicated in the case of objects, as we'll see later).

Things are different when a subroutine uses a variable that is defined outside the subroutine. That variable exists independently of the subroutine, and it is accessible to other parts of the program, as well as to the subroutine. Such a variable is said to be global to the subroutine, as opposed to the local variables defined inside the subroutine. The scope of a global variable includes the entire class in which it is defined. Changes made to a global variable can have effects that extend outside the subroutine where the changes are made. You've seen how this works in the last example in the previous section, where the value of the global variable, `gamesWon`, is computed inside a subroutine and is used in the `main()` routine.

It's not always bad to use global variables in subroutines, but you should realize that the global variable then has to be considered part of the subroutine's interface. The subroutine uses the global variable to communicate with the rest of the program. This is a kind of sneaky, back-door communication that is less visible than communication done through parameters, and it risks violating the rule that the interface of a black box should be straightforward and easy to understand. So

before you use a global variable in a subroutine, you should consider whether it's really necessary.

I don't advise you to take an absolute stand against using global variables inside subroutines. There is at least one good reason to do it: If you think of the class as a whole as being a kind of black box, it can be very reasonable to let the subroutines inside that box be a little sneaky about communicating with each other, if that will make the class as a whole look simpler from the outside.

Source : http://math.hws.edu/javanotes/c4/s3.html