3.3: Locality of Reference

With the introduction of functions comes the notion of local variables. Local variables are those that are declared inside one function. By using local variables instead of global ones, a function can hide what it is doing from the rest of the program. This is another Good Thing, for a number of reasons.

First, local variables can be created afresh every time a function is called. (Well, this is true for most languages nowadays.) For example, in initialization, previous calls of the function won't influence the results of future ones. This property of local variables also makes *recursion* possiblethe ability of a function to call itself (see below). Recursion can be a powerful technique if used properly.

Secondly, by employing local variables, you can use common variable names in many different functions without the different uses interfering with one another. For example, using "i" for loop counters (or "x" and "y" for coordinates) in many different functions is fine, as long as these variables are all local to the functions.

Third, and perhaps most important, by using local variables, you the programmer can be sure that those variables are not being changed from anywhere in the program except from the function in which they are declared. Local variables cannot be referred to outside of the function they're declared in. This is an enormous advantage while you're debugging: anytime you can narrow down the search for misbehaving code, you're ahead of the game.

Local variable use is one special case of the general idea of "locality of data." Different languages support locality in different ways, and we'll soon see that object-oriented languages have their own special technique.

In C, variables declared at file scope (outside of any function) can be declared static which in that language means that the variables can be accessed (read and/or written) only by functions defined in that one file of source code. By placing related groups of functions in separate files, along with the data they manipulate, a program can be divided into logically separate, mostly noninteracting parts, with well-defined interfaces. (Visual Basic supports this notion via the "PRIVATE" statement. Variables declared PRIVATE cannot be accessed outside of the module in which they're declared.) C also allows the "static" keyword to be applied to functions as well. "Static" tells the compiler that no code outside of that source file can call the function; using it, the programmer can hide implementation details inside of these separate modules. Hiding implementation is good because it gives you the flexibility to change the innards of a module with the knowledge that you're not breaking any other code in the program. In Visual Basic, functions and subroutines can be declared PRIVATE to make them invisible outside of the module in which they are declared.

Locality in general is a tremendous boon to anyone who needs to maintain code. Code that keeps all of its data in globally accessible variables can be a nightmare to understand and debug. Restricting access makes the interactions between parts of a program clearer and easier to understand.

Another important side effect of using only local data is the possibility of easy reuse. If you write a function that accesses several global variables, then (in general) it can be used only in the program in which you originally wrote it; it is too tangled up in the rest of the code to be extracted and moved elsewhere. On the other hand, if it uses only local data and exchanges information with the rest of the program through parameters, you can easily reuse it in every program you write.

All of these meanings of locality are important and strongly represented in Java. Java's units of locality comparable to C's file and Visual Basic's module are classes and packages (a package is simply a group of classes.) Hiding things inside of classes and packages is both easy and strongly encouraged. We'll learn how in the next module.