[Next Section | Chapter Index | Main Index]

Section 4.1

Black Boxes

A SUBROUTINE CONSISTS OF INSTRUCTIONS for performing some task, chunked together and given a name. "Chunking" allows you to deal with a potentially very complicated task as a single concept. Instead of worrying about the many, many steps that the computer might have to go though to perform that task, you just need to remember the name of the subroutine. Whenever you want your program to perform the task, you just call the subroutine. Subroutines are a major tool for dealing with complexity.

A subroutine is sometimes said to be a "black box" because you can't see what's "inside" it (or, to be more precise, you usually don't **want** to see inside it, because then you would have to deal with all the complexity that the subroutine is meant to hide). Of course, a black box that has no way of interacting with the rest of the world would be pretty useless. A black box needs some kind of interface with the rest of the world, which allows some interaction between what's inside the box and what's outside. A physical black box might have buttons on the outside that you can push, dials that you can set, and slots that can be used for passing information back and forth. Since we are trying to hide complexity, not create it, we have the first rule of black boxes:

> **The interface of a black box should be fairly straightforward, well-defined, and easy to understand.**

Are there any examples of black boxes in the real world? Yes; in fact, you are surrounded by them. Your television, your car, your mobile phone, your refrigerator.... You can turn your television on and off, change channels, and set the volume by using elements of the television's interface—on/off switch, remote control, don't forget to plug in the power—without understanding anything about how the thing actually works. The same goes for a mobile phone, although the interface in that case is a lot more complicated.

Now, a black box does have an inside—the code in a subroutine that actually performs the task, or all the electronics inside your television set. The inside of a black box is called its implementation. The second rule of black boxes is that:

> **To use a black box, you shouldn't need to know anything about its implementation; all you need to know is its interface.**

In fact, it should be possible to **change** the implementation, as long as the behavior of the box, as seen from the outside, remains unchanged. For example, when the insides of TV sets went from using vacuum tubes to using transistors, the users of the sets didn't need to know about it—or even know what it means. Similarly, it should be possible to rewrite the inside of a subroutine, to use more efficient code for example, without affecting the programs that use that subroutine.

Of course, to have a black box, someone must have designed and built the implementation in the first place. The black box idea works to the advantage of the implementor as well as the user of the black box. After all, the black box might be used in an unlimited number of different situations. The implementor of the black box doesn't need to know about any of that. The implementor just needs to

make sure that the box performs its assigned task and interfaces correctly with the rest of the world. This is the third rule of black boxes:

> **The implementor of a black box should not need to know anything about the larger systems in which the box will be used.**

In a way, a black box divides the world into two parts: the inside (implementation) and the outside. The interface is at the boundary, connecting those two parts.

By the way, you should **not** think of an interface as just the physical connection between the box and the rest of the world. The interface also includes a specification of what the box does and how it can be controlled by using the elements of the physical interface. It's not enough to say that a TV set has a power switch; you need to specify that the power switch is used to turn the TV on and off!

To put this in computer science terms, the interface of a subroutine has a semantic as well as a syntactic component. The syntactic part of the interface tells you just what you have to type in order to call the subroutine. The semantic component specifies exactly what task the subroutine will accomplish. To write a legal program, you need to know the syntactic specification of the subroutine. To understand the purpose of the subroutine and to use it effectively, you need to know the subroutine's semantic specification. I will refer to both parts of the interface—syntactic and semantic —collectively as the contract of the subroutine.

The contract of a subroutine says, essentially, "Here is what you have to do to use me, and here is what I will do for you, guaranteed." When you write a subroutine, the comments that you write for the subroutine should make the contract very clear. (I should admit that in practice, subroutines' contracts are often inadequately specified, much to the regret and annoyance of the programmers who have to use them.)

For the rest of this chapter, I turn from general ideas about black boxes and subroutines in general to the specifics of writing and using subroutines in Java. But keep the general ideas and principles in mind. They are the reasons that subroutines exist in the first place, and they are your guidelines for using them. This should be especially clear in Section 4.7, where I will discuss subroutines as a tool in program development.

You should keep in mind that subroutines are not the only example of black boxes in programming. For example, a class is also a black box. We'll see that a class can have a "public" part, representing its interface, and a "private" part that is entirely inside its hidden implementation. All the principles of black boxes apply to classes as well as to subroutines.

[Next Section | Chapter Index | Main Index]