18.6 Recursion and the Method-Call Stack

In Chapter 6, the *stack* data structure was introduced in the context of understanding how Java performs method calls. We discussed both the *method-call stack* and *stack frames*. In this section, we'll use these concepts to demonstrate how the program-execution stack handles *recursive* method calls.

Let's begin by returning to the Fibonacci example—specifically, calling method `fibonacci` with the value `3`, as in Fig. 18.6. To show the *order* in which the method calls' stack frames are placed on the stack, we've lettered the method calls in Fig. 18.7.

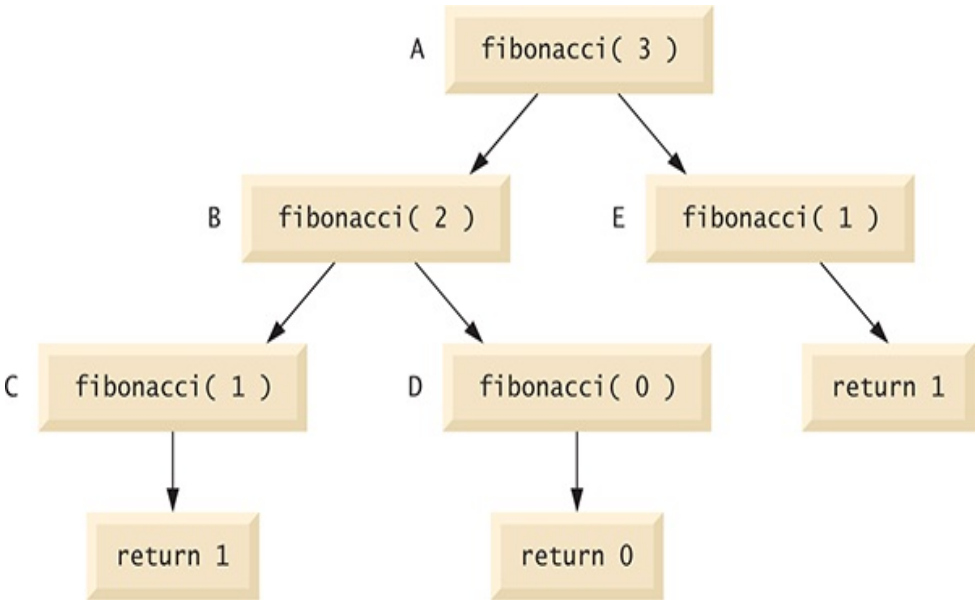

Fig. 18.7

Method calls made within the call `fibonacci(3)`.

Description

When the first method call (`A`) is made, a *stack frame* containing the value of the local variable `number` (`3`, in this case) is *pushed* onto the *program-execution stack*. This stack, including the stack frame for method call `A`, is illustrated in part (a) of Fig. 18.8. [*Note:* We use a simplified stack here. An actual program-execution stack and its stack frames would be more complex than in Fig. 18.8, containing such information as where the method call is to *return* to when it has completed execution.]

Within method call `A`, method calls `B` and `E` are made. The original method call has not yet completed, so its stack frame remains on the stack. The first method call to be made from within `A` is method call `B`, so the stack frame for method call `B` is pushed onto the stack on top of the one for method call `A`. Method call `B` must execute and complete before method call `E` is made.

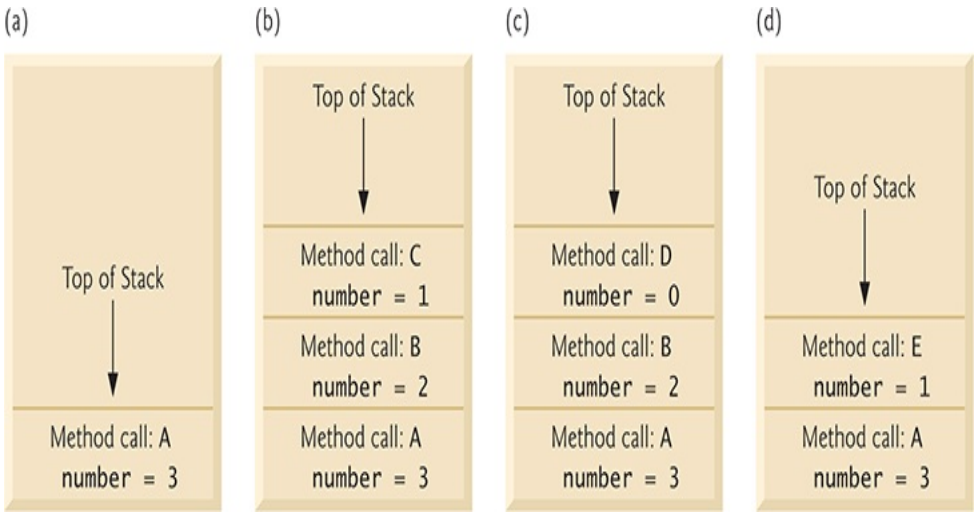

Fig. 18.8

Method calls on the program-execution stack.

Description

Within method call `B`, method calls `C` and `D` will be made. Method call `C` is made first, and its stack frame is pushed onto the stack [part (b) of Fig. 18.8]. Method call `B` has not yet finished, and its stack frame is still on the method-call stack. When method call `C` executes, it makes no further method calls, but simply returns the value `1`. When this method returns, its stack frame is popped off the top of the stack. The method call at the top of the stack is now `B`, which continues to execute by performing method call `D`. The stack frame for method call `D` is pushed onto the stack [part (c) of Fig. 18.8]. Method call `D` completes without making any more method calls and returns the value `0`. The stack frame for this method call is then popped off the stack.

Now, both method calls made from within method call `B` have returned. Method call `B` continues to execute, returning the value `1`. Method call `B` completes, and its stack frame is popped off the stack. At this point, the stack frame for method call `A` is at the top of the stack and the method continues its execution. This method makes method call `E`, whose stack frame is now pushed onto the stack [part (d) of Fig. 18.8]. Method call `E` completes and returns the value `1`. The stack frame for this method call is popped off the stack, and once again method call `A` continues to execute.

At this point, method call `A` will not make any other method calls and can finish its execution, returning the value `2` to `A`'s caller (`fibonacci(3) = 2`). `A`'s stack frame is popped off the stack. The executing method is always the one whose stack frame is at the top of the stack, and the stack frame for that method contains the values of its local variables.