Chapter 8 Recursion

The subject of this chapter is *recursion*, an implicit application of the stack ADT. A *recursive method* is a method that carries out its task by making a call(s) to itself. Each *recursive call* of a (correctly implemented) recursive method reduces the problem size, i.e., recursion is a divide-and-conquer approach to problem solving. When the problem size becomes sufficiently small, the solutions of all the smaller subproblems are combined to arrive at a solution to the original problem. As we will see in this chapter and the next, recursion is a powerful problem-solving technique.

8.1 Recursive Definitions

The first step in solving a problem recursively is to formulate a *recursive definition* of the problem.

recursive definition: a self-referential definition, i.e., a definition that defines an expression in terms of itself

As a first example, let us consider the case of raising a real number, x, to a nonnegative integer power, n. A standard definition for x^n is shown below. This definition makes it apparent that x^n can be computed using the loop that follows the definition.

$$x^n = \begin{cases} 1, \text{if } n = 0 \\ \overbrace{x \cdot x \cdot x \cdot \ldots \cdot x \cdot x}^{n}, \text{if } n > 0 \end{cases}$$

```
// Compute x^n using a loop.

double result = 1;
for (int j = 1; j <= n; j++)
    result *= x;
```

The recursive definition of x^n shown below is more concise but appears to offer little guidance to the programmer.

$$x^n = \begin{cases} 1, \text{if } n = 0 \\ x \cdot x^{n-1}, \text{if } n > 0 \end{cases}$$

This definition does provide us with enough information to program a solution, however. Consider method `power`.

```
// Compute x^n recursively.

public class Recurse
{
    /*
     precondition: This method computes x^n, where n is assumed
                   to be nonnegative. A call to this method with
                   negative n  will cause the program to crash.
    postcondition: x^n has been returned.
    */

    public static double power(double x, int n)
    {
        if (n == 0)      // This is called the base case. The
            return 1.0;  // base case controls the depth of the
                         // recursion, i.e., how many recursive
                         // calls are made.

        return x * power(x, n - 1);  // This is just the recursive
                                     // definition restated in
    }                                // Java!
}
```

Method `power` will in fact compute x^n, but how it does so may be far from obvious. In order to understand `power`'s behavior we must look *beneath* the code above, for recursion turns on the use of a hidden stack, the *run-time stack*, for passing arguments.

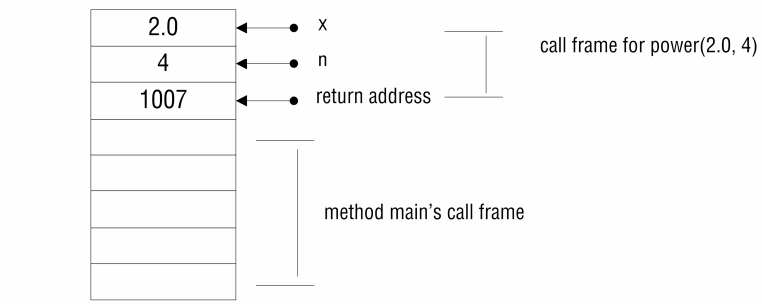
8.2 Method Arguments and the Run-Time Stack

Available to any running program is a stack called the run-time stack. If the program in question takes the form of machine code, i.e., if the program was written in a truly compiled language, then the run-time stack is part of the running program. Java is not a truly compiled language; a Java class file contains not machine code but byte code. Byte code is a language intermediate between Java and machine language. Java byte code cannot be executed directly by the CPU; byte code must be *interpreted*, i.e., executed by a machine-language program. The machine-language program that executes Java byte code is called the *Java Virtual Machine* (*JVM*). The run-time stack available to a Java program resides within the JVM.

When a method is called, the method's arguments and its *return address* are pushed onto the run-time stack before execution of the method begins. A method's return address marks the location at which program execution should continue after the method has returned. The method's arguments, its return address, and any other pertinent data that have been pushed onto the run-time stack are collectively referred to as the method call's *stack frame*, or *call frame*, or *activation record*. An example appears in the code snippet and figure below.

```
public class Function
{
    public static double power(double x, int n)
    {
        double result = 1;
        for (int j = 1; j <= n; j++)
            result *= x;
        return result;
    }
}
.
.
.
public class Application
{
    /*
    Method main() contains a call to the power() method defined
    above. For the sake of this example, assume that the call's
    return address is 1007. power() must return to address 1007
    so that its return value can be assigned to 'result'.
    */

    public static void main(String[] args)
    {
        double result;
1007    result = Function.power(2.0, 4);
        System.out.println("2^4 = " + result);
    }
}
```

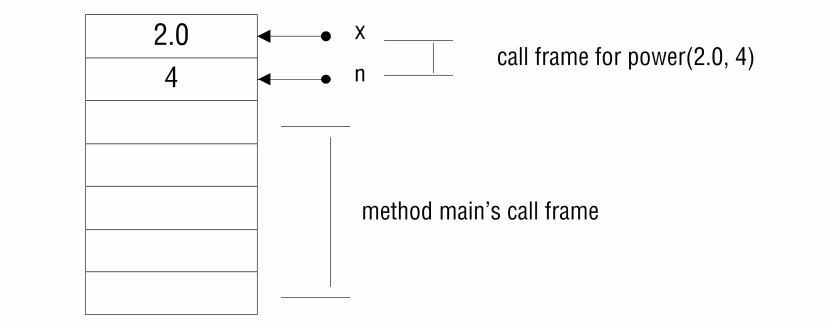


During its execution, method `power` accesses its arguments by their addresses on the run-time stack. When the method's execution ends, its call frame is popped off of the run-time stack and program execution continues at address 1007.

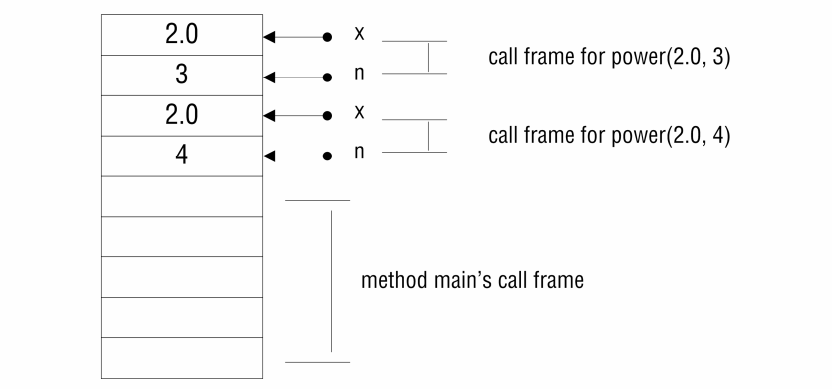
Observe that `power`'s call frame is stacked on top of `main`'s call frame. This stacking of call frames is what makes recursion possible. Let us see how by looking behind the scenes of a call to the recursive `power` method of class `Recurse`. Consider the call shown below.

```
double result = Recurse.power(2.0, 4);
```

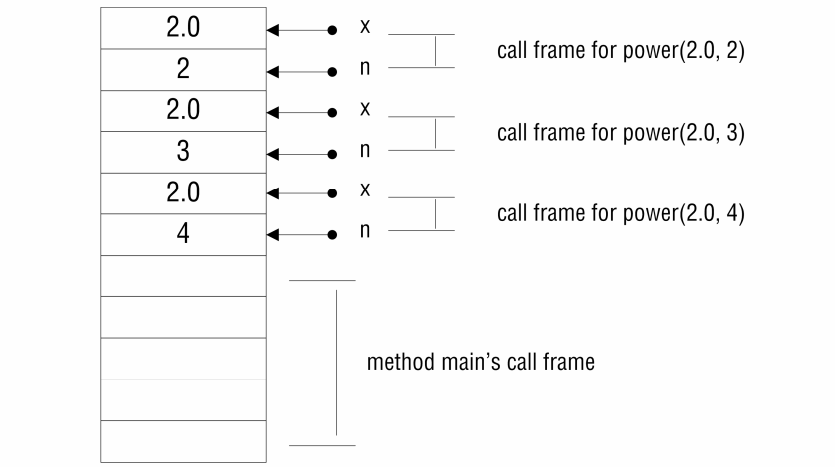
This initial, or outer, call to `power` results in the construction of the call frame shown below. The call's return address has been omitted.



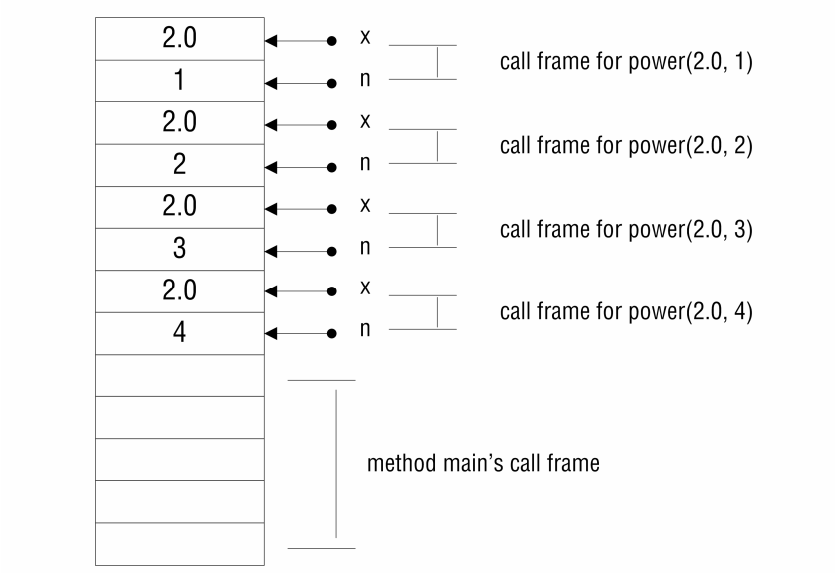
Because 4 > 0, we have not yet reached the base case. Thus the outer call to `power` does not return 1.0 but instead proceeds to the method's second return statement, which makes a second call to `power` with $n = 3$. The outer call cannot perform its multiplication or return its result until the second call returns its value, and so the call frame for the outer call remains on the run-time stack and the call frame for the second call is built on top of the first, as shown below.



Because 3 > 0, the base case is still irrelevant. Consequently, the second call to `power` makes a third call with $n = 2$. The second call cannot return its result until the third call returns. Thus the call frame for the third call is built on top of those from the first and second calls.

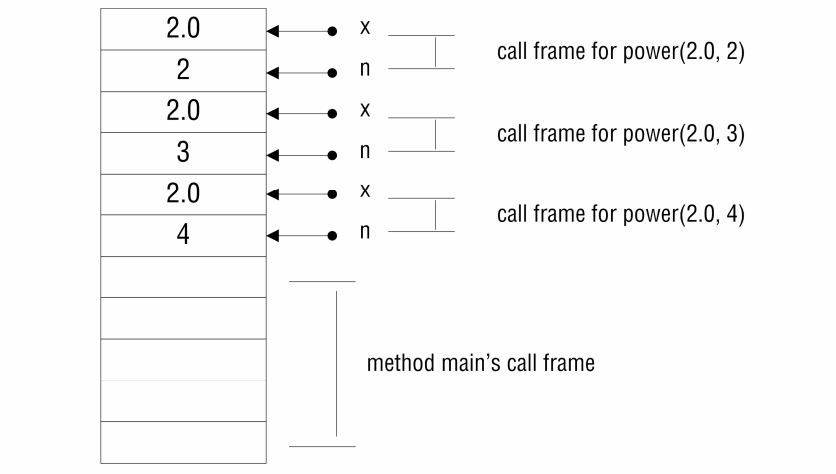


We still have not arrived at the base case, and so yet another call frame with $n = 1$ is stacked upon the preceding frames.



The call to `power` for which $n = 1$ now makes a call to `power` with $n = 0$. The call frame for `power(2.0, 0)` is placed on top of the others but it does not stay there for long because $n = 0$ is the base case. Because `power(2.0, 1)` is waiting for the return value from `power(2.0, 0)`, the return address for the final call is the return statement within `power(2.0, 1)`. Hence the call frame for `power(2.0, 0)` is popped from the run-time stack and the value 1.0 is returned to `power(2.0, 1)`. The run-time stack is again in the state shown in the last figure and `power(2.0, 1)` has what it needs to perform its multiplication and returns its value, 2.0.

Next, `power(2.0, 1)` returns its value to `power(2.0, 2)`, after having popped its call frame off of the run-time stack. This leaves the run-time stack in the state shown below and gives `power(2.0, 2)` what it needs to perform its multiplication and return its value, 4.0.



This process of resurfacing continues until the outer call has received the value 8.0 from the second call. The outer call can then compute its return value, remove its call frame from the run-time stack, and return 16.0 to the assignment statement that began the recursion. After the outer call has returned no remnants of the recursion remain on the run-time stack.

The run-time stack, unlike an instance of our `Stack` class, has a fixed size. If a recursive method makes too many recursive calls, the call frames will fill the run-time stack, causing a `java.lang.StackOverflowError` to be thrown. Stack overflow may occur if a recursive method's precondition is not met, if a recursive method lacks an appropriate base case, or if recursion goes too deep to be supported by the run-time stack despite the presence of an appropriate base case.

8.3 More Examples

Here we examine two more classic examples of recursive problem solving: the *factorial function* and the *Fibonacci numbers*.

8.3.1 The Factorial Function

The factorial function, which pervades mathematics, presents a nice opportunity for using recursion. Both recursive and non-recursive definitions of factorial appear below along with a recursive implementation.

$$n! = \begin{cases} 1, \text{if } n = 0 \\ n(n-1)(n-2)\cdot\ldots\cdot 3\cdot 2\cdot 1, \text{if } n > 0 \end{cases}$$

$$n! = \begin{cases} 1, \text{if } n = 0 \\ n(n-1)!, \text{if } n > 0 \end{cases}$$

```
public class Recurse
{
    /*
     precondition: The factorial function is undefined for
                   negative n. If a negative argument is passed
                   to this method, then the method will cause
                   stack overflow.
    postcondition: The factorial of n has been returned.
    */

    public static long factorial(int n)
    {
        if (n == 0)
            return 1;
        return n * factorial(n - 1);
    }
}
```

8.3.2 The Fibonacci Sequence

The *n*th Fibonacci number gives the number of pairs of rabbits *n* months after a single pair begins breeding, assuming that newborns begin breeding at two months of age[1]. The Fibonacci numbers also describe many other natural phenomena, and there is even a collection called the Fibonacci heap. The first few Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,....

[1] Eric W. Weisstein. "Fibonacci Number." From *MathWorld*--A Wolfram Web Resource. http://mathworld.wolfram.com/FibonacciNumber.html

The *n*th Fibonacci number is equal to the sum of its two predecessors. Thus the Fibonacci sequence is inherently recursive.

$$F(n) = \begin{cases} 1, \text{if } n = 1 \text{ or } n = 2 \\ F(n-1) + F(n-2), \text{if } n > 2 \end{cases}$$

The implementation appears below.

```
public class Recurse
{
    /*
     precondition: F(n) is defined only for positive n.
    postcondition: The nth Fibonacci number has been returned.
    */

    public static int Fibonacci(int n)
    {
        if (n == 1 || n == 2)
            return 1;
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    }
}
```

8.4 The Pros and Cons of Using Recursion

Recursion and looping are the two kinds of programmatic *iteration*, or repetition. Is recursion more powerful than looping? In other words, are there algorithms that can be implemented with recursion but not with loops? Conversely, is looping more powerful than recursion? The answer to both questions is no, for it can be shown that looping and recursion are equivalent, i.e., any algorithm that can be implemented with a loop can also be implemented with recursion, and vice versa. What, then, are the advantages of recursion, and does it have any drawbacks?

The primary advantage of recursion is that it often means less work for the programmer. Although loop implementations for `power`, `factorial`, and `Fibonacci` are not challenging to code, their recursive implementations are somewhat shorter and, at least on the surface, easier to understand. And there are algorithms that are quite easy to implement recursively but much more challenging to implement using loops. Traversing a binary tree is an example of such an algorithm, as we will see in Chapter 13.

Recursion also has drawbacks. Building and removing call frames is time consuming relative to the record keeping required by loops, and so a recursive method tends to execute slower than an equivalent looping method. Both the looping and the recursive implementations of `factorial`, for example, have linear running times, but the loop implementation is faster by a constant factor.

There is also the matter of stack overflow. The loop implementation of `Fibonacci` obviously cannot overflow the run-time stack, but the recursive version will overflow the stack for a disappointingly small value of *n*.

Exercises

1. Add the recursive methods specified below to class `Recurse`.

```
 precondition: n >= 0
postcondition: The method has printed "Go Lions" n times, followed
               by "Yay Team," followed by "Go Lions" n times.

cheer(PrintStream out, int n);

example:

cheer(System.out, 2);

Go Lions
Go Lions
Yay Team
Go Lions
Go Lions

 precondition: ch is in the range '0' through '9'.
postcondition: The method has printed a pattern of digits to the
               stream as follows:

               1. If ch is '0', the output is '0'.
               2. For other values of ch, the output consists of
                  a. the previous digit character, (ch - 1)
                  b. ch itself
                  c. the previous digit character

               There is no newline printed at the end of the
               output.

digits(PrintStream out, char ch);

example:

digits(System.out, '1');

010

digits(System.out, '3');

010201030102010
```

```
 precondition: m <= n
postcondition: The function has printed 2 * (n - m + 1) lines of
               output to the stream. The first line contains n
               asterisks, the next line contains n - 1 asterisks,
               and so on down to m asterisks. Then the pattern is
               repeated backwards, from m up to n.

triangle(PrintStream out, int m, int n);

example:

triangle(System.out, 2, 4);

****
***
**
**
***
****

 precondition: N/A
postcondition: The function has returned x^n.

double power(double x, int n);

example:

double result = power(2.0, 8);  // result is 256.0

result = power(2.0, -4);        // result is 1 / 16 = 0.0625

 precondition: 'size' is equal to the length of string 'str'.
postcondition: The characters of 'str' have been printed to the
               stream in reverse order.

revString(PrintStream out, String str, int size);

example:

revString(System.out, "Spiderman", 9);

namredipS
```

2. Implement the binary search algorithm using recursion.
3. Draw binary trees to show the recursive calls that result from calling method `Fibonacci` with $n = 3, 5, 6$, and 7. Conclude that the number of method calls is in $O(2^n)$.
4. On your system, what value of n causes method `Fibonacci` to overflow the run-time stack?
5. Is it possible to set the JVM stack size? If so, how is it done?