Eliminating Recursion

Some algorithms lend themselves to a recursive approach, some don’t. As we’ve seen, the recursive triangle() and factorial() methods can be implemented more efficiently using a simple loop. However, various divide-and-conquer algorithms, such as mergesort, work very well as recursive routines.

Often an algorithm is easy to conceptualize as a recursive method, but in practice the recursive approach proves to be inefficient. In such cases, it’s useful to transform the recursive approach into a non-recursive approach. Such a transformation can often make use of a stack.

Recursion and Stacks

There is a close relationship between recursion and stacks. In fact, most compilers implement recursion by using stacks. As we noted, when a method is called, the compiler pushes the arguments to the method and the return address (where control will go when the method returns) on the stack, and then transfers control to the method. When the method returns, it pops these values off the stack. The arguments disappear, and control returns to the return address.

Simulating a Recursive Method

In this section we’ll demonstrate how any recursive solution can be transformed into a stack-based solution. Remember the recursive triangle() method? Here it is again:

```java
int triangle(int n)
{
    if(n==1)
        return 1;
    else
        return( n + triangle(n-1) );
}
```

We’re going to break this algorithm down into its individual operations, making each operation one case in a switch statement. (You can perform a similar decomposition using goto statements in C++ and some other languages, but Java doesn’t support goto.)

The switch statement is enclosed in a method called step(). Each call to step() causes one case section within the switch to be executed. Calling step() repeatedly will eventually execute all the code in the algorithm.

The triangle() method we just saw performs two kinds of operations. First, it carries out the arithmetic necessary to compute triangular numbers. This involves checking if n is 1, and adding n to the results of previous recursive calls. However, triangle() also performs the operations necessary to manage the method itself, including transfer of control, argument access, and the return address. These operations are not visible by looking at the code; they’re built into all methods. Here, roughly speaking, is what happens during a call to a method:

- When a method is called, its arguments and the return address are pushed onto a stack.
- A method can access its arguments by peeking at the top of the stack.
- When a method is about to return, it peeks at the stack to obtain the return address, and then pops both this address and its arguments off the stack and discards them.

The stackTriangle.java program contains three classes: Params, StackX, and StackTriangleApp. The Params class encapsulates the return address and the method’s argument, n; objects of this class are pushed onto the stack. The StackX class is similar to those in other chapters, except that it holds objects of class Params. The StackTriangleApp class contains four methods: main(), recTriangle(), step(), and the usual getInt() method for numerical input.

The main() routine asks the user for a number, calls the recTriangle() method to calculate the triangular number corresponding to n, and displays the result.

The recTriangle() method creates a StackX object and initializes codePart to 1. It then settles into a while loop, where it repeatedly calls step(). It won’t exit from the loop until step() returns true by reaching case 6, its exit point. The step() method is basically a large switch statement in which each case corresponds to a section of code in the original triangle() method.

// stackTriangle.java
import java.io.*; // for I/O

// evaluates triangular numbers, stack replaces recursion

class Params // parameters to save on stack
{
    public int n;
    public int returnAddress;
    public Params(int nn, int ra)
    {
        n=nn;
        returnAddress=ra;
    }
} // end class Params

class StackX
{
    private int maxSize; // size of StackX array
    private Params[] stackArray;
    private int top; // top of stack

    public StackX(int s) // constructor
    {
        maxSize = s; // set array size
        stackArray = new Params[maxSize]; // create array
        top = -1; // no items yet
    }

    public void push(Params p) // put item on top of stack
    {
        stackArray[++top] = p; // increment top, insert item
    }

    public Params pop() // take item from top of stack
    {
        return stackArray[top--]; // access item, decrement top
    }

    public Params peek() // peek at top of stack
    {
        return stackArray[top];
    }
} // end class StackX

class StackTriangleApp
{
    static int theNumber;
    static int theAnswer;
    static StackX theStack;
    static int codePart;
    static Params theseParams;

    public static void main(String[] args) throws IOException
    {
        System.out.print("Enter a number: ");
        theNumber = getInt();
    }
}
recTriangle();
System.out.println("Triangle="+theAnswer);
} // end main()

public static void recTriangle()
{
    theStack = new StackX(10000);
    codePart = 1;
    while( step() == false) // call step() until it’s true
    ; // null statement
}

public static boolean step()
{
    switch(codePart)
    {
    case 1: // initial call
        theseParams = new Params(theNumber, 6);
        theStack.push(theseParams);
        codePart = 2;
        break;
    case 2: // method entry
        theseParams = theStack.peek();
        if(theseParams.n == 1) // test
        {
            theAnswer = 1;
            codePart = 5; // exit
        }
        else
            codePart = 3; // recursive call
        break;
    case 3: // method call
        Params newParams = new Params(theseParams.n - 1, 4);
        theStack.push(newParams);
        codePart = 2; // go enter method
        break;
    case 4: // calculation
        theseParams = theStack.peek();
        theAnswer = theAnswer + theseParams.n;
        codePart = 5;
        break;
    case 5: // method exit
        theseParams = theStack.peek();
        codePart = theseParams.returnAddress; // (4 or 6)
        theStack.pop();
        break;
    case 6: // return point
        return true;
    } // end switch
    return false;
}

public static String getString() throws IOException
{
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
String s = br.readLine();
return s;
}

public static int getInt() throws IOException
{
    String s = getString();
    return Integer.parseInt(s);
}

This program calculates triangular numbers, just as the triangle.java program. Here’s some sample output:
Enter a number: 100
Triangle=5050

Figure shows how the sections of code in each case relate to the various parts of the algorithm.

The program simulates a method, but it has no name in the listing because it isn’t a real Java method. Let’s call this simulated method simMeth(). The initial call to simMeth() (at case 1) pushes the value entered by the user and a return value of 6 onto the stack and moves to the entry point of simMeth() (case 2).
At its entry (case 2), simMeth() tests whether its argument is 1. It accesses the argument by peeking at the top of the stack. If the argument is 1, this is the base case and control goes to simMeth()’s exit (case 5). If not, it calls itself recursively (case 3).

This recursive call consists of pushing n-1 and a return address of 4 onto the stack, and going to the method entry at case 2.

On the return from the recursive call, simMeth() adds its argument n to the value returned from the call. Finally, it exits (case 5). When it exits, it pops the last Params object off the stack; this information is no longer needed.

The return address given in the initial call was 6, so case 6 is the place where control goes when the method returns. This code returns true to let the while loop in recTriangle() know that the loop is over.

Note that in this description of simMeth()’s operation we use terms like argument, recursive call, and return address to mean simulations of these features, not the normal Java versions.

If you inserted some output statements in each case to see what simMeth() was doing, you could arrange for output like this:

Enter a number: 4
case 1. theAnswer=0 Stack:
case 2. theAnswer=0 Stack: (4, 6)
case 3. theAnswer=0 Stack: (4, 6)
case 2. theAnswer=0 Stack: (4, 6) (3, 4)
case 3. theAnswer=0 Stack: (4, 6) (3, 4)
case 2. theAnswer=0 Stack: (4, 6) (3, 4) (2, 4)
case 3. theAnswer=0 Stack: (4, 6) (3, 4) (2, 4)
case 2. theAnswer=0 Stack: (4, 6) (3, 4) (2, 4) (1, 4)
case 5. theAnswer=1 Stack: (4, 6) (3, 4) (2, 4) (1, 4)
case 4. theAnswer=1 Stack: (4, 6) (3, 4) (2, 4)
case 5. theAnswer=3 Stack: (4, 6) (3, 4) (2, 4)
case 4. theAnswer=3 Stack: (4, 6) (3, 4)
case 5. theAnswer=6 Stack: (4, 6) (3, 4)
case 4. theAnswer=6 Stack: (4, 6)
case 5. theAnswer=10 Stack: (4, 6)
case 6. theAnswer=10 Stack:
Triangle=10

The case number shows what section of code is being executed. The contents of the stack (consisting of Params objects containing n followed by a return address) are also shown. The simMeth() method is entered four times (case 2) and returns four times (case 5). Only when it starts returning does theAnswer begin to accumulate the results of the calculations.

**What Does This Prove?**

In stackTriangle.java (Listing 6.7) we have a program that more or less systematically transforms a program that uses recursion into a program that uses a stack. This suggests that such a transformation is possible for any program that uses recursion, and in fact this is the case.

With some additional work, you can systematically refine the code we show here, simplifying it and even eliminating the switch statement entirely to make the code more efficient.
In practice, however, it’s usually more practical to rethink the algorithm from the beginning, using a stack-based approach instead of a recursive approach.

```java
// stackTriangle2.java
// evaluates triangular numbers, stack replaces recursion
import java.io.*; // for I/O

class StackX
{
    private int maxSize; // size of stack array
    private int[] stackArray;
    private int top; // top of stack

    public StackX(int s) // constructor
    {
        maxSize = s;
        stackArray = new int[maxSize];
        top = -1;
    }

    public void push(int p) // put item on top of stack
    { stackArray[++top] = p; }

    public int pop() // take item from top of stack
    { return stackArray[top--]; }

    public int peek() // peek at top of stack
    { return stackArray[top]; }

    public boolean isEmpty() // true if stack is empty
    { return (top == -1); }
}

class StackTriangle2App
{
    public static void main(String[] args) throws IOException
    {
        System.out.print("Enter a number: ");
        theNumber = getInt();
        stackTriangle();
        System.out.println("Triangle="+theAnswer);
    }

    public static void stackTriangle()
    {
    }
```
theStack = new StackX(10000); // make a stack
theAnswer = 0; // initialize answer
while(theNumber > 0) // until n is 1,
{
    theStack.push(theNumber); // push value
    --theNumber; // decrement value
}
while( !theStack.isEmpty() ) // until stack empty,
{
    int newN = theStack.pop(); // pop value,
    theAnswer += newN; // add to answer
}

public static String getString() throws IOException
{
    InputStreamReader isr = new InputStreamReader(System.in);
    BufferedReader br = new BufferedReader(isr);
    String s = br.readLine();
    return s;
}

public static int getInt() throws IOException
{
    String s = getString();
    return Integer.parseInt(s);
}

} // end class StackTriangle2App

Here two short while loops in the stackTriangle() method substitute for the entire step() method of the stackTriangle.java program. Of course, in this program you can see by inspection that you can eliminate the stack entirely and use a simple loop. However, in more complicated algorithms the stack must remain. Often you’ll need to experiment to see whether a recursive method, a stack-based approach, or a simple loop is the most efficient (or practical) way to handle a particular situation.