Think Java

CHAPTER 8: RECURSIVE METHODS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Up to this point, we've been using while and for loops whenever we've needed to repeat something. Methods that use iteration are called iterative. They are straight-forward, but sometimes there are more elegant solutions.
- In this chapter, we will explore one of the most magical things that a **method** can do: invoke itself to solve a smaller version of the same problem. **A method that invokes itself is called recursive.**

Topics

- What is a recursive method?
- Base Case
- Recursive Pattern

LECTURE 1

Recursive void methods

A Simple Recursive Method – countdown()

- Consider the following example:

```
public static void countdown(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        System.out.println(n);
        countdown(n - 1);
    }
}
```

- The name of the method is countdown; it takes a single integer as a parameter. If the parameter is zero, it displays the word “Blastoff”. Otherwise, it displays the number and then invokes itself, passing n - 1 as the argument.

What happens if we invoke countdown(3) from main?

The execution of countdown begins with n == 3, and since n is not zero, it displays the value 3, and then invokes itself...
The execution of countdown begins with n == 2, and since n is not zero, it displays the value 2, and then invokes itself...
The execution of countdown begins with n == 1, and since n is not zero, it displays the value 1, and then invokes itself...
The execution of countdown begins with n == 0, and since n is zero, it displays the word "Blastoff!" and then returns.
The countdown that got n == 1 returns.
The countdown that got n == 2 returns.
The countdown that got n == 3 returns.

coundDown

- And then you're back in main. So the total output looks like:

```
3
2
1
Blastoff!
```

newLine() and threeLine()

- As a second example, we'll rewrite the methods **newLine** and **threeLine** from Section 4.3. Here they are again:

```
public static void newLine() {
    System.out.println();
}
public static void threeLine() {
    newLine();
    newLine();
    newLine();
}
```

nLines()

Although these methods work, they would not help if we wanted to display two newlines, or maybe 100. A more general alternative would be:

```
public static void nLines(int n) {
    if (n > 0) {
        System.out.println();
        nLines(n - 1);
    }
}
```

This method takes an integer, **n**, as a parameter and displays n newlines. The structure is similar to countdown. As long as **n** is greater than zero, it displays a newline and then invokes itself to display (n−1) additional newlines. The total number of newlines is 1 + (n − 1), which is just what we wanted: **n**.

LECTURE 2

Recursive stack diagrams

Stack Diagram for countDown

- In the Section 4.6, we used a stack diagram to represent the state of a program during a method invocation. The same kind of diagram can make it easier to interpret a recursive method.
- Remember that every time a method gets called, Java creates a new frame that and variables. Figure 8.1 is a stackcontains the current method's parameters diagram for countdown, called with n == 3.

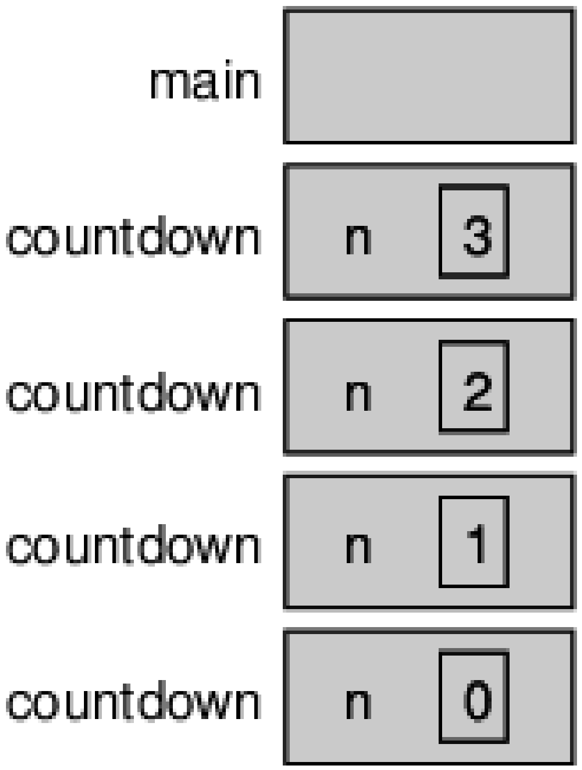


Figure 8.1: Stack diagram for the countdown program.

Stack Diagram for countdown – Call Frame

By convention, the frame for main is at the top, and the stack of other frames grows down. That way, we can draw stack diagrams on paper without needing to guess how far they will grow. The **frame** for main is empty because main does not have any variables. (It has the parameter args, but since we're not using it, we left it out of the diagram.)

There are four frames for countdown, each with a different value for the parameter n. The last frame, with n == 0, is called the base case. It does not make a recursive call, so there are no more frames below it.

StackOverflowError

- If there is no base case in a recursive method, or if the base case is never reached, the stack would grow forever – at least in theory. In practice, the size of the stack is limited. If you exceed the limit, you get a **StackOverflowError**.
- For example, here is a recursive method without a base case:

```
public static void forever(String s) {
    System.out.println(s);
    forever(s);
}
```

- This method displays the string until the stack overflows, at which point it throws an error. Try this example on your computer – you might be surprised by how long the error message is!

LECTURE 3

Value returning methods

Recursion

- To give you an idea of what you can do with the tools we have learned, let's look at methods that evaluate recursively-defined mathematical functions.
- A recursive definition is similar to a "circular" definition, in the sense that the definition refers to the thing being defined. Of course, a truly circular definition is not very useful:

  recursive:

  An adjective used to describe a method that is recursive.
- If you saw that definition in the dictionary, you might be annoyed. Then again, if you search for "recursion" on Google, it displays "Did you mean: recursion" as an inside joke. People fall for that link all the time.

Read the Recursive Pattern

- Many mathematical functions are defined recursively, because that is often the simplest way. For example, the factorial of an integer n, which is written n!, is defined like this:

$$0! = 1$$

$$n! = n \cdot (n-1)!$$

- Don’t confuse the mathematical symbol !, which means factorial, with the Java operator !, which means not. This definition says that factorial(0) is 1, and that factorial(n) is n * factorial(n - 1).

In-Class Demo Program

- Base case
- Recursive Formula

FACTORIAL.JAVA

Factorial

- So factorial(3) is 3 * factorial(2); factorial(2) is 2 * factorial(1); factorial(1) is 1 * factorial(0); and factorial(0) is 1. Putting it all together, we get 3 * 2 * 1 * 1, which is 6.
- If you can formulate a recursive definition of something, you can easily write a Java method to evaluate it. The first step is to decide what the parameters and return type are. Since factorial is defined for integers, the method takes an int as a parameter and returns an int.

Add Base Case

```
public static int factorial(int n) {
    return 0;  // stub
}
```

- Next, we think about the base case. If the argument happens to be zero, we return 1.

```
public static int factorial(int n) {
    if (n == 0) {
        return 1;
    }
    return 0;  // stub
}
```

Add The Recursion

- Otherwise, and this is the interesting part, we have to make a recursive call to find the factorial of n−1, and then multiply it by n.

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

- To illustrate what is happening, we'll use the temporary variables recurse and result. In each method call, recurse stores the factorial of n − 1, and result stores the factorial of n.

Execution Flow for Factorial

- The flow of execution for this program is similar to countdown from Section 8.1. If we invoke factorial with the value 3:

```
Since 3 is not zero, we skip the first branch and calculate the
factorial of n-1...
Since 2 is not zero, we skip the first branch and calculate the
factorial of n-1...
Since 1 is not zero, we skip the first branch and calculate the
factorial of n-1...
Since 0 is zero, we take the first branch and return the value 1
immediately.
The return value (1) gets multiplied by n, which is 1, and the result is
returned.
The return value (1) gets multiplied by n, which is 2, and the result is
returned.
The return value (2) gets multiplied by n, which is 3, and the result,
6, is returned to whatever invoked factorial(3).
```

Stack diagram

- Figure 8.2 shows what the stack diagram looks like for this sequence of method invocations. The return values are shown being passed up the stack. Notice that recurse and result do not exist in the last frame, because when n == 0 the code that declares them does not execute.

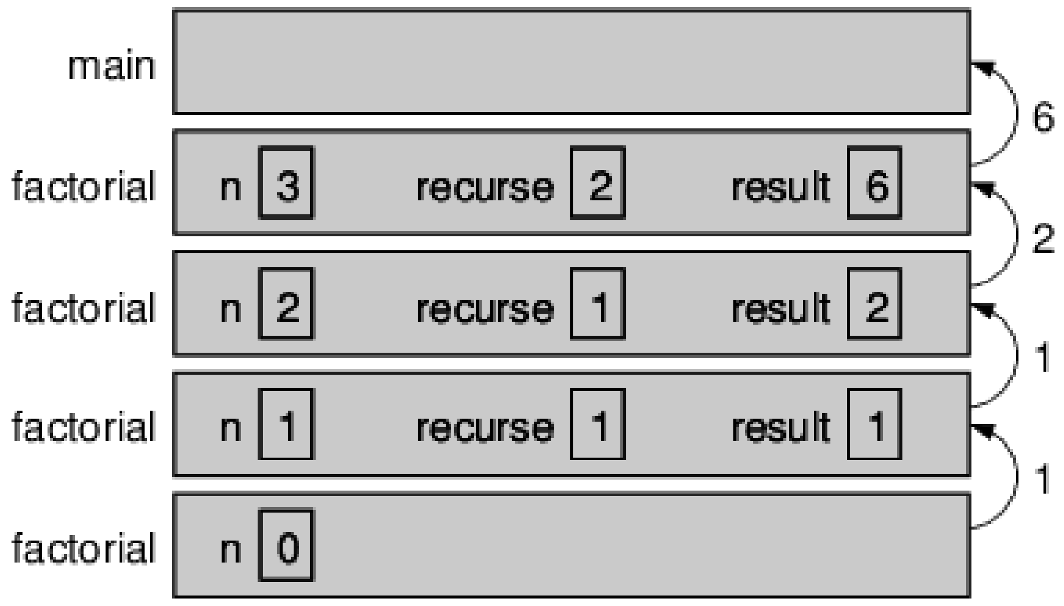


Figure 8.2: Stack diagram for the factorial method.

LECTURE 4

The leap of faith

Leap of Faith

- Following the flow of execution is one way to read programs, but it can quickly become overwhelming. An alternative way to understand recursion is the **leap of faith**: when you come to a method invocation, instead of following the flow of execution, you assume that the method works correctly and returns the appropriate value.
- In fact, you are already practicing this leap of faith when you use methods in the Java library. When you invoke **Math.cos** or **System.out.println**, you don't examine or think about the implementations of those methods. You just assume that they work properly.

Abstraction

- The same is true of other methods. For example, consider the method from Section 5.7 that determines whether an integer has only one digit:

```
public static boolean isSingleDigit(int x) {
    return x > -10 && x < 10;
}
```

- Once you convince yourself that this method is correct – by examining and testing the code – you can just use the method without ever looking at the implementation again.

Leap of Faith

- Recursive methods are no different. When you get to a recursive call, don't try to follow the flow of execution. Instead, you should assume that the recursive call produces the desired result.
- For example, "Assuming that I can find the factorial of n−1, can I compute the factorial of n?" Yes you can, by multiplying by n.

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

Leap of Faith

- Notice how similar this implementation (with the temporary variables removed) is to the original mathematical definition. There is essentially a **one-to-one** correspondence.

$$0! = 1$$

$$n! = n \cdot (n-1)!$$

- Of course, it is strange to assume that the method works correctly when you have not finished writing it. But that's why it's called the **leap of faith**!

Recursive Fibonacci Function

- Another common recursively-defined mathematical function is the Fibonacci sequence, which has the following definition:

```
fibonacci(1) = 1
fibonacci(2) = 1
fibonacci(n) = fibonacci(n−1) + fibonacci(n−2)
```

- Translated into Java, this function is:

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

Leap of Faith

- If you try to follow the flow of execution here, even for small values of n, your head will explode. But if we take a leap of faith and assume that the two recursive invocations work correctly, then it is clear, looking at the definition, that our implementation is correct.

LECTURE 5

Binary number system

Binary Numbers

- Before introducing the next recursive example, we need to discuss how integers are represented by a computer.
- You are probably aware that computers can only store 1's and 0's. That's because, at the end of the day, processors and memory are made up of billions of tiny on-off switches.
- The value 1 means a switch is on; the value 0 means a switch is off. All types of data, whether integer, floating-point, text, audio, video, or something else, need to be represented by 1's and 0's.
- Fortunately, mathematicians solved this problem centuries ago. We can represent any integer as a binary number. The following table shows the first eight numbers in binary and decimal (the number system we normally use).

Binary	Decimal
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7

Table 8.1: The first eight binary numbers.

Digits and Bits

- In the decimal system, each part of a number is referred to as a “digit”. For example, the number 456 has three digits. In the binary system, each part of a number is referred to as a “bit”. The number 10111 in binary has five bits.
- When you hear the phrase “64-bit computer”, it means that the processors and memory use 64 bits to store integers. That is where the limits for data types like int and long come from.

Weighted Digits

- Decimal numbers are based on powers of 10, because there are 10 possible values for each digit. For example, the number 456 has 6 in the 1's place, 5 in the 10's place, and 4 in the 100's place. So the value is 400 + 50 + 6.

4	5	6
10^2	10^1	10^0

Binary Numbers

- Binary numbers are based on powers of 2, because there are 2 possible values for each bit. For example, the number 10111 has 1 in the 1's place, 1 in the 2's place, 1 in the 4's place, 0 in the 8's place, and 1 in the 16's place. So the value is 16 + 0 + 4 + 2 + 1, which is 23 in decimal.

1	0	1	1	1
2^4	2^3	2^2	2^1	2^0

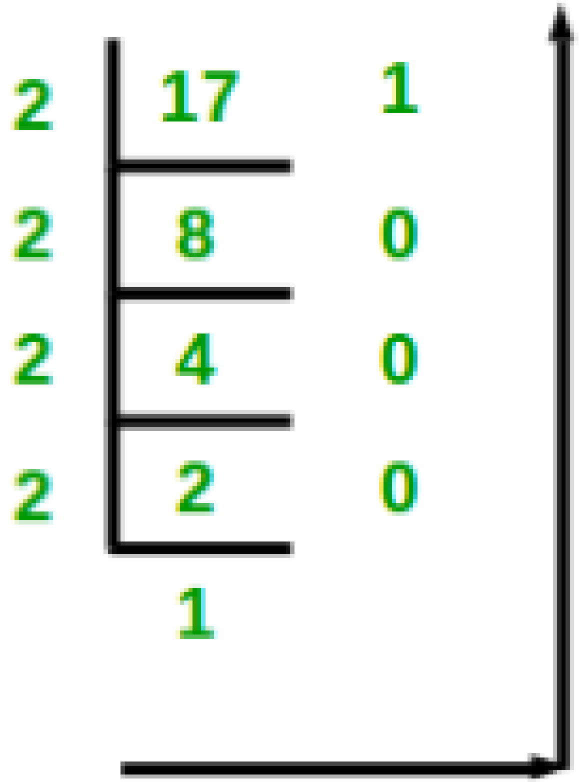
Recursive Generation of Bits

- To convert from decimal to binary, we simply need to divide the number by two repeatedly until we reach zero. When you divide by two, the remainder will be either 0 or 1. If you keep track of the remainders, you'll have your binary number.

```
23 / 2 is 11 remainder 1
11 / 2 is  5 remainder 1
 5 / 2 is  2 remainder 1
 2 / 2 is  1 remainder 0
 1 / 2 is  0 remainder 1
```

- Reading these remainders from bottom to top, 23 in binary is 10111

Decimal number : 17



Binary number: 10001

In-Class Demo Program

- Base case
- Recursive Formula

BINARY.JAVA

LECTURE 6

Recursive binary method

countup()

- At the start of the chapter, the countdown example had three parts: (1) it checked the base case, (2) displayed something, and (3) made a recursive call. What do you think happens if you reverse steps 2 and 3, making the recursive call before displaying?

```
public static void countup(int n) {
    if (n == 0) {
        System.out.println("Blastoff!");
    } else {
        countup(n - 1);
        System.out.println(n);
    }
}
```

countup()

- The stack diagram is the same as before, and the method is still called n times. But now the **System.out.println** happens just before each recursive call returns. As a result, it counts up instead of down:

```
Blastoff!
1
2
3
```

displayBinary()

- We can apply this idea to solve our binary conversion problem. Here is a recursive method that displays the binary value of any positive integer:

```
public static void displayBinary(int value)
{
    if (value > 0) {
        displayBinary(value / 2);
        System.out.print(value % 2);
    }
}
```

Stack Diagram for displayBinary

- If value is zero, displayBinary does nothing (that's the base case). If the argument is positive, the method divides it by two and calls displayBinary recursively.
- When the recursive call returns, the method displays one digit of the result and returns (again). Figure 8.3 illustrates this process.

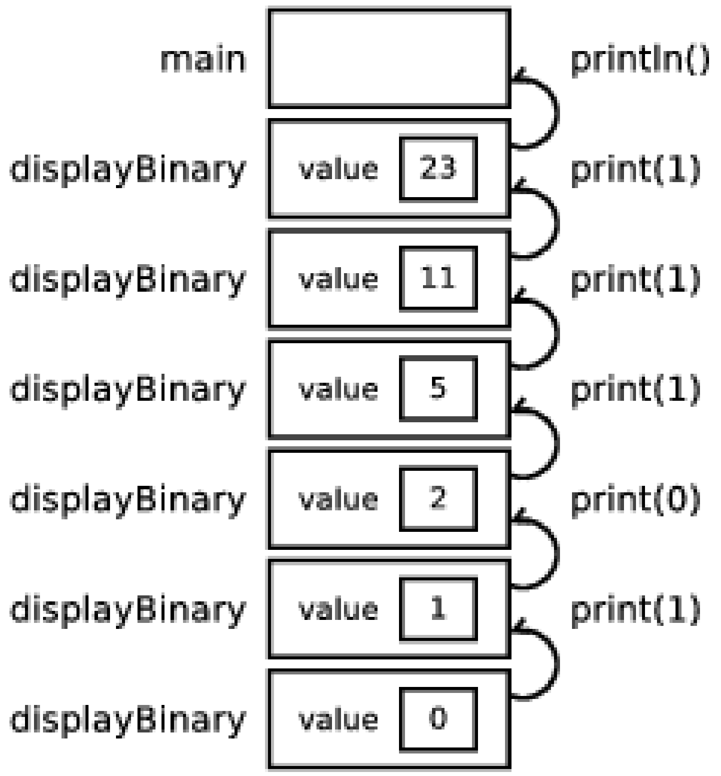


Figure 8.3: Stack diagram for the displayBinary method.

displayBinary()

- The leftmost digit is near the bottom of the stack, so it gets displayed first. The rightmost digit, near the top of the stack, gets displayed last. After invoking **displayBinary**, we use println to complete the output.

```
displayBinary(23);
System.out.println();
// output is 10111
```

LECTURE 7

CodingBat problems

codingBat

- In the past several chapters of this book, you've seen conditions, methods, loops, strings, arrays, and recursion. A great resource for practicing all of these concepts is `CodingBat.com`.
- CodingBat is a free website of live programming problems developed by Nick Parlante, a Computer Science lecturer at Stanford. As you work on these problems, CodingBat will save your progress (if you create an account).
- To conclude this chapter, we will look at two problems in the Recursion-1 section of CodingBat. One of them deals with strings, and the other deals with arrays. Both of them have the same recursive idea: check the base case, look at the current index, and recursively handle the rest.
- The first problem is available at `http://codingbat.com/prob/p118230`:

Lab

NOX AND ARRAY11 ON CODINGBAT

Recursion-1 noX

- Given a string, compute recursively a new string where all the ’x’ chars have been removed.

```
noX("xaxb") → "ab"
noX("abc") → "abc"
noX("xx") → ""
```

noX

- When solving recursive problems, it helps to think about the base case first. The base case is the easiest version of the problem; for **noX**, it’s when you’re given the empty string. If the string is empty, there are no x’s to be removed.

```
if (str.length() == 0) {
    return "";
}
```

- Next comes the more difficult part. To solve a problem recursively, you need to think of a simper instance of the same problem. For noX, it’s removing all the x’s from a shorter string.

noX

- To find an x, we only need to look at one character. So we can recursively call noX on the rest of the string (the substring at index 1). Here is the solution:

```
char c = str.charAt(0);
if (c == 'x') {
    return noX(str.substring(1));
} else {
    return c + noX(str.substring(1));
}
```

- The else block “saves” the character if it’s not an x. Otherwise, the x is “removed” by the first returnstatement.

Recursion-2 array11

- The second problem is available at http://codingbat.com/prob/p135988:

- Given an array of ints, compute recursively the number of times that the value 11 appears in the array.

```
array11([1, 2, 11], 0) → 1
array11([11, 11], 0) → 2
array11([1, 2, 3, 4], 0) → 0
```

array11

- This problem uses the convention of passing the index as an argument. So the base case is when we’ve reached the end of the array. At that point, we know there are no more 11’s.

```
if (index >= nums.length) {
    return 0;
}
```

Recursion-2 array11

- Next we look at the current number (based on the given index), and check if it's an 11. After that, we can recursively check the rest of the array. Similar to the noX problem, we only look at one integer per method call.

```
if (nums[index] == 11) {
    return 1 + array11(nums, index + 1);
} else {
    return array11(nums, index + 1);
}
```

array11

- You can run these solutions on CodingBat by pasting them into the provided method definition. But don't forget to paste both parts: the base case, and the recursive step.
- To see how these solutions actually work, you might need to step through them with a debugger (see Appendix A.6) or Java Tutor (http://pythontutor.com/java.html). Then try to solve several CodingBat problems of your own.
- Learning to think recursively is an important aspect of learning to think like a computer scientist. Many algorithms can be written concisely with recursive methods that perform computations on the way down, on the way up, or both

Homework

Homework

- Chapter Exercise: all exercise problems.
- Project 8