Data Structures

Recursion

Professor: Evan Korth
New York University

Road Map

- Introduction to Recursion
- Recursion Example #1: World's Simplest Recursion Program
- Visualizing Recursion
  - Using Stacks
- Recursion Example #2
- Computing Factorials
  - Iterative Approach
- Computing Factorials
  - Recursive Approach
- Reading, recursion is covered in **Chapter 5 of our book** but we are only covering a fraction of that material today

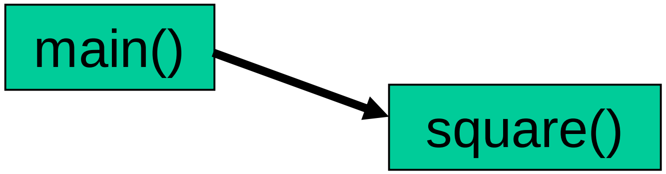
Reading Schedule

- Review of 101 material
  - Chapters 1, 2 and 5 (recursion)
- Asymptotic Analysis (Tuesday)
  - Chapter 4
- Linked lists (Thursday and the following Tuesday)
  - Chapter sections 3.1, 3.2 and 3.4
- Recitation stars tomorrow

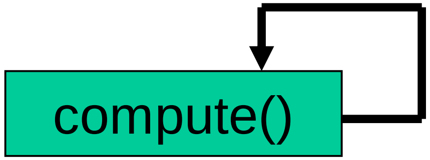
Introduction to Recursion

Introduction to Recursion

- "Normally", we have methods that call other methods.
  - For example, the `main()` method calls the square`()` method.



- Recursive Method:
  - A recursive method is a method that calls itself.



Why use Recursive Methods?

- In computer science, some problems are more easily solved by using recursive methods.
- In this course, will see many of examples of this.
- For example:
  - Traversing through directories of a file system.
  - Traversing through a tree of search results.
  - Some sorting algorithms recursively sort data
- For today, we will focus on the basic structure of using recursive methods.

World's Simplest Recursion Program

World’s Simplest Recursion Program

```
public class Recursion
{
    public static void main (String args[])
    {
        count(0);
        System.out.println();
    }

    public static void count (int index)
    {
        System.out.print(index);
        if (index < 2)
            count(index+1);
    }
}
```

This program simply counts from 0-2:

012

This is where the recursion occurs. You can see that the count() method calls itself.

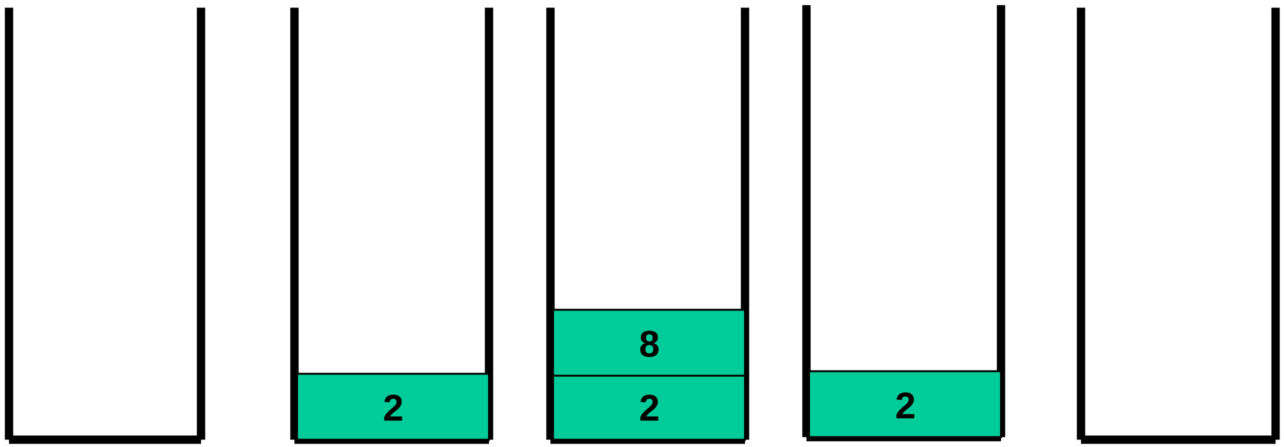
Visualizing Recursion

- To understand how recursion works, it helps to visualize what's going on.
- To help visualize, we will use a common concept called the *Stack*.
- A stack basically operates like a container of trays in a cafeteria. It has only two operations:
  - Push: you can push something onto the stack.
  - Pop: you can pop something off the top of the stack.
- Let's see an example stack in action.

Stacks

The diagram below shows a stack over time.

We perform two pushes and pops.

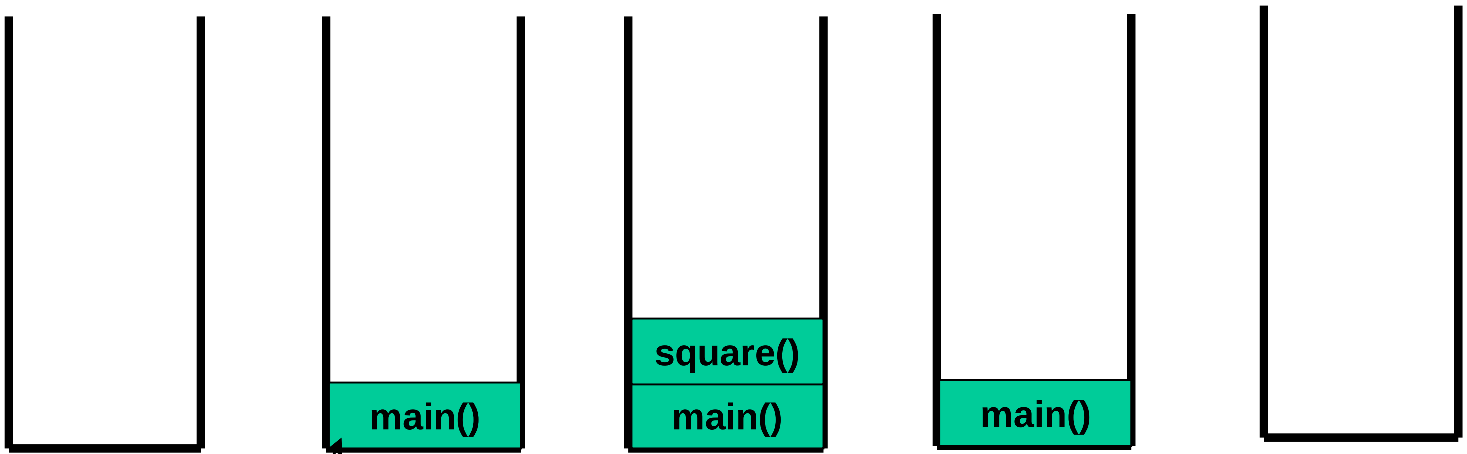


Time: 0	Time 1:	Time 2:	Time 3:	Time 4:
Empty Stack	Push "2"	Push "8"	Pop: Gets 8	Pop: Gets 2

Stacks and Methods

- When you run a program, the computer creates a stack for you. This is called the program stack.
- Each time you invoke a method, the method's activation record is placed on top of the stack.
- When the method returns or exits, the method is popped off the stack.
- The diagram on the next page shows a sample stack for a simple Java program.

Stacks and Methods



Time: 0

Empty Stack

Time 1:

Push: main()

Time 2:

Push: square()

Time 3:

Pop: square()

returns a value.

method exits.

Time 4:

Pop: main()

returns a value.

method exits.

This is called an activation record or stack frame. Usually, this actually grows downward.

Stacks and Recursion

- Each time a method is called, you *push* the method on the stack.
  - Save the position in the calling method
  - Push the called method's activation frame on the stack
  - Begin execution of the called method
- Each time the method returns or exits, you *pop* the method off the stack.
  - Pop the current method off the stack
  - Continue execution of the method which called it
- If a method calls itself recursively, you just push another copy of the method onto the stack.
- We therefore have a simple way to visualize how recursion really works.

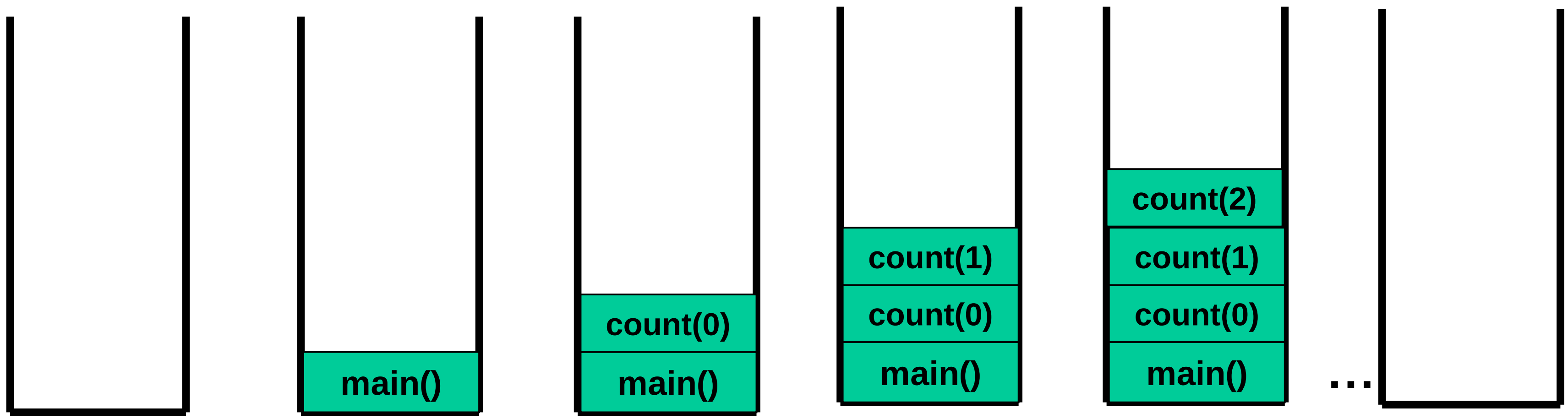
Back to the Simple Recursion Program

- Here's the code again. Now, that we understand stacks, we can visualize the recursion.

```
public class Recursion1V0
{
    public static void main (String args[])
    {
        count(0);
        System.out.println();
    }

    public static void count (int index)
    {
        System.out.print(index);
        if (index < 2)
            count(index+1);
    }
}
```

Stacks and Recursion in Action



Time: 0

Empty Stack

Time 1:

Push: main()

Time 2:

Push: count(0)

Time 3:

Push: count(1)

Time 4:

Push: count(2)

Times 5-8:

Pop everything

Inside count(0):
print (index); → 0
if (index < 2)
    count(index+1);

Inside count(1):
print (index); → 1
if (index < 2)
    count(index+1);

Inside count(2):
print (index); → 2
if (index < 2)
    count(index+1);

This condition now fails!

Hence, recursion stops, and we proceed to pop all methods off the stack.

Recursion, Variation 1

What will the following program do?

```
public class Recursion1V1
{
    public static void main (String args[])
    {
        count(3);
        System.out.println();
    }

    public static void count (int index)
    {
        System.out.print(index);
        if (index < 2)
            count(index+1);
    }
}
```

Recursion, Variation 2

What will the following program do?

```
public class Recursion1V2
{
    public static void main (String args[])
    {
        count(0);
        System.out.println();
    }

    public static void count (int index)
    {
        if (index < 2)
            count(index+1);
        System.out.print(index);
    }
}
```

Note that the print statement has been moved to the end of the method.

Recursion, Variation 3

What will the following program do?

```
public class Recursion1V0
{
    public static void main (String args[])
    {
        count(3);
        System.out.println();
    }

    public static void count (int index)
    {
        System.out.print(index);
        if (index > 2)
            count(index+1);
    }
}
```

First two rules of recursion

- **Base case**: You must always have some base case which can be solved without recursion
- **Making Progress**: For cases that are to be solved recursively, the recursive call must always be a case that makes progress toward the base case.

Problem: Not working towards base case

- In variation #3, we do not work towards our base case. This causes infinite recursion and will cause our program to crash.
- Java throws a StackOverflowError error.

Recursion Example #2

Recursion Example #2

```
public class Recursion2
{
    public static void main (String args[])
    {
        upAndDown(1);
        System.out.println();
    }

    public static void upAndDown (int n)
    {
        System.out.print ("\nLevel: " +  n);
        if (n < 3)
            upAndDown (n+1);
        System.out.print ("\nLEVEL: " + n);
    }
}
```

Recursion occurs here.

Computing Factorials

Factorials

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;

- For example:

  1! = 1

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

If you study this table closely, you will start to see a pattern.

Seeing the Pattern

- Seeing the pattern in the factorial example is difficult at first.
- But, once you see the pattern, you can apply this pattern to create a recursive solution to the problem.
- Divide a problem up into:
  - What we know (call this the base case)
  - Making progress towards the base
    - Each step resembles original problem
    - The method launches a new copy of itself (recursion step) to make the progress.

Factorials

- Computing factorials are a classic problem for examining recursion.
- A factorial is defined as follows:

  n! = n * (n-1) * (n-2) …. * 1;

- For example:

  1! = 1 (Base Case)

  2! = 2 * 1 = 2

  3! = 3 * 2 * 1 = 6

  4! = 4 * 3 * 2 * 1 = 24

  5! = 5 * 4 * 3 * 2 * 1 = 120

If you study this table closely, you will start to see a pattern.

The pattern is as follows:

You can compute the factorial of any number (n) by taking n and multiplying it by the factorial of (n-1).

For example:

5! = 5 * 4!

(which translates to 5! = 5 * 24 = 120)

Recursive Solution

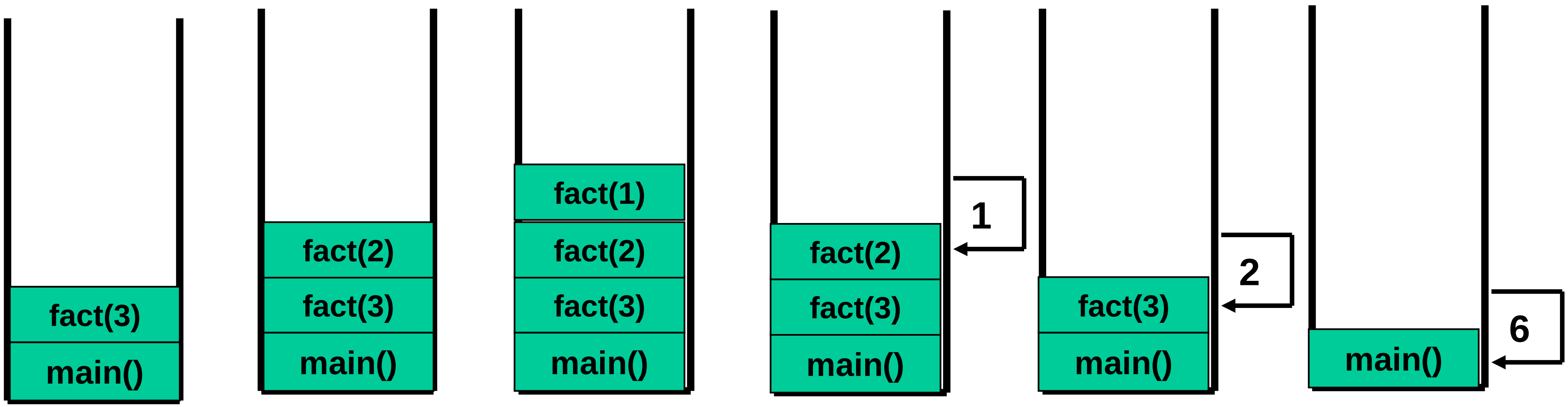
```
public class FindFactorialRecursive
{
    public static void main (String args[])
    {
        for (int i = 1; i < 10; i++)
                System.out.println ( i + "! = " + findFactorial(i));
    }

    public static int findFactorial (int number)
    {
        if ( number <= 1)
                return 1;
        else
                return (number * findFactorial (number-1));
    }
}
```

Base Case.

Making progress

Finding the factorial of 3



Time 2:

Push: fact(3)

Time 3:

Push: fact(2)

Time 4:

Push: fact(1)

Time 5:

Pop: fact(1)

returns 1.

Time 6:

Pop: fact(2)

returns 2.

Time 7:

Pop: fact(3)

returns 6.

Inside findFactorial(3):

if (number <= 1) return 1;

else return (3 * factorial (2));

Inside findFactorial(2):

if (number <= 1) return 1;

else return (2 * factorial (1));

Inside findFactorial(1):

if (number <= 1) return 1;

else return (1 * factorial (0));

Recursion pros and cons

- All recursive solutions can be implemented without recursion.
- Recursion is "expensive". The expense of recursion lies in the fact that we have multiple activation frames and the fact that there is overhead involved with calling a method.
- If both of the above statements are true, why would we ever use recursion?
- In many cases, the extra "expense" of recursion is far outweighed by a simpler, clearer algorithm which leads to an implementation that is easier to code.
- Ultimately, the recursion is eliminated when the compiler creates assembly language (it does this by implementing the stack).

Tail recursion

- Tail recursion is when the last line of a method makes the recursive call.
- In this case, you have multiple active stack frames which are unnecessary because they have finished their work.
- It is easy to rid your program of this type of recursion. These two steps will do so:
  - Enclose the body of the method in a while loop
  - Replace the recursive call with an assignment statement for each method argument.
- Most compilers do this for you. Note: I said "most".

Revisit recursive factorial solution

Just follow our two steps

```
public class Test
{
    public static void main (String args[])
    {
        for (int i = 1; i < 10; i++)
            System.out.println ( i + "! = " + findFactorial(i));
    }
    public static int findFactorial (int number)
    {
        if (number <= 1)
            return 1;
        else
            return (number * findFactorial (number-1));
    }
}
```

```
public class Test
{
    public static void main (String args[])
    {
        for (int i = 1; i < 10; i++)
            System.out.println ( i + "! = " + findFactorial(i));
    }
    public static int findFactorial (int number)
    {
        int answer = 1;
        while ( number > 1)
        {
            answer = answer * number;
            number--;
        }
        return answer;
    }
}
```

Example Using Recursion: The Fibonacci Series

- Fibonacci series
  - Each number in the series is sum of two previous numbers
    - e.g., 0, 1, 1, 2, 3, 5, 8, 13, 21…

      fibonacci(0) = 0
      fibonacci(1) = 1
      fibonacci(n) = fibonacci(n - 1) + fibonacci(n – 2)

    - fibonacci(0) and fibonacci(1) are base cases

```
import javax.swing.JOptionPane;

public class FibonacciTest
{
    public static void main (String args[])
    {
        long number, fibonacciValue;
        String numberAsString;
        numberAsString = JOptionPane.showInputDialog("What Fib value do you want?");
        number = Long.parseLong( numberAsString );

        fibonacciValue = fibonacci( number );

        System.out.println (fibonacciValue);
        System.exit (0);

    }

    // recursive declaration of method fibonacci
    public static long fibonacci( long n )
    {
        if ( n == 0 || n == 1 )
            return n;
        else
            return fibonacci( n - 1 ) + fibonacci( n - 2 );
    } // end method fibonacci
} // end class FibonacciTest
```

Four basic rules of recursion

- **Base case**: You must always have some base case which can be solved without recursion
- **Making Progress**: For cases that are to be solved recursively, the recursive call must always be a case that makes progress toward the base case.
- **Design Rule**: Assume that the recursive calls work.
- **Compound Interest Rule**: Never duplicate work by solving the same instance of a problem in separate recursive calls.

From Data Structures and Algorithms by Mark Allen Weiss

Fibonacci problem

- Which rule do we break in the Fibonacci solution?

```
import javax.swing.JOptionPane;

public class FibonacciTest
{
        static long [] array = new long [100];

        public static void main (String args[])
        {
                long number, fibonacciValue;
                String numberAsString;
                numberAsString = JOptionPane.showInputDialog("What Fib value do you want?");
                number = Long.parseLong( numberAsString );

                fibonacciValue = fibonacci( number );

                System.out.println (fibonacciValue);
                System.exit (0);

        }

        // recursive declaration of method fibonacci
        public static long fibonacci( long n )
        {
                if ( n == 0 || n == 1 )
                        return n;
                else if (array[(int)n] != 0)
                        return array[(int)n];
                else
                        {
                                array[(int)n] = fibonacci( n - 1 ) + fibonacci( n - 2 );
                                return array[(int)n];
                        }

        } // end method fibonacci
} // end class FibonacciTest
```

One more thing to watch out for

- Circular recursion occurs when we stop making progress towards the base case. For example:
  - We continuously call a(x) from within a(x)
  - a(x) calls a(x+1) then a(x+1) calls a(x)