Recursion

Lecture 13

What is Recursion?

- A method of programming in which a method refers to itself in order to solve a problem

- Never necessary
  - In some situations, results in simpler and/or easier-to-write code
  - Can often be more expensive in terms of memory + time

Exercise

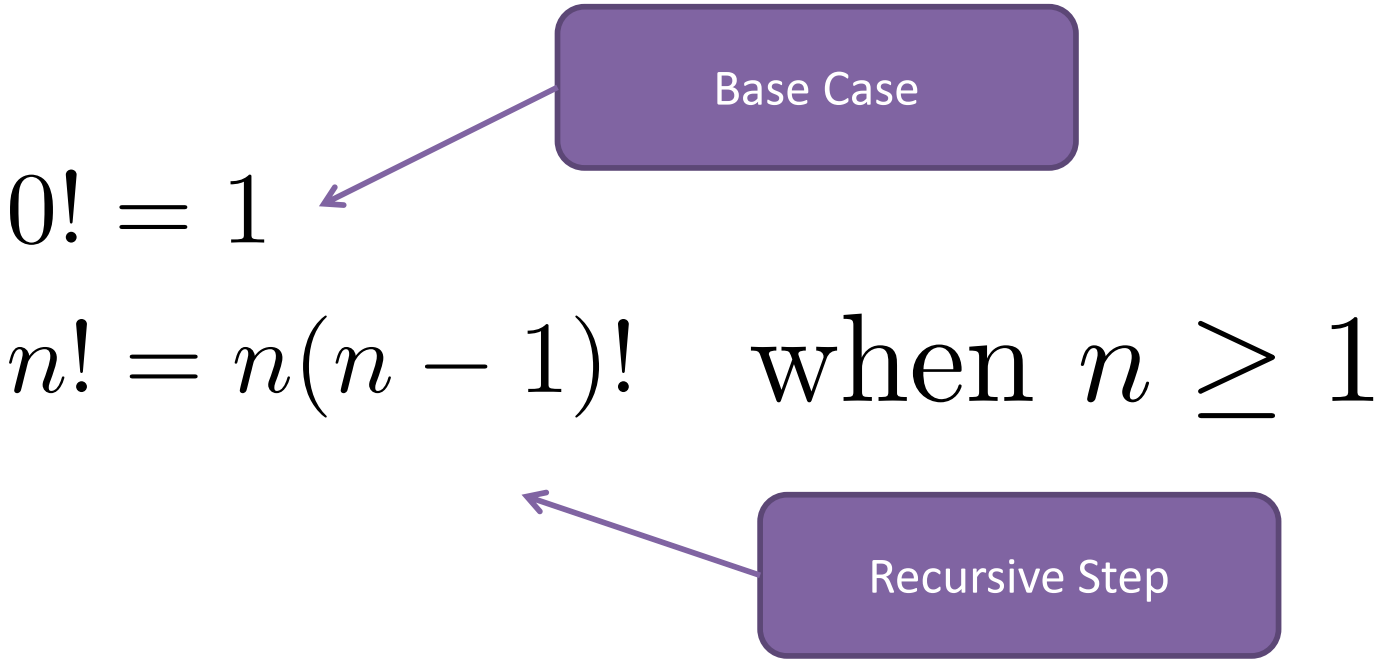
Write a factorial method that takes as input an integer (assumed to be >= 0) and returns as an integer the result

$$n! = \prod_{k=1}^{n} k = 1 * 2 * 3 * \ldots * n$$

Solution

```
public static int factorial(int n) {
   int result = 1;
   for (int i=2; i<=n; i++) {
      result *= i;
   }
   return result;
}
```

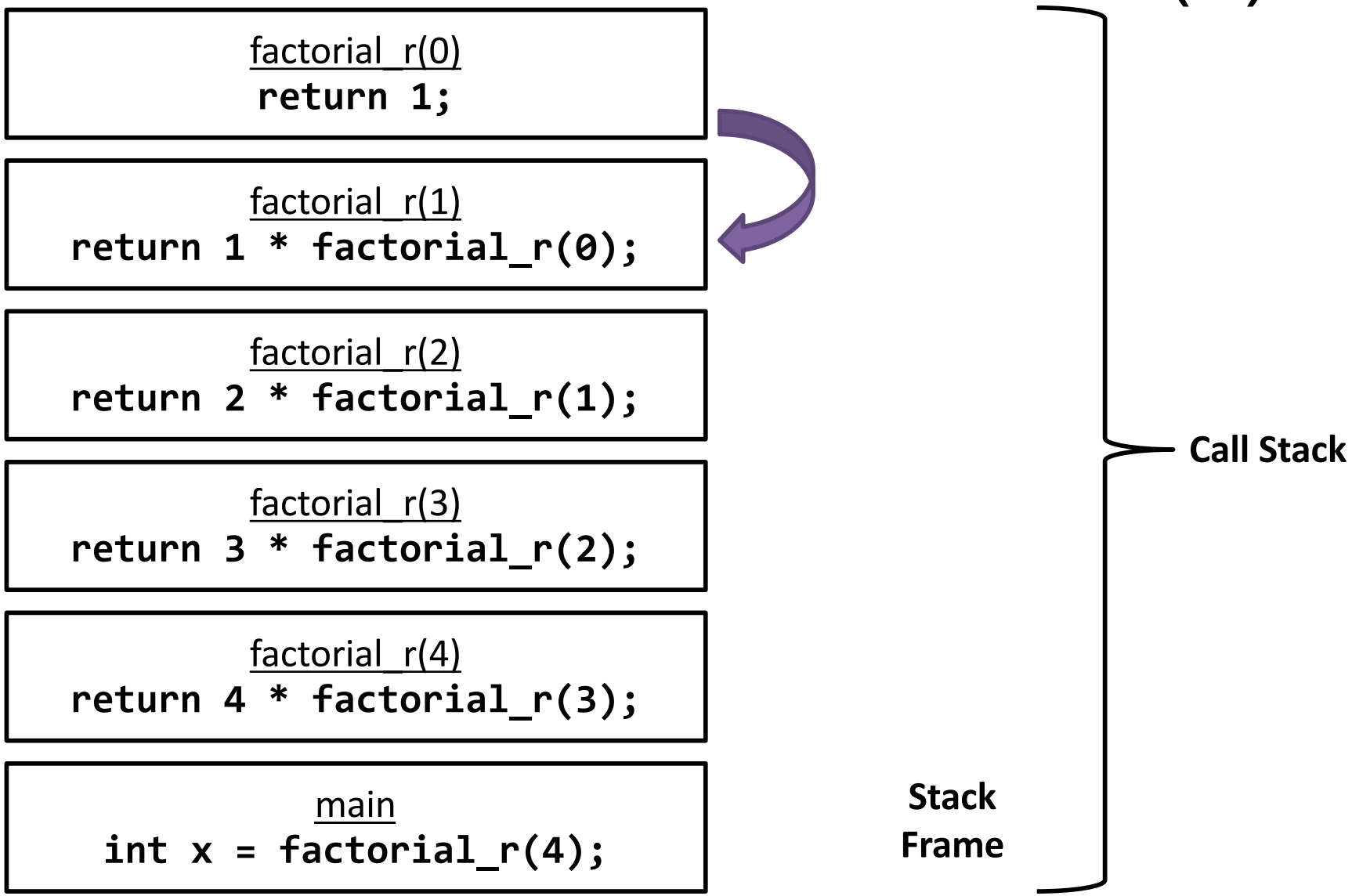
Consider a Recursive Definition



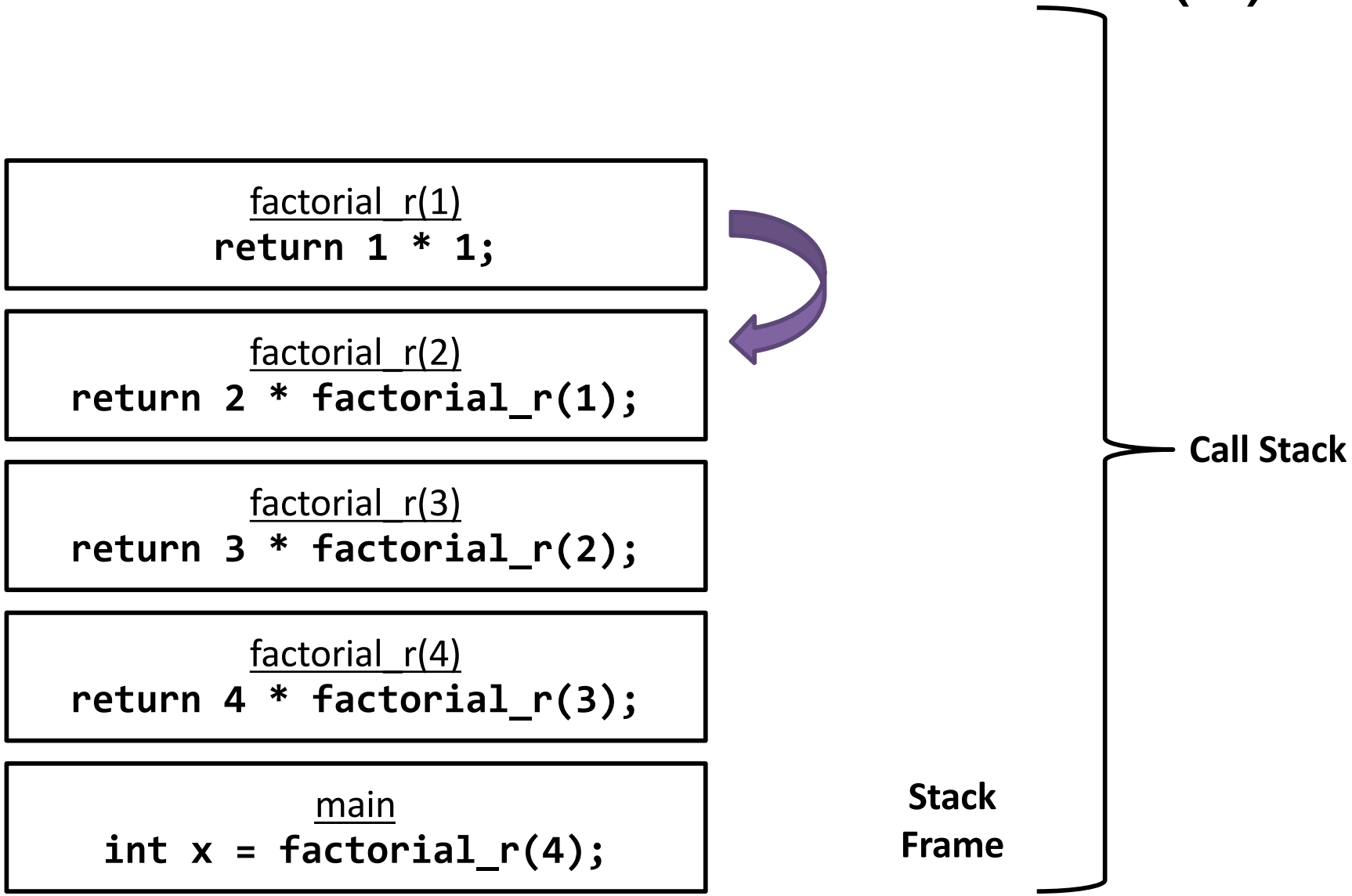
Conversion to Code

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

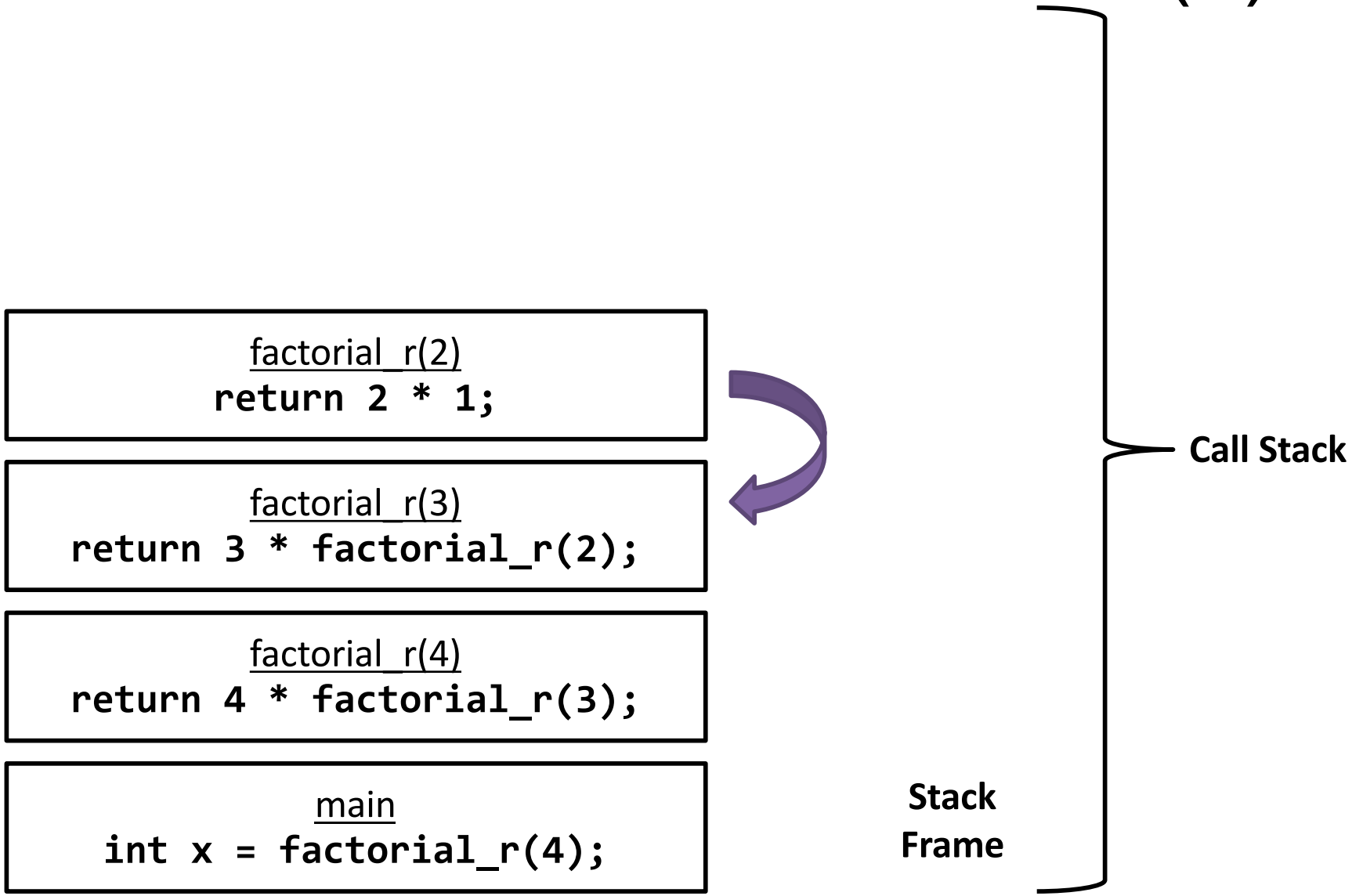
How the Code Executes (1)



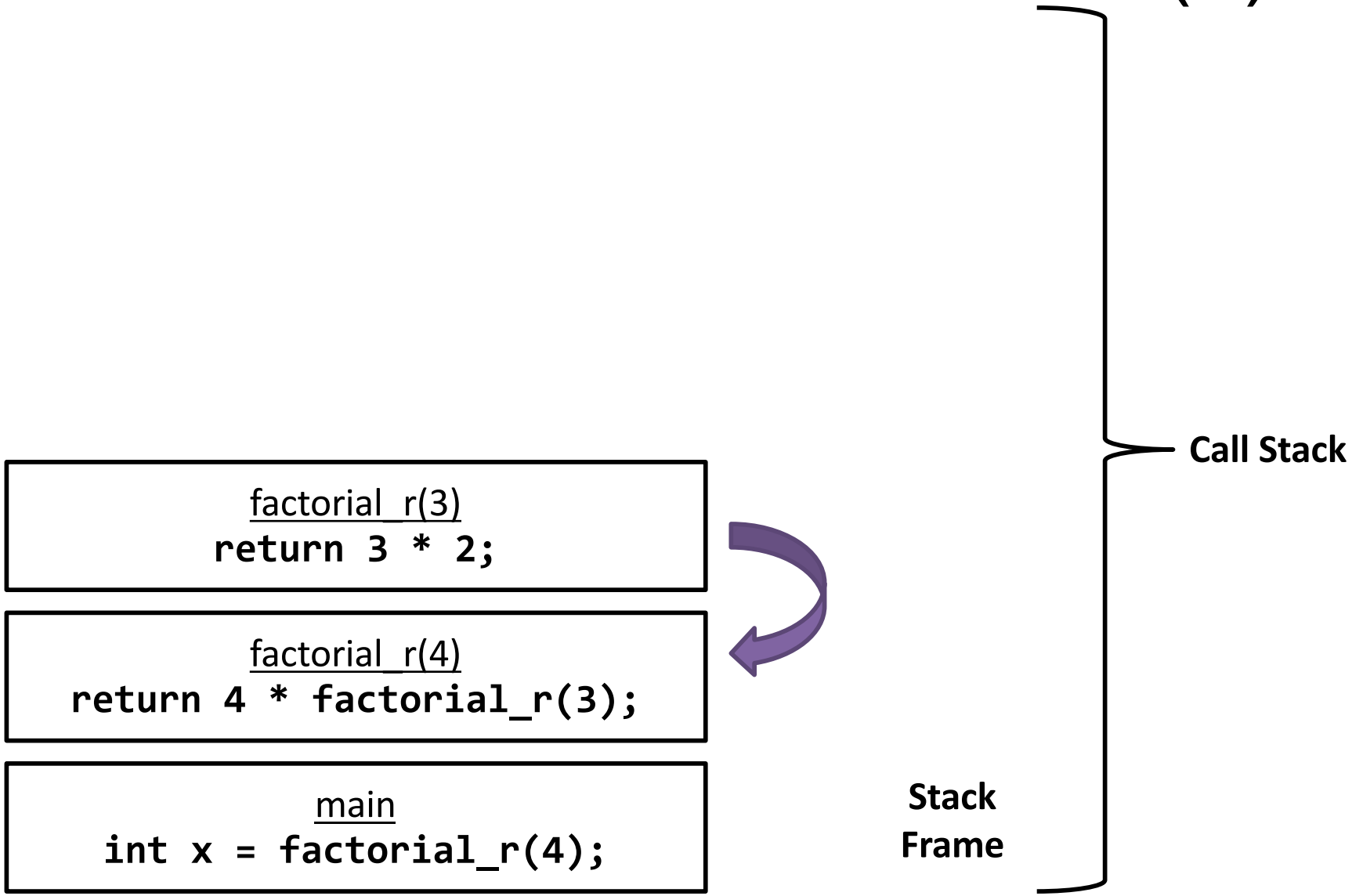
How the Code Executes (2)



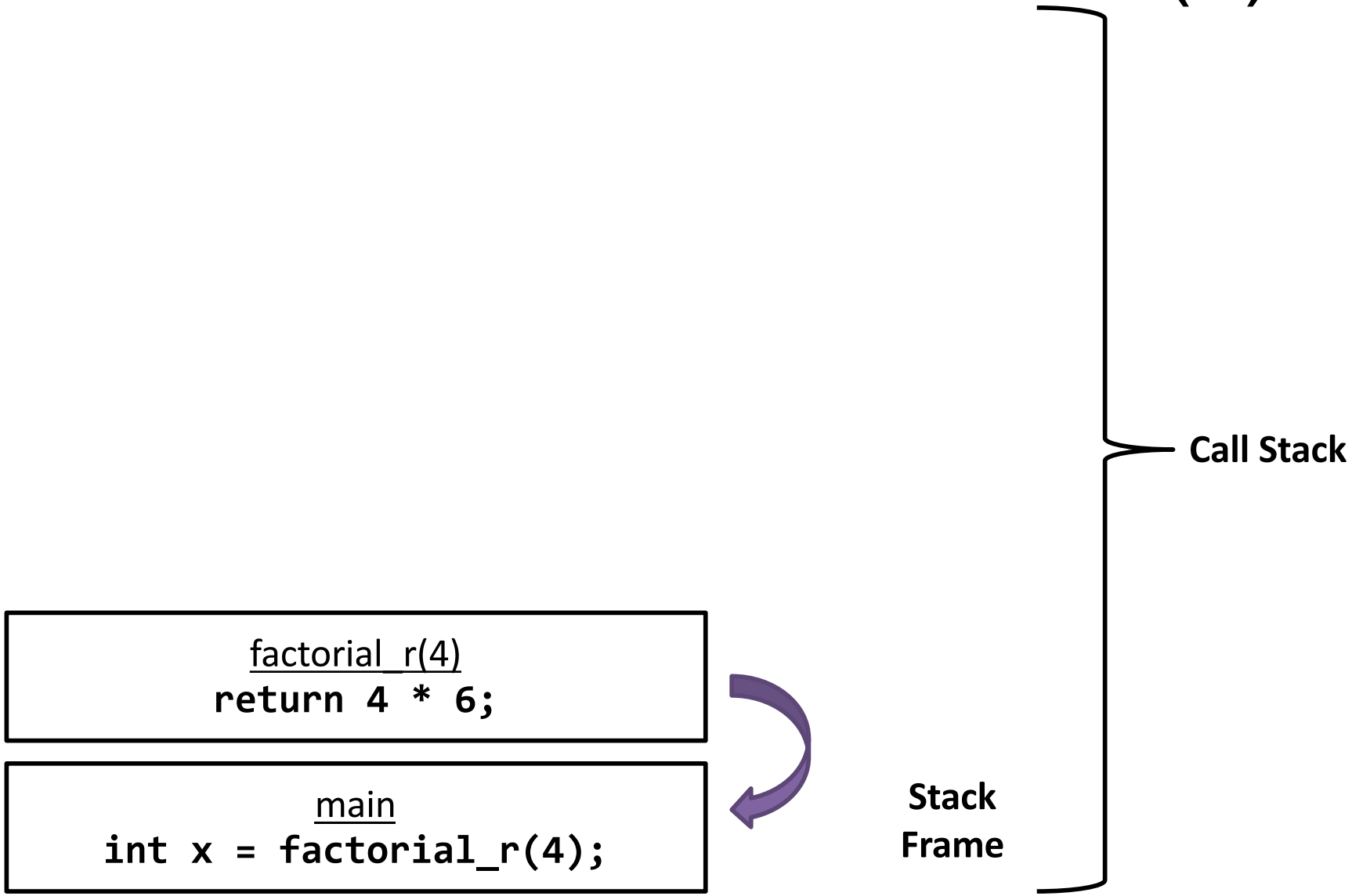
How the Code Executes (3)



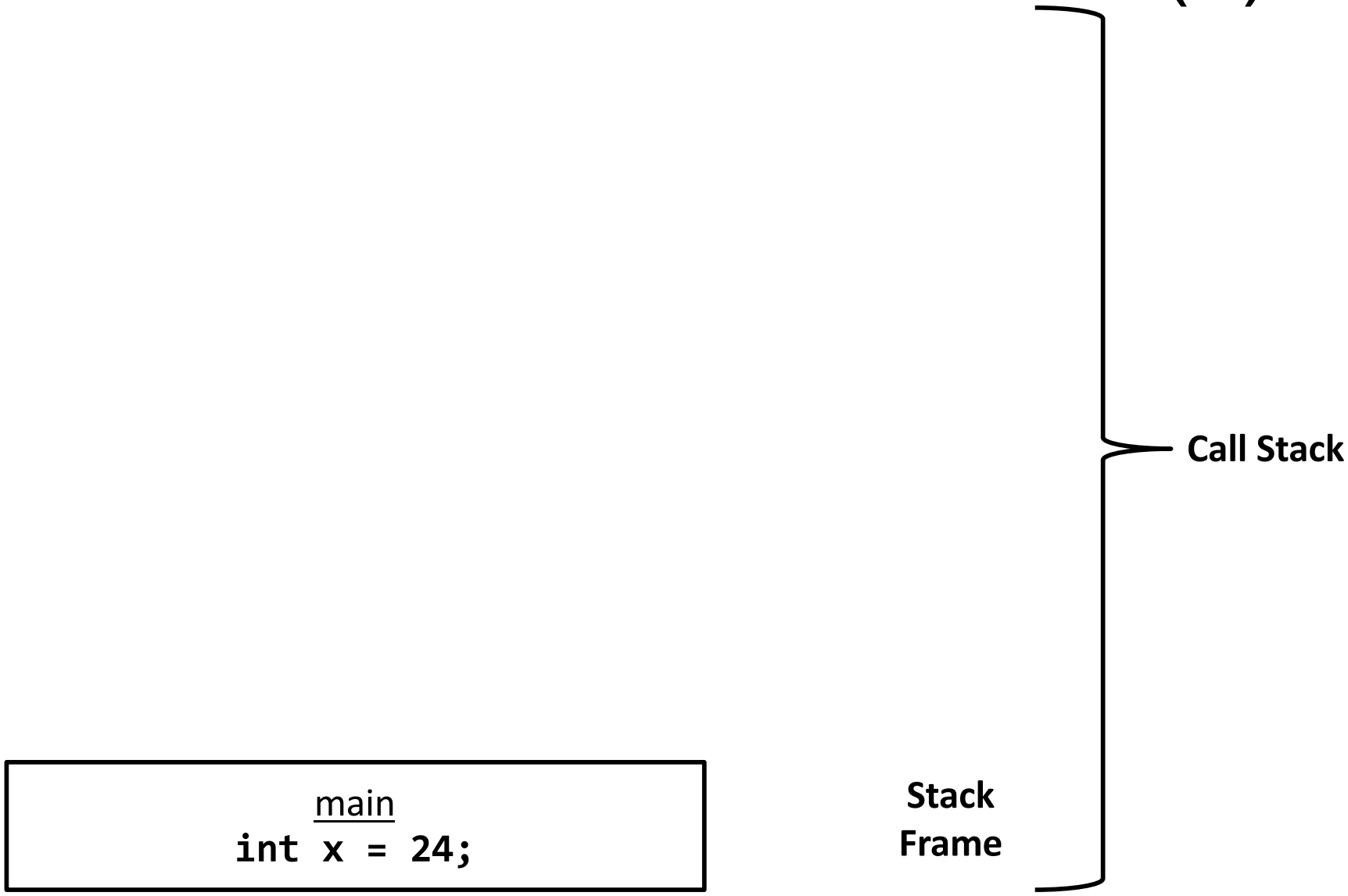
How the Code Executes (4)



How the Code Executes (5)



How the Code Executes (6)



Solving via Recursion

- In general, to solve a problem using recursion, break it into sub-problems

- If a sub-problem is similar to the original problem, just smaller in size, you can apply the same approach to solve the sub-problem recursively
  - Always make sure to have a base case, which is when the sub-problem has become "too small"

Exercise

Write a recursive method to print “Mind Blown!” *n* times, without using a loop.

Solution

```
public static void mindBlown(int n) {
   if (n >= 1) {
      System.out.printf("Mind Blown!%n");
      mindBlown(n-1);
   } // The base case is n == 0
}
```

Exercise

Write a recursive method **`power`** that takes in two integer arguments (**`base`**, **`exponent`**) and returns $\mathbf{base}^{\mathbf{exponent}}$ using no libraries. Assume exponent will be non-negative

Solution

```
public static int power(int base, int exponent) {
   // base case
   if (exponent == 0)
      return 1;

   // recursive step
   return base * power(base, exponent-1);
}
```

Exercise

Write a recursive method **`verticalDigits`** that outputs each digit of an integer to the screen on its own line. For example:

```
verticalDigits(1234);
1
2
3
4
```

Solution

```
public static void verticalDigits(int n) {
   if (n < 10) {
      System.out.printf("%d%n", n);
   } else {
      verticalDigits(n / 10);
      System.out.printf("%d%n", n % 10);
   }
}
```

Exercise

Write a recursive method **`verticalDigits2`** that outputs each digit of an integer to the screen on its own line. For example:

```
verticalDigits2(1234);
4
3
2
1
```

Solution

```
public static void verticalDigits2(int n) {
   if (n < 10) {
      System.out.printf("%d%n", n);
   } else {
      System.out.printf("%d%n", n % 10);
      verticalDigits2(n / 10);
   }
}
```

Exercise

In mathematics, the Fibonacci sequence is a sequence of integers:

0, 1, 1, 2, 3, 5, 8, 13, 21, …

Or, more formally:

$$F_n = F_{n-1} + F_{n-2}$$

$$F_0 = 0, F_1 = 1$$

Write the recursive **`fib`** method, which takes one integer argument.

Solution

```
public static int fib(int n) {
  if (n == 0)
    return 0;
  else if (n == 1)
    return 1;
  else
    return fib(n-1) + fib(n-2);
}
```

Exercise

Write a recursive method to compute all permutations of a supplied string and return them as a list

```
getPerms("abc")
[abc, acb, bac, bca, cab, cba]
```

Solution

```
public static String insertCharAt(String word,
                                  char c,
                                  int i) {
      final String start = word.substring(0, i);
      final String end = word.substring(i);
      return start + c + end;
}
```

```
getPerms("") -> [""]
getPerms("c") -> ["c"]
getPerms("bc") -> ["bc","cb"]
getPerms("abc") -> ["abc","bac","bca",
                    "acb”,"cab","cba"]
```

```
public static List<String> getPerms(String str) {
      if (str == null) {
            return null;
      }

      final ArrayList<String> perms =
            new ArrayList<>();
      if (str.length() == 0) {
            perms.add("");
            return perms;
      }

      final char first = str.charAt(0);
      final String remainder = str.substring(1);
      final List<String> words =
            getPerms(remainder);
      for (String word : words) {
            for (int j=0; j<=word.length(); j++) {
                  perms.add(
                    insertCharAt(word, first, j));
            }
      }
      return perms;
}
```

Recursion vs. Iteration

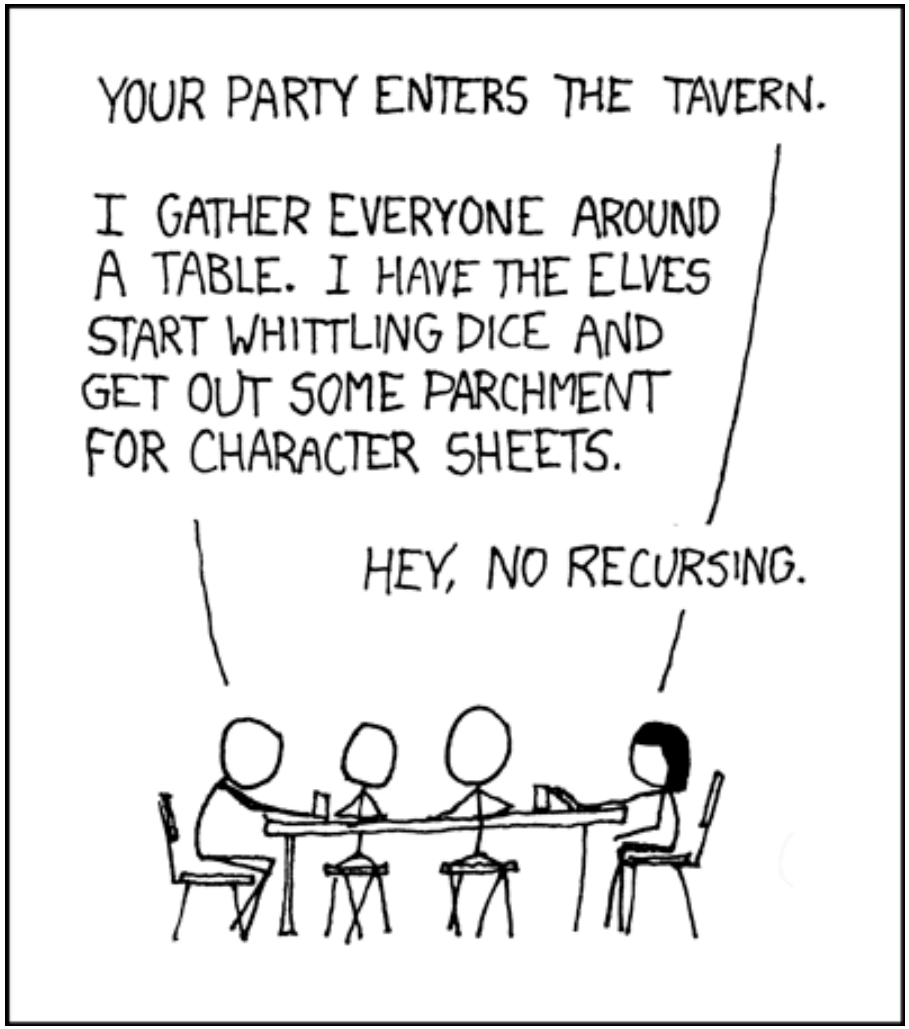
- Recursion is an alternative form of program control – essentially repetition without a loop

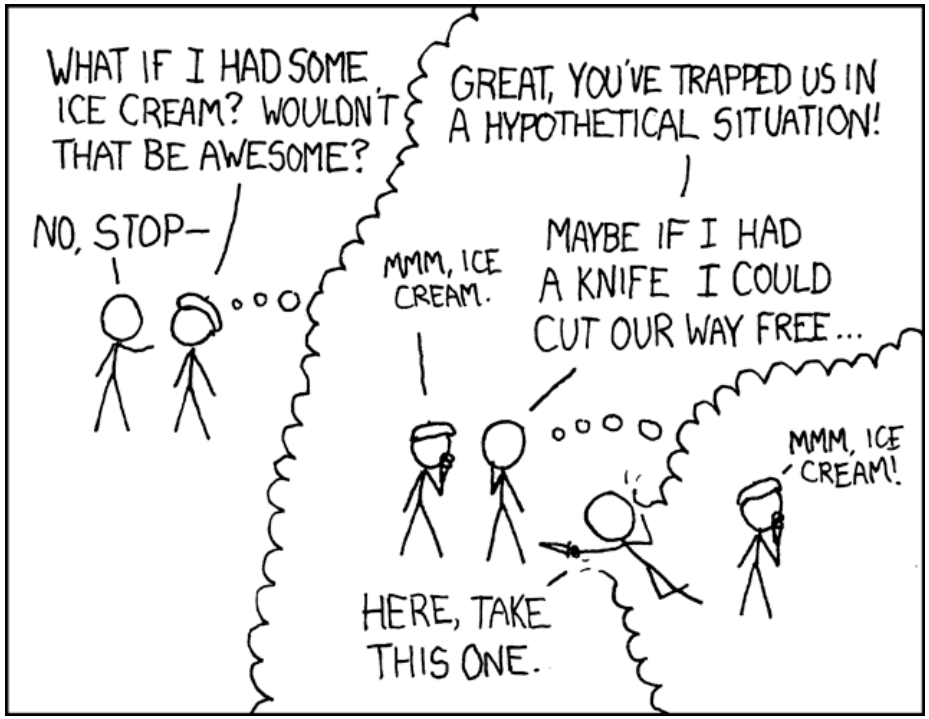
- Recursion bears substantial overhead
  - Each time the program calls a method, the system must assign space for all of the method's local variables and parameters
  - This can consume considerable memory and requires extra time to manage the additional space
  - Too much recursion = exceeding the call-stack memory limit = one way to cause a **stack overflow**

Take Home Points

- Recursive methods are methods that call themselves
- Recursion is an alternative to iteration (i.e. looping)
  - Sometimes simpler to write
  - Comes at computational expense, is more limited in depth than iterative approaches

XKCD Says…


YOUR PARTY ENTERS THE TAVERN.
I GATHER EVERYONE AROUND A TABLE. I HAVE THE ELVES START WHITTLING DICE AND GET OUT SOME PARCHMENT FOR CHARACTER SHEETS.
HEY, NO RECURSING.


WHAT IF I HAD SOME ICE CREAM? WOULDN'T THAT BE AWESOME?
NO, STOP—
GREAT, YOU'VE TRAPPED US IN A HYPOTHETICAL SITUATION!
MMM, ICE CREAM.
MAYBE IF I HAD A KNIFE I COULD CUT OUR WAY FREE...
MMM, ICE CREAM!
HERE, TAKE THIS ONE.