Chapter 18

Recursion

Characteristics of Recursion

All recursive methods have the following characteristics:

- One or more base cases (the simplest case) are used to stop recursion.
- Every recursive call reduces the original problem, bringing it increasingly closer to a base case until it becomes that case.

- In general, to solve a problem using recursion, you break it into subproblems.
- If a subproblem resembles the original problem, you can apply the same approach to solve the subproblem recursively.
- This subproblem is almost the same as the original problem in nature with a smaller size.

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

n! = n * (n-1)!

animation

Computing Factorial

factorial(4)

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

animation

Computing Factorial

factorial(4) = 4 * factorial(3)

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

animation

Computing Factorial

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2

animation

Computing Factorial

factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2
= 4 * 6

animation

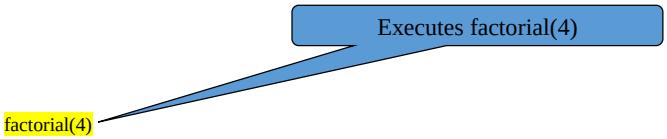
Computing Factorial

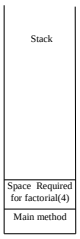
factorial(0) = 1;
factorial(n) = n*factorial(n-1);

factorial(4) = 4 * factorial(3)
= 4 * 3 * factorial(2)
= 4 * 3 * (2 * factorial(1))
= 4 * 3 * (2 * (1 * factorial(0)))
= 4 * 3 * (2 * (1 * 1)))
= 4 * 3 * (2 * 1)
= 4 * 3 * 2
= 4 * 6
= 24

animation

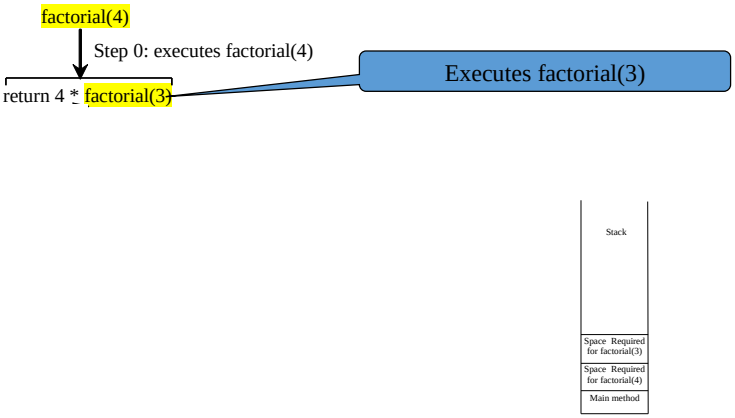
Trace Recursive factorial





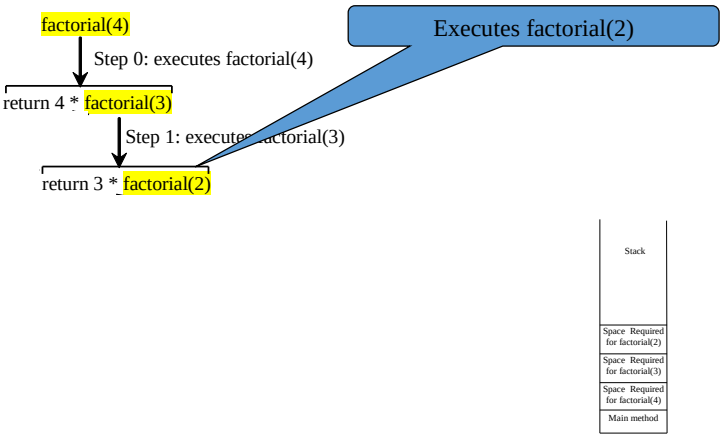
animation

Trace Recursive factorial



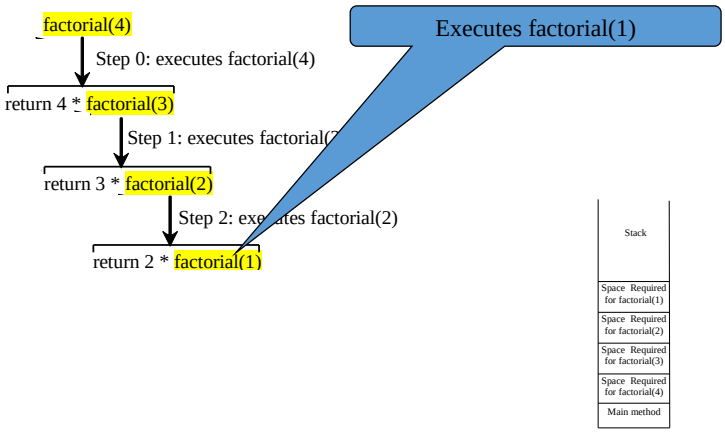
animation

Trace Recursive factorial



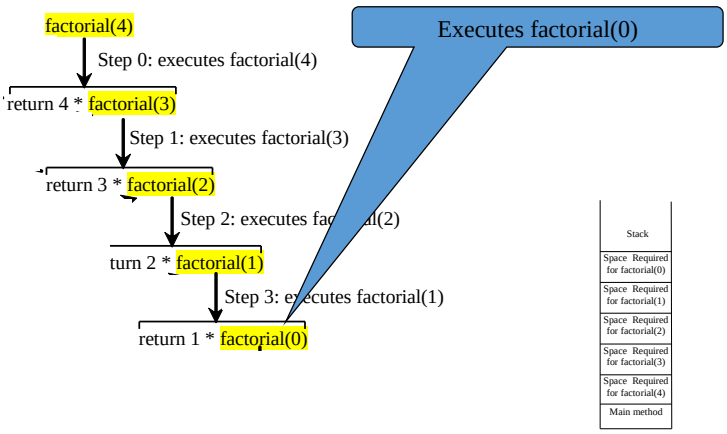
animation

Trace Recursive factorial



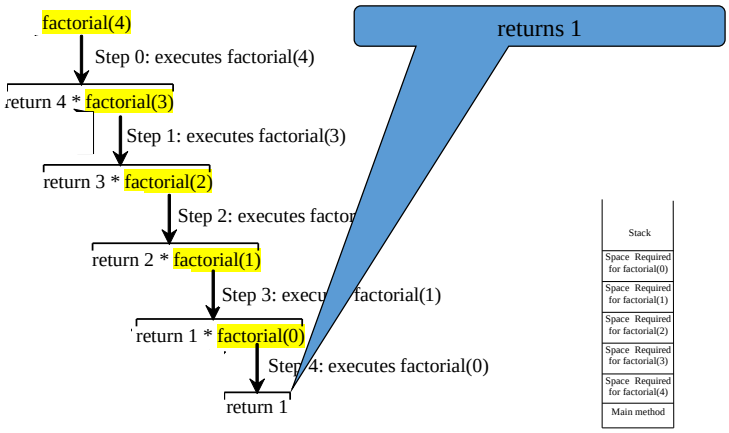
animation

Trace Recursive factorial



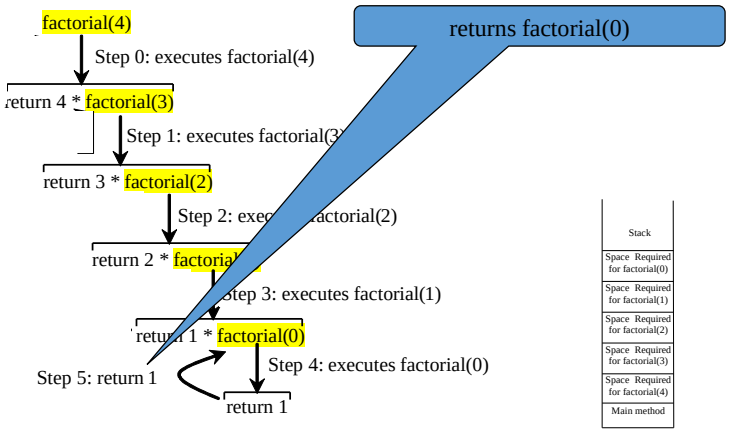
animation

Trace Recursive factorial



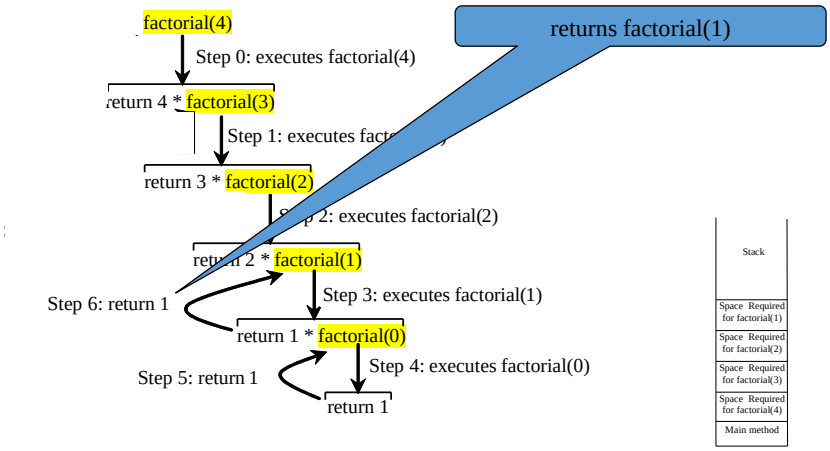
animation

Trace Recursive factorial



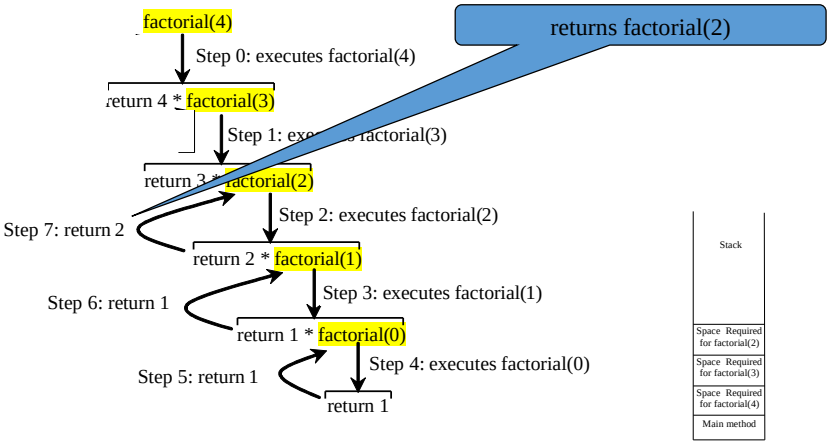
animation

Trace Recursive factorial



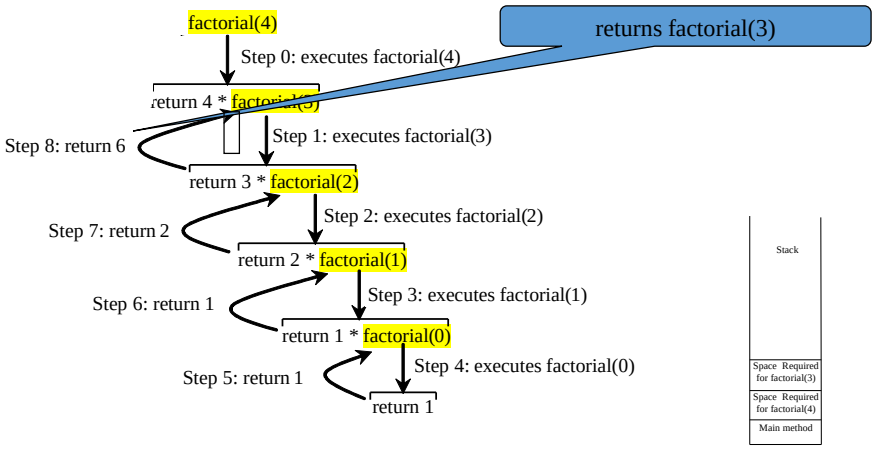
animation

Trace Recursive factorial



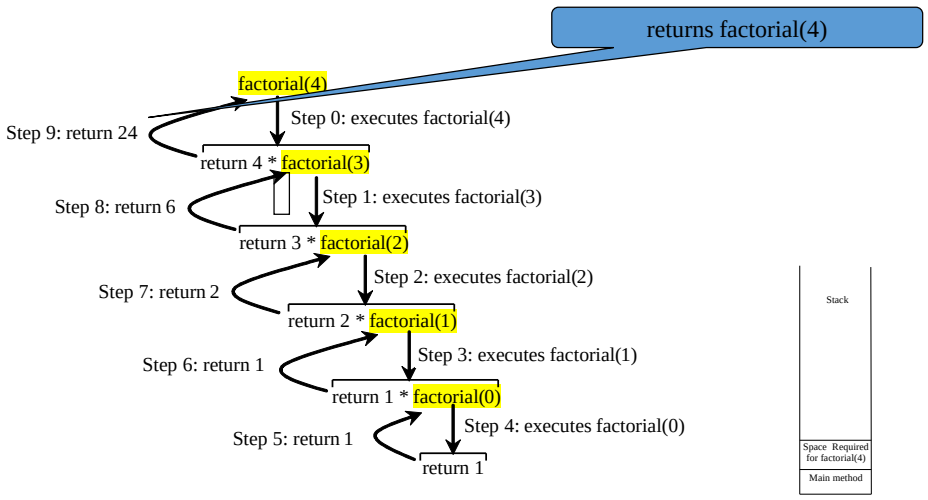
animation

Trace Recursive factorial

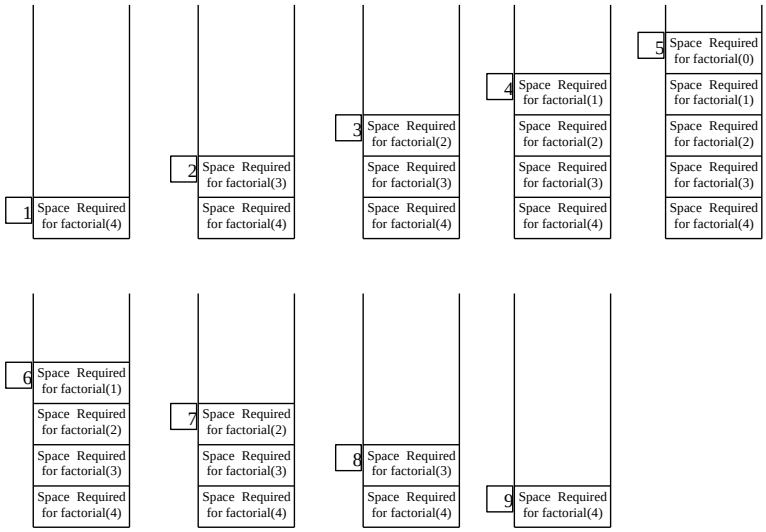


animation

Trace Recursive factorial



factorial(4) Stack Trace



Computing Factorial: Implementation

LISTING 18.1 ComputeFactorial.java

```
1  import java.util.Scanner;
2
3  public class ComputeFactorial {
4    /** Main method */
5    public static void main(String[] args) {
6      // Create a Scanner
7      Scanner input = new Scanner(System.in);
8      System.out.print("Enter a nonnegative integer: ");
9      int n = input.nextInt();
10
11     // Display factorial
12     System.out.println("Factorial of " + n + " is " + factorial(n));
13   }
14
15   /** Return the factorial for the specified number */
16   public static long factorial(int n) {
17     if (n == 0) // Base case                                  base case
18       return 1;
19     else
20       return n * factorial(n - 1); // Recursive call         recursion
21   }
22 }
```

If recursion does not reduce the problem in a manner that allows it to eventually converge into the base case or a base case is not specified, *infinite recursion* can occur.

Problem Solving Using Recursion

- Consider a simple problem of printing a message for n times.
- You can break the problem into two subproblems:
  - one is to print the message one time and
  - the other is to print the message for n-1 times.
- The second problem is the same as the original problem with a smaller size.
- The base case for the problem is n==0.
- You can solve this problem using recursion as follows:
  nPrintln("Welcome", 5);

```
public static void nPrintln(String message, int times) {
  if (times >= 1) {
    System.out.println(message);
    nPrintln(message, times - 1);
  } // The base case is times == 0
}
```


Chapter Summary

- A *recursive method* is one that invokes itself. For a recursive method to terminate, there must be one or more *base cases*.
- *Recursion* is an alternative form of program control. It is essentially repetition without a loop control.