Recursion

Recursion

- The role of recursive functions in programming is to break complex problems down to a small and solvable problem.

- The solvable problem is known as the ***base case***.

- A recursive function is designed to terminate when it reaches its *base case*.

Recursion

- A recursive function is one that calls itself.

Any problem with the function?

```
void myRecursionFun (void)
{
  System.out.println("This is a recursive function.“);
  myRecursionFun();
}
```

- The function above displays the string "This is a recursive function.", and then calls itself again

Problem 1: Factorial Function

In mathematics, the notation **n!** represents the factorial of an integer number ***n***. The factorial of a number is defined as:

```
n! =  1 * 2 * 3 * ...(n-1) * n      if n = 2, 3, …
n! = 1                              if n = 0 or 1
```

- **How to implement it using recursive function? But first let's try with iteration (loops)**
  - **Could you do it in 3 minutes?**

```
int factorial_withLoop(int num)
{
  int factor = 1;
  for(int i=1; i<=num; i++) factor = factor * i;
  return factor;
}
```

Factorial Function

- Define the factorial of a number, using recursion as:

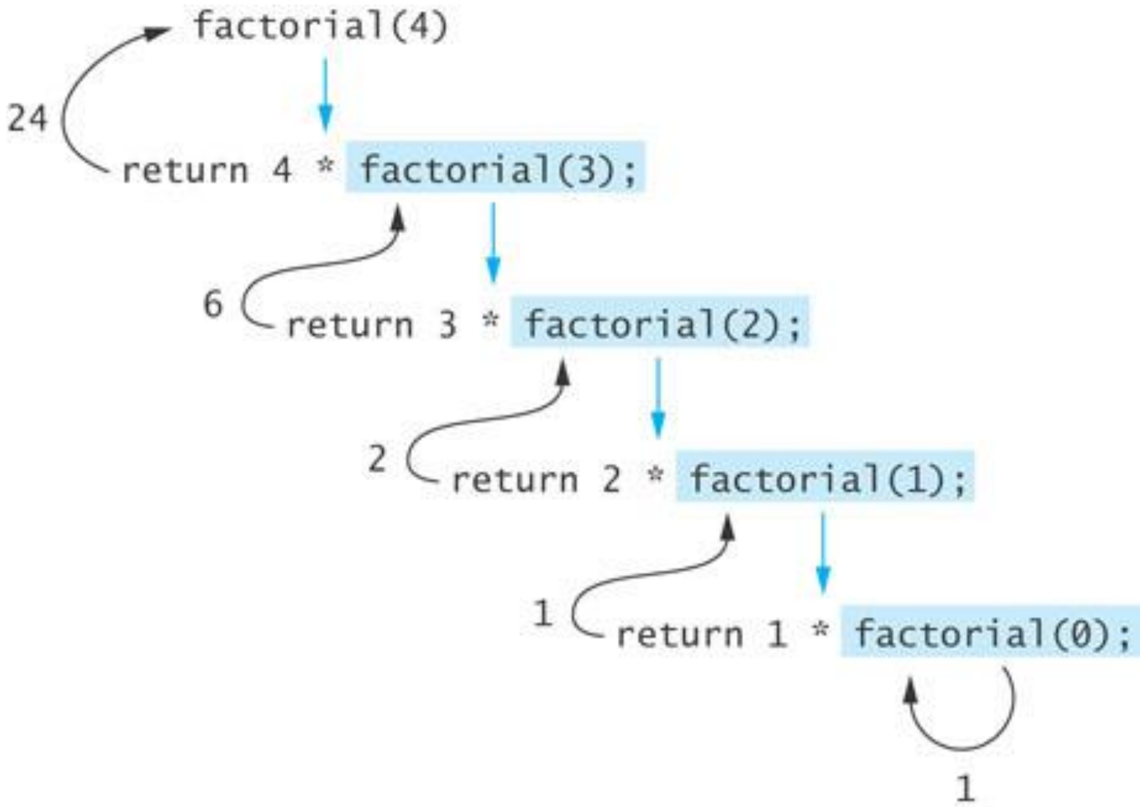
```
n! =  1 * 2 * 3 * ...(n-1) * n      if n = 2, 3, …
n! = 1                              if n = 0 or 1
                                    (note, 0! = 0)
Factorial(n) =    n * Factorial(n - 1)  if n > 1
                                  1      if n = 0 or 1
```

```
int factorial(int num)
{
  if (num >= 1)
        return num * factorial(num - 1);
  else
        return 1;
}
```

Factorial Function

- Recursive trace for the call
    - Unwinding the recursion

```
static int factorial(int num)
{
  if (num >= 1)
    return num * factorial(num - 1);
  else
    return 1;
}
```



Recursion

- A problem of a given size *N* can be reduced to one or smaller versions of the same problem (recursive case(s))

- There must be at least one case (the base case), for a small value of *N*, that can be solved directly

- Identify the base case(s) and solve it/them directly

- Combine the solutions to the smaller problems to solve the larger problem

Problem 2: Fibonacci numbers

- Some mathematical problems are designed to be solved recursively. One example is the calculation of *Fibonacci numbers*:

0, 1, 1, 2, 3, 5, 8, 13, ?

21

21, 34

21, 34, 55, 89, 144, 233, …

Fibonacci numbers

- The Fibonacci series can be defined as:

$$\begin{cases} F_0 = 0, \\ F_1 = 1, \\ F_N = F_{N-1} + F_{N-2} \qquad \textit{for } N \geq 2 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, … …

Fibonacci numbers

- A recursive function to calculate the **nth** number in the Fibonacci series:

$$
\begin{cases}
F_0 = 0, \\
F_1 = 1, \\
F_N = F_{N-1} + F_{N-2} & \text{for } N \geq 2
\end{cases}
$$

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

Recursion

- Recursion can solve many programming problems that are difficult to conceptualize and solve linearly – we will see a few later.

- Recursive algorithms can
  - compute factorials
  - compute a greatest common divisor
  - process data structures (strings, arrays, linked lists, etc.)
  - search efficiently using a binary search
  - find a path through a maze, and more

Recursion

- Using recursion has advantages:
  - Complex tasks can be broken down into simpler problems.
  - Code using recursion is usually shorter and more elegant.
  - Sequence generation is cleaner with recursion than with iteration (loops).

Recursion - more examples

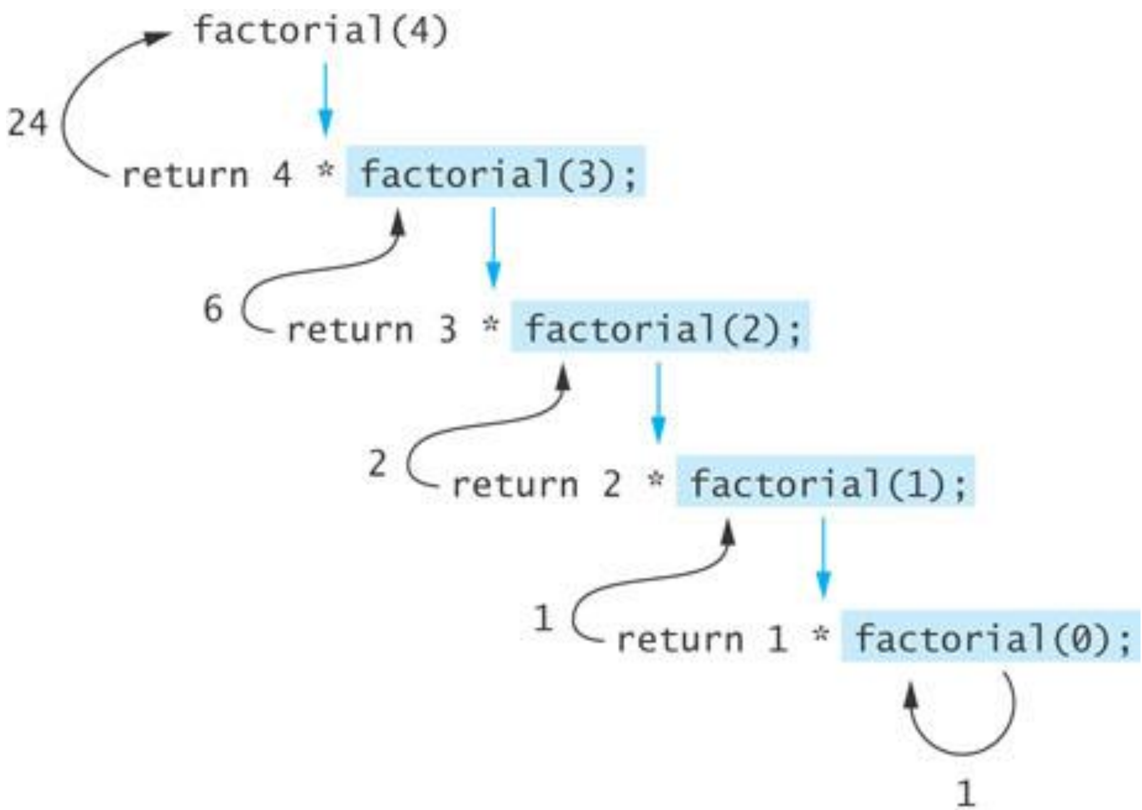
- Use recursion to solve the following questions:
  - Reverse a string
  - Count down: given an integer N, print "N, N-1, N-2, … , 1, 0"
  - Sum all numbers from N to 1
  - print each character of an input string

```
String StrReverse (String str)
{
        // … …
}
void CountDown (int N)
{
        // … …
}
void SumAll (int N)
{
        // … …
}
void PrintEachCharacter(String str)
{
        // … …
}
```

Factorial Function

- Question:
  - How are the returning addresses kept?

```
static int factorial(int num)
{
  if (num >= 1)
    return num * factorial(num - 1);
  else
    return 1;
}
```

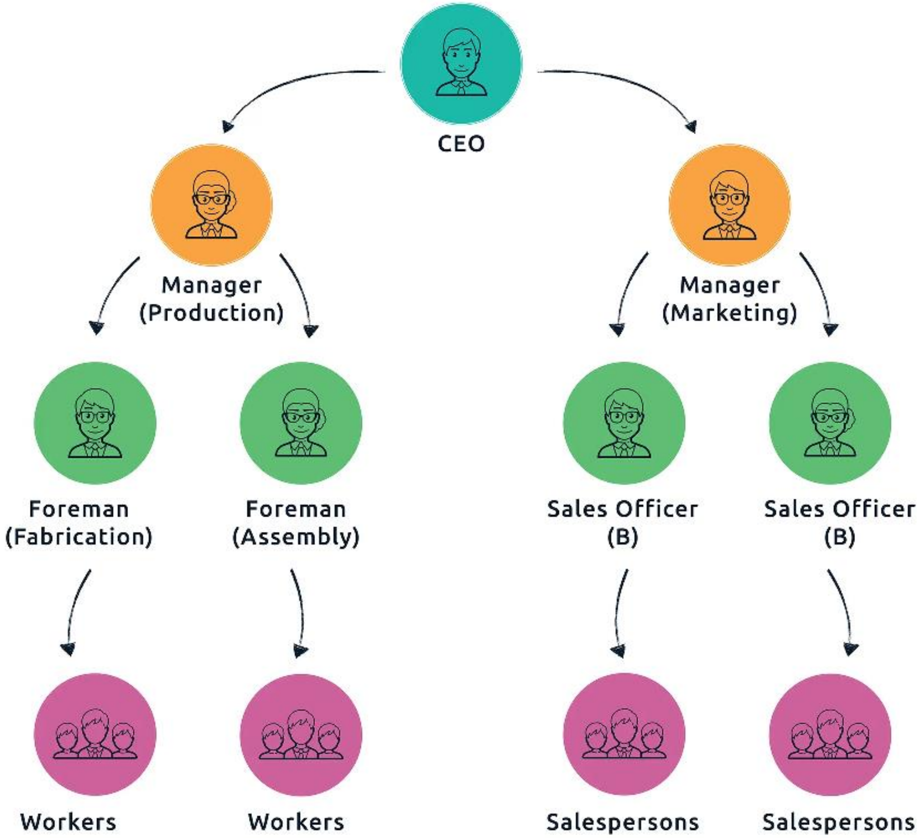


Recursive call – stack

- Stack:
  - A pile of objects, that are typically arranged neatly

Recursive call - stack

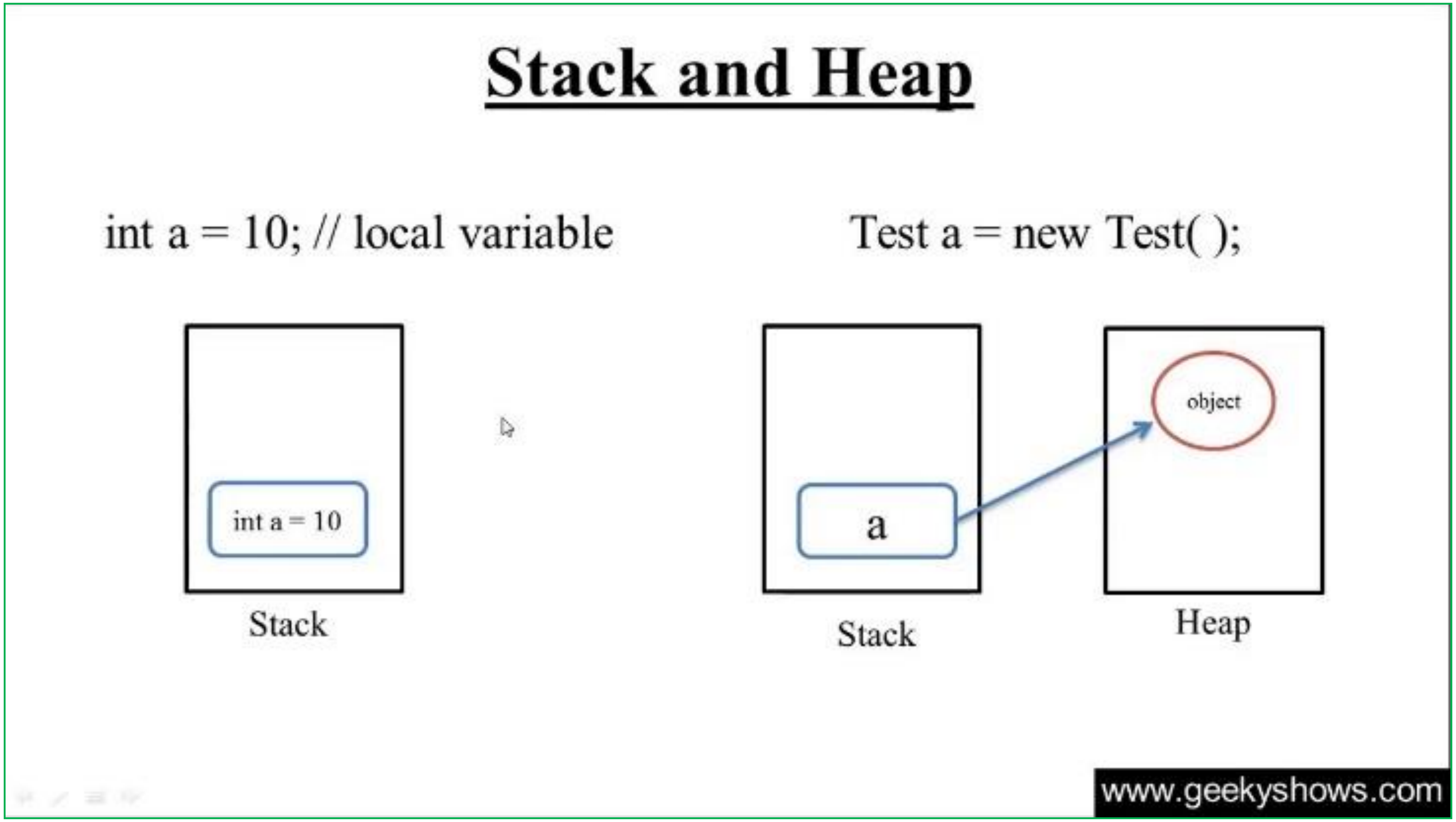
- Run-time stack for recursive vs. employees in an office tower
    - The employee on the bottom level carries out part of the instructions, calls the employee on the next level up and is put on hold
    - The employee on the next level completes part of the instructions and calls the employee on the next level up and is put on hold
    - And then …



https://digitalleadership.com/blog/organizational-structure/

Recursive call – stack

- Stack vs. Heap in memory



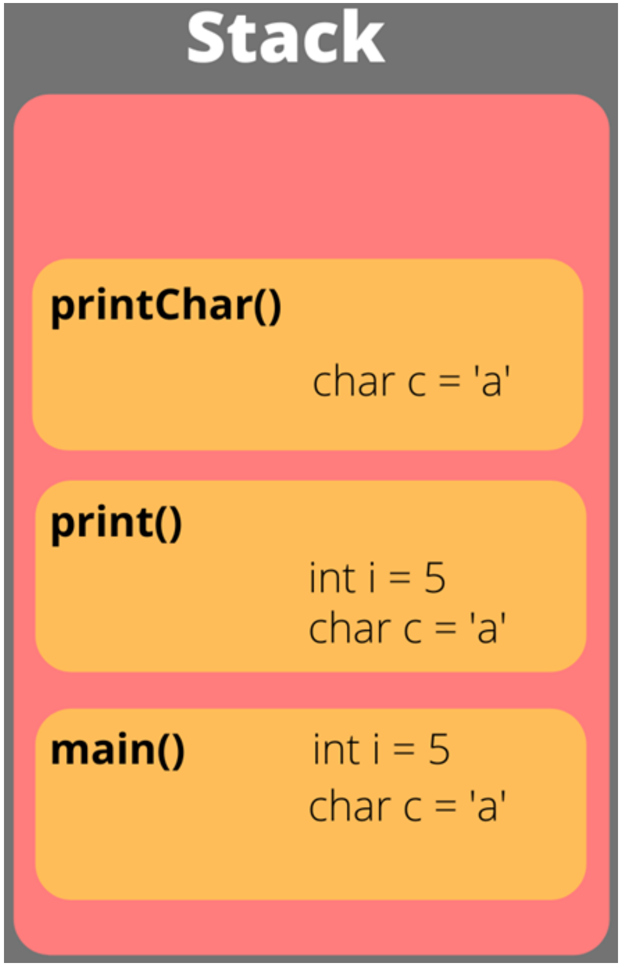
Recursive call – stack

- Stack vs. Heap in memory
  - Stack memory:
    - Keeps all primitive data resides and the references to other objects are also stored in it.
    - When a method is called, a new block for that method is added to the stack. This block will contain all the primitive data and references that are needed by the method.
      - Memory is automatically allocated when a new method is called and deallocation also takes place automatically when a method returns
    - Stack is much smaller in size.
    - Stack memory is very efficient and memory access is also faster.

Recursive call – stack

- Stack memory:
  - primitive data, references to objects, method blocks

```
public class MemoryTest
{
   public static void main(String[] arg)
   {
          int i = 5;
          char c = 'a’;
          print(i, c);
          System.out.println(“Done!”);
   }

   static void print(int i, char c)
   {      System.out.println(i);
          printChar(c);
   }

   static void printChar(char c)
   {
          System.out.println(c);
   }
}
```



Recursive call – stack

- Stack vs. Heap in memory

  - Heap memory:
    - all class instances or objects are allocated memory. As discussed above, the references to these objects are stored in the Stack
    - Memory space is allocated manually by the programmer when new objects are created.
    - Java uses a Garbage Collection mechanism to free up space.

Recursive call – stack

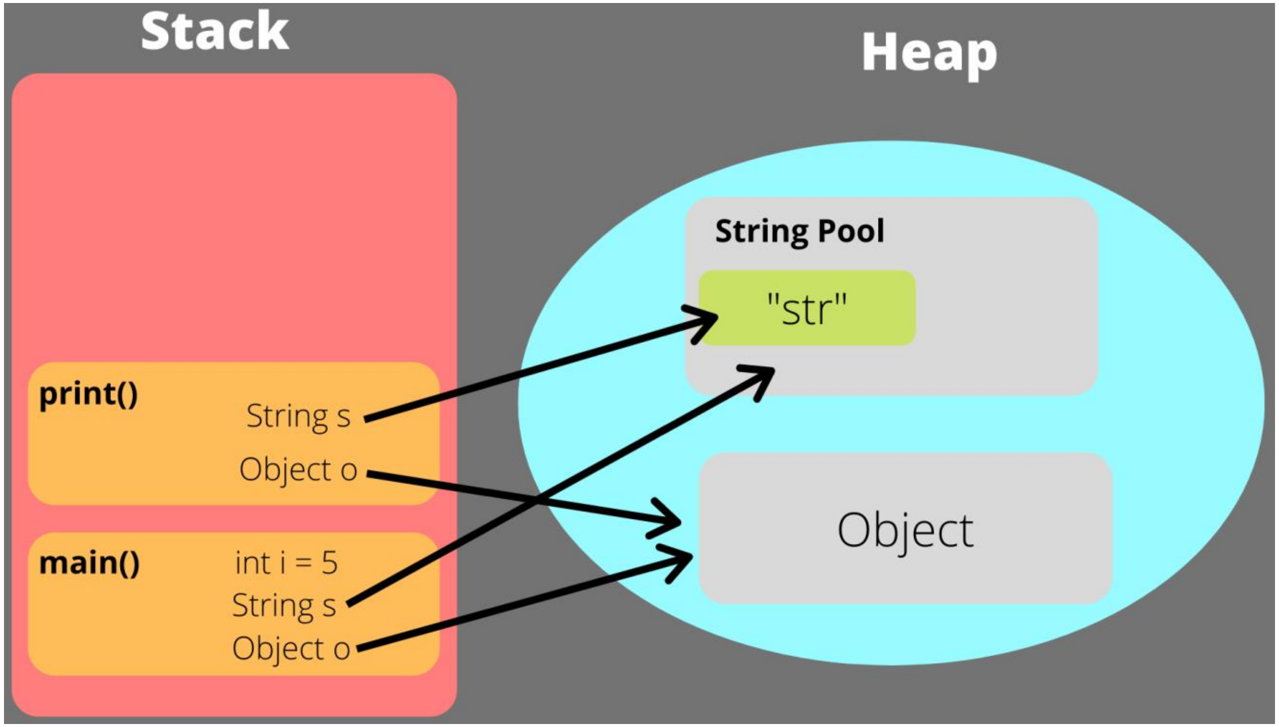
- Stack vs. Heap in memory

```
public class MemoryTest
{
  public static void main(String[] arg)
  {
        int i = 5;
        String s = ‘str’;
        Object o = new Object();
        print(o, c);
  }

  static void print(String s, Object o)
  {     System.out.println(s);
        System.out.println(o);
  }
}
```

Recursive call – stack

- Stack vs. Heap in memory

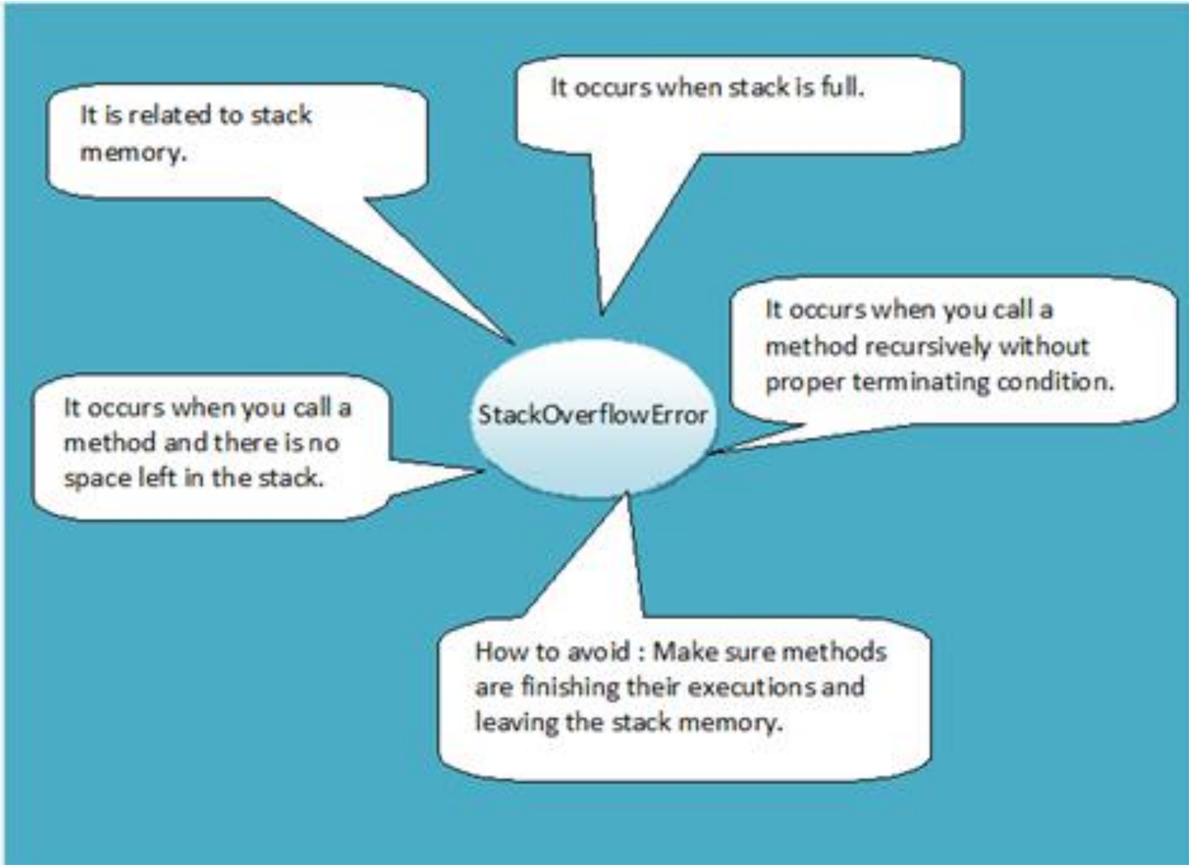
```
public static void main(String[] arg)  {
  int i = 5;  String s = 'str'; Object o = new Object();
        print(o, c);
  }

  static void print(String s, Object o) {
      System.out.println(s);    System.out.println(o);
  }
```



Recursive call – stack

- Stack vs. Heap in memory
  - Exception to throw:
    - We may encounter the **StackOverflowError** if we run out of the Stack memory.

```
void myRecursionFun ()
{
    myRecursionFun();
}
```



Recursive call – stack

- Stack vs. Heap in memory
  - Exception to throw:
    - If we run out of heap memory, then the JVM throws the ***OutOfMemoryError***.

```
class Person
{
        String name, address;
        int age, idNum;
        Job job;
        Food food;
        Family family;
        … …
}
```

```
public static void outOfMemoryError()
{
        while(true)
        {
                new Person();
        }
}
```

Recursion vs. Loop

- There are similarities between recursion and iteration
  - In iteration, a loop repetition condition determines whether to repeat the loop body or exit from the loop
  - In recursion, the condition usually tests for a base case

- You can almost always write an iterative solution to a problem that is solvable by recursion
- A recursive algorithm may be simpler than an iterative algorithm and thus easier to write, code, debug, and read

Recursion vs. Loop

- However, the recursion may have overhead, more than what you expect
  - Let’s revisit Fibonacci question:

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

  - Is this efficient? How is it compared to an approach using iteration instead?

Recursion vs. Loop

- However, the recursion may have overhead, more than what you expect
  - Let's revisit Fibonacci question:
    - **Iterative version**

$F_0 = 0,$

$F_1 = 1,$

$F_N = F_{N-1} + F_{N-2} \quad \text{for } N \geq 2$

```
public long fib_iterative(int n)
{
    if(n == 0) return 0;
    else if (n == 1) return 1;
    else
    {
      long temp = 1;
      long oldValue = 1;
      long newValue = 1;
      for (int i = 3; i <= n; i++)
      {
        newValue = oldValue + temp;
        temp = oldValue;
        oldValue = newValue;
      }
      return newValue;
    }
}
```

Recursion vs. Loop

- However, the recursion may have overhead, more than what you expect
  - ► Let's revisit Fibonacci question:
    - Recursive version
    - Iterative version

    - ?? Which one runs faster?
      - – Let's demo it

Recursion vs. Loop

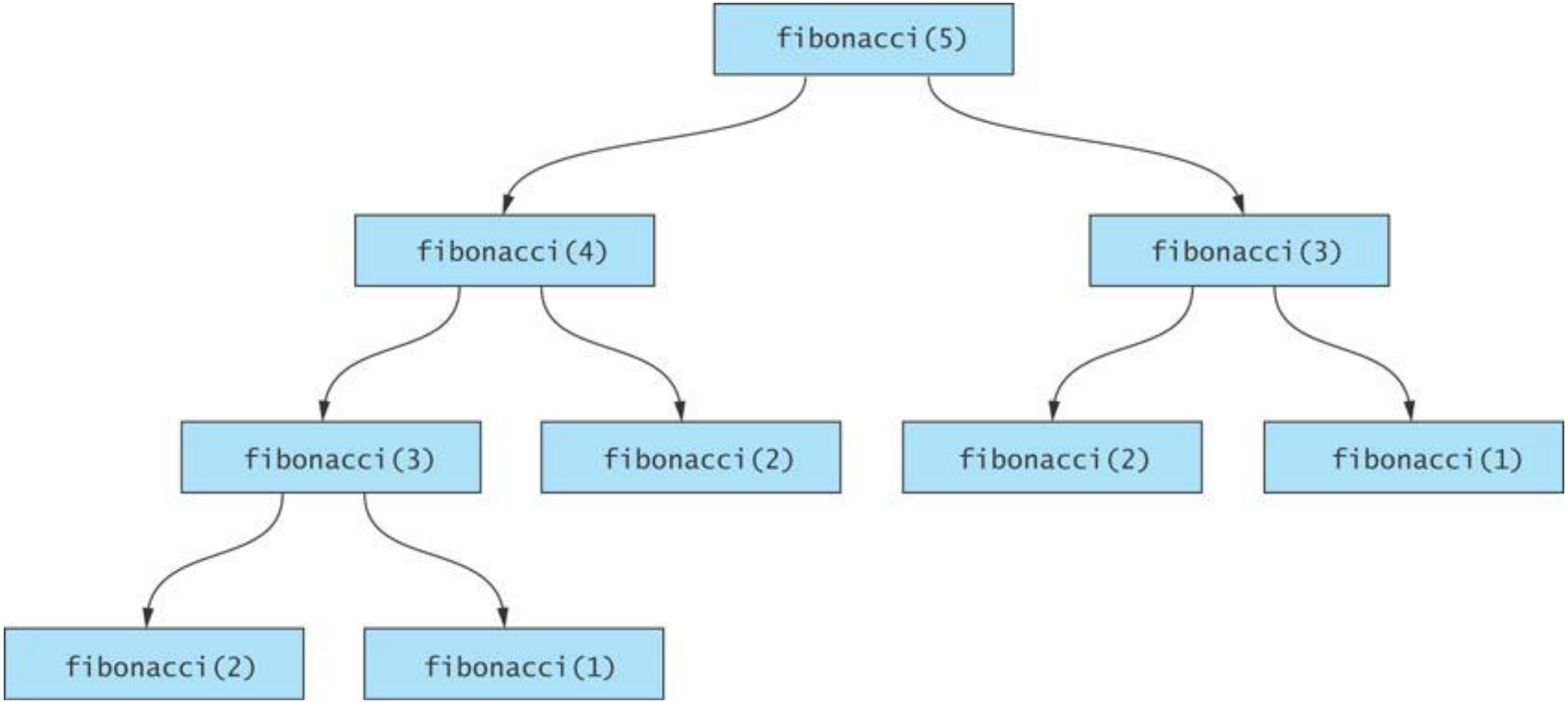
- However, the recursion may have overhead, more than what you expect
  - ► Let's revisit Fibonacci question:
    - Recursive version
    - Iterative version

    - ?? Which one runs faster?
      - – Let's demo it

    - Obviously, the recursion is slower than the 'loop'! Why?

Fibonacci numbers

- Stacked calls and efficiency



Recursion vs. Loop

- Recursive methods often have slower execution times relative to their iterative counterparts
- The overhead for loop repetition is smaller than the overhead for a method call and return
- If it is easier to conceptualize an algorithm using recursion, then you should code it as a recursive method
- The reduction in efficiency usually does not outweigh the advantage of readable code that is easy to debug

Recursion vs. Loop

- A question from AIME (American Invitational Mathematics Examination), a challenging competition since 1983
  - 15 question, 3-hour examination, each answer is an integer number between 0 to 999.
  - Here is a question from 1984:

    - The function *f* is defined on the set of integers and satisfies:

$$f(n) = \begin{cases} n - 3 & \text{if } n \geq 1000 \\ f(f(n+5)) & \text{if } n < 1000 \end{cases}$$

Find *f*(84)

Factorial Function

- Using run-time **stack** to keep track of returning address (and relevant local variables, if any)