Lesson 4
Recursion

Wholeness of the Lesson

Computation of a function by recursion involves repeated self-calls of the function. Recursion is implicit also at the design level when a reflexive association is present. Recursion mirrors the self-referral dynamics of consciousness, on the basis of which all creation emerges.

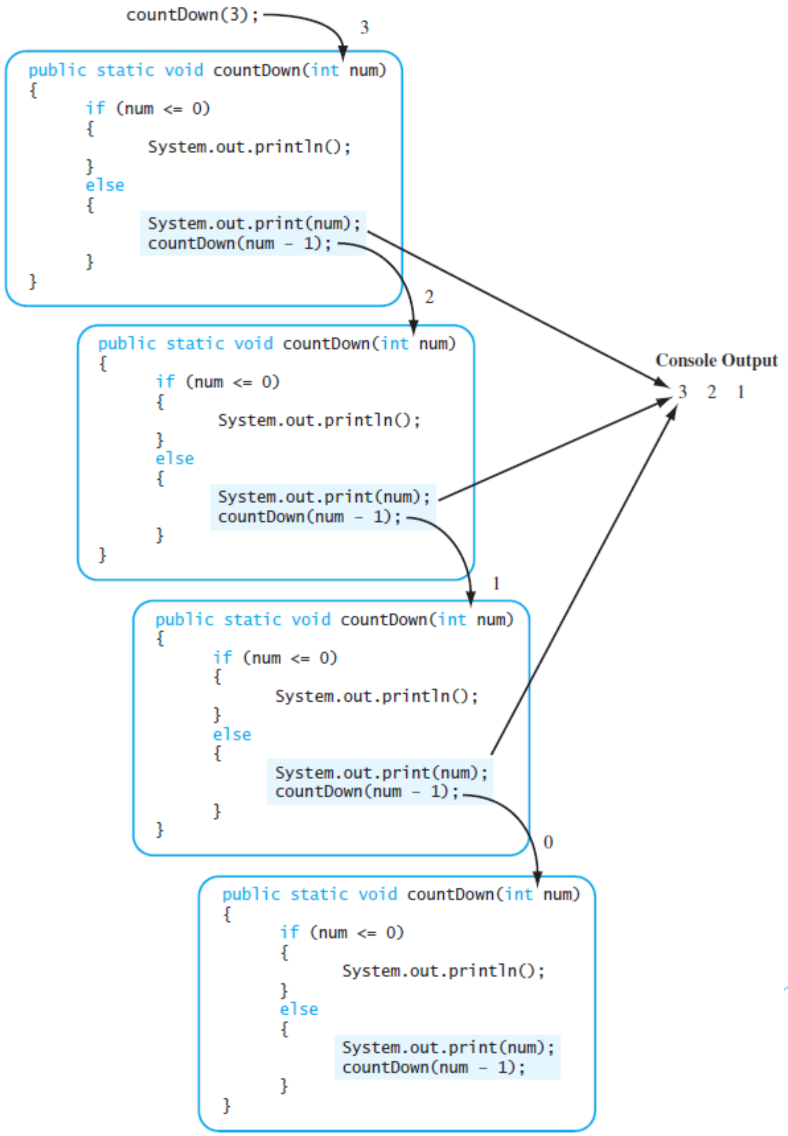
Recursion

- Recursion is an approach to solving problems using a function that calls itself as a subroutine.
- Recursion reduces a problem into one or more simpler versions of itself.
- A Java method can be recursive, or exhibits recursion, if it calls itself in its body.

Problem: Countdown from N to 0

```
public class RecursionSolution {

    public static void countdown(int num){

        if(num <= 0)
            return;        // base case

        System.out.println(num);
        --num;

        countdown(num);

    }

    public static void main(String[] args) {
        countdown(3);
    }
}
```

Problem: Countdown from N to 0



```
countDown(3);
```

```
public static void countDown(int num)
{
        if (num <= 0)
        {
                System.out.println();
        }
        else
        {
                System.out.print(num);
                countDown(num – 1);
        }
}
```

```
public static void countDown(int num)
{
        if (num <= 0)
        {
                System.out.println();
        }
        else
        {
                System.out.print(num);
                countDown(num – 1);
        }
}
```

```
public static void countDown(int num)
{
        if (num <= 0)
        {
                System.out.println();
        }
        else
        {
                System.out.print(num);
                countDown(num – 1);
        }
}
```

```
public static void countDown(int num)
{
        if (num <= 0)
        {
                System.out.println();
        }
        else
        {
                System.out.print(num);
                countDown(num – 1);
        }
}
```

Design a Recursive Algorithm

- There must be at least one case (the base case), for a small value of *n*, that can be solved directly
- A problem of a given size *n* can be reduced to one or more smaller versions of the same problem (recursive case(s))
- Identify the base case(s) and solve it directly
- Devise a strategy to reduce the problem to smaller versions of itself while making progress toward the base case
- Combine the solutions to the smaller problems to solve the larger problem

Problem: Finding the Length of a String

```
if the string is empty (has no characters)
        the length is 0
else
        the length is 1 plus the length of the string that
        excludes the first character
```

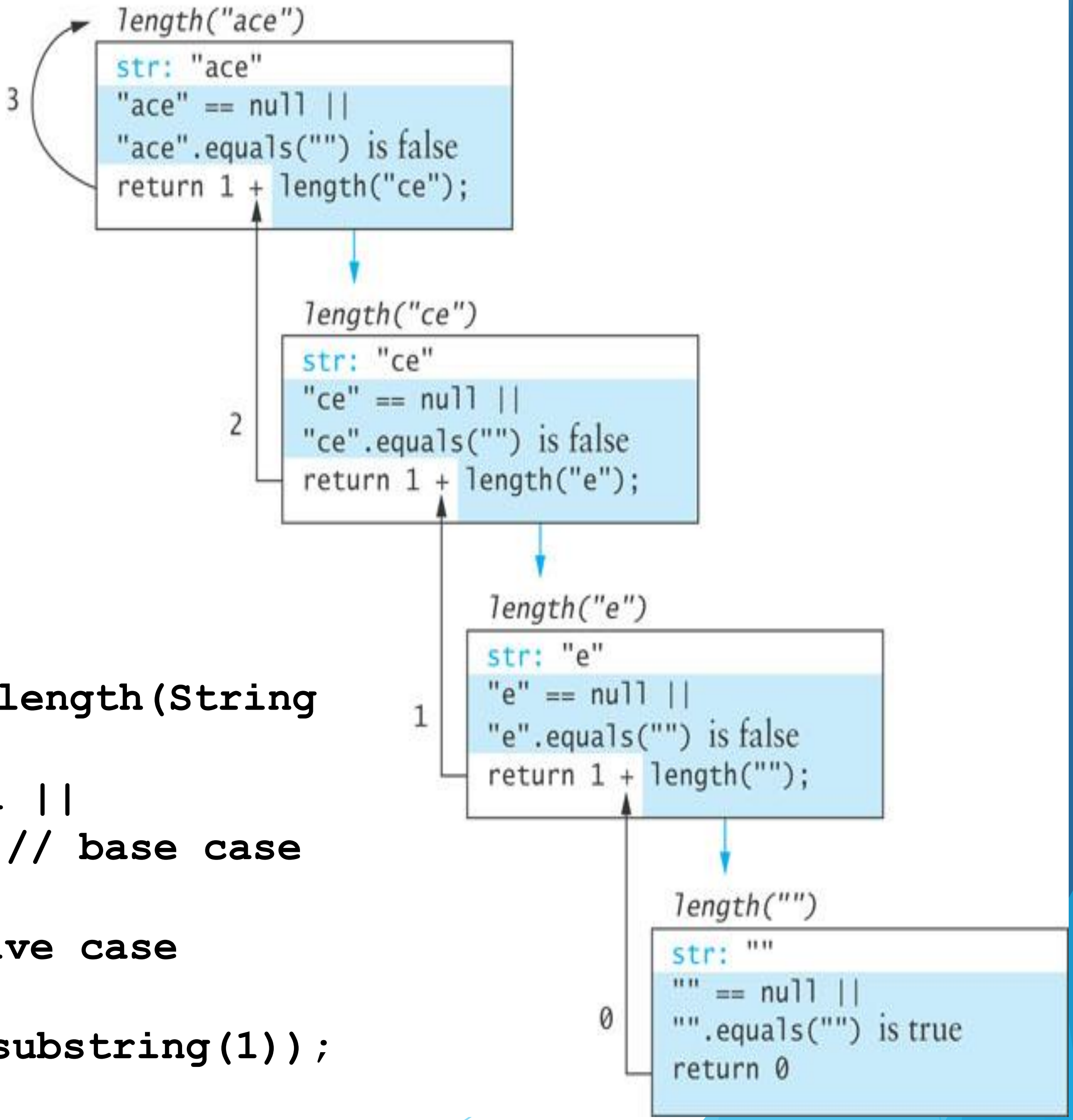
Finding the Length of a String (cont.)

```java
public class Solution {

    public static int getLengthOfString(String s){
        if (s.equals("") || s == null){
            return 0;
        }

        int sub_len = getLengthOfString(s.substring(1));
        int len = 1 + sub_len;

        return len;
    }
    public static void main(String[] args) {
        System.out.println(getLengthOfString("hello"));
    }
}
```

Run-Time Stack and Activation Frames

- Java maintains a run-time stack on which it saves new information in the form of an *activation frame*
- The activation frame contains storage for
  - method arguments
  - local variables (if any)
  - the return address of the instruction that called the method
- Whenever a new method is called (recursive or not), Java pushes a new activation frame onto the run-time stack and return will remove the activation frame from the stack and return to the previous call.

Stack Trace



```
public static int length(String
str) {
    if (str == null ||
    str.equals(""))// base case
        return 0;
    else // Recursive case
        return 1 +
        length(str.substring(1));
}
```

Problem: Sum of 1 to N

Method : Sum(n) = 1 + 2 + 3 + ...+n

Sum(1) = 1

Sum(2) = 1 + 2 => Sum(2) = Sum(1) + 2

Sum(2) = 1 + 2

Sum(3) = 1 + 2 + 3 => Sum(3) = Sum(2) + 3

Sum(n-1) = 1 + 2 + 3 + ... + (n-2) + (n-1)

Sum(n) = 1 + 2 + 3 + ... + (n-2) + (n-1) + n

Sum(n) = Sum(n-1) + n

Finding the sum of n integer

Sum(n) = Sum(n-1) + n

Recursive relationship.

This relationship establishes the general case or recursive case.

Solution is expressed in terms of solutions to smaller versions of the same problem

All other cases are base cases.

Solution is obtained directly.

Recursive case :

Sum(n) = Sum(n-1) + n (if n > 1)

Base case

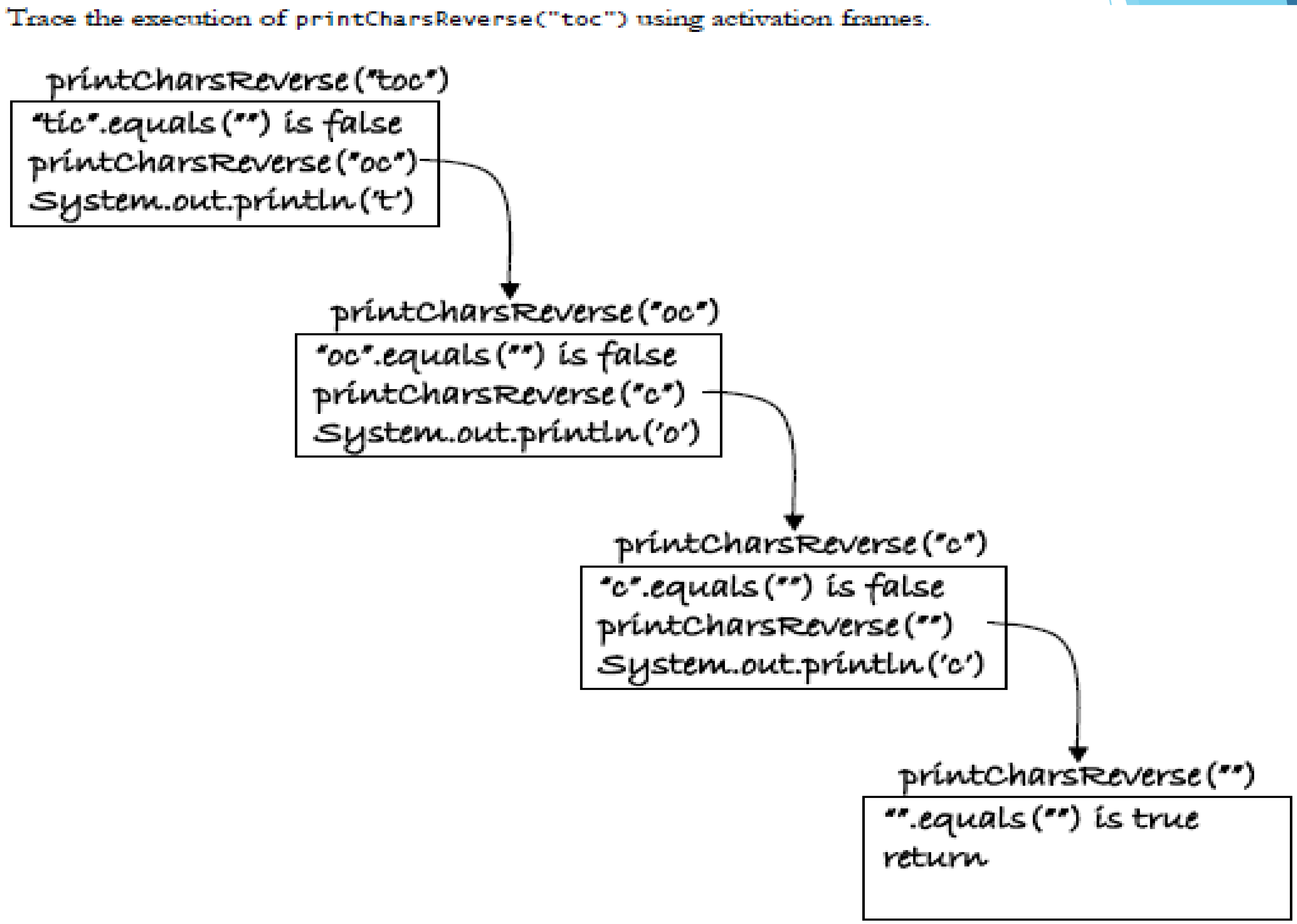
Sum(1) = 1

```
public static int Sum(int n)
{
    if (n == 1)                 //base case
      return 1;
    else
      return Sum(n-1) + n; //general case
}
```

Write a Recursive Algorithm for Printing String Characters in Reverse

```
/** Recursive method printCharsReverse
    Problem : The argument string is displayed in reverse,
           one character per line
    @param str :The input string
*/
public static void printCharsReverse(String str) {
    if (str == null || str.equals(""))
        return;
    else {
        printCharsReverse(str.substring(1));
        System.out.println(str.charAt(0));
    }
}
```

Draw the Stack diagram



Trace the execution of printCharsReverse("toc") using activation frames.

Main Point

When a recursion involves many redundant computations, one tries to write an iterative version of the method (using loops). Likewise, though all healing can in principle be done on the level of consciousness, when consciousness is not yet sufficiently established in its home, many steps of healing may be required to obtain the desired result.

Mathematical & Data Structure Examples

- Factorial
- Fibonacci
- Linear Search

Problem: Factorial Function

The factorial function on input n computes the product of the positive integers less than or equal to n.

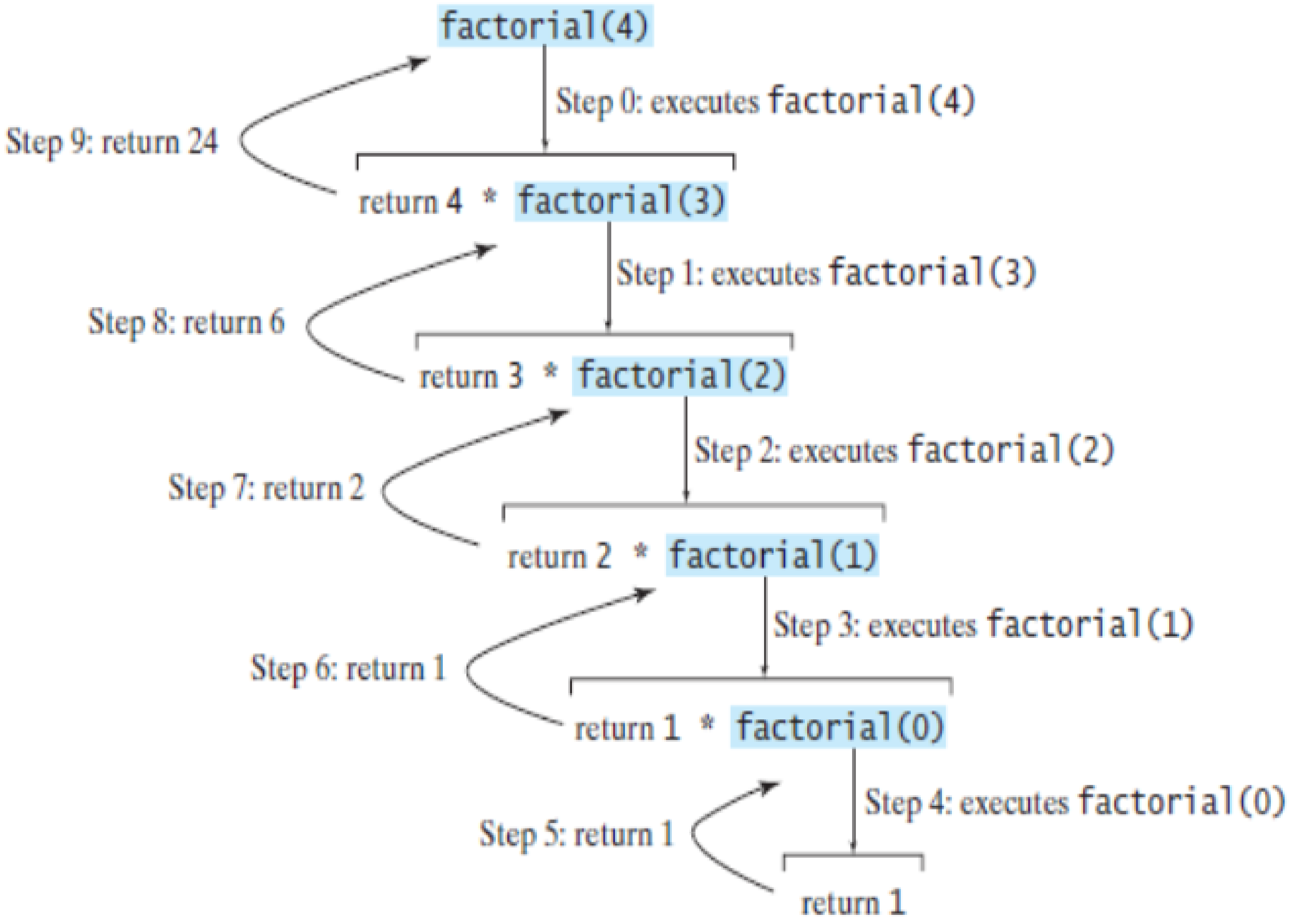
For example, 5! = 5*4*3*2*1 = 120

```
public static int fact(int num) {
    if(num == 0 || num == 1) {     //base case
        return 1;
    }
    return num * fact(num-1);
}
```

Factorial Class

```
public class factorial{
    public static void main(String[] args) {
        System.out.println("Factorial of 6 = " + fact(6));
        System.out.println("Factorial of 10 = " + fact(10));
    }

    public static int fact(int num) {
        if(num == 0 || num == 1)
            return 1;
        else
            return num * fact(num - 1);
    }
}
```

Stack Trace of Factorial



What is the Output ?

```
public class test {
    public static void main(String[] args) {
        xMethod(1234567);
    }
    public static void xMethod(int n) {
        if (n > 0) {
        System.out.print(n % 10);
        xMethod(n /  10);
        }
     }
}
// Output : 7654321
```

Fibonacci Numbers

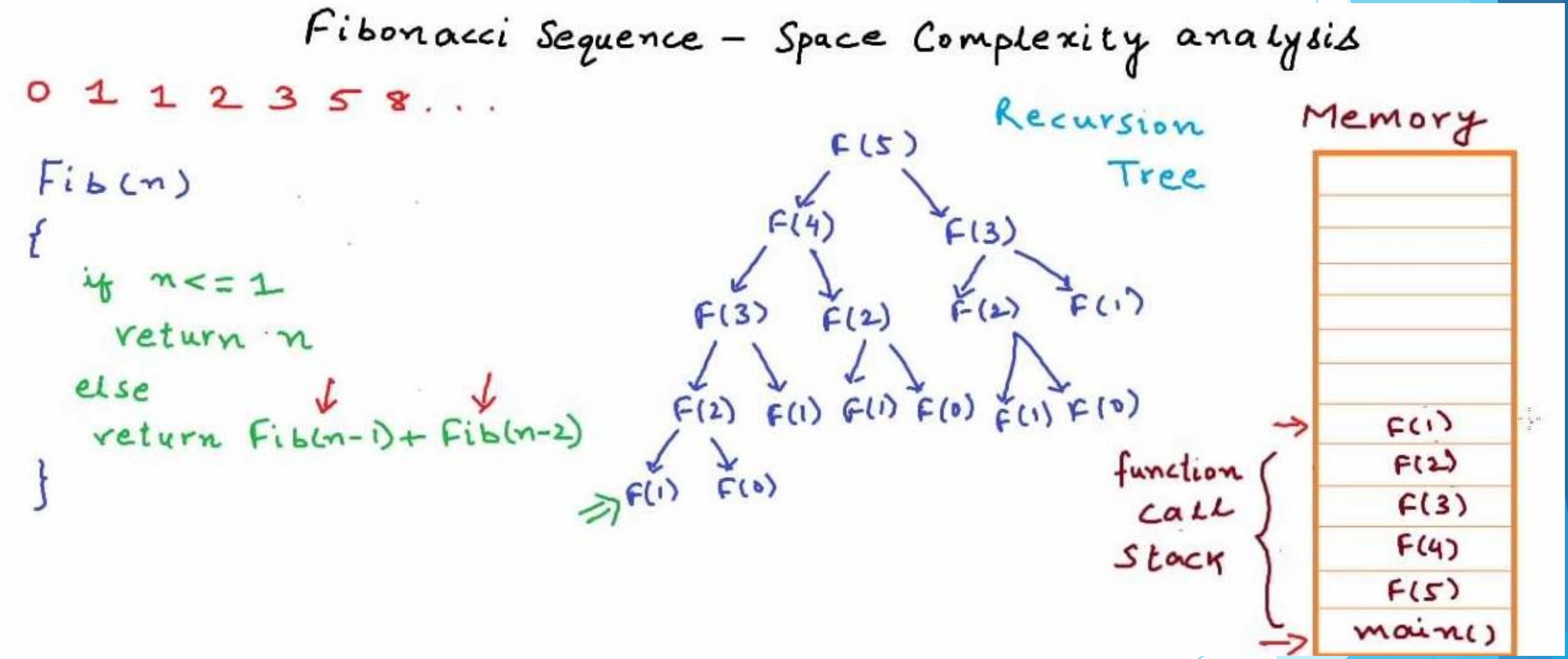
$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, F_4 = 3, F_5 = 5, \ldots,$

OR

$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ for $n > 1$

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

Fibonacci Numbers



```
Fib(n)
{
    if n<=1
       return n
    else
      return Fib(n-1)+ Fib(n-2)
}
```

Linear Search

- Searching an array can be accomplished using recursion
- Base cases for recursive search:
  - Empty array, target can not be found;

    result is -1
  - First element of the array being searched = target;

    result is the subscript of first element
- The recursive step searches the rest of the array, excluding the first element

Linear Search

```
private static int linearSearch(Object[] items, Object target, int
posFirst) {
        if (posFirst == items.length) {
            return -1;
        } else if (target.equals(items[posFirst])) {
            return posFirst;
        } else {
            return linearSearch(items, target, posFirst + 1);
        }
    }
```

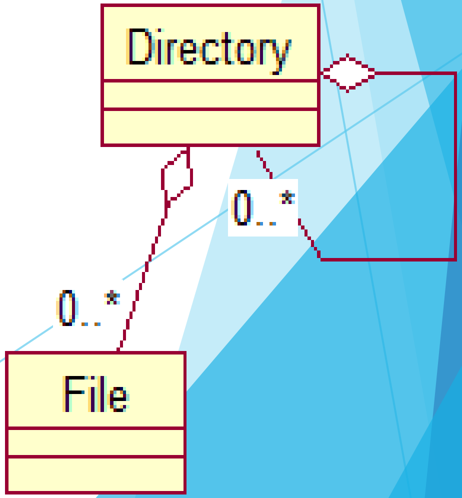
Recursion Versus Iteration

- 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 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

Object-based Recursion

- In Java, one can work with files and directories – each is represented by a particular Java class (to be discussed later). Suppose we want to write a Java method that searches for a particular file. This task will require recursion. To see what is involved, we represent the structure of a directory in the following class diagram:

Here there is a *reflexive association* from Directory to itself. This relationship at the design level suggests recursion at implementation level



Strategy

To search for a given file *file* in a given directory *dir*, the recursive strategy is:

- Get all the files and other directories that lie in the given directory *dir*
- For each of these files, compare with the given file *file* – if the same, return true
- For each directory *d* among the directories found in *dir*, recursively search for *file*
- Return false

Implementation

Rather than discuss the implementation of Directory and File in Java, we give pseudo-code to show how such a search is to be done. (See Lesson 13 for more details on File.)

```
//this is pseudo-code - not Java code
boolean searchForFile(Object file, Object startDir) {

  Object[] fileSystemObjects = startDir.getContents();

  for(Object o: fileSystemObjects) {
      //base case
      if(isFile(o) && isSameFile(o,file)) {
          return true;
      }

      if(isDirectory(o)) {
           // Recursive case
          searchForFile(file, o);
      }
  }
  //file not found in startDir
  return false;
}
```

Summary

- A Java method is ***recursive***, or exhibits recursion, if in its body it calls itself.
- A recursion is ***valid*** if the following criteria are met:
  - The method must have a base case which returns a value without making a self-call.
  - For every input to the method, the sequence of self-calls eventually leads to a self-call in which the base case is accessed.
- Sometimes recursion leads to redundant computations, which lead to slow running times (like Fibonacci). In such cases, an implementation using iteration instead of recursion should be done.
- When recursion is used to provide utility function support, the public method signature that is exposed to the client reveals only the parameters that are relevant for the client – not the special parameters that may be needed to implement the recursion.

Including Unit Testing in Your Work Environment

- A quick way to test your code as you develop is to run it from the main method as we have been doing
- A better way that is more reusable and useful for larger-scale team development is to have a parallel test project and use Junit.
- Refer : Junit-Test-File.doc from the Democode

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Recursion creates from self-referral activity

1. In Java, it is possible for a method to call itself.
2. For a self-calling method to be a legitimate recursion, it must have a base case, and whenever the method is called, the sequence of self-calls must converge to the base case.

---

3. **Transcendental Consciousness:** TC is the self-referral field of existence, at the basis of all manifest existence.
4. **Wholeness moving within itself**: In Unity Consciousness, one sees that all activity in the universe springs from the self-referral dynamics of wholeness. The "base case" – the reference point – is always the Self, realized as Brahman.