

Recursion

Lecture 7

Object-Oriented Programming

Agenda

- Why Recursion?
- Definition
- Examples
- The Three Rules of Recursion
- Recursive Methods
- Activation Records
- Tree
- Equilateral Triangle
- Koch's Snowflake
- Towers of Hanoi
- Recursive Factorial Demo
- Fibonacci Series

Russian Dolls

Each doll contains a smaller doll which in turn contains a smaller doll...



Lecture 7

Object-Orie

3

Why Recursion?

- Why learn recursion?
 - New mode of thinking.
 - Powerful programming paradigm.
- Many computations are naturally self-referential.
 - Many Sorting Techniques.
 - A folder contains files and other folders.
 - Factorial
 - Trees and Spirals

Lecture 7

Object-Oriented Programming

4

Recursion Defined

- Models problems that are *self-similar*
 - Decompose whole task into smaller, simpler sub-tasks that are similar
 - Thus, each sub-task can be solved by applying similar technique
- Whole task solved by combining solutions to sub-tasks
 - Special form of *divide and conquer*
- Task is defined in terms of itself
 - In Java, modeled by method that calls itself
 - Requires *base case* (case simple enough to be solved directly, without recursion) to end recursion; otherwise *infinite recursion*

Lecture 7

Object-Oriented Programming

5

Recursion

- Recursion solves a problem by solving a smaller instance of the same problem.
- Think *divide*, *conquer*, and *glue* when all the subproblems have the same “shape” as the original problem.



Lecture 7

Object-Oriented Programming

6

Russian Dolls

- Take Example of the Russian Dolls
 - Each doll contains a smaller doll which in turn contains a smaller doll...
- How do you know that there are no more dolls?
- Could we apply the same procedure to count the participants of this class?



Lecture 7

Object-Oriented Programming

7

Counting the Number of Students

- Lets try it out....

```

count (C)
  if (C == 1)
  {
      return 1
  }
  else
  {
      count (C-1)
      n = n + 1
  }

```

Lecture 7

Object-Oriented Programming

8

Remember Turtles

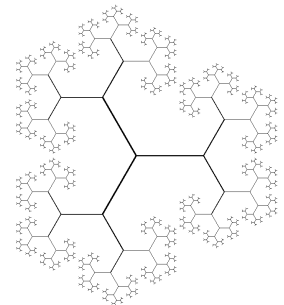
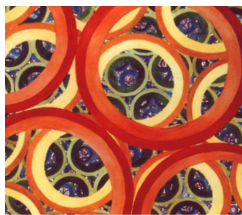
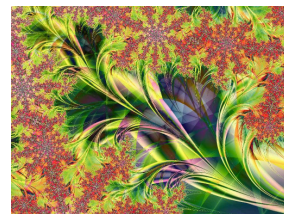
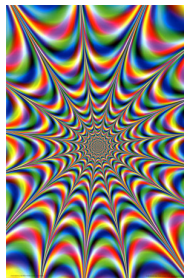
- Use turtle to draw *fractal* shapes:
 - complex shapes that are composed of smaller, simpler copies of some pattern
 - branch of mathematics developed by Benoit Mandelbrot: characteristic is self-similarity
 - natural for recursion!

Lecture 7

Object-Oriented Programming

9

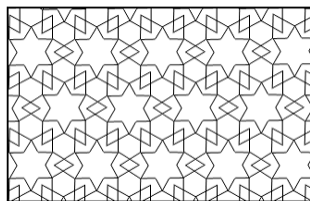
Some Example of Fractals



Lecture 7

Object-Oriented Programming

Some Islamic Art is Fractals Also



Lecture 7

Object-Oriented Programming

11

Drawing a Spiral

- Some Simple Fractal examples:
 - spiral, tree, and snowflake
- Let's start with the simplest: a spiral

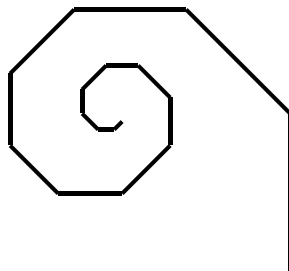
The spiral starts at a particular point. It is made of successively shorter lines, each line at an angle to the previous one. The length of the first side (the longest one), spiral's angle, and amount by which to decrement spiral's sides can be specified by the user.

Lecture 7

Object-Oriented Programming

12

Drawing a Spiral



Which angle are we turning?

Lecture 7

Object-Oriented Programming

13

The Three Rules of Recursion

- Approach every recursive problem as if it were a journey; if you follow these rules, you will complete the journey successfully.

- **RULE 1:** Find out how to take just one step.
- **RULE 2:** Break each journey down into one step plus a smaller journey.
- **RULE 3:** Know when to stop.

Drawing a Spiral Using the Turtle

- First Step: Move turtle forward to draw line and turn some degrees. What's next?



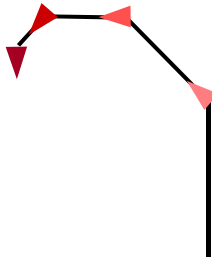
Lecture 7

Object-Oriented Programming

15

Drawing a Spiral Using the Turtle

Draw smaller line and turn! Then another, and another...



Lecture 7

Object-Oriented Programming

16

Recursive Methods

- Recursive methods
 - So far we are used to a method containing different message sends to *this*, but now we send the same message to *this*
 - Method must handle successively smaller versions of original task
- Method's variable(s):
 - As with separate methods, each invocation (message send) has its own copy of parameters and local variables, and shares access to instance variables
 - Record of code, all parameters and local variables is called *activation record*

Lecture 7

Object-Oriented Programming

17

Activation Records

- With recursion,
 - Many activations of single method may exist at once
 - At base case, as many exist as depth of recursion
 - Each has its own copy of local variables, parameters
 - Each activation of a method is stored on the “activation stack”

Very Important
Concept

Lecture 7

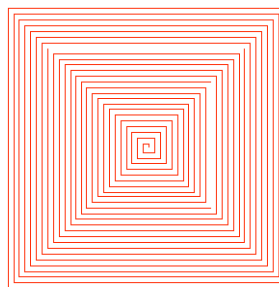
Object-Oriented Programming

18

Activation Record of the Spiral

- `public void spiral(steps,
 angle, length, increment)`
- e.g. `spiral(100 , 90 , 0 , 3)`

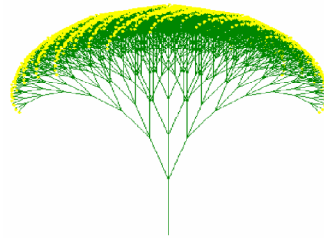
After 100 Steps



Tree

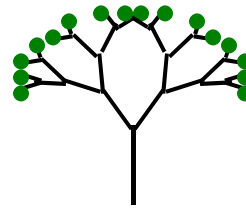
The tree is composed of a trunk that splits into two, smaller branches that sprout in opposite directions at the same angle. Each branch then splits as the trunk did until the sub-branch can no longer be seen, then it is drawn as a leaf. The length of a tree's main trunk, angle at which branches sprout, and amount to decrement each branch can be specified by user.

Or a Broccoli or a Cauliflower



Drawing a Tree using Turtles

- Broccoli or Cauliflower are essentially trees
- Compare each left branch to corresponding right branch:
 - Right branch is simply rotated copy of left branch
- Branches are themselves smaller trees!
 - branches are themselves smaller trees!
 - Branches are themselves smaller trees!
 - ...
- Our tree is recursive!
 - Base case is leaf



Lecture 7

Object-Oriented Programming

23

Drawing a Tree

```
public void tree(int levels, double
                length, double angle, double shrink)
```

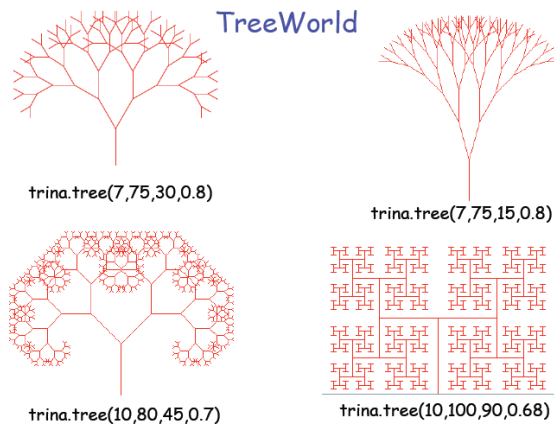
- *Levels* states the number of branches the tree is going to have.
- *Length* states the length of the initial branch.
- *Angle* states the relative angle of the two sibling branches.
- *Shrink* is the factor with which the branch length is going to shrink.

Lecture 7

Object-Oriented Programming

24

Types of Trees



Lecture 7

Object-Oriented Programming

25

Equilateral Triangle

- What is an equilateral triangle?
- How can you draw an equilateral triangle using a turtle?

Lecture 7

Object-Oriented Programming

26

Koch's Snowflake

- Invented by Swedish mathematician, Helge von Koch, in 1904; also known as *Koch Island*

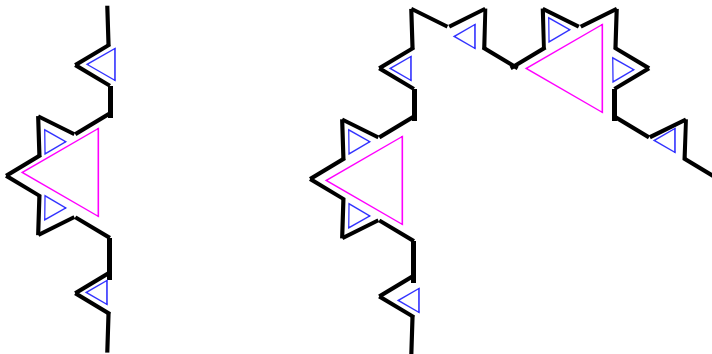
The snowflake is created by taking an equilateral triangle and partitioning each side into three equal parts. Each side's middle part is then replaced by another equilateral triangle (with no base) whose sides are one third as long as the original. This process is repeated for each remaining line segment. The length of the initial equilateral triangle's side can be given by the user.

Lecture 7

Object-Oriented Programming

27

Koch's Snowflake

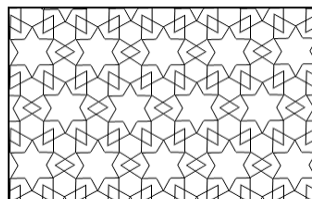
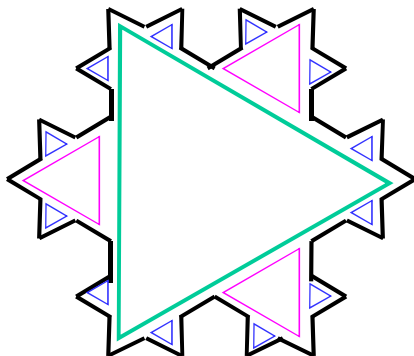


Lecture 7

Object-Oriented Programming

28

Koch's Snowflakes



Any Similarity?

Lecture 7

Object-Oriented Programming

29

Towers of Hanoi

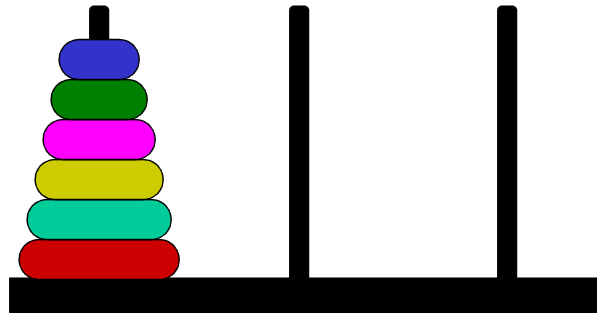
- Game invented by French mathematician Edouard Lucas in 1883
- **Goal:** move tower of n disks, each of a different size, from left-most peg to right-most peg
- **Rule 1:** no disk can be placed on top of smaller disk
- **Rule 2:** only one disk can be moved at once

Lecture 7

Object-Oriented Programming

30

Towers of Hanoi



Lecture 7

Object-Oriented Programming

31

Towers of Hanoi

- One disk:
 - move disk to final pole
- Two disks:
 - use one disk solution to move top disk to intermediate pole
 - use one disk solution to move bottom disk to final pole
 - use one disk solution to move top disk to final pole
- Three disks
 - use two disk solution to move top disks to intermediate pole
 - use one disk solution to move bottom disk to final pole
 - use two disk solution to move top disks to final pole
- In general (for n disks)
 - use $n - 1$ disk solution to move top disks to intermediate pole
 - use one disk solution to move bottom disk to final pole
 - use $n - 1$ disk solution to move top disks to final pole

Lecture 7

Object-Oriented Programming

32

Recursive Factorial Demo

by Robert Sedgewick and Kevin Wayne

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

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

n = 3

environment

fact(3)

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

The diagram shows an environment frame on the left with a black header containing the text `n = 3` and a green body containing the text `environment`. To its right is a function object for `fact(3)`. The function object has a dark blue header with `fact(3)` and a light blue body containing the following code:

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

Lecture 7 Object-Oriented Programming 35

The diagram shows an environment frame on the left with a black header containing the text `n = 3` and a green body containing the text `environment`. To its right is a function object for `fact(3)`. The function object has a dark blue header with `fact(3)` and a light blue body containing the following code:

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

Lecture 7 Object-Oriented Programming 36

n = 3
environment

```
fact(3)
static int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

n = 2
environment

```
fact(2)
static int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

Lecture 7 Object-Oriented Programming 37

n = 3
environment

```
fact(3)
static int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

n = 2
environment

```
fact(2)
static int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}
```

Lecture 7 Object-Oriented Programming 38

The diagram illustrates the execution of the factorial function `fact(3)`. It shows two environment frames: `n = 3 environment` and `n = 2 environment`. The `n = 3 environment` frame is positioned above the `n = 2 environment` frame. The `fact(3)` function body is shown in a box, with the line `else return n * fact(n-1);` highlighted in blue. The `fact(2)` function body is shown in a box below it, with the line `else return n * fact(n-1);` highlighted in blue.

```

n = 3
environment

fact(3)
static int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}

n = 2
environment

fact(2)
static int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}

```

Lecture 7 Object-Oriented Programming 39

The diagram illustrates the execution of the factorial function `fact(3)`. It shows three environment frames: `n = 3 environment`, `n = 2 environment`, and `n = 1 environment`. The `n = 3 environment` frame is positioned above the `n = 2 environment` frame, which is above the `n = 1 environment` frame. The `fact(3)` function body is shown in a box, with the line `else return n * fact(n-1);` highlighted in blue. The `fact(2)` function body is shown in a box below it, with the line `else return n * fact(n-1);` highlighted in blue. The `fact(1)` function body is shown in a box below it, with the line `else return n * fact(n-1);` highlighted in blue.

```

n = 3
environment

fact(3)
static int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}

n = 2
environment

fact(2)
static int fact(int n) {
  if (n == 0) return 1;
  else return n * fact(n-1);
}

n = 1
environment

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

```

Lecture 7 Object-Oriented Programming 40

The diagram illustrates the call stack for the recursive function `fact(3)`. It shows three frames, each representing a call to the function with a different value of `n`:

- fact(1) frame:** Located at the bottom. The label is `n = 1 environment`. The code is:


```
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```
- fact(2) frame:** Located in the middle. The label is `n = 2 environment`. The code is:


```
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```
- fact(3) frame:** Located at the top. The label is `n = 3 environment`. The code is:


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

Lecture 7 Object-Oriented Programming 41

The diagram illustrates the call stack for the recursive function `fact(3)`. It shows three frames, each representing a call to the function with a different value of `n`:

- fact(1) frame:** Located at the bottom. The label is `n = 1 environment`. The code is:


```
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```
- fact(2) frame:** Located in the middle. The label is `n = 2 environment`. The code is:


```
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```
- fact(3) frame:** Located at the top. The label is `n = 3 environment`. The code is:


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

Lecture 7 Object-Oriented Programming 42

The diagram illustrates the call stack for the recursive function `fact(3)`. It shows four stacked frames, each representing a different value of `n` (0, 1, 2, and 3). Each frame contains the function's code and is associated with an environment. The frames are arranged from bottom to top, with `fact(0)` at the base and `fact(3)` at the top. The code in each frame is:

```

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

Each frame is labeled with its corresponding `n` value and the word "environment". The frames are stacked such that each higher `n` frame is partially obscured by the one below it, showing the recursive calls. The bottom frame is labeled "n = 0 environment" and the top frame is labeled "n = 3 environment".

Lecture 7 43

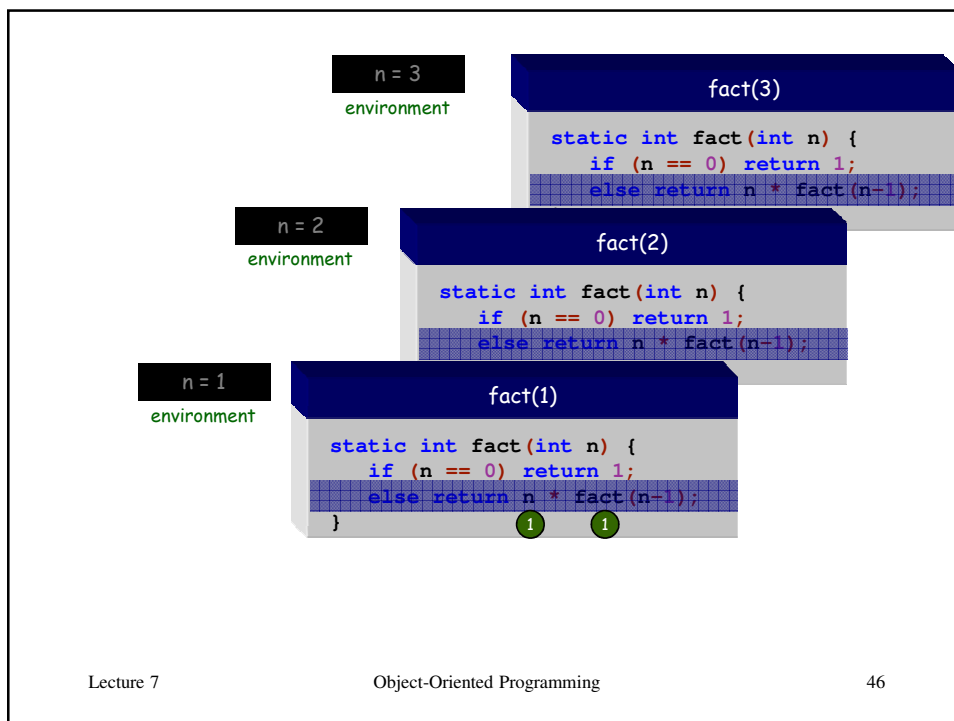
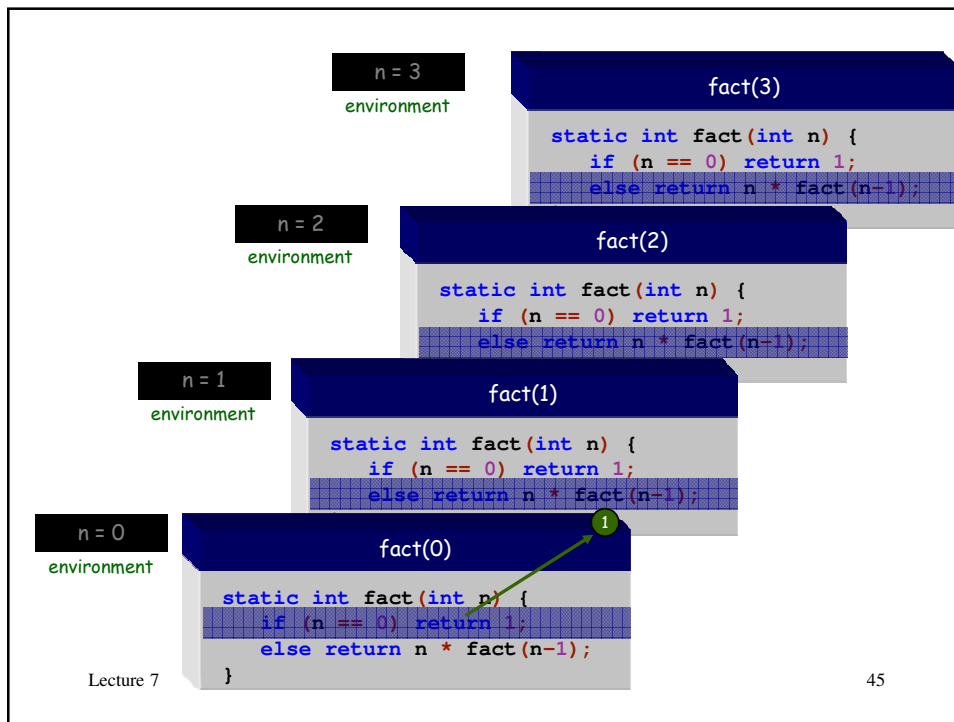
This diagram is identical to the one above, showing the call stack for `fact(3)` with environments for `n=0, 1, 2, 3`. It displays the recursive function's code in four stacked frames, each associated with an environment. The frames are arranged from bottom to top, with `fact(0)` at the base and `fact(3)` at the top. The code in each frame is:

```

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

Each frame is labeled with its corresponding `n` value and the word "environment". The frames are stacked such that each higher `n` frame is partially obscured by the one below it, showing the recursive calls. The bottom frame is labeled "n = 0 environment" and the top frame is labeled "n = 3 environment".

Lecture 7 44



n = 3 environment

```
fact(3)
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

n = 2 environment

```
fact(2)
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

n = 1 environment

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

Lecture 7 Object-Oriented Programming 47

n = 3 environment

```
fact(3)
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
```

n = 2 environment

```
fact(2)
static int fact(int n) {
    if (n == 0) return 1;
    else return n * fact(n-1);
}
2 1
```

Lecture 7 Object-Oriented Programming 48

The diagram illustrates the execution of a recursive factorial function. It shows two environment frames: one for `n = 3` and another for `n = 2`. The `n = 3` environment is associated with a `fact(3)` code block. The `n = 2` environment is associated with a `fact(2)` code block. A green arrow points from the `fact(n-1)` call in the `fact(2)` code to the `fact(2)` code block, indicating a recursive call. The `fact(2)` code block has a green circle with the number '1' under the closing brace and a green circle with the number '2' under the `fact(n-1)` call. The `fact(3)` code block has a green circle with the number '2' under the `fact(n-1)` call.

Lecture 7 Object-Oriented Programming 49

The diagram shows the environment frame for `n = 3` and the `fact(3)` code block. The `fact(3)` code block has a green circle with the number '3' under the closing brace and a green circle with the number '2' under the `fact(n-1)` call.

Lecture 7 Object-Oriented Programming 50

`n = 3`
environment

`fact(3)`

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

3
2

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

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

6

```
% java Factorial
6
```

Lecture 7

Object-Oriented Programming

51

Fibonacci Series

- Fibonacci series
 - 0, 1, 1, 2, 3, 5, 8, 13, 21...
 - fibonacci(0) = 0
 - fibonacci(1) = 1
 - $\text{fibonacci}(n) = \text{fibonacci}(n - 1) + \text{fibonacci}(n - 2)$
 - fibonacci(0) and fibonacci(1) are base cases
 - Each number in the series is sum of two previous numbers

Fibonacci Code

```
public long fibonacci( long n )
{
    // base case
    if ( n == 0 || n == 1 )
        return n;

    // recursive step
    else
        return fibonacci( n - 1 ) + fibonacci( n - 2 );

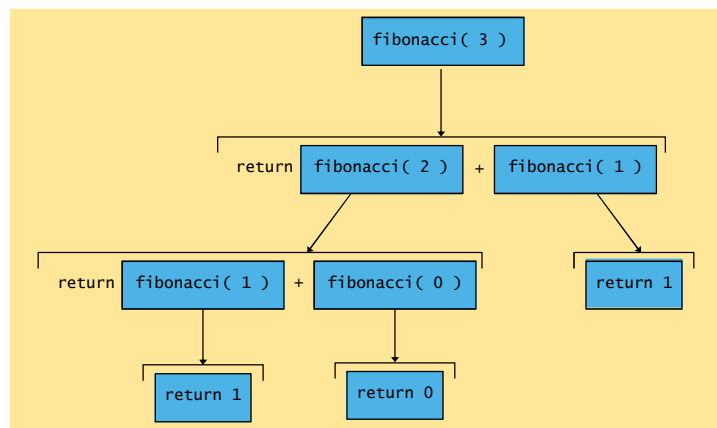
} // end method fibonacci
```

Lecture 7

Object-Oriented Programming

53

Fibonacci Executed



Lecture 7

Object-Oriented Programming

54

Reading

Book Name: Object Oriented Programming in Java™ - A Graphical Approach

Author: Kathryn E. Sanders & Andries van Dam

Content: Chapter # 12

Acknowledgements

- While preparing this course I have greatly benefited from the material developed by the following people:
 - Andy Van Dam (Brown University)
 - Mark Sheldon (Wellesley College)
 - Robert Sedgewick and Kevin Wayne (Princeton University)
 - Mark Guzdial and Barbara Ericsson (Georgia Tech)
 - Richard Halterman (Southern Adventist University)