

Recursion

COSC 1P03 – Lecture 08 (Spring 2024)

Maysara Al Jumaily

amaysara@brocku.ca

Brock University



Tuesday June 18, 2024

Total slides: 40

Lecture Outline

- 01 Introduction
- 02 Three Conditions Must be Satisfied
- 03 A Simple and Complete Recursion Example
- 04 A More Complex Recursion Example
- 05 Instance Variables and Recursion
- 06 Golden Conclusion of Recursion

What is Recursion

- Recursion is a method calling itself.

- For example:

```
private void recursion(int x){  
    recursion(x); //recursively calling itself.  
}
```

- The above example shows what recursion is but it is very poor!
- This is because we have to satisfy three conditions to obtain a successful recursive method.

Three Conditions that must be Satisfied

All conditions must be satisfied to obtain valid recursion.

- **Condition 1:** A recursive method must have a base case and a recursive case.
- **Condition 2:** We will recursively call a method finite amount of times.
 - by finite, we mean n times (n can be any number: 0, 1, 5, 31, 1021, etc) but **not** infinity.
 - You will get `StackOverflowException` if you recursively loop infinitely many times.
- **Condition 3:** Every time the method calls itself, we **must** get closer to the base case.
 - Once we hit the base case, the base case is designed to stop the recursion.

Why Previous Example is Poor?

- The example,

```
private void recursion(int x){  
    recursion(x); //recursively calling the method.  
}
```

is poor because of three reasons:

- It didn't have a base case.
- It recursively looped infinite times (will eventually give `StackOverflowException`).
- Since the base case didn't exist, every time it recursively called itself, it didn't get closer to the *non-existence* base case.

A Simple and Complete Recursion Example

Goal: Create a recursive method

```
private int sum(int x){  
    :  
}
```

which returns the sum recursively from

$$x + (x - 1) + \dots + 1 + 0.$$

Writing `System.out.println(sum(5));` should print out 15 by performing:

$$5 + 4 + 3 + 2 + 1 + 0 = 15.$$

$$5 + 4 + 3 + 2 + 1 + 0 = 15.$$

Approach:

- The first thing is to think about the base case. It should stop the recursion. What is it?
- **Base case:** Stop recursion when x is 0.
- Secondly, what should be the recursive call/recursive case?
 - If the current number passed is 5, then, the next should be 4.
 - If the current number passed is 4, then, the next should be 3.
 - If the current number passed is 3, then, the next should be 2.
 - If the current number passed is x , then, the next should be $x - 1$.
- **Recursive case:**
 - Informally, it will be something along $x + (x - 1)$.
 - More formally, it will be $x + \text{sum}(x - 1)$. We will discuss this furthermore.

A Simple and Complete Recursion Example Continued

Taking care of the base case:

```
private int sum(int x){  
    //base case (once reached we stop the recursion).  
    if(x == 0){  
        //no recursion here  
        return 0;  
    }  
    //more code should be here for the recursive call.  
}
```


A Simple and Complete Recursion Example Continued

The complete method (with the base case and recursive call) will be:

```
private int sum(int x){  
    //base case (once reached, we stop the recursion).  
    if(x == 0){  
        //no recursion here  
        return 0;  
    } else { //perform recursive case/recursive call.  
        //we said it will be x + (x - 1).  
        return x + sum(x - 1);  
    }  
}
```

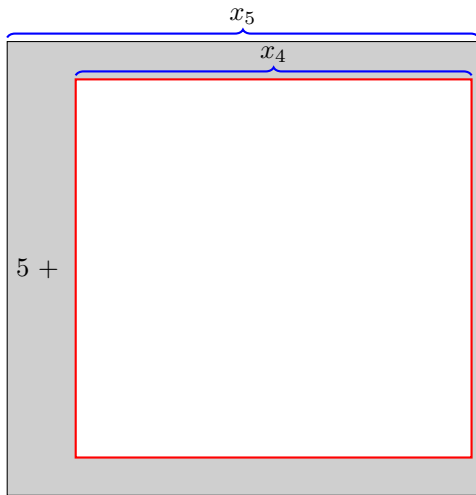
The next slides will explain this visually using memory.

Visualization Notation

- Recursion will always have a two-way pass to complete recursion.
 - We first *inspect* and then *evaluate*.
- The three keywords used (*inspect*, *evaluate* and *snapshot*) are only used in this presentation. They are not found in other textbooks.
- Grey boxes represent inspecting and green boxes represent evaluating.
- The labels x_5, x_4, x_3, x_2, x_1 , and x_0 are there to visually label what is going on. They don't practically exist.

Inspecting 2

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

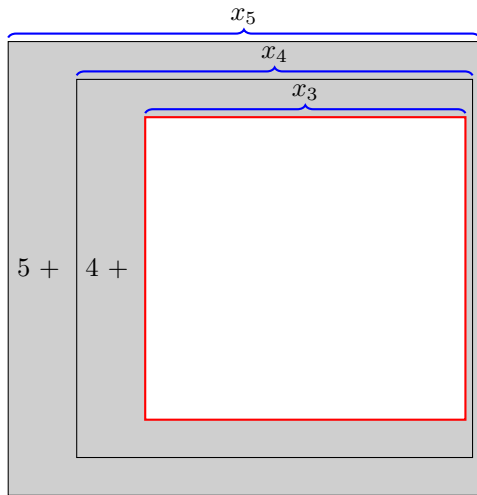


Memory

\vdots
$x_5 = 5 + x_4$
\vdots

Inspecting 4

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

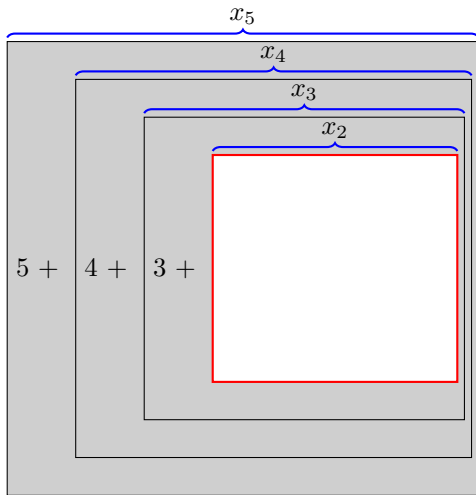


Memory

\vdots
$x_4 = 4 + x_3$
$x_5 = 5 + x_4$
\vdots

Inspecting 6

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

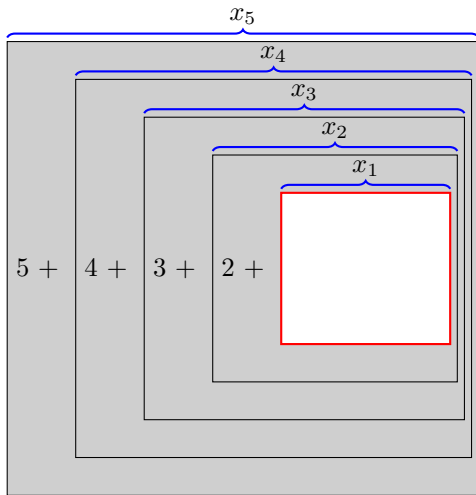


Memory

\vdots
$x_3 = 3 + x_2$
$x_4 = 4 + x_3$
$x_5 = 5 + x_4$
\vdots

Inspecting 8

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

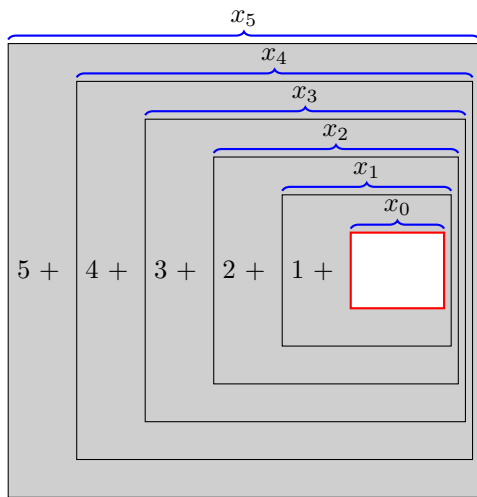


Memory

\vdots
$x_2 = 2 + x_1$
$x_3 = 3 + x_2$
$x_4 = 4 + x_3$
$x_5 = 5 + x_4$
\vdots

Inspecting 10

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

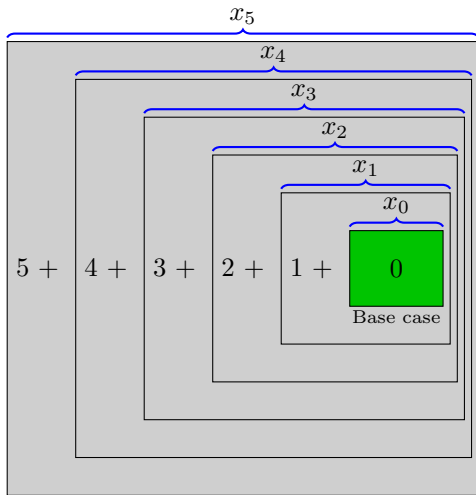


Memory

\vdots
$x_1 = 1 + x_0$
$x_2 = 2 + x_1$
$x_3 = 3 + x_2$
$x_4 = 4 + x_3$
$x_5 = 5 + x_4$
\vdots

Evaluating 2

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

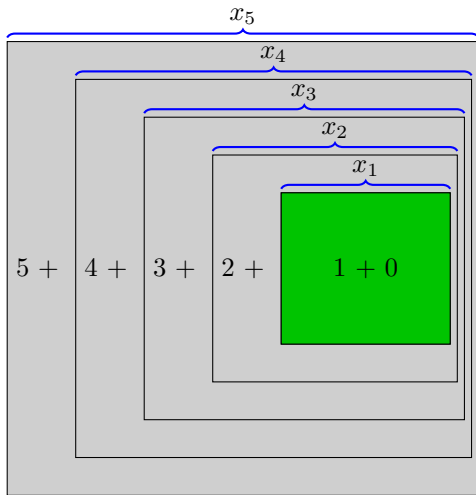


Memory

\vdots
$x_0 = 0$
$x_1 = 1 + x_0$
$x_2 = 2 + x_1$
$x_3 = 3 + x_2$
$x_4 = 4 + x_3$
$x_5 = 5 + x_4$
\vdots

Evaluating 4

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

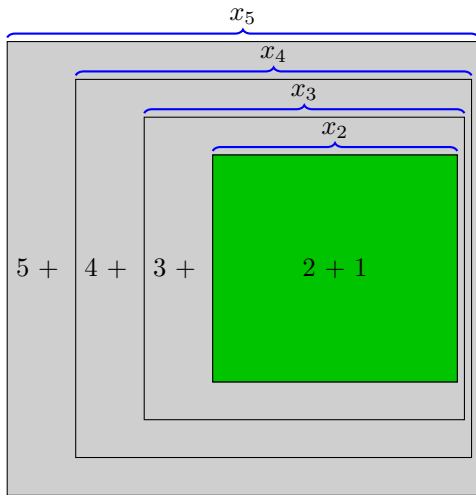


Memory

\vdots
$x_1 = 1 + \overset{x_0}{0} = 1$
$x_2 = 2 + x_1$
$x_3 = 3 + x_2$
$x_4 = 4 + x_3$
$x_5 = 5 + x_4$
\vdots

Evaluating 6

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

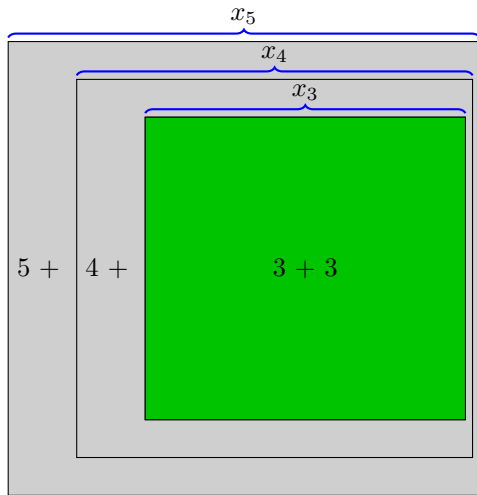


Memory

\vdots
$x_2 = 2 + x_1 = 3$
$x_3 = 3 + x_2$
$x_4 = 4 + x_3$
$x_5 = 5 + x_4$
\vdots

Evaluating 8

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

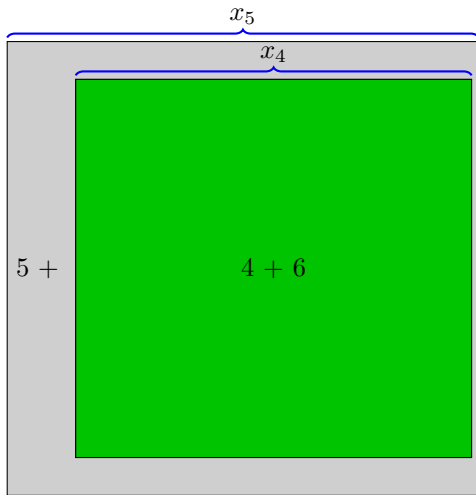


Memory

\vdots
$x_3 = 3 + \overset{x_2}{3} = 6$
$x_4 = 4 + x_3$
$x_5 = 5 + x_4$
\vdots

Evaluating 10

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.

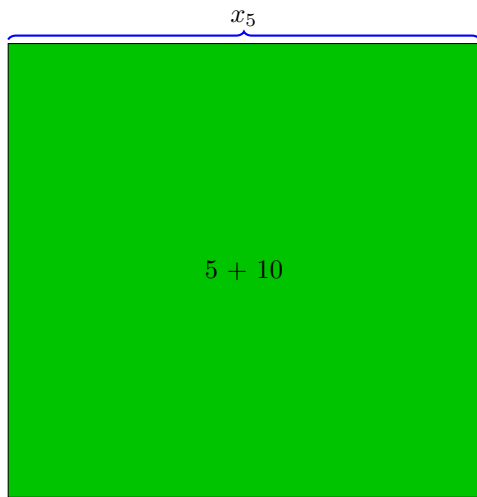


Memory

\vdots
$x_4 = 4 + \overset{x_3}{6} = 10$
$x_5 = 5 + x_4$
\vdots

Evaluating 12

Goal: $5 + 4 + 3 + 2 + 1 + 0 = 15$.



Memory

\vdots
$x_5 = 5 + 10^{x_4} = 15$
\vdots

Output

Calling `System.out.println(sum(5));` will recursively find 15 as the answer and prints it on the console. Keep in mind that only 15 is printed out, the process of obtaining $0, 1, \dots, 5$ was only in memory, it is not visible to the user.

Remarks

- As long as we code the method correctly, the following next bullet points will be executed automatically in order.
- Every time we inspect, we take a snapshot of the current variables involved to store in memory.
 - A snapshot contains the current values of local variables used and any code that needs to be executed **after** the recursive call (this will be discussed next slide).
- Every time we inspect, we take a snapshot of the current variables and code that needs to be executed involved to store in memory.
- Once we reach the base case, we can then start removing the snapshots away from memory.
- When evaluating, recursion will remove (in order) the newest snapshot, then second newest, . . . , and lastly, the oldest snapshot.
- This order of inserting and removing uses a *stack* LIFO (last-in-first-out).

A More Complex and Complete Recursion Example

Consider the following method `printEven(int n)` which accepts an `n` and prints out:

$$0, 2, \dots, (n - 4), (n - 2), n.$$

i.e., the even numbers between 0 and n (including 0 and including n). For example, calling `printEven(6);` should print out (in order) from smallest to largest:

$$0, 2, 4, 6.$$

```
/**
 * Prints (in order) the even numbers between 0 and n.
 *
 * @param n non-negative even number.
 */
private void printEven(int n){
    :
}
```


0, 2, 4, 6

Approach:

- The first thing is to think about the base case. It should stop the recursion. What is it (don't let the order fool you)?
- **Base case:** Stop recursion when `n` is 0.
- Secondly, what should be the recursive call/recursive case?
 - If the current number passed is 6, then the next should be 4.
 - If the current number passed is 4, then the next should be 2.
 - If the current number passed is 2, then the next should be 0.
 - If the current number passed is n , then the next should be $n - 2$.
- **Recursive case:**
 - Informally, it will be something along `(n - 2)`.
 - More formally, it will be `printEven(n - 2)`.

A Simple and Complete Recursion Example Continued

- The complete method (with the base case and recursive call) will be:

```
private void printEven(int n){  
    //base case (once reached, we stop the recursion).  
    if(n == 0){  
        //no recursion here  
        return;  
    } else { //perform recursive case/recursive call.  
        //we said it will be (n - 2).  
        printEven(n - 2); //calling itself  
    }  
}
```

- Where should `System.out.println();` be place in the code? What should we pass to it?

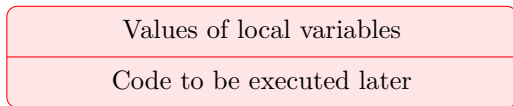
A Simple and Complete Recursion Example Continued

```
private void printEven(int n){  
    //(1) place print statement here  
    if(n == 0){//base case (stop the recursion).  
        System.out.println(n);//print 0  
        return;//no recursion, return nothing  
    } else { //perform recursive case/call.  
        //(2) place print statement here  
        printEven(n - 2);//calling itself  
        //save whatever code below in the snapshot  
    }  
    //(3) place print statement here  
}
```

- `return;` breaks out of the method (similar to `break;` in a loop).
- Try yourself, place `System.out.println(n);` in (1), (2), (3). Note, have a single `System.out.println(n);` in method.

Defining an Activation Record (Snapshot)

- A snapshot is what other books refer to as *Activation Record*.
- In recursion, a snapshot is defined to store necessary information:
 - The values of local variables.
 - The code *after* the recursive call, *i.e.*, `System.out.println(n);`.
 - Other complex information which we will not include here such as dynamic link etc.
- A snapshot in this presentation is visually represented as:

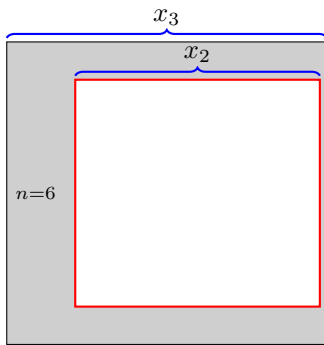


- Every time we call the recursive case, Java will automatically take a snapshot and place it in memory.
- Note, the simple example we discussed earlier only stored the local variable (`n`). There was no need to use the complete snapshot.

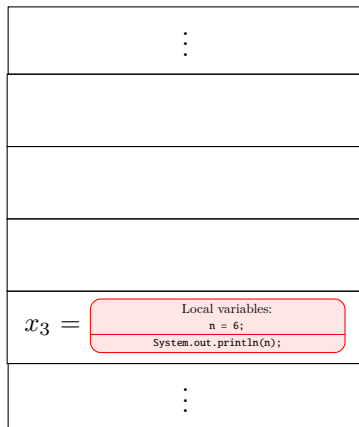
Inspecting 2

Goal: 0, 2, 4, 6

```
private void printEven(int n){  
    if(n == 0){  
        //base case  
        System.out.println(n);  
        return;  
    } else {  
        //recursive call  
        printEven(n - 2);  
        System.out.println(n);  
    }  
}
```



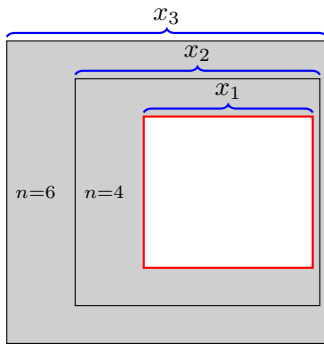
Memory



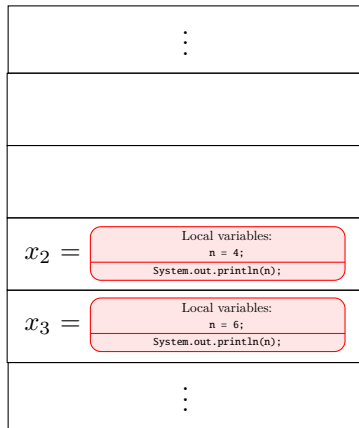
Inspecting 4

Goal: 0, 2, 4, 6

```
private void printEven(int n){  
    if(n == 0){  
        System.out.println(n);  
        return;  
    } else {  
        printEven(n - 2);  
        System.out.println(n);  
    }  
}
```



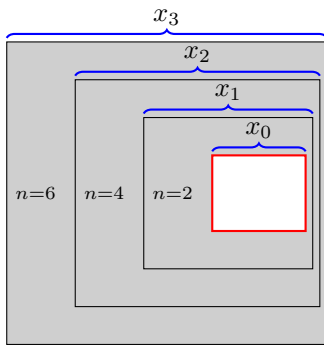
Memory



Inspecting 6

Goal: 0, 2, 4, 6

```
private void printEven(int n){  
    if(n == 0){//base case  
        System.out.println(n);  
        return;  
    } else {//recursive call  
        printEven(n - 2);  
        System.out.println(n);  
    }  
}
```



Memory

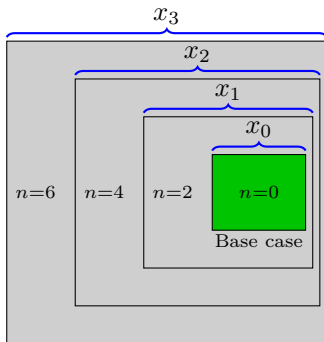
	⋮
$x_1 =$	<div>Local variables: <code>n = 2;</code> <code>System.out.println(n);</code></div>
$x_2 =$	<div>Local variables: <code>n = 4;</code> <code>System.out.println(n);</code></div>
$x_3 =$	<div>Local variables: <code>n = 6;</code> <code>System.out.println(n);</code></div>
	⋮

Evaluating 2

Goal: 0, 2, 4, 6

Output: x_0
0

```
private void printEven(int n){  
    if(n == 0){  
        System.out.println(n);  
        return;  
    } else {  
        printEven(n - 2);  
        System.out.println(n);  
    }  
}
```



Memory

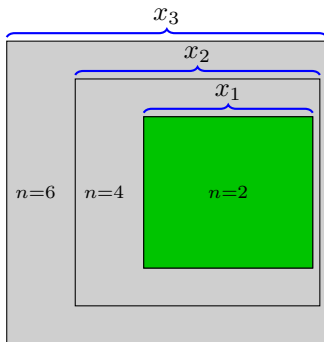
	⋮
$x_0 =$	<div>Local variables: <code>n = 0;</code> <code>System.out.println(n);</code></div>
$x_1 =$	<div>Local variables: <code>n = 2;</code> <code>System.out.println(n);</code></div>
$x_2 =$	<div>Local variables: <code>n = 4;</code> <code>System.out.println(n);</code></div>
$x_3 =$	<div>Local variables: <code>n = 6;</code> <code>System.out.println(n);</code></div>
	⋮

Evaluating 4

Goal: 0, 2, 4, 6

Output: x_0 , x_1
0, 2

```
private void printEven(int n){  
    if(n == 0){  
        System.out.println(n);  
        return;  
    } else {  
        printEven(n - 2);  
        System.out.println(n);  
    }  
}
```



Memory

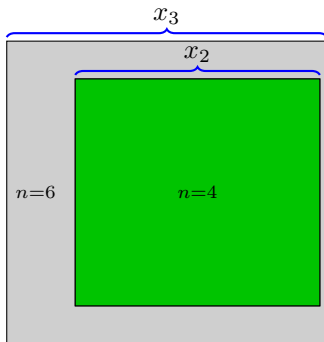
	⋮
$x_1 =$	Local variables: n = 2; System.out.println(n);
$x_2 =$	Local variables: n = 4; System.out.println(n);
$x_3 =$	Local variables: n = 6; System.out.println(n);
	⋮

Evaluating 6

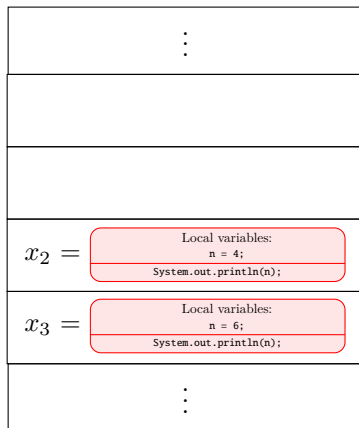
Goal: 0, 2, 4, 6

Output: x_0 , x_1 , x_2
0, 2, 4

```
private void printEven(int n){  
    if(n == 0){  
        System.out.println(n);  
        return;  
    } else {  
        printEven(n - 2);  
        System.out.println(n);  
    }  
}
```



Memory

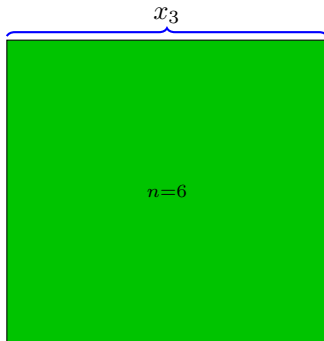


Evaluating 8

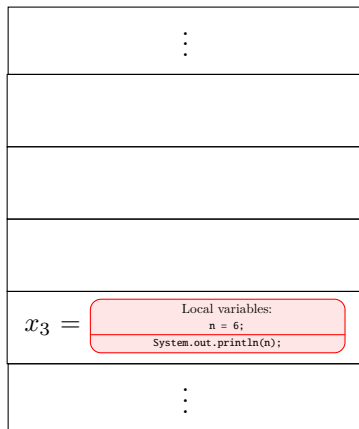
Goal: 0, 2, 4, 6

Output: x_0 , x_1 , x_2 , x_3
0, 2, 4, 6

```
private void printEven(int n){  
    if(n == 0){  
        System.out.println(n);  
        return;  
    } else {  
        printEven(n - 2);  
        System.out.println(n);  
    }  
}
```



Memory



Instance Variables and Recursion

- The instance variables are **NOT** stored inside of activation records.
- There will be one and only one value of an instance variable and is accessed/modified by all recursive calls.
- This is not a difficult concept to understand because it is the equivalent to having one instance variable and three normal/non-recursive methods. These three methods will “share” that instance variable and all of them have the ability to access/modify it.
- Here is an example on how instance variables are managed during recursion:
 - Suppose we have variable `int x = 1;` as an instance variable.
 - Throughout all of the recursive calls, it will contain the value `1` and is *not* stored in the activation record.
 - Suppose only the first recursive call will increment it by 1, then `x` is now `2`.
 - Now, assume that all other recursive calls will print out the value `x`, but not modify it. Since all recursive calls share the variable `x`, they will all print the current value of `x`, which is `2`.
 - Suppose that the recursive logic has ended and we left the recursive method. The value of `x`, is still `2`, because one recursive call incremented it by 1.

The Golden Rule of Recursion

- The three conditions must be satisfied:
 - **Condition 1:** A recursive method must have a base case and recursive case.
 - **Condition 2:** We will recursively call a method *finite* amount of times. Otherwise, you will encounter `StackOverflowException`.
 - **Condition 3:** Every time the method calls itself, we **must** get closer to the base case.
- Every time we inspect (**just before the program recursively call itself**), a snapshot is taken and placed it in memory.
- Once the base case is reached, we evaluate by popping out the snapshots from the memory in a last-in-first-out fashion.
- In more complex scenarios, recursion doesn't follow a linear pattern of inspecting first then evaluating until we have no snapshots (activation records) in the memory. It could be that we inspect, inspect, inspect, evaluate, inspect, evaluate, evaluate, inspect, inspect, evaluate, etc. The pattern continues until there are no snapshots left in memory, our recursion is then complete!
- Instance variables are **NOT** stored in the activation records and can be accessed/modified by all recursive calls.

Extra thoughts about recursion I

- A typical recursive sequence is referred to as Fibonacci sequence that is defined as two base cases $f_0 = 0$ and $f_1 = 1$. The recursive case is $f_n = f_{n-1} + f_{n-2}$, or coding-wise:

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

- The \mathcal{O} notation of the Fibonacci sequence is exponential (which is very bad and slow!), it is $\mathcal{O}(2^n)$
- Any recursive method could be converted to an iterative method and vice-versa.

Extra thoughts about recursion II

- The iterative approach is much faster $\mathcal{O}(n)$, and given as:

```
public int fibonacciIterative(int n) {  
    int prev = 0;  
    int current = 1;  
    int total = 0;  
    for (int i = 0; i < n; i++) {  
        prev = current;  
        current = total;  
        total = prev + current;  
    }  
    return total;  
}
```

Passing a Parameter Into a Recursive Call

- Suppose we wanted to store the current value of an `int`eger and also pass the next number to the recursive call. For example, the current parameter `x` has the value `5` and wanted to pass the value `6` to the recursive call.
- It is tempting to write `x++;`, but this will increase and update the variable `x`. Instead, we don't want to change it but pass the next `int`eger value. Hence, we would place the value of `x` in the parameter of the recursive method and add one to it:
`recursion(x + 1);`
- That way, the value of `x` has not been modified, and the value plus one is passed to the recursive call.