# Recursion

## COSC 1P03 – Tutorial 7 (Winter 2021)

Maysara Al Jumaily
amaysara@brocku.ca

Brock University

March $17^{\text{th}}$ – $18^{\text{th}}$, 2021

# Presentation Outline

# The Recordings of Tutorials

- Must be logged-in to access the recordings of tutorials. Tutorials are Wednesdays (6pm – 7pm) and Thursdays (11am – 12pm):

| | Wednesdays | Thursdays |
|---|---|---|
| | Wednesdays Playlist | Thursdays Playlist |
| Tutorial 01 | January $20^{th}$, 2021 | January $21^{st}$, 2021 |
| Tutorial 02 | January $27^{th}$, 2021 | January $28^{th}$, 2021 |
| Tutorial 03 | February $03^{rd}$, 2021 | February $04^{th}$, 2021 |
| Tutorial 04 | February $10^{th}$, 2021 | February $11^{th}$, 2021 |
| Tutorial 05 | March $3^{rd}$, 2021 | March $4^{th}$, 2021 |
| Tutorial 06 | March $10^{th}$, 2021 | March $11^{th}$, 2021 |
| Tutorial 07 | March $17^{th}$, 2021 | March $18^{th}$, 2021 |
| Tutorial 08 | March $24^{th}$, 2021 | March $25^{th}$, 2021 |
| Tutorial 09 | March $31^{st}$, 2021 | April $1^{st}$, 2021 |
| Tutorial 10 | April $7^{th}$, 2021 | April $8^{th}$, 2021 |

# What is Recursion

- Recursion a is method calling itself.

# What is Recursion

- Recursion a is method calling itself.
- For example:

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

# What is Recursion

- Recursion a is 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!

# What is Recursion

- Recursion a is 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 musts be satisfied to obtain valid recursion.

- **Condition 1:** A recursive method must have a base case and a recursive case.

# Three Conditions that must be Satisfied

**All** conditions musts 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.

# Three Conditions that must be Satisfied

**All** conditions musts 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.

# Three Conditions that must be Satisfied

**All** conditions musts 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.

# Three Conditions that must be Satisfied

**All** conditions musts 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.

# Three Conditions that must be Satisfied

**All** conditions musts 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,

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

  is poor because of three reasons:

# Why Previous Example is Poor?

- The example,

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

  is poor because of three reasons:
  - It didn’t have a base case.

# 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`).

# Why Previous Example is Poor?

- The example,

```java
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

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

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

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

# A Simple and Complete Recursion Example Continued

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

**Approach:**

- The first thing is to think about the base case. It should stop the recursion. What is it?

# A Simple and Complete Recursion Example Continued

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

**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.

# A Simple and Complete Recursion Example Continued

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

**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?

# A Simple and Complete Recursion Example Continued

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

**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.

# A Simple and Complete Recursion Example Continued

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

**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.

# A Simple and Complete Recursion Example Continued

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

**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.

# A Simple and Complete Recursion Example Continued

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

**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 $x - 1$.

# A Simple and Complete Recursion Example Continued

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

**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:**

# A Simple and Complete Recursion Example Continued

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

**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)`.

# A Simple and Complete Recursion Example Continued

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

**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 + 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);
    }
}
```

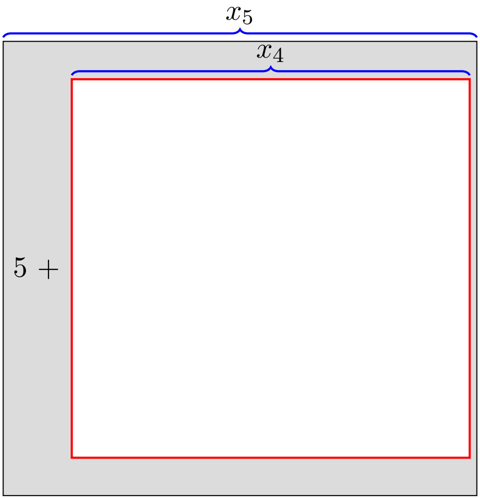
The next slides will explain this visually using memory.

# 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 1

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

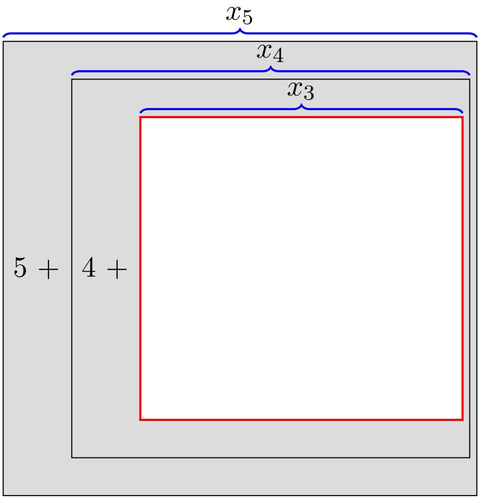



Memory

| |
|---|
| $\vdots$ |
| |
| |
| |
| |
| |
| $x_5 = 5 + x_4$ |
| $\vdots$ |

# Inspecting 2

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

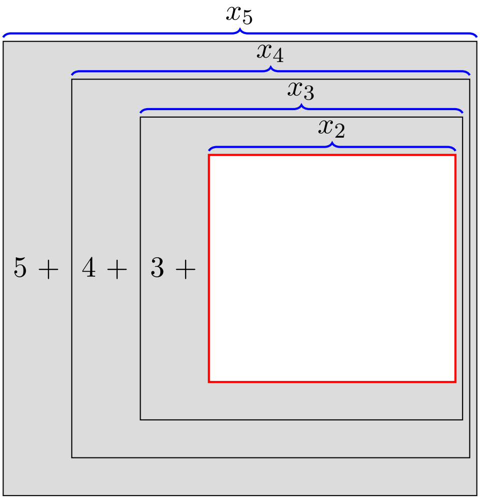



Memory

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

# Inspecting 3

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

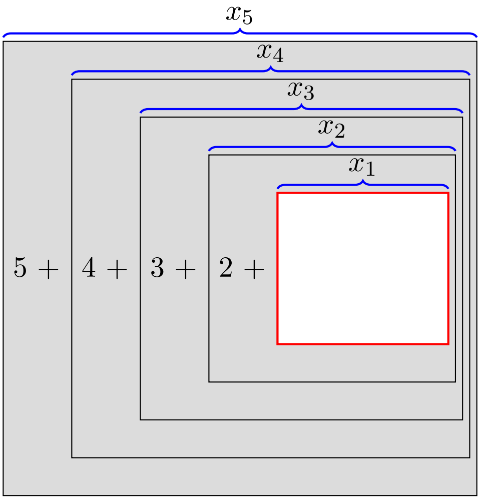



Memory

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

# Inspecting 4

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

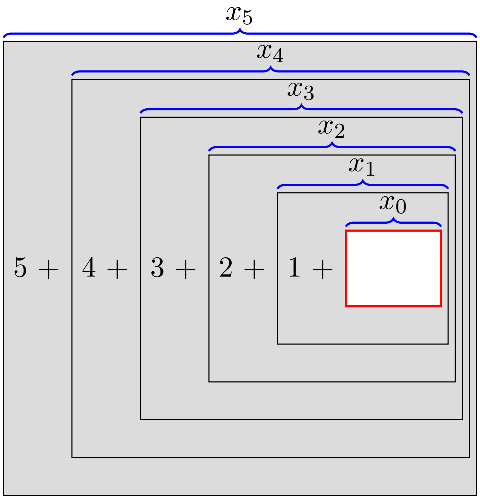



Memory

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

# Inspecting 5

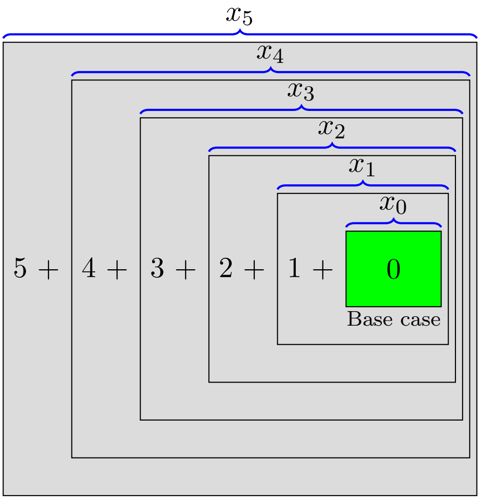
**Goal:** $5 + 4 + 3 + 2 + 1 + 0 = 15.$




Memory

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

**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 2

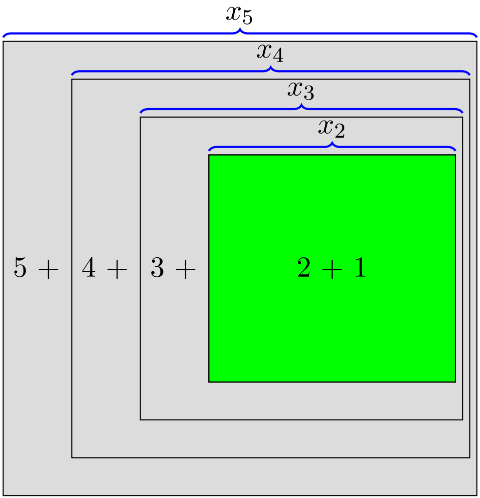
**Goal:** $5 + 4 + 3 + 2 + 1 + 0 = 15.$

$$\overbrace{5 + \overbrace{4 + \overbrace{3 + \overbrace{2 + \overbrace{1 + 0}^{x_1}}^{x_2}}^{x_3}}^{x_4}}^{x_5}$$

| 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 3

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

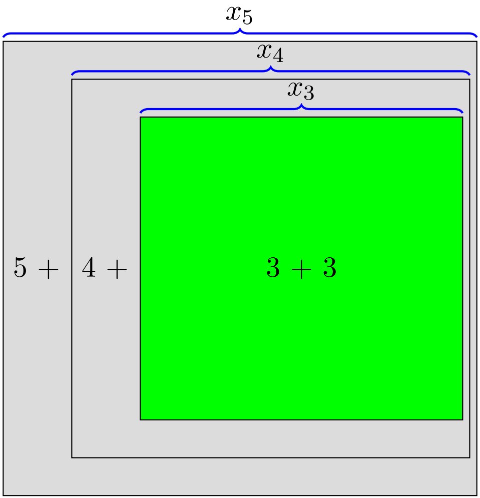



Memory

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

# Evaluating 4

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

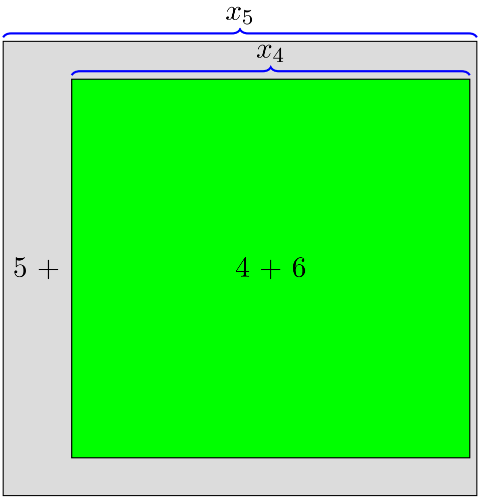



Memory

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

# Evaluating 5

**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 6

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

$$\overbrace{5 + 10}^{x_5}$$

Memory

| |
|---|
| $\vdots$ |
| |
| |
| |
| |
| |
| $x_5 = 5 + \overset{x_4}{10} = 15$ |
| $\vdots$ |

# 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 $0, 1, \ldots, 5$ 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:

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

*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){
          ⋮
}
```

# A More Complex and Complete Recursion Example Continued

$$0, 2, 4, 6$$

**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)?

$$0, 2, 4, 6$$

**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.

# A More Complex and Complete Recursion Example Continued

$$0, 2, 4, 6$$

**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?

$$0, 2, 4, 6$$

**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.

# A More Complex and Complete Recursion Example Continued

$$0, 2, 4, 6$$

**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.

# A More Complex and Complete Recursion Example Continued

$$0, 2, 4, 6$$

**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.

$$0, 2, 4, 6$$

**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 $n - 2$.

# A More Complex and Complete Recursion Example Continued

$$0, 2, 4, 6$$

**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:**

$$0, 2, 4, 6$$

**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)`.

$$0, 2, 4, 6$$

**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

```java
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
        System.out.println(n);//print statement
    }
    //(#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 a snapshot

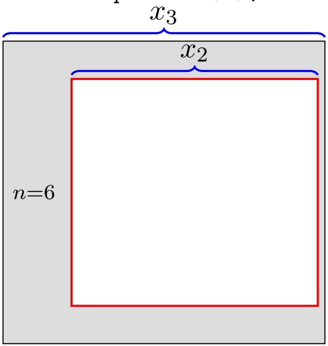
- A snapshot is what other books refer to as *Activation Record*.
- In recursion, a snapshot is defined to store necessary information:
  - The values of local variables.
  - The code *after* the recursive call, *i.e.*, `System.out.println(n);`.
  - Other complex information which we will not include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

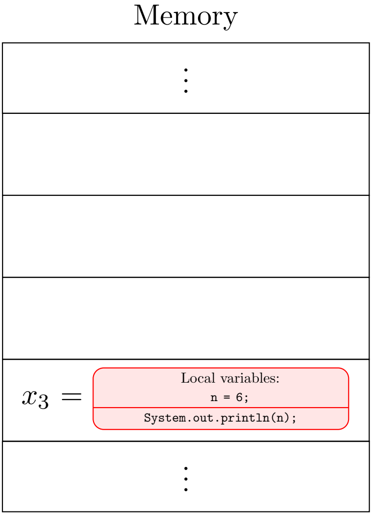
| |
|---|
| Values of local variables |
| Code to be executed later |

- Every time we call the recursive case, Java will automatically a take 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.

**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);
    }
}
```

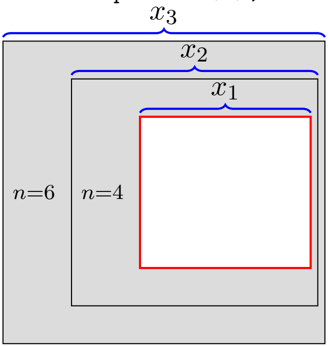


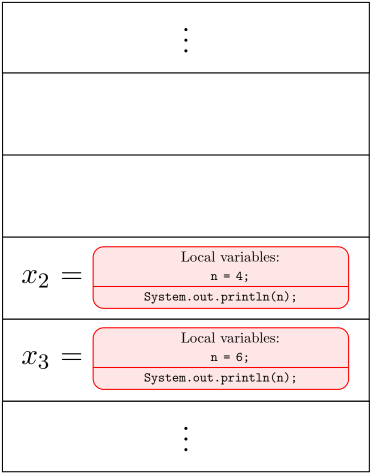
# Inspecting 2

**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);
    }
}
```



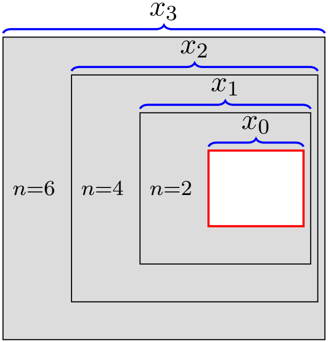

Memory

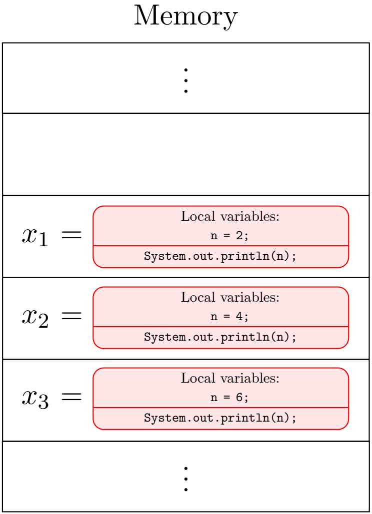


# Inspecting 3

**Goal:** 0, 2, 4, 6

```java
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);
    }
}
```

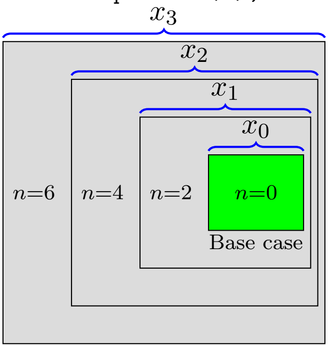


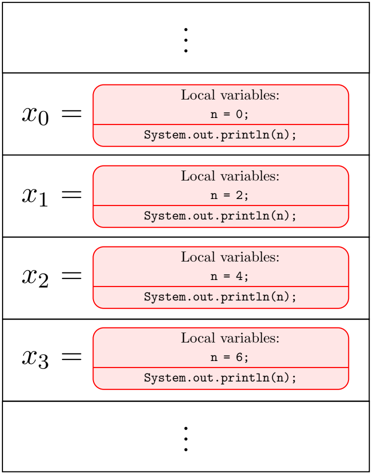
# Evaluating 1

**Goal:** 0, 2, 4, 6 **Output:** $\overset{x_0}{0}$

```java
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

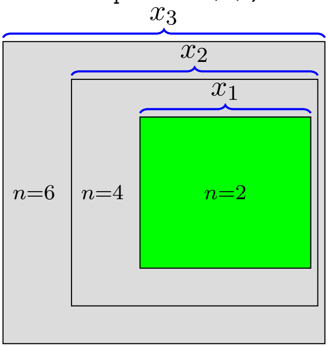


# Evaluating 2

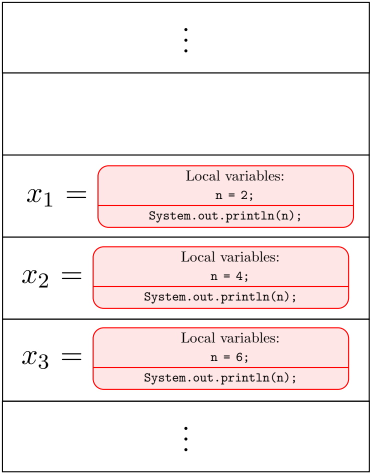
**Goal:** 0, 2, 4, 6

**Output:** $\overset{x_0}{0}, \overset{x_1}{2}$

```java
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);
    }
}
```



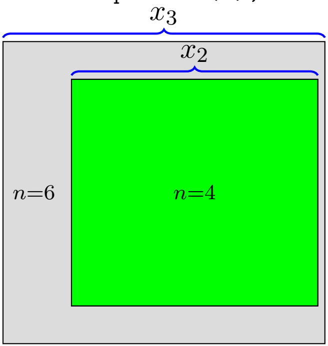

Memory

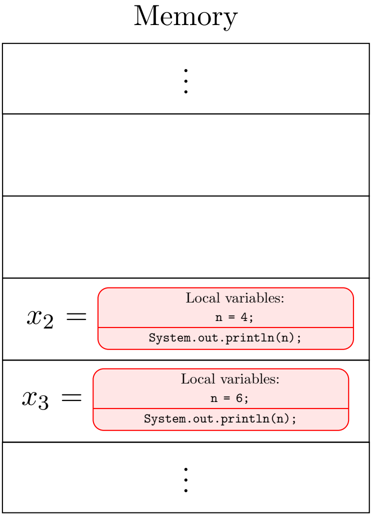


# Evaluating 3

**Goal:** 0, 2, 4, 6 **Output:** $\overset{x_0}{0}, \overset{x_1}{2}, \overset{x_2}{4}$

```java
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);
    }
}
```

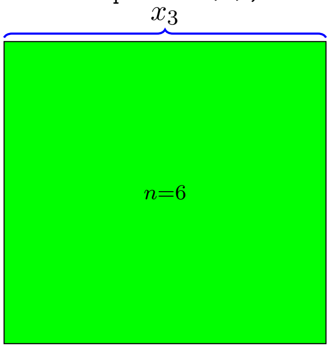


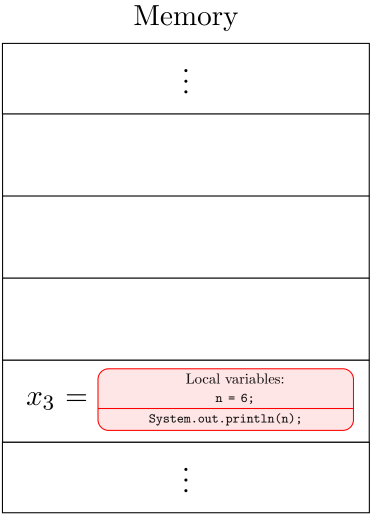



# Evaluating 4

**Goal:** 0, 2, 4, 6 **Output:** $\overset{x_0}{0}, \overset{x_1}{2}, \overset{x_2}{4}, \overset{x_3}{6}$

```java
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);
    }
}
```



# Output

Calling `printEven(6);` will recursively print the following in order: $0, 2, 4, 6$.

# The Golden Rule of Recursion

- The three conditions must be satisfied:

# The Golden Rule of Recursion

- The three conditions must be satisfied:
  - **Condition 1:** A recursive method must have a base case and recursive case.

# 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`.

# 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.

# 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.

# 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.

# 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!