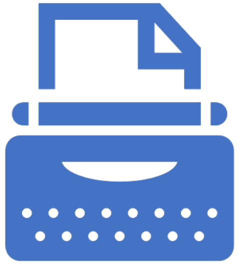
CMPS 200: Introduction to Programming Using JAVA

LECTURE 7 – Decomposition, Abstraction, Methods

Maurice J. KHABBAZ, Ph.D.

Last Time



Output Formatting:

`NumberFormat` Class

`DecimalFormat` Class

`printf()` method

Wrapper Classes:

`Character` Class

`Integer` Class

`Double` Class

Today

Structuring Programs:
Hiding Details

Methods

Specifications

Keywords:
`return` v.s. `print`, `println`

Scope

How To Write Code?

So far:

- Covered language mechanisms.
- Know how to write different files for each computation.
- Each file is some piece of code.
- Each code is a sequence of instructions.

Problems with this approach:

- Easy for small-scale problems.
- Messy for larger problems.
- Hard to keep track of details.
- How to know that the right info supplied to the right part of the code.

Good Programming

Code	Measure	Introduce	Go
More code not necessarily a good thing.	Measure good programmers by the amount of functionality.	Introduce functions	Go for mechanisms that achieve decomposition and abstraction.

Example: Projector

- It’s a black box.
- I don’t know how it works.
- I know, however, its interface: input and output.
- Connect electronic device to it that can communicate with that input.
- It somehow converts an image from input to the wall and magnifies it.
- **Astraction Idea**: Do not need to know how projector works to use it.

Example: Projector

Projecting a large Olympics image: Decomposed into separate tasks for separate projectors.

Each projector takes input and produces separate output.

All projectors work together to produce larger image.

Decomposition Idea: Different devices work together to achieve an end goal.

Apply These Concepts To Programming

Create Structure With Decomposition

Recall

Projector Example: Separate devices.

Programming

Divide code into methods/modules:

- Are **self-contained**.
- Used to **break up** code.
- Are **reusable**.
- Keep code **organized**.
- Keep code **coherent**.

This Lecture

Achieve decomposition with **methods**.

Later

Achieve decomposition with **classes**.

Supress Details With Abstraction

Projector Example:

How-to-use instructions are sufficient.	No need to know how to build one.

Programming: think of a piece of code as a black box

Cannot see details.	Do not need to see details.	Do not want to see details.	Hide tedious coding details.

Achieve abstraction with **method specifications** (*a.k.a.* **commented code**)

Method

- A group of statements that is given a name.
- When **invoked** (*i.e.* called) a method specifies the code to be executed:
  - Statements pertaining to an invoked method are executed sequentially.
  - Once done, control returns to the location of the call and execution continues.

Methods

Write reusable pieces/chunks of code: **methods / functions**

Not executed in program until **called** or **invoked**

Method characteristics

Header

- **Return Type**
- **Name**
- **Parameters** (0 or more)
- **Descriptions / Comments** (optional)
- **Body**

Method: Header Syntax

- A **method declaration** begins with a method header:

Input parameters list

```
<return_type> <name> (<type1> <v1>, <type2> <v2>, ...)
```

- The input parameters list specifies the **type** and **name** of each parameter:
  - These are delimited by two parentheses `( )`.
  - If the method has no input parameters the `( )` are left empty.
- The name of a parameter in the method declaration is called **formal parameter**.
- **Example:**

return type name input parameter

```
boolean isEven(int i)
```

Remarks:

1. If a method accepts a parameter, it is **illegal** to call it without passing a value for that parameter.
2. The **value passed** for a method's parameter must be of the **correct type**.

Method: Body

- A method's header is followed by that met
  - Enclosed between two curly brac
- **Syntax:**

<retu   <v1>, <t

2>;

Execute

...

nother

Keyword

re

valuated and its val

point (<val> type

- **Exa**

boolean

ide isEven() method.");

0);

!! WAIT … THIS IS NOT FINAL YET !!

MORE STUFF TO KNOW

MORE TO DO TO BE ABLE TO INVOKE A METHOD

More Stuff To Know: Return Type

- In method header:
  - <return_type> → primitive type or class name.

- When method:
  - Returns a value:
    - It must have a `return` statement.
  - Does not return a value:
    - Return type is a reserved word **`void`**.
    - Must not have a `return` statement.

Example:

```
void isEven(int i) {
    if (i % 2 == 0)
        System.out.print(i + "is even.");
    else
        System.out.print(i + "is odd.");
}
```

- Upon completion of method execution:
  - Control is returned to calling point.

`return` V.S. `print`

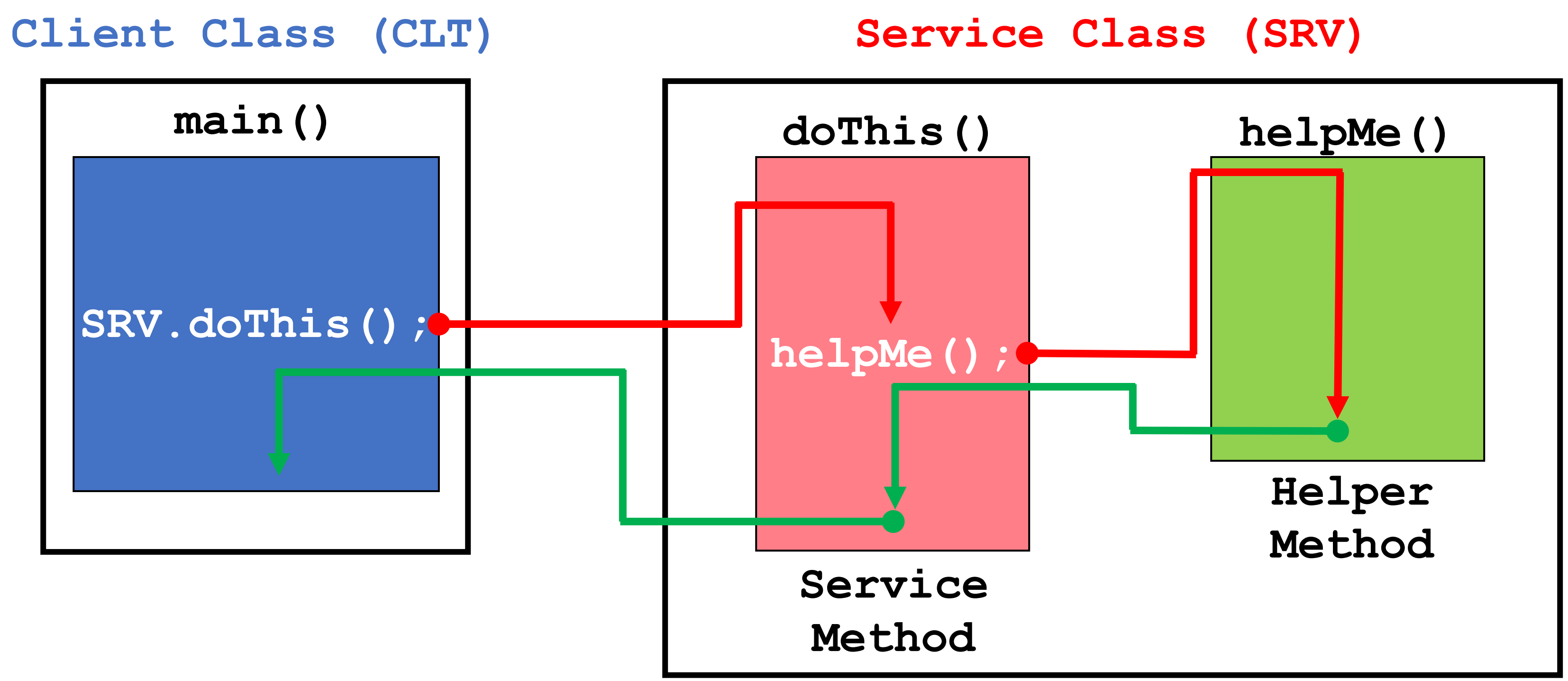
- Only **meaningful inside methods**.
- Only **one return executed per method**.
- **Code after `return` skipped**.
- Associated with value:
  - **Value assigned to method caller**.

- Used mainly **outside methods**:
  - Can be used inside as well.
- Many `prints` may be executed.
  - Inside and outside methods.
- **Code after `print` executed**.
- **May have a value** associated to it:
  - Value **outputted to console**.

More Stuff To Know: Method Access Visibility

- Access visibility is:
  - Applied to a method depends on the purpose of the method.
  - Part of the method's header (before `<return_type>`)
- **Syntax:** **`<visibility>`** `<return_type> <name> (<parameters>)`
- Typically, methods are declared with **`public`** access visibility:
  - They are called and accessed from anywhere within or outside of a class.
  - Methods with `public` visibility are known as **service methods**.
- Opposite to `public` is the **`private`** visibility modifier:
  - Methods with `private` visibility are only invokable from within their enclosing class.
  - Such methods are referred to as **helper methods**.
- More on this topic in upcoming lectures…

More Stuff To Know: Method Invocations



- method must be `static`.
- `static` modifier after visibility.
- **Remark:** non-static method cannot be referenced (invoked) from a static context.

- If the calling point and invoked/called method are:
  - In different classes → Invoke through the name of the method's class.
    → **or** through the name of an object of that class.
  - Within the same class → Only the method's name is required.

More Stuff To Know: Local Data

- **Local variables** can be declared inside a method.
- Formal parameters of a method are also local variables for that method.
- When a method completes execution → **all local variables deleted**.

What To Do Next?

- **Need a method?**
  - Determine the method's visibility (*i.e.* `public`, `private`, …)
  - Determine if you need the method to be `static` or not.
  - Choose the method's return type (`void` if no return is needed).
  - Give the method an appropriate name.
  - List the method's formal parameters and their types.
  - Lay out the body the method (`return` statement if method returns a value).
  - Put all the above in the appropriate class.

- **Ready to go:** Invoke the method throughout the program.

Methods: Example 1

Write a JAVA program that takes from the user an integer `n` and, then, calls a function called `isEven()` that takes `n` as a parameter and returns `true` if `n` is even and `false` otherwise. Based on this returned value, the program must display a message saying whether `n` is even or odd.

formal parameter

```
import java.util.Scanner;

public class EvenOdd {
   public static boolean isEven(int i) {
      return i % 2 == 0;
   }
   public static void main(String[] args) {
      Scanner keyboard = new Scanner(System.in);
      int n;
      System.out.print("Enter an integer: "); n = keyboard.nextInt();
      if (isEven(n)) System.out.println(n + "is even.");
      else System.out.println(n + "is odd.");
   }
}
```

actual parameter

Methods: Example 2

Write a JAVA program that draws the figure on the right

```
* * * *
 * * *
  * *
   *
   *
  * *
 * * *
* * * *
* * * *
 * * *
  * *
   *
   *
  * *
 * * *
* * * *
```

Solution:

Primitive Method to print an upward arrow

```
public static void upArrow() {
  System.out.println(“   *   ”);
  System.out.println(“  * *  ”);
  System.out.println(“ * * * ”);
  System.out.println(“* * * *”);
}
```

Primitive Method to print an downward arrow

```
public static void downArrow() {
  System.out.println(“* * * *”);
  System.out.println(“ * * * ”);
  System.out.println(“  * *  ”);
  System.out.println(“   *   ”);
}
```

Main Method

```
public static void main(String[] args){
  for (int i = 1; i <= 2; i++){
       downArrow();
       upArrow();
  }
}
```

Methods: Example 3

Write a JAVA program that takes from the user an integer `n` and, then, calls a function called `factorial()` that takes `n` as a parameter and returns `n!`. The program must, then, display this result to the screen.

```
import java.util.Scanner;

public class Factorial {
   public static int factorial(int n) {
      int fact = 1;
      for (int i = 2; i <= n; i++) fact *= i;
      return fact;
   }
   public static void main(String[] args) {
      Scanner keyboard = new Scanner(System.in);
      int n;
      System.out.print("Enter n: ");
      n = keyboard.nextInt();
      System.out.println("n! = " + factorial(n));
   }
}
```

Remarks On Value Semantics:

- When primitive variables are passed as actual parameters, their values are copied.
- Modifying the parameter values will not affect the original variable passed in.

Variable Scope

- Upon a method call:
  - **Formal parameter** gets bound to the value of **actual parameter**.
- New **scope**/**frame**/**environment** created when a function executes.
- The **scope** is a **mapping of names to objects**.
- **Example:**

Formal Parameter

```
public static int f(int x){
   x = x + 1;
   System.out.println("in f(x): x = " + x);
   return x;
}
```

Method Declaration

Actual Parameter

```
int z = f(3)
```

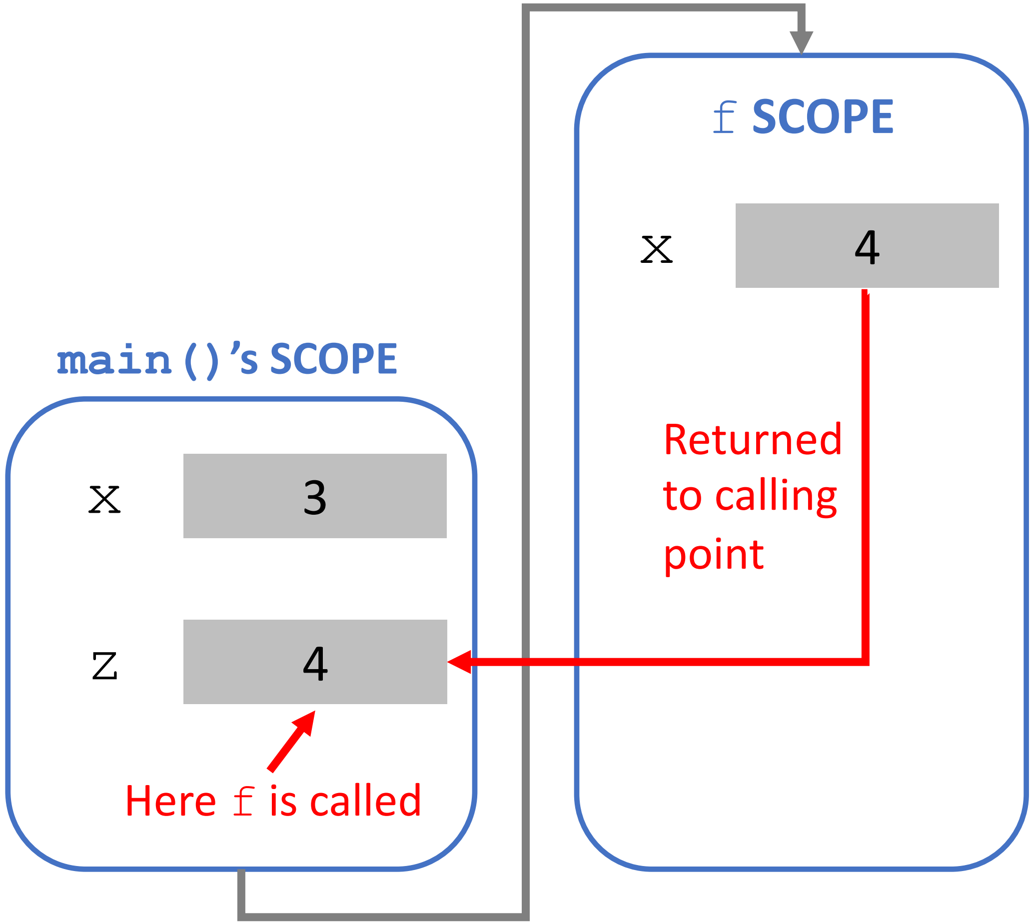
Main method calls `f()` and assigns its returned value to a variable `z`

Variable Scope

class defines a Global Scope
that includes `f()` and `main()`'s declarations

```
public class VarScope{
  public static int f(int x){
    x = x + 1;
    S.o.pln("in f(x): x = ", x);
    return x
  }
  public static void main(String[] args){
    int x = 3;
    int z = f(x);
  }
}
```

`main()`'s includes the declarations of `x` and `z`



Example 1: Variable Scope

A Method With Multiple `return` Statements

Write a JAVA program that takes from the user an integer `n` and, then, calls a function called `reciprocal()` that takes `n` as a parameter and returns its reciprocal to be printed in a fractional format (*i.e.*, `1/n`)

Solution:

```
import java.util.Scanner;
public class Reciprocal {
   public static String reciprocal(int n){
      if (n == 0) return "ERROR: Division by 0.";
      else if (n == 1) return "1";
      else return "Reciprocal of " + n + " is: 1/" + n;
   }
   public static void main(String[] args){
     Scanner k = new Scanner(System.in);
     System.out.print("Enter n: "); n = k.nextInt();
     System.out.println(reciprocal(n));
   }
}
```

Example 2: Variable Scope
Value Semantic Example

- What will be the output of the following JAVA code?

```
public class ValueSemanticExample{
    public static void main(String[] args) {
        int x = 17;
        doubleNumber(x);
        System.out.println(“x = ” + x);

        int number = 42;
        doubleNumber(number);
        System.out.println(“number = ” + number);
    }

    public static void doubleNumber(int number) {
        System.out.println(“Initial value = ” + number);
        number += 2;
        System.out.println(“Final value = ” + number);
    }
}
```

CONSOLE OUTPUT

```
Initial value = 17
Final value = 19
x = 17
Initial value = 42
Final value = 44
number = 42
```

Functions As Arguments

```
public static void f_a(){
    System.out.println("inside f_a");
}
public static int f_b(int y){
    System.out.println("inside f_b");
    return y;
}
public static double f_c(double z){
    System.out.println("inside f_c");
    return z;
}
```

method calls from main() method

```
f_a();
System.out.println(5+f_b(2));
System.out.println(f_c(f_b(6)));
```

Call f_a() takes no parameters

Call f_b() takes one parameter 2

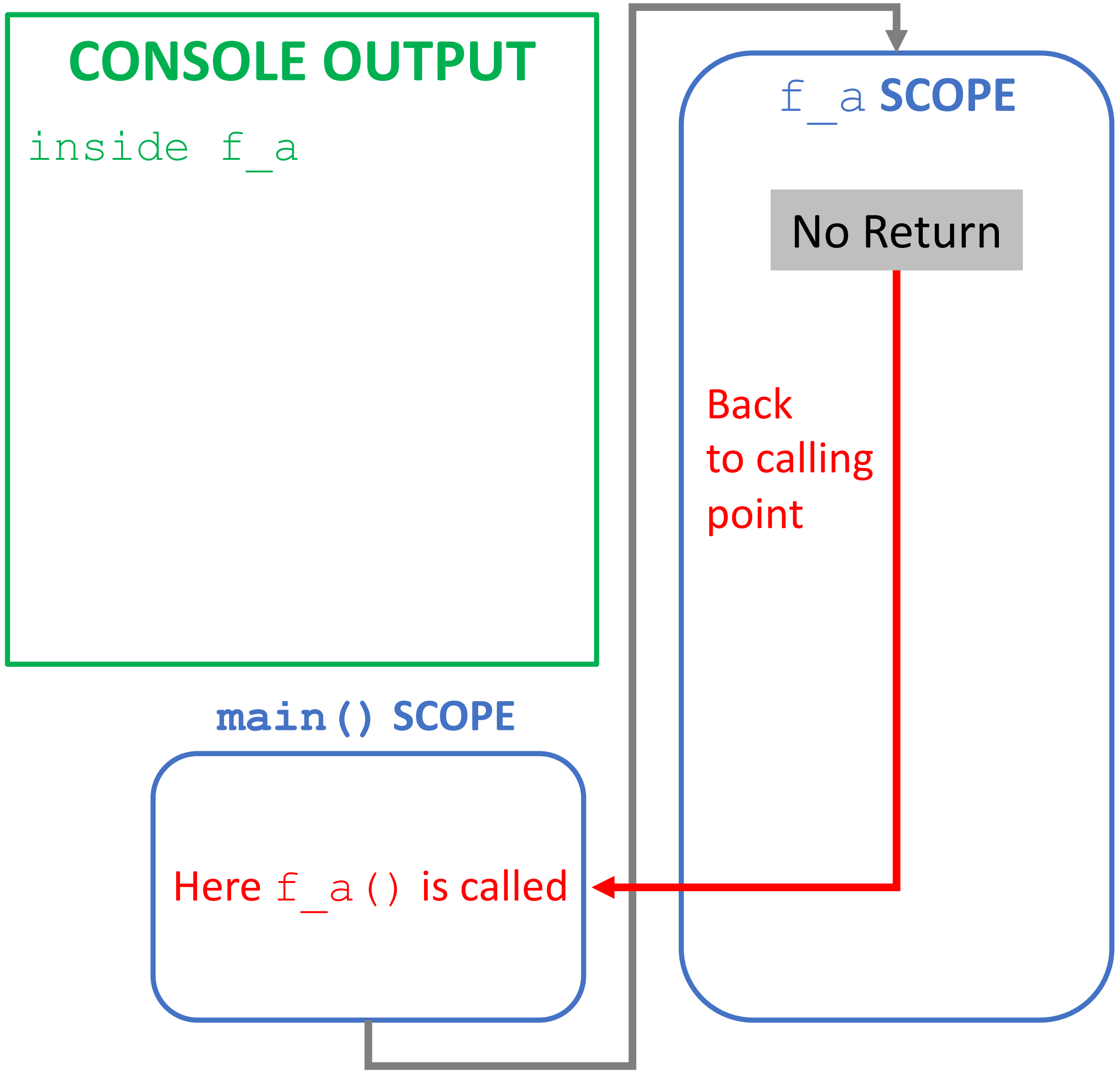
Call f_c() takes one parameter, which is another function

Functions As Arguments

```
public static void f_a(){
    System.out.println("inside f_a");
}
public static int f_b(int y){
    System.out.println("inside f_b");
    return y;
}
public static double f_c(double z){
    System.out.println("inside f_c");
    return z;
}
```

method calls from main() method

```
f_a();
System.out.println(5+f_b(2));
System.out.println(f_c(f_b(6)));
```



Functions As Arguments

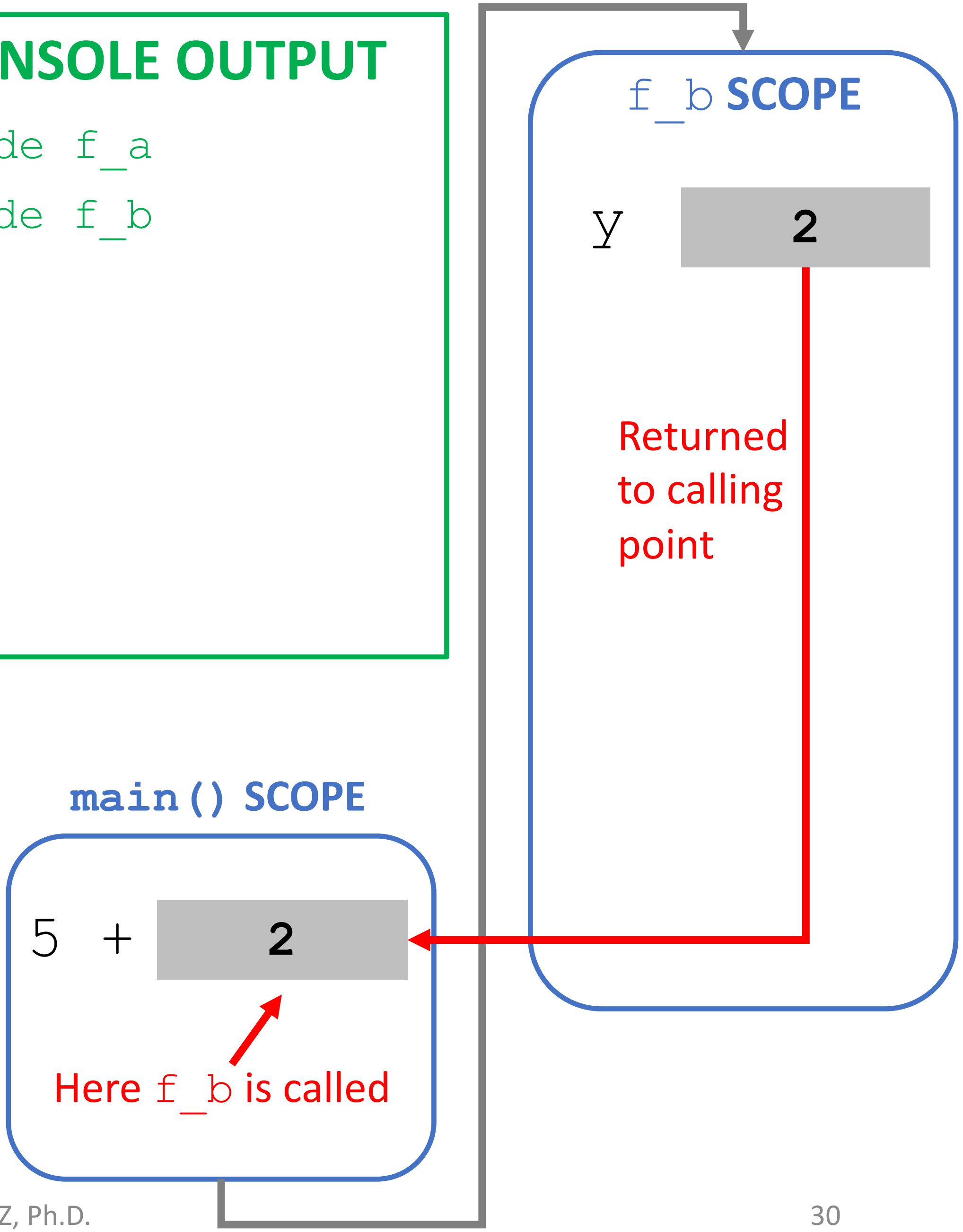
```
public static void f_a(){
    System.out.println("inside f_a");
}
public static int f_b(int y){
    System.out.println("inside f_b");
    return y;
}
public static double f_c(double z){
    System.out.println("inside f_c");
    return z;
}
```

method calls from `main()` method

```
f_a();
System.out.println(5+f_b(2));
System.out.println(f_c(f_b(6)));
```

CONSOLE OUTPUT

```
inside f_a
inside f_b
7
```



Functions As Arguments

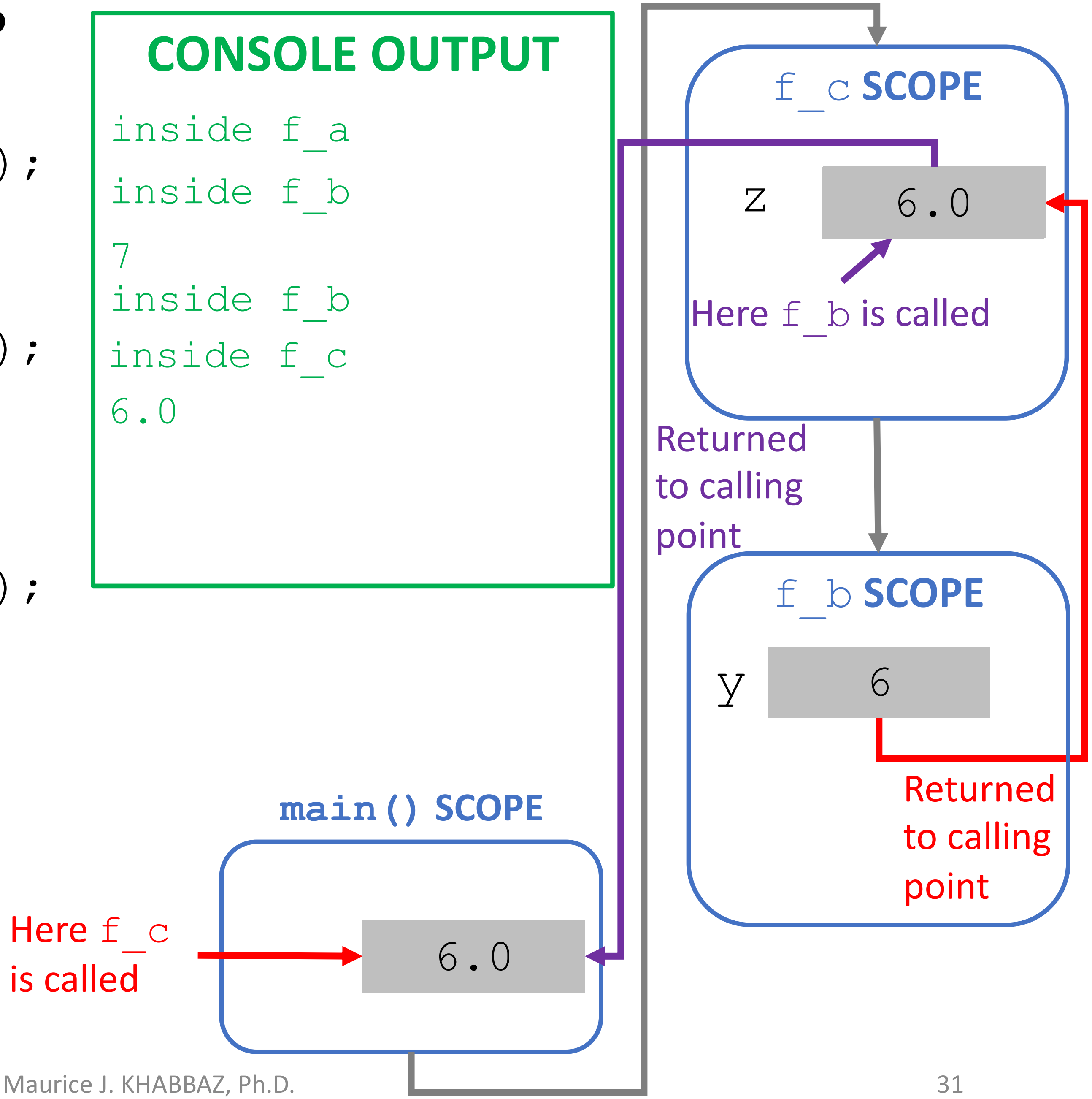
```
public static void f_a(){
    System.out.println("inside f_a");
}
public static int f_b(int y){
    System.out.println("inside f_b");
    return y;
}
public static double f_c(double z){
    System.out.println("inside f_c");
    return z;
}
```

method calls from `main()` method

```
f_a();
System.out.println(5+f_b(2));
System.out.println(f_c(f_b(6)));
```

CONSOLE OUTPUT

```
inside f_a
inside f_b
7
inside f_b
inside f_c
6.0
```



Another Scope Example With More Details

Formal Parameter

```
public static int g(int x){
    x++;
    h();
    return x;
}
```

Function g(x) has a scope that includes the declaration of one variable x and calls h()

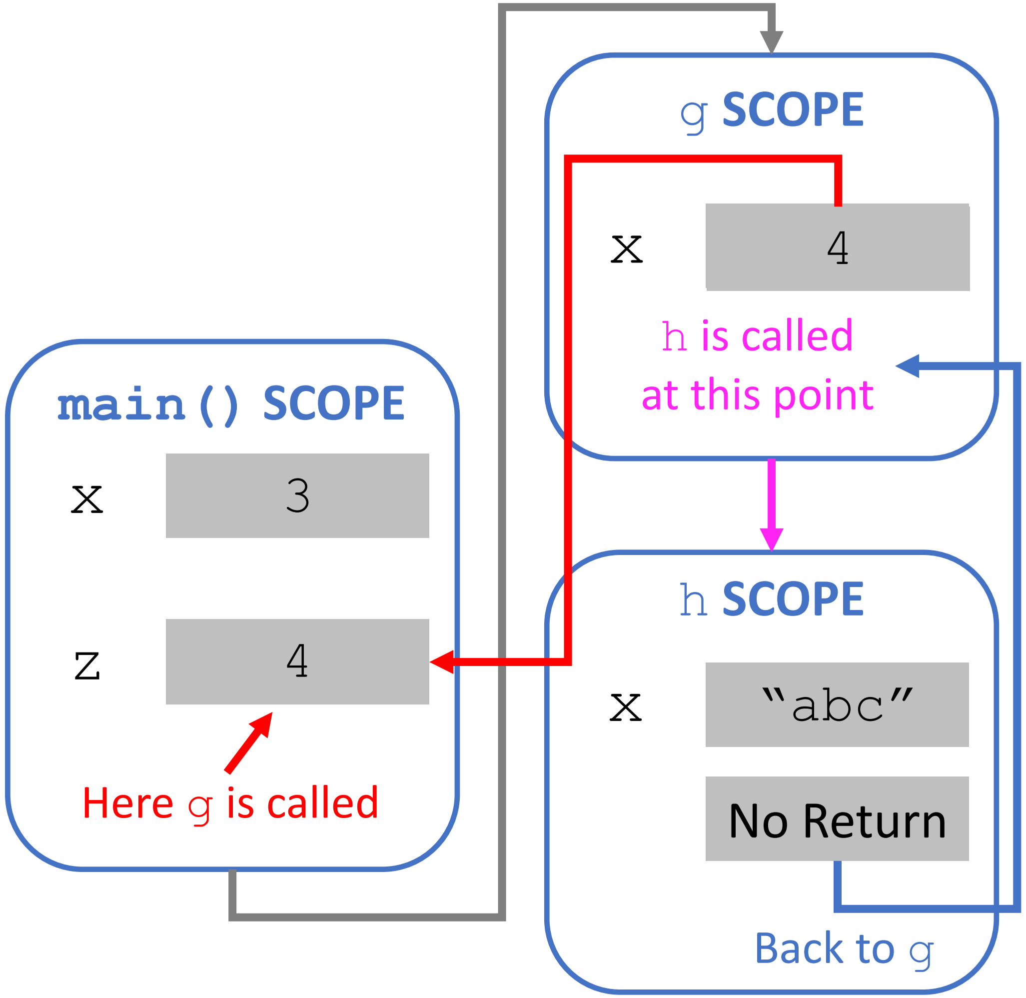
```
public static void h(){
    String x = "abc";
}
```

Function h() scope defines one variable x

```
int x = 3;
int z = g(x);
```

Actual Parameter

main() method defines two variables x and z and calls g()



ASCII Art: Recall The Box Drawing Pseudocode

- Draw a (`width` × `height`) box using two symbols: border and inner.

Top Line: print `width` border symbol.

Body: for the remaining `height - 2` lines

```
for (each of the height - 2 lines) {
    print a border symbol.
    print width - 2 inner symbol.
    print a border symbol.
}
```

Bottom Line: print width border symbol.

Examples

```
************
*          *
*          *
*          *
*          *
*          *
************
```

```
0000000
0111110
0111110
0111110
0000000
```

```
++++++++++++
+**********+
+**********+
+**********+
+**********+
+**********+
++++++++++++
```

Practice Exercise: ASCII Art Using Methods

This problem has the objective of implementing a ASCII Art Box Drawing Algorithm (AABDA) through the implementation of three fundamental methods, namely:

- **Method 1:** called **`repeat()`** that takes two input parameters, namely: *i*) a character `c` and *ii*) an integer `i`. The method will then print `c` to the screen `i` times without returning any result to its calling point.
- **Method 2:** called **`line()`** takes two input parameters, namely: *i*) a character `c` and *ii*) an integer `i`. This method will use the above-implemented `repeat()` method to print out `c` to the screen `i` times followed by a line break (*i.e.* new empty line) without returning any result to its calling point.
- **Method 3:** called **`box()`** that takes four input parameters, namely: *i*) a character `br` representing a box's border symbol, *ii*) a character `in` representing the inner symbol of that box, *iii*) an integer `h` representing the box's height and *iv*) an integer `w` representing the box's width. When invoked from the program's `main()` method, `box()` will make use of the above-implemented `line()` and `repeat()` methods to print out the entire box as specified without returning any result to its calling point.

Finally, the **`main()`** method will request from the user to interactively enter the border and inner characters as well as the width and height of the box and then call the method `box()` passing to it the appropriate actual parameters in the order specified in the above description of this latter method.

Conclusion: Decomposition and Abstraction

- Powerful together.

- Code:
  - Can be **used and reused** many times.
  - Has to be **debugged only once**!