UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

Software Development II

Lecture 5:Methods & Recursion

Reading : Java for everyone Chapter:5

From Last Week

- Arrays
  - Declaration
  - Indices
  - Access
  - Length
  - Enhanced loop
  - Search
  - Copy
  - 2D arrays
  - Search & Sort Algorithms

Today’s outline

- What is a method
- Input and outputs
- Syntax
- Declaration and use
- Variables and constants scope
- Overloading
- Recursion

Methods

- The purpose of using methods is to break up a program into smaller, reusable pieces of software.
- **Methods are a collection of statements (code) that are grouped together to perform a specific task. They are equivalent to Python functions.**
- While some methods are predefined - that is written and included as part of the Java environment, most methods will be written by the programmer.

Example

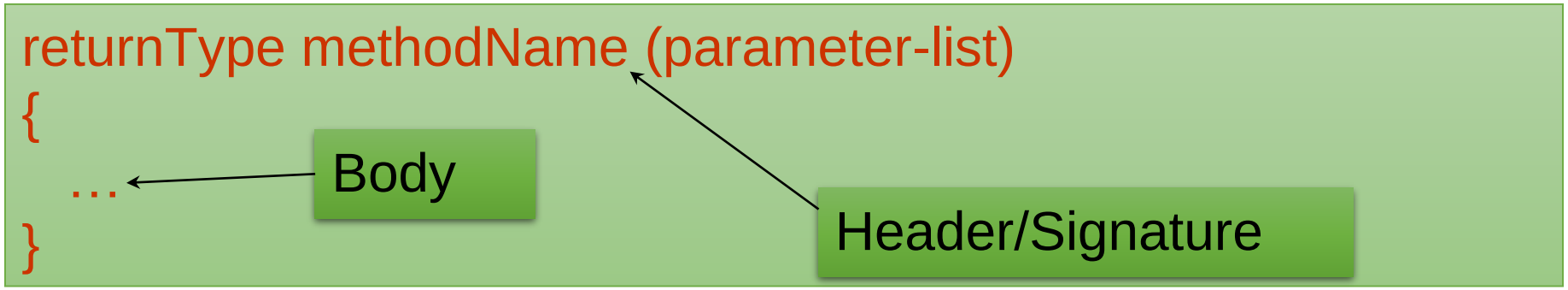
```
for(int i=0;i<10;i++)
{
   System.out.println(i);
}

// repeat for the 2nd time
for(int j=20;j<30;j++)
{
   System.out.println(j);
}

// repeat once again
for(int k=40;k<50;k++)
{
   System.out.println(k);
}
```

```
public class Main {

    // method to avoid repetition
    public static void printNum(int start, int end)
    {
        for(int i=start;i<end;i++)
        {
            System.out.println(i);
        }
    }

    public static void main(String[] args)
    {
        printNum(0,10);
        printNum(20,30);
        printNum(40,50);
    }

}
```

Method Declaration

- We have so far used methods such as main() and will now look at how we can create methods of our own.
- To define a method:
  - give it a name
  - specify the method's return type or choose **void**
  - specify the types of parameters and give them names or keep the parenthesis empty.
  - write the method body
  - test the method

Method Declaration



- A method is always defined inside a class.
- A method returns a value of the specified type unless it is declared void; the return type can be any primitive data type or a class type.
- A method's parameters can be of any primitive data types or class types.

Methods: Access Level

```
public class Methods {
    public static void main(String[] args){
        print_hello();
        print_hello();
        print_hello();
    }

    private static void print_hello(){
        System.out.println("Hello world!");
    }

    public static void print_bye(){
        System.out.println("Bye!");
    }
}
```

Main method (the one that will execute first when we run the program

Output:

```
Hello world!
Hello world!
Hello world!
```

Method name

Only accessible from the class `Methods`

Accessible from outside the class `Methods`
(We will see later when this is useful)

Method Declaration - Example

Output parameter type

Method name

Input parameter data type and name

Input parameter data type and name

```
private static double multiply_numbers(double x, double y)
{
  double mul = x * y;
  return mul;
}
```

Method body (implementation)

Output parameter (must match data type)

Invoking a Method (Method Call)

- We invoke (or 'call') a method by stating:
  - Its name (identifier)
  - The values to be taken by its parameters

- Example:

Method name

```
double result= multiply_numbers(2.0,5.0)
```

Same type as method output

Input parameter1

Input parameter2

Also Valid

```
double result;
double number_1 = 2.0;
double number_2 = 2.0;
result = multiply_numbers(number_1, number_2));
```

Passing Parameters

- So the values that are supplied to the method as parameters can be:
  - *constant* values, such as **`12.3`**
  - *expressions*, such as **`7.5+5.6`**
  - *variables*, such as in **`sideLength=12.3`**
    - not a named parameter. Only initialize when passing

- Where an expression is used, it is evaluated first and then the result is copied to the method.
- Where a variable is used, its value is copied to the method and the variable remains unchanged -> pass by value.

Formal & Actual Parameters

- The formal parameters are:
  - The identifiers used when writing the method signature.
  - Their use is local to the method
- The actual parameters are:
  - the parameters in the method call (those being passed to the method).
- Actual parameters must match the formal parameters in number and type.

Exercise

- Write a method called **calcTotal** to add two numbers that are given as parameters and return the total.
- Invoke **calcTotal()** inside the main method.

Returning Information

- The rules of Java only allow us to pass information into a method through the parameters.
- To get results out of a method, we turn it into an expression and return a value of a particular type.
- Storing a returned value after the call

```
double result = cubeVolume(12.3);
```

- The methods were of type void which means that they do not return any value.

Method Comments

- Whenever you write a method, you should comment its behavior
- Method comments explain:
  - The purpose of the method
  - The meaning of the parameter variables
  - The return value
  - Any special requirements

```
/**
  Computes the volume of a cube.
  @param sideLength the side length of the cube
  @return the volume
*/
public static double cubeVolume(double sideLength)
```

Methods: Step by step

1. Define input parameters
2. Define output parameters
3. Implement method
4. Call method

```
private static void check_email(String email){
  if (email.contains("@") && email.contains(".")){
    System.out.println("Email is correct.");
  }
  else System.out.println("Email is not correct.");
}

public class Methods {
   public static void main(String[] args) {
      String email = "bla bla bla";
      check_email(email);

      email = "bla@bla";
      check_email(email);

      email = "bla@bla.com";
      check_email(email);
   }
}
```

Methods: call other methods

```
public class Methods {
   public static void main(String[] args) {
      String email = "bla@bla.com";
      check_email(email);
   }
}

private static void check_email(String email){
   if (email.contains("@")){
     check_dot(email);
   }
   else System.out.println("Email is not correct.");
}

private static void check_dot(String email){
   if (email.contains(".")){
     System.out.println("Email is correct.");
   }
   else System.out.println("Email is not correct.");
}
```

Recap : Method Types

With no input and no output:

```
private static void print_error(){
  System.out.println("Error. Try again");
}
```

With input and output:

```
private static int add_numbers(int x, int y)
{
  int sum = x + y;
  return sum;
}
```

Recap : Method Types

With input but with no output

```
private static void check_larger_than_10(int number){
  if(number>10){
    System.out.println("Larger than 10.");
  }
  else System.out.println("Not larger than 10.");
}
```

With no input but with output

```
private static int random_number_10(){
  int number = (int) (Math.random()*10+1.0);
  return number;
}
```

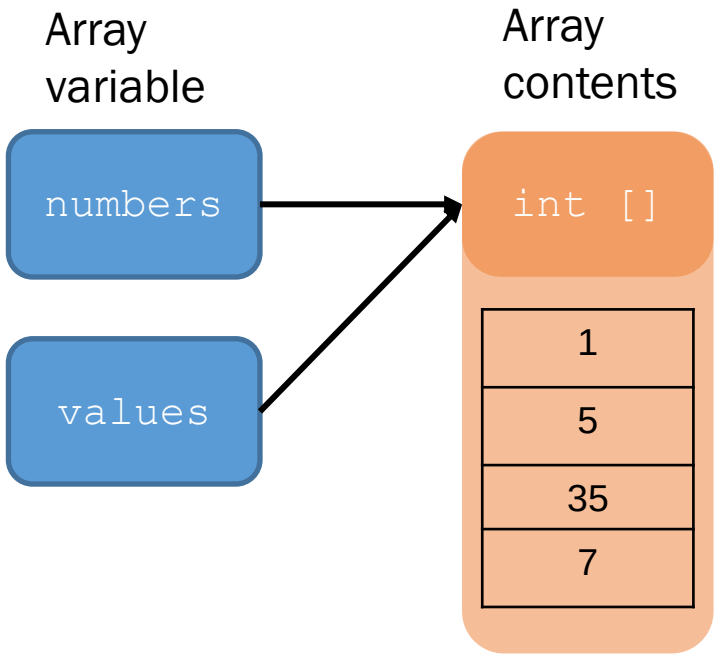
Re-cap: Arrays: copy

REMINDER: An array variable contains a **reference** to the array content. The **reference** is the location of the array contents (in memory)

```
int[] numbers = new int [];
numbers = {1, 5, 35, 7};
```

```
int[] values = numbers;
```

Will copy the reference only, so both will point to the same memory.



What happens when we modify *values*?

Methods: arrays as inputs

- With arrays we have to be careful.

```
int[] array = {2, 4, 6, 8};
add_1(array);
```

We don't modify the value of array here, but we call the function `add_1`.

```
for (int value : array){
  System.out.println(value);
}
```

The values of `array` have changed

Output = 3, 5,7,9.

```
private static void add_1(int[] array)
{
  for(int i = 0; i < array.length; i++){
    array[i]++;
  }
}
```

Methods: when should we use them?

- **Reusability:**
  - To avoid code repetition
  - If there is a bug (error), you only have to correct it once.
  - The more code you write, the higher the chance of including bugs.
- **Easy modification:**
  - If you need to modify it, you only have to do it once.
- **Readability:**
  - To improve understanding.

```
public class Methods {
    public static void main(String[] args) {
        String email = "bla bla bla";
        if (email.contains("@") && email.contains(".")){
            System.out.println("Email is correct.");
        }
        else System.out.println("Email is not correct.");

        email = "bla@bla";
        if (email.contains("@") && email.contains(".")){
            System.out.println("Email is correct.");
        }
        else System.out.println("Email is not correct.");

        email = "bla@bla.com";
        if (email.contains("@") && email.contains(".")){
            System.out.println("Email is correct.");
        }
        else System.out.println("Email is not correct.");
    }
}
```

Same code

Variable Scope

- Variables can be declared
  - Inside a method
    - Known as "local variable"
    - Availability inside the method
    - Parameters are local variables
- Inside a block of code {}
  - If variable declared inside {}
- Outside method
  - Sometimes called Global scope
  - Use and change inside any method
- Instance/member variables
  - Declare inside a class

Example of Scopes

- sum is a local variable in main
- square is only visible inside the for loop block
- i is only visible inside the for loop

```
public static void main(String[] args)
{
  int sum = 0;
  for (int i = 1; i <= 10; i++)
  {
    int square = i * i;
    sum = sum + square;
  }
  System.out.println(sum);
}
```

sum

i

square

The scope of a variable is the part of the program in which it is visible.

Local Variables of Methods

- Variables declared inside one method are not visible to other methods
- sideLength is local to main
- Using it outside main will cause a compiler error

```
public static void main(String[] args)
{
  double sideLength = 10;
  int result = cubeVolume();
  System.out.println(result);
}

public static double cubeVolume()
{
  return sideLength * sideLength * sideLength; // ERROR
}
```

Reusing names for local variables

- Variables declared inside one method are not visible to other methods
  - result is local to square and result is local to main
  - They are two different variables and do not overlap

```
public static int square(int n)
{
  int result = n * n;
  return result;
}

public static void main(String[] args)
{
  int result = square(3) + square(4);
  System.out.println(result);
}
```

result

result

Re-using names for block variables

- Variables declared inside one block are not visible to other methods
  - i is inside the first for block and i is inside the second
  - They are two different variables and do not overlap

```
public static void main(String[] args)
{
  int sum = 0;
  for (int i = 1; i <= 10; i++)
  {
    sum = sum + i;
  }
  for (int i = 1; i <= 10; i++)
  {
    sum = sum + i * i;
  }
  System.out.println(sum);
}
```

i

i

Overlapping Scope

- Variables (including parameter variables) must have unique names within their scope
  - n has local scope and n is in a block inside that scope
  - The compiler will complain when the block scope n is declared

```
public static int sumOfSquares(int n)
{
  int sum = 0;
  for (int i = 1; i <= n; i++)
  {
    int n = i * i; // ERROR
    sum = sum + n;
  }
  return sum;
}
```

Local n

block scope n

Global and Local Overlapping

- Global and Local (method) variables can overlap
  - The local same will be used when it is in scope
  - No access to global same when local same is in scope

```
public class Scoper
{
  public static int same;   // 'global'
  public static void main(String[] args)
  {
    int same = 0;     // local
    for (int i = 1; i <= 10; i++)
    {
      int square = i * i;
      same = same + square;
    }
    System.out.println(same);
  }
}
```

same

same

Variables in different scopes with the same name will compile, but it is not a good idea

Constants scope

```
public class myClass {

    static final float VAR_1 = 1;

    public static void main(String[] args)
    {
      final float VAR_2 = 2;
      System.out.println(VAR_1);
      System.out.println(VAR_2);
    }

    private static void myFunction()
    {
      final float VAR_3 = 3;
      System.out.println(VAR_1);
      System.out.println(VAR_3);
    }
}
```

Scope of VAR_2

Scope of VAR_3

Scope of constant VAR_1

Exercises : Predict the output

```
public class myClass {

   public static void main(String[] args) {
     int num = 10;
     int newNum = myFunction(num);
     System.out.println(newNum);
   }

   private static int myFunction(int num){
     int result = num + num;
     return result;
  }
}
```

```
public class myClass {

   public static void main(String[] args) {
     int a = 2;
     int b = 10;
     int newNum = myFunction(a, b);
     System.out.println(newNum);
   }

   private static int myFunction(int b, int a)
   {
     int result = (a * 2) + b ;
     return result;
   }
}
```

Exercises : Predict the correct Method Declaration

```
public class myClass {

   public static void main(String[] args) {
     double a = 2.5;
     int b = 10;
     double result = myFunction(a, b);
     System.out.println(result);
   }

   xxxx xxxx xxxx myFunction(xxx a, xxx b){
     double result = a + b;
     return result;
   }
}
```

Method Overloading

Overloading happens when we have **multiple methods with the same name** in the same class but different parameters (input or output).

Method Overloading

- Method signature: name, number, and type of parameters

```
private static int add(int valueA, int valueB) {}
private static int add(int valueA, int valueB, int valueC) {}
```

- You can use the same name of the method for :
  - Different types of parameters
  - Different number of parameters

- Also called “static polymorphism”

Return Value and Signature

- The return value is not included in the signature

```
private static int add(int valueA, int valueB) {}
 private static int add(int valueA, int valueB, int valueC) {}  //OK

private static float add(int valueA, int valueB, int valueC) {}  //Not OK
```

- Third method is not possible to implement.

Method Overloading : Example 1

```
public class Main {

    public void test(double a,double b){}

    public void test(int a,int b){}

    public void test(int c, double d){}

    public void test(double e, int f){}

    public static void main(String[] args)
    {

        Main m=new Main();
        m.test(12.3,12.2);
        m.test(12,12);
        m.test(12,12.2);
        m.test(12.3,12);

    }

}
```

Method Overloading : Example 2

```
private static void multiply_numbers(double x, double y)
{
  System.out.println(x * y);
}

private static double multiply_numbers(double x, double y)
{
  double mul = x * y;
  return mul;
}

private static double multiply_numbers(double x, double y, double z)
{
  double mul = x * y * z;
  return mul;
}
```

Instead of defining 3 methods that should do the same thing, we overload one.

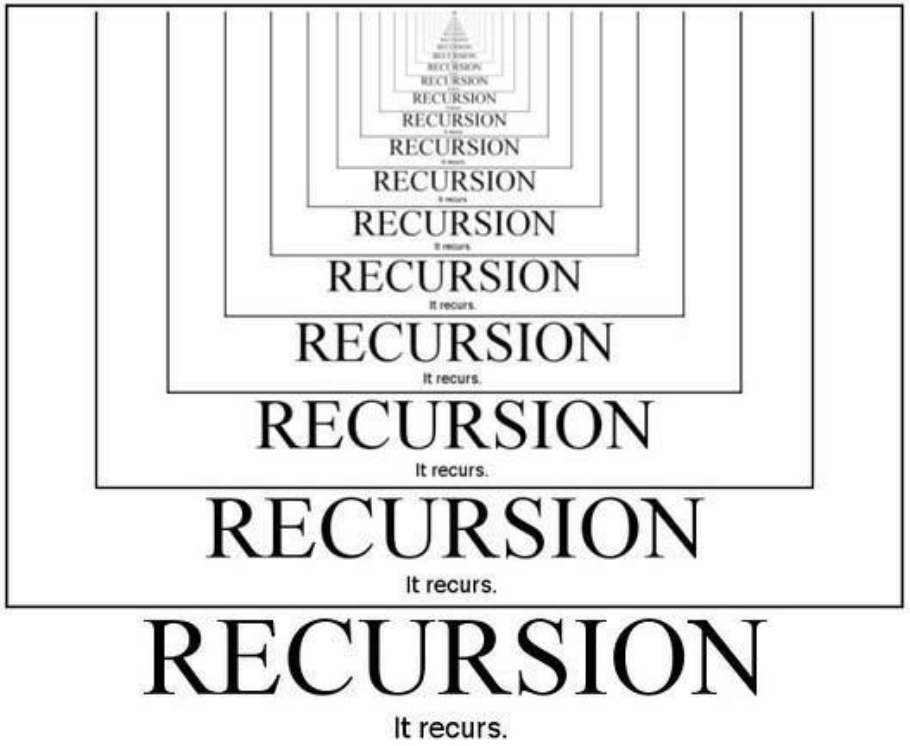
When we call the method, it will execute the one that the data type of the parameters **match better**

Predict the Output

```
public class myClass {
 public static void main(String[] args) {
  double a = 2.5;
  double b = 1.5;
  double result = multiply_numbers(a, b);
 }

 private static int multiply_numbers(int x, int y)
 {
   System.out.println("Method 1");
   return x * y;
 }

 private static double multiply_numbers(double x, double y)
 {
  System.out.println("Method 2");
  return x * y;
 }
}
```

Recursion

To Understand Recursion, you must first understand Recursion.



Recursive Methods

- A recursive method is a method that calls itself
- A recursive computation solves a problem by using the solution of the same problem with simpler inputs
- For a recursion to terminate, there must be special cases for the simplest inputs
- Every recursive call must simplify the task in some way.
- There must be special cases to handle the simplest tasks directly.

Recursion : Example 01

```
public class Main {
   static double myPower(double number, int powerOf)
   {
       if(powerOf ==0){ //special case
           return 1;
       }
       else
       {
           return  number * myPower(number,powerOf -1);
       }
   }

    public static void main(String[] args)
    {
        double result=myPower(2,3);
        System.out.println(result);
    }
}
```

Recursive Calls and Returning

- Assume the developer calls the method `myPower(2,4)`
  - `The call myPower(2,4) calls myPower(2,3)`
    - `The call myPower(2,3) calls myPower(2,2)`
      - `The call myPower(2,2) calls myPower(2,1)`
        - `The call myPower(2,1) calls myPower(2,0)`
          - `myPower(2,0) returns 1`
        - `myPower(2,1) returns 2*1`
      - `myPower(2,2) returns 2*2`
    - `Mypower(2,3) returns 2*4`
  - `myPower(2,4) finally returns 2*8`

Recursive Triangle Example

```
public static void printTriangle(int sideLength)
{
  if (sideLength < 1) { return; }

  printTriangle(sideLength - 1);
  for (int i = 0; i < sideLength; i++)
  {
    System.out.print("[]");
  }
  System.out.println();
}
```

Special Case

Recursive Call

```
[]
[][]
[][][]
[][][][]
```

Print the triangle with side length 3.
Print a line with four [].

- The method will call itself (and not output anything) until `sideLength` becomes < 1
- It will then use the return statement and each of the previous iterations will print their results
  - 1, 2, 3 then 4

Recursive Calls and Returns

- The call `printTriangle(4)` calls `printTriangle(3)`.
  - The call `printTriangle(3)` calls `printTriangle(2)`.
    - The call `printTriangle(2)` calls `printTriangle(1)`.
      - The call `printTriangle(1)` calls `printTriangle(0)`.
        - The call `printTriangle(0)` returns, doing nothing.
      - The call `printTriangle(1)` prints `[]`.
    - The call `printTriangle(2)` prints `[][]`.
  - The call `printTriangle(3)` prints `[][][]`.
- The call `printTriangle(4)` prints `[][][][]`.

Recursion Example 02

- Example: Compute factorials. The factorial of a number is that number multiplied by all of the numbers below it until 1. Factorial (6) = 6 * 5 * 4 * 3 * 2 * 1 = 720.

Factorial (6) = 6 * 5 * 4 * 3 * 2 * 1 = 720.

Factorial (5) = 5 * 4 * 3 * 2 * 1 = 120.

Factorial (4) = 4 * 3 * 2 * 1 = 24.

Factorial (3) = 3 * 2 * 1 = 6.

Factorial (2) = 2 * 1 = 2.

Factorial (1) = 1.

```
private static int factorial(int num) {
  if (num > 1) {
    return num * factorial(num - 1);
  }
  else{
    return 1;
  }
}
```

Independent Study

- Complete Recommended reading *: **Java for Everyone** - Chapter 05*
- Tryout all coding examples provided in lecture slides using code editor and observe output.
- Attempt all exercises provided in lecture slides using code editor and discuss your issues during tutorials.
- Complete Formative test week 05 (Available in Blackboard Week5 folder).
- Attempt all questions in tutorial 04 and submit to BB before deadline

Independent Study

- Read recommended chapters in ebooks.
- Complete feedback questions in BB
- Submit tutorial answers.
- Work on possible sections in Coursework.

Thank You!!