

# CS 162FZ: Introduction to Computer Science II

## Lecture 02

### Methods I

Prof. Chun-Yang Zhang

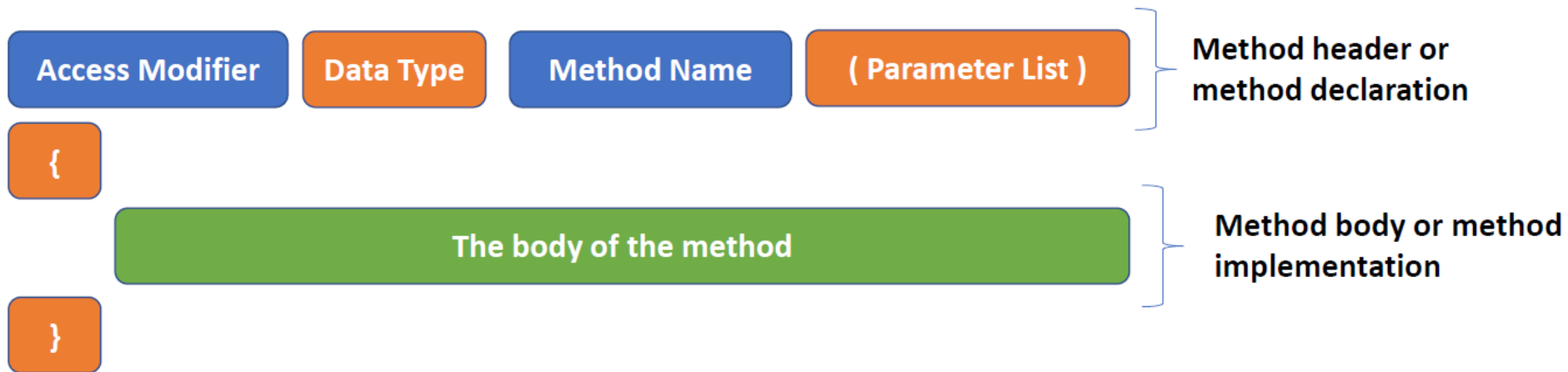
---

# Methods

- A **method** is a program module that contains a series of statements that carry out a task.
  - Methods are often used to define **reusable** code, organize and simplify coding.
  - Methods are sometimes called *Procedures*, *Functions* or *Operations*.
-

# Defining a Method

- A method definition consists of an access modifier (optional), data type of returning value, a method name, a list of parameters and a body.



# Access Modifiers

Access modifiers can be public , protected , private or unspecified (no explicit modifier).

- **private** : The most restricted access modifier. A private method can only be accessed within the class of its definition. It cannot be access from outside of the class.
- **Unspecified** : If the access level of a method is unspecified, the method can only be accessed within the package. It cannot be accessed from outside the package.
- **protected** : The access level of a protected methods is within the package and/or outside the package through child class.
- **public** : No restriction on the access to public

# Data Type of Returning Values

- A method may return a value. The data type in the method declaration is thus the data type of the value the method returns.
- To return a value from a method, use the keyword **return**
- A method may not return a value. The data type in the method declaration is specified using the keyword **void**
- If a method returns a value, it is called a **value returning method (or function)**. If a method does not return a value, it is often called a **void method (or procedure)**

# Examples

```
public class TestMethods {  
    public int addTwoNumber(int a, int b) {  
        int c = a + b;  
        return c;  
    }  
}
```

```
public class TestMethods {  
    public void PrintTwoNumber(int a, int b) {  
        System.out.println(a + " " + b);  
    }  
}
```

---

# Method Body

- A method body contains a set of Java statements that carries out some specified tasks.
  - The body of a method can be left empty. A method with empty body is called a **stub**.
  - A stub often return *void* or be declared in an **interface** that often is used to express common operations.
-

# Examples

## A Stub

```
public class TestMethods {  
    public void aStubRight() {  
    }  
}
```

## Syntax Error

```
public class TestMethods {  
    public int aStubWrong() {  
    }  
}
```

## A Method Declaration in an Interface

```
public interface TestMethod3 {  
    public int add(int a, int b);  
}
```



# Stubs

- Software developers often create an empty method as a placeholder to make a software system design logically complete, then filling in the body at later stage.

## Syntax Error

```
public class TestMethods {  
    public int aStubWrong() {  
    }  
}
```

## Correcting by returning a pseudo value

```
public class TestMethods {  
    public int aStubWrong() {  
        return 0;  
    }  
}
```

# Method Body-Example1

The code to be executed by the method is contained inside **curly braces**. The **main()** Method in Java is special because it is the method that is always run first by the java Virtual Machine (JVM), Each line in the main() method is executed in sequence from the top down. The following two blocks of code provide sample method in Java

```
public class MethodExample1
{
    public static void main (String args [])
    {
        //This is how we call the method printGreeting()
        printGreeting();
    }
    // This method is public so it can be called and run by any other program
    // Note that the return type is void and the parameter list is empty
    public static void printGreeting()
    {
        System.out.println("Hello there!");
    }
}
```

---

## Keyword: Static

- Static methods are also known as **class method** that are declared with a class.
  - If you do not declare the method to be static then it must be called by creating an object of the class. This is known as an **instance method**.
  - Static methods are commonly used to **perform calculations that are independent of any object** that might be defined in a class.
-

# Method Body cont'd

```
public class MethodsExample2 {  
    public static void main (String args [])  
    { String myName = "Aoife";  
        //This is how we call the method printMyName(String name)  
        printMyName("Brian");  
    }  
    // This method is public so it can be called and run by any other program  
    // Note that the return type is void and that it takes one parameter of type String  
    public static void printMyName(String name)  
    {  
        System.out.println("Hello there " + name);  
    }  
}
```

The example methods above are **public** so they can be called and run by any other program from anywhere because they are in a public class. These methods are also **static**, which means that the methods belong to the class and that when you call it from outside the class you need to call it in certain format.

---

# Calling a static method from outside

The required format is:

The class name followed by a dot, followed by the method name

For example to call the method `printGreeting()` which is contained in the `MethodsExample1` class we could use:

**`MethodExample1. printGreeting()`**

---

# Calling a static method from outside

## Example

```
package Lecture02;
/*
 *
 * This is a program that will call the printMyName method from the
 * MethodsExample class above
 *
 */
public class OutsideClassExample{

    public static void main(String args[]) {

        MethodsExample2.printMyName("CYZhang"); }
}
```

# Method Example3

The following example contains a static method that print the following header information:

```
*****  
+++++++welcome+++++++  
*****
```

The method should be called `printHead()`, and should be called from the `main()` method.

```
public class MethodExample3 {  
    public static void main(String[] args) {  
        printHead();  
    }  
    public static void printHead() {  
        System.out.println("*****");  
        System.out.println("+++++++welcome+++++++");  
        System.out.println("*****");  
    }  
}
```

# Method Example4

The following example contains a static method that takes a single parameter of type **int** and prints a row of stars with the same length as the same integer.

```
public class MethodsExample4
{
    public static void main (String args[])
    {
        int n = 20;
        //Calling the printStars method and passing n as a parameter
        printStars(n);
    }
    //Creating a method that accept and integer n
    public static void printStars(int n)
    {
        for (int i = 0; i<n ;i++)
        {
            System.out.print("*");
        }
        System.out.println();
    }
}
```



# Multiple Parameters (1)

All the methods that we write are capable of taking multiple parameters. To do this, we separate individual parameter with **commas**.

```
public class MethodsClass
{
    public static void main (String args[])
    {
        double d1 = 10.5;
        double d2 = 22.8;
        sum(d1,d2);

        int a = -4;
        int b = 9;
        int c = 5;
        getSmallest(a,b,c);
        average(a,b);
    }
}
```

# Multiple Parameters (2)

```
public static void getSmallest(int x, int y, int z)
{
    if(x<y && x<z)
    {
        System.out.println(x+ " is the smallest");
    }
    else if(y<x && y<z)
    {
        System.out.println(y + " is the smallest");
    }
    else
    {
        System.out.println(z + " is the smallest");
    }
}
```

---

# Multiple Parameters (3)

```
public static void average(int a, int b)
{
    System.out.println((a + b)/2);
}
```

```
public static void sum( double x, double y)
{
    System.out.println("The sum of " + x + " + " + y + "=" + (x+y));
}
```

---

# Parameter Names—same names

You can use a same name for the parameters when you call a method (actual parameters) to the names used in the method signature (formal parameters).

```
public class MethodScope1
{
    // instance variables - replace the example below with your own
    public static void main(String[] args)
    {
        int inValue = 4;
        //Actual parameters
        twoTimes(inValue);

        } //same names
    // Formal parameters public static void twoTimes(int inValue)
    public static void twoTimes(int inValue)
    {

        System.out.println("The result is "+inValue*2);

    }
}
```

# Parameter Names—different names

You can also use different names for the parameters when you call a method (actual parameters) to the names used in the method signature (formal parameters).

```
public class MethodScope2
{
    // instance variables - replace the example below with your own
    public static void main(String[] args)
    {
        int inValue = 4;
        //Actual parameters
        int result=twoTimes(inValue);
        System.out.println("The result is "+result);

        } //same names
    // Formal parameters public static void twoTimes(int inValue)
    public static int twoTimes(int Value)
    {
        int answer= Value*2;
        return answer;
    }
}
```

# Scope of Identifiers

You cannot use variables `mainValue` and `inValue` interchangeably.

```
public class MethodScope2
{
    // instance variables - replace the example below with your own
    public static void main(String[] args)
    {
        int inValue = 4;
        //Actual parameters
        int result=twoTimes(inValue);
        System.out.println("The result is "+result);

        } //same names
    // Formal parameters public static void twoTimes(int inValue)
    public static int twoTimes(int Value)
    {
        int answer= Value*2;
        return answer;
    }
}
```

# Returning Values from Methods

Methods typically are used for two types of tasks:

1. Perform a task and print the result. These methods return “void” or nothing.
2. Perform a task and then return a value back to the method from where it was called.

```
public class MethodScope2
{
    // instance variables - replace the example below with your own
    public static void main(String[] args)
    {
        int inValue = 4;
        //Actual parameters
        int result=twoTimes(inValue);
        System.out.println("The result is "+result);

        } //same names
    // Formal parameters public static void twoTimes(int inValue)
    public static int twoTimes(int Value)
    {
        int answer= Value*2;
        return answer;
    }
}
```

# Sigma Example

The sigma of a number is the sum of all numbers from 1 to that number inclusive. For example

$$\text{Sigma}(0) = 0$$

$$\text{Sigma}(1) = 1$$

$$\text{Sigma}(2) = 1 + 2$$

$$\text{Sigma}(3) = 1 + 2 + 3$$

$$\text{Sigma}(4) = 1 + 2 + 3 + 4$$

...

$$\text{Sigma}(n) = 1 + 2 + \dots + n-2 + n-1 + n$$

We could use a loop as

```
int result = 0;
for (int i=1; i<=n; i++) {
    result +=i;
}
```



# Sigma Example

Imaging having to write this piece of code over and over again. You can see the benefit of putting it in a method and making this code **re-usable**.

```
public class SigmaMethod{
    public static void main(String args[]){
        int myNum = 10, sum =0;
        System.out.println("My sigma for the number "+myNum+" is: "+mySigma(myNum));
    }

    public static int mySigma(int n){
        int result = 0;
        for (int i=1; i<=n; i++) { |
            result +=i;
        }
        return result;
    }
}
```

Add a method header and create a suitable method body (with a return statement)

# Sigma Example

What if we wanted to find the sigma of a range of numbers, let's say between 1 and 10? We would have to add a loop as

```
public class SigmaMethod2 {  
    public static void main(String args[]){  
        int n = 10, sum =0, result = 0;  
        for (int i = 1; i <= n;i++) {  
            result = mySigma(i);  
            System.out.println("The sigma for the number "+ i +" is: "+ result);  
        }  
    }  
  
    public static int mySigma(int n){  
        int result = 0;  
        for (int i=1; i<=n; i++) {  
            result +=i;  
        }  
        return result;  
    }  
}
```

# Factorial Example

Just like the Sigma function, you have seen the factorial function previously when you looked at the iteration. We could put it in a method

```
public class FactorialMethod
{
    public static void main (String args[])
    {
        int fact = 0;
        int n=5;
        fact = factorial(n);
        System.out.println("The factorial is " + fact);
    }

    public static int factorial (int n)
    { int result = 1;
      for(int i = 1; i<=n; i++)
      { |
        result = result*i;
      }
      return result;
    }
}
```

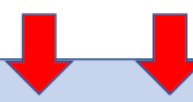
---

# Parameter List

- Variables defined in the method header are called formal parameters or simply **parameters**
  - Parameters are optional. A method may have one or more parameters or not at all.
  - Each parameter in a method must be declared separately, i.e., each parameter in the list must have a data type and a name.
  - Multiple parameters are separated using a comma sign “,”
-

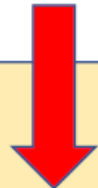
# Examples

Two Parameters



```
public class TestMethods {  
    public int addTwoNumber(int a, int b) {  
        int c = a + b;  
  
        return c;  
    }  
}
```

No Parameters



```
public class TestMethods {  
  
    public void PrintTwoNumber( ) {  
        System.out.println("test");  
    }  
}
```

---

# Method Signature

- A method's **signature** is the combination of the method name and the number of parameters , the data types of parameters and the order of the appearance of parameters in the method declaration.
  - A method call must match the called method's signature.
-

---

# Method Overloading

When we are defining methods within the same class, we can have multiple methods with the same name. **Two or more methods within the same class can share the same name, as long as their parameter declarations are different.** This is called **method overloading**.

These overloaded methods can be distinguished by :

- The types of their parameters.
  - The number of their parameters.
-

# Method Overloading-example

```
public class MethodOverload
{
    public static void main(String args[])
    {
        timesTwo(2);
        timesTwo(2.0);
    }

    public static void timesTwo(int y)
    {
        System.out.println("Im in the method handling integers only");
        System.out.println(y*2);
    }

    public static void timesTwo(double y)
    {
        System.out.println("Im in the method handling doubles only");
        System.out.println(y*2.0);
    }
}
```



# Method Overloading-example

```
public class TestOverLoadMethods{
    public static void main(String args[]){
        int x = 8 , y = 5 , r = 45;
        double z = 5.1, n = 6.5;
        int result = mult(x,y);    //---> mult(8,5)
        System.out.println("Result: " + result);
        double result1 = mult(z,n);
        System.out.println("Result: " + result1);
        mult(x,y,r);
    }

    public static int mult(int x, int y, int z){
        return (x * y * z);
    }

    public static int mult(int x, int y){
        return (x * y);
    }

    public static double mult(double x, double y){
        return (x * y);
    }
}
```

---

# Why overload methods?

- Facilitate the programming by reducing the number of identifiers.

multInt () + multFloat () + multDouble () +...  
=>mult()

# Method Overloading-example

```
public int method(int a, double b) {  
    return 0;  
}
```

```
public int method(double a, int b) {  
    return 1;  
}
```

```
public int method(int a) {  
    return 2;  
}
```

```
public int method(double a) {  
    return 3;  
}
```

Overloaded methods must have different parameter lists. You can NOT overload methods based on different modifiers or return types.

# Ambiguous Overloading

```
public class TestA0 {  
    public static void main(String[] args) {  
        TestA0 taol = new TestA0();  
  
        int num1 = 10;  
        int num2 = 20;  
        System.out.println(taol.add(num1, num2));  
    }  
}
```

```
private double add(int a, double b) {  
    return (a + b);  
}
```

Find the closest match!

```
private double add(double a, double b) {  
    return (a + b);  
}
```

Results: 30.00

```
public class TestA0 {  
    public static void main(String[] args) {  
        TestA0 taol = new TestA0();  
  
        int num1 = 10;  
        int num2 = 20;  
        System.out.println(taol.add(num1, num2));  
    }  
}
```

```
private double add(int a, double b) {  
    return (a + b);  
}
```

```
private double add(double a, int b) {  
    return (a + b);  
}
```

Error! Ambiguous Overloading!