School of Science & Engineering
Department of CSE
Canadian University of Bangladesh

Lecture-5: Methods

Semester: Summer 2024

Object-Oriented Problem Solving

Methods

Based on Chapter 6 of "Introduction to Java Programming" by Y. Daniel Liang.

Outline

- Introduction (6.1)
- Defining a method (6.2)
- Calling a method (6.3)
- A void Method Example (6.4)
- Passing Arguments by Values (6.5)
- Modularizing Code (6.6)
- Overloading methods (6.8)
- The scope of variables (6.9)

Introduction

- A *method* is a collection of statements grouped together to perform an operation.
- *Methods* can be used to define reusable code and organize and simplify code.

Example of a Reusable Code

```
int sum = 0;
for ( int i = 1; i <= 10; i++)
    sum += i;
System.out.println("Sum from 1 to 10 is "+ sum);
```

Compute the sum from 1 to 10

```
int sum = 0;
for ( int i = 20; i <= 37; i++)
    sum += i;
System.out.println("Sum from 20 to 37 is "+ sum);
```

Compute the sum from 20 to 37

```
int sum = 0;
for ( int i = 35; i <= 49; i++)
    sum += i;
System.out.println("Sum from 35 to 49 is "+ sum);
```

Compute the sum from 35 to 49

Example of a Reusable Code (Cont.)

- It would be nice to write the common code once and reuse it.
- This is achieved by:
  - *Defining* a method that contains the common code.
  - Reuse it by *invoking* it with different values.

```
public static int sum (int i1, int i2){
    int result= 0;
    for ( int i = i1; i <= i2; i++)
        result += i;
    return result;
}
```

```
public static void main (String [] args){
    System.out.println("Sum from 1 to 10 is" + sum (1, 10) );
    System.out.println("Sum from 20 to 37 is" + sum (20, 37) );
    System.out.println("Sum from 35 to 49 is" + sum (35, 49) );
}
```

```
public class SumCalculator {

    public static int sum(int i1, int i2) {
        int result = 0;
        for (int i = i1; i <= i2; i++) {
            result += i;
        }
        return result;
    }

    public static void main(String[] args) {
        System.out.println("Sum from 1 to 10 is " + sum(1, 10));
        System.out.println("Sum from 20 to 37 is " + sum(20, 37));
        System.out.println("Sum from 35 to 49 is " + sum(35, 49));
    }
}
```

Defining a Method

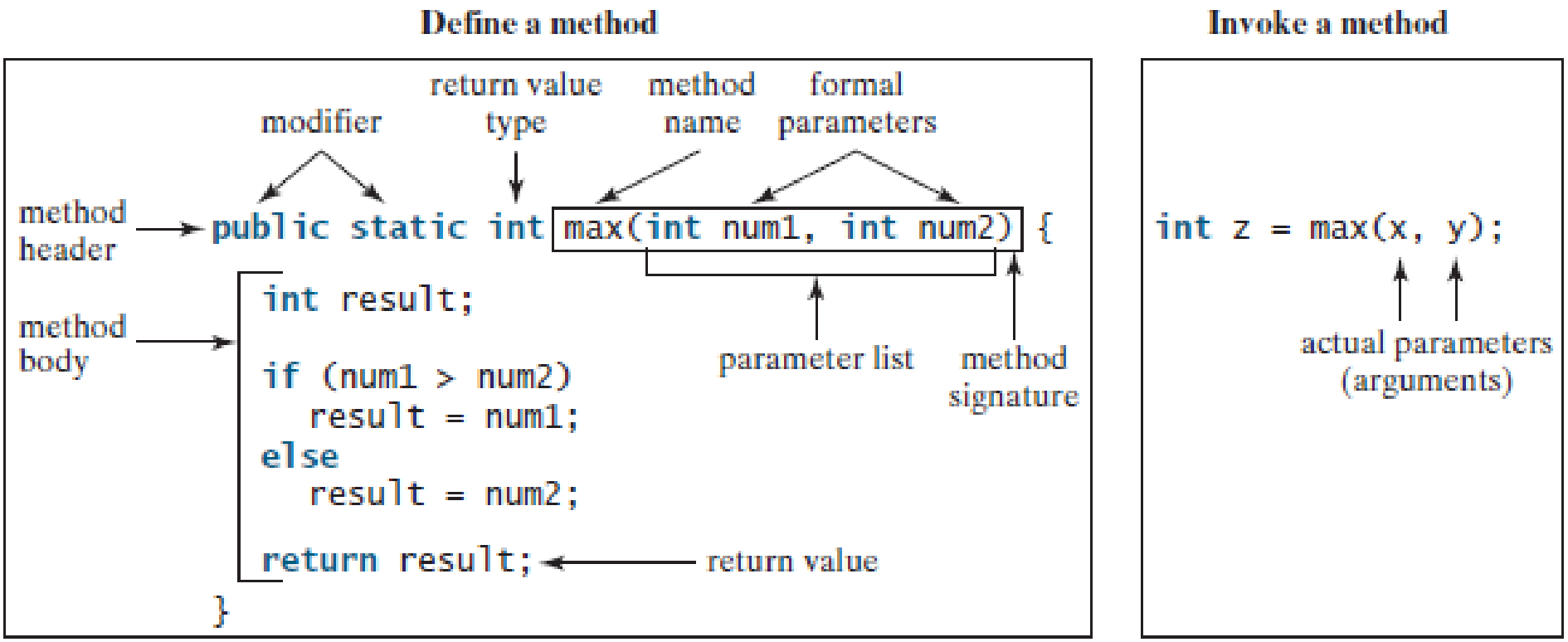
- A method definition consists of:
  - Return value type.
  - Method name.
  - Parameters.
  - Body.
- The syntax for defining a method is:

```
modifier returnValueType methodName (list of parameters){
        //method body
}
```

Defining a Method (Cont.)

- The *returnValueType* is the data type of the value the method returns.
- Some methods perform desired operations without returning a value.
  - In this case, the *returnValueType* is the keyword *void*.
- If a method returns a value, it is called a *value-returning method;* otherwise it is called a *void method*.

Method Definition: An



- In a method definition, you define what the method is to do.

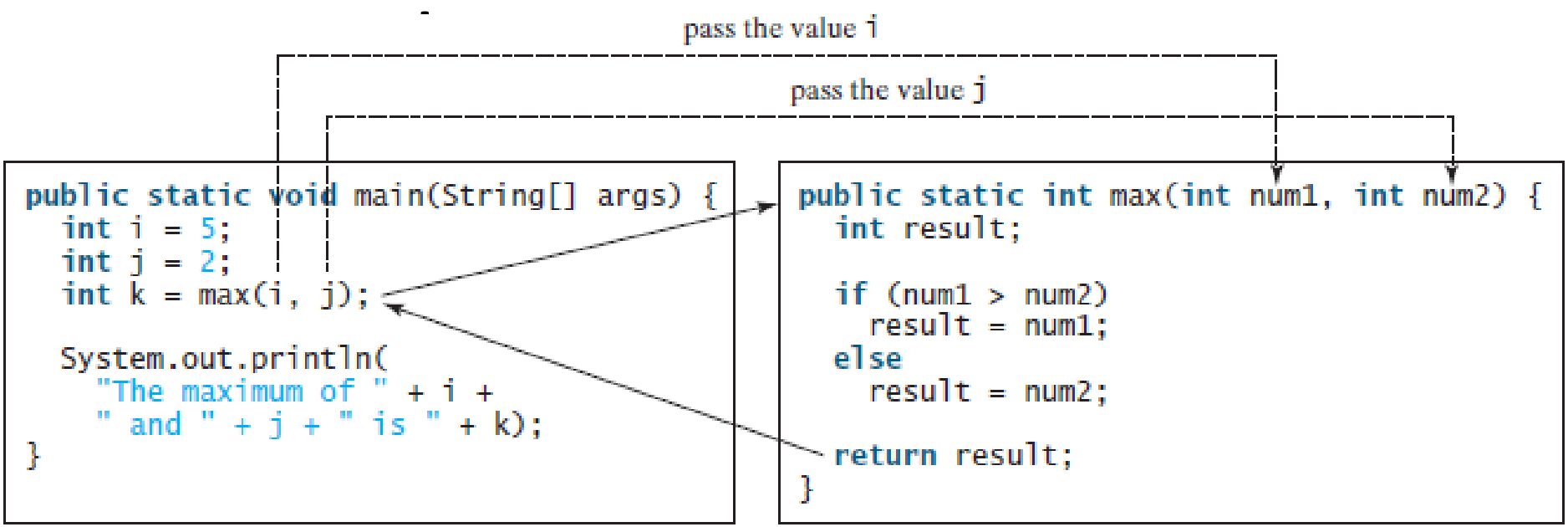
Calling a method

- Calling a method executes the code in the method.
- There are two ways to call a method, depending on whether the method returns a value or not.
- If a method returns a value, a call to the method is usually treated as a value.
  - *int larger = max(3, 4);*

  //calls max(3, 4) and assigns the result of the method to the variable larger.
  - *System.out.println(max(3, 4));*

  //prints the return value of the method call max(3, 4).
- If a method returns *void*, a call to the method must be a statement.
  - For example, the method *println* returns *void*. The following call is a statement:

Method Invocation: An



- When a program calls a method, program control is transferred to the called method.
- A called method returns control to the caller when:
  - Either its return statement is executed, or
  - Its method-ending closing brace is reached.

TestMax.java

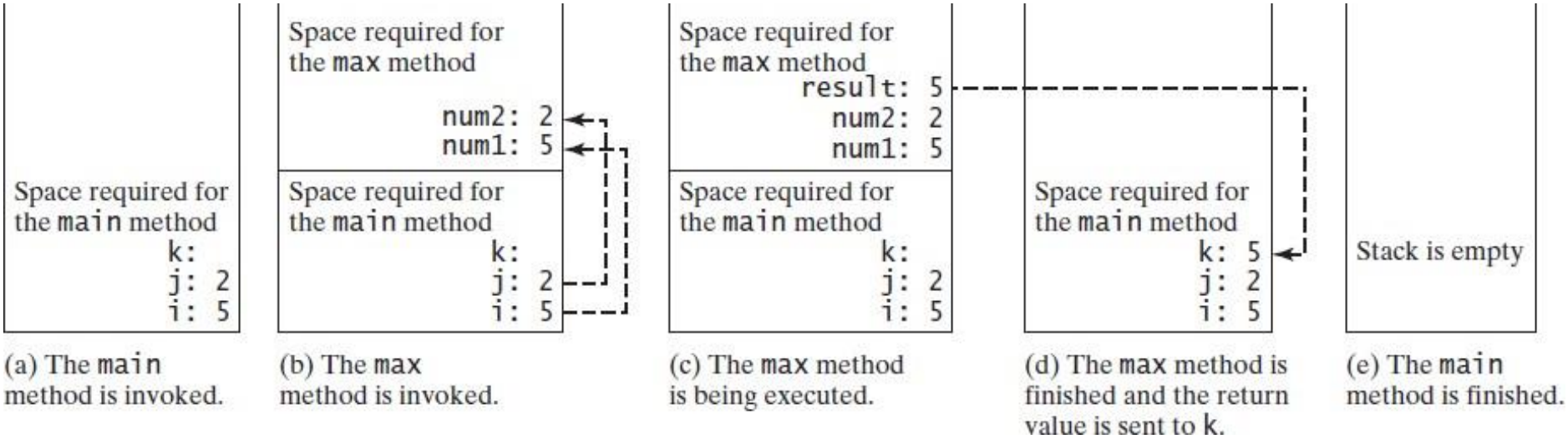
LISTING 6.1 TestMax.java

```
public class TestMax {
/** Main method */
public static void main(String[] args) {
int i = 5;
int j = 2;
int k = max(i, j);
System.out.println("The maximum of " + i + " and " + j + " is " + k);
 }

/** Return the max of two numbers */
public static int max(int num1, int num2) {
int result;

if (num1 > num2)
result = num1;
else
result = num2;

return result;
}
}
```

What happens when a method is invoked?

- Each time a method is invoked, the system creates an *activation record*.
  - *Activation record* stores parameters and variables for the method .
  - *Activation record* is placed in an area of memory known as the *call stack*, or simply the *stack*.
- When a method invokes another method, the caller's activation record is kept intact, and a new activation record is created.
- When a method finishes its work and returns to its caller, its activation record is removed from the stack.
- A call stack stores methods in last-in, first-out fashion.

What happens when a method is invoked? (Example)



(a) The `main` method is invoked.

(b) The `max` method is invoked.

(c) The `max` method is being executed.

(d) The `max` method is finished and the return value is sent to `k`.

(e) The `main` method is finished.

A *void* Method Example

- A *void* method does not return a value.

```
public static void printGrade(double score){
        if (score >= 90.0)
                System.out.println ('A');
        else if (score >= 80.0)
                System.out.println ('B');
        else if (score >= 70.0)
                System.out.println ('C');
        else if (score >= 60.0)
                System.out.println ('D');
        else
                System.out.println ('F');
}
```

Example of calling this method:

```
System.out.print ("The grade is ");
printGrade(78.5);
```

Passing Arguments by Values

- When calling a method, you need to provide *arguments*, which must match the *parameters* defined in the method signature in:
  - Order
  - Number.
  - Compatible type.

```
public static void nPrintln (String message, int
    n){  for (int i =0; i < n; i++)
        System.out.println(message);
}
```

```
nPrintln ("Hello",
3);
```

```
nPrintln (3, "Hello");
```

Passing Arguments by Values (Cont.)

- When you invoke a method with an argument, the value of the argument is passed to the parameter.
  - This is referred to as *pass-by-value*.
- If a value of a variable is passed as an argument to a parameter, <u>the variable is not affected</u>, regardless of the changes made to the parameter inside the method.

Passing Arguments by Values (Example)

```
public class Increment{
    public static void main (String [] args){
        int x = 1;
        System.out.println("Before the call, x is "+  x);
        increment (x);
        System.out.println("After the call, x is "+    x);
    }

    public static void increment (int
        n){  n++;
        System.out.println("n inside the method is " + n);
    }
}
```

Passing Arguments by Values (Example Cont.)

```
Before the call, x is 1
n inside the method is 2?
After the call, x is 1
```

Value of x does not change

Modularizing Code

- Modularizing makes the code:
  - Clear and easy to read.
    - Isolates parts used to perform specific computations from the rest of the code.
  - Easy to maintain and debug.
    - Narrows the scope of debugging.
  - Reusable.
    - Code can be reused by other programs.

Overloading Methods

- Overloading methods enables you to define the methods with the same name as long as their signatures are different.
- Methods overloading is having two or more methods that have the *same name*, but *different parameter lists* within one class.
- The Java compiler determines which method to use based on the *method signature*.
  - It finds the most specific method for a method invocation.

Overloading Methods: An Example

```
public static int max (int num1, int
      num2){  if (num1 > num2)
              return num1;
      else return num2;
}
```

```
public static double max (double num1, double num2){
      if (num1 > num2)
              return num1;
      else return num2;
}
```

```
public static double max (double num1, double num2,
double num3){
      return max (max(num1, num2), num3);
}
```

max(3.0,4.5)

max(3,4)

max(3.1,4.5,5.5)

max(2,2.5)

The Scope of Variables

- The *scope* of a variable is the part of the program where the variable can be referenced.
- A variable defined inside a method is referred to as a *local variable*.
  - The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
  - A local variable must be declared and assigned a value before it can be used.
- A parameter is actually a local variable.
  - The scope of a method parameter covers the entire method.

The Scope of Variables (Cont.)

A variable declared in the initial-action part of a for-loop header has its scope in the entire loop.

A variable declared inside a for-loop body has its scope limited in the loop body from its declaration to the end of the block.

The scope of i

The scope of j

```
public static void method1() {
    .
    .
    for (int i = 1; i < 10; i++) {
        .
        .
        int j;
        .
        .
        .
    }
}
```

The Scope of Variables (Cont.)

- You can declare a local variable with the same name in different blocks in a method.
- But you cannot declare a local variable twice in the same block or in nested blocks.

It is fine to declare i in two nonnested blocks.

```
public static void method1() {
  int x = 1;
  int y = 1;

  for (int i = 1; i < 10; i++) {
    x += i;
  }

  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

It is wrong to declare i in two nested blocks.

```
public static void method2() {

  int i = 1;
  int sum = 0;

  for (int i = 1; i < 10; i++)
    sum += i;
  }

}
```