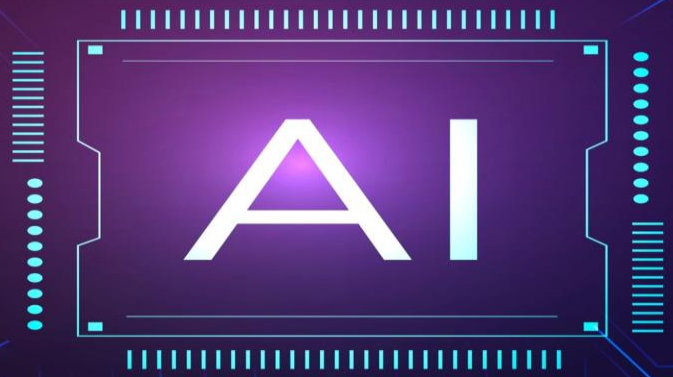


AP Computer Science A

Java Programming Essentials [Ver. 3.0]

Unit 2: Structured Programming

CHAPTER 6: METHODS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Features for basic functions.
- Method as class function or instance function.
- Calling Method, Parameters, Return Values, and Activation Records
- Procedural Abstraction
- Overloading
- Recursion
- DNA Computing and Craps game

Function

LECTURE 1

Functions

- Functions are "self contained" modules of code that accomplish a specific task. Functions usually "take in" data, process it, and "return" a result. Once a function is written, it can be used over and over and over again. Functions can be "called" from the inside of other functions.

Why do we Write Functions?

1. They allow us to conceive of our program as a bunch of sub-steps. (Each sub-step can be its own function. When any program seems too hard, just break the overall program into sub-steps!)
2. They allow us to reuse code instead of rewriting it.
3. Functions allow us to keep our variable namespace clean (local variables only "live" as long as the function does). In other words, function_1 can use a variable called i, and function_2 can also use a variable called i and there is no confusion. Each variable i only exists when the computer is executing the given function.
4. Functions allow us to test small parts of our program in isolation from the rest. This is especially true in interpreted languages, such as Matlab, but can be useful in C, Java, ActionScript, etc.

Steps to Writing a Function

1. Understand the purpose of the function.
2. Define the data that comes into the function from the caller (in the form of parameters)!
3. Define what data variables are needed inside the function to accomplish its goal.
4. Decide on the set of steps that the program will use to accomplish this goal. (The Algorithm)

Functional Abstraction

- A function is a named subprogram that performs a specific task when it is called.
- The function's specification is a description of what the function does.
- Functions use a parameter list for data flow from the caller to the function and from the function to the caller.

Scope and Lifetime of Variables

- The scope of a variable defines the section of the code in which the variable is visible. As a general rule, variables that are defined within a block are not accessible outside that block.
- The lifetime of a variable refers to how long the variable exists before it is destroyed. Destroying variables refers to deallocating the memory that was allotted to the variables when declaring it. We have written a few classes till now.
- You might have observed that not all variables are the same. The ones declared in the body of a method were different from those that were declared in the class itself. There are three types of variables: instance variables, formal parameters or local variables and local variables.

Formal and Actual Parameters

An **identifier** is a name used for a class, a variable, a method, or a parameter. The following definitions are useful:

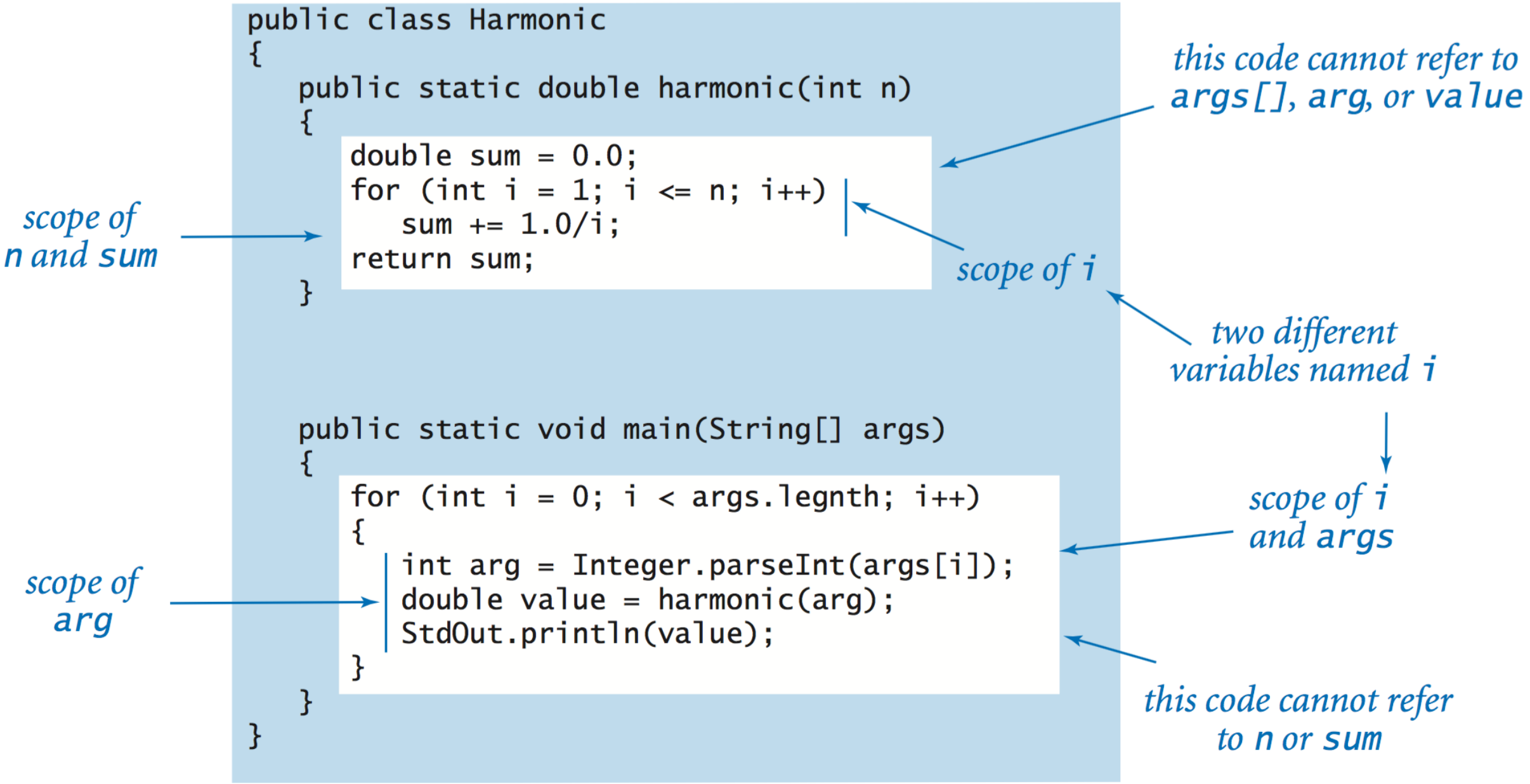
- **formal parameter** — the identifier used in a method to stand for the value that is passed into the method by a caller.
  - For example, amount is a formal parameter of processDeposit
- **actual parameter** — the actual value that is passed into the method by a caller.
  - For example, the 200 used when processDeposit is called is an actual parameter.
  - actual parameters are often called **arguments**

Return statement

- In programming, return is a statement that tells the program to leave the subroutine and return to the return address, directly after where the subroutine was called.
- In most programming languages, the return statement is either return or return value, where value is a variable or other information coming back from the subroutine. Below is an example of how the return statement could be used in the Perl programming language.

Function and Method

- A **function** is a piece of code that is called by name. It can be passed data to operate on (i.e. the parameters) and can optionally return data (the return value). All data that is passed to a function is explicitly passed.
- A **method** is a piece of code that is called by a name that is associated with an object. In most respects it is identical to a function except for two key differences:
  1. A method is implicitly passed the object on which it was called.
  2. A method is able to operate on data that is contained within the class (remembering that an object is an instance of a class - the class is the definition, the object is an instance of that data).


public class Harmonic
{
public static double harmonic(int n)
{
double sum = 0.0;
for (int i = 1; i <= n; i++)
sum += 1.0/i;
return sum;
}
public static void main(String[] args)
{
for (int i = 0; i < args.legnth; i++)
{
int arg = Integer.parseInt(args[i]);
double value = harmonic(arg);
StdOut.println(value);
}
}
}
this code cannot refer to args[], arg, or value
scope of n and sum
scope of i
two different variables named i
scope of i and args
scope of arg
this code cannot refer to n or sum

Method

LECTURE 2

Opening Problem: Code Reusability

Find the sum of integers from 1 to 10, from 20 to 37, and from 35 to 49, respectively.

Opening Problem: Code Reusability

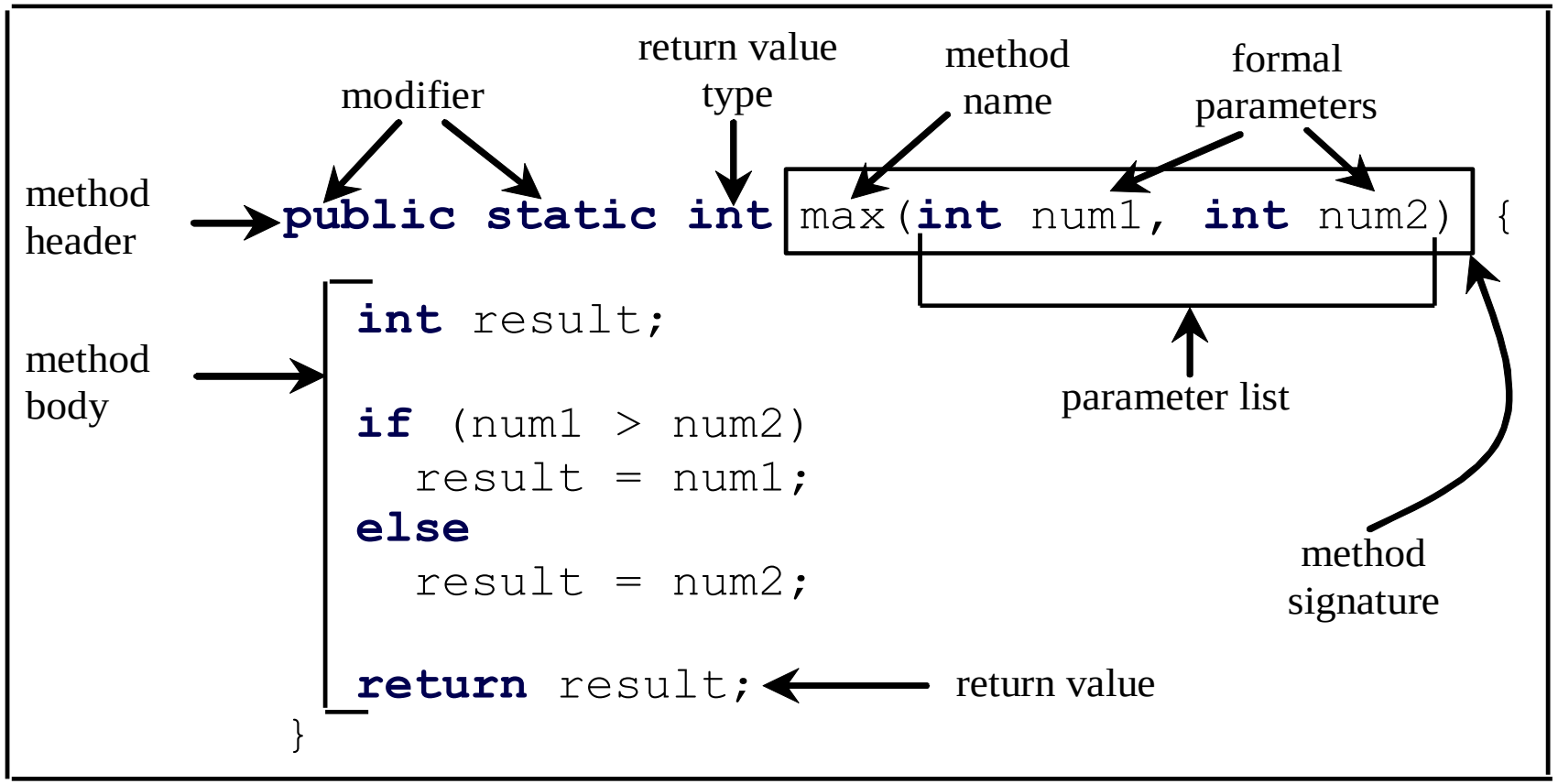
```
int sum = 0;      /* redundancy */
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

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

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

```
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 Methods

A method is a collection of statements that are grouped together to perform an operation.

Define a method



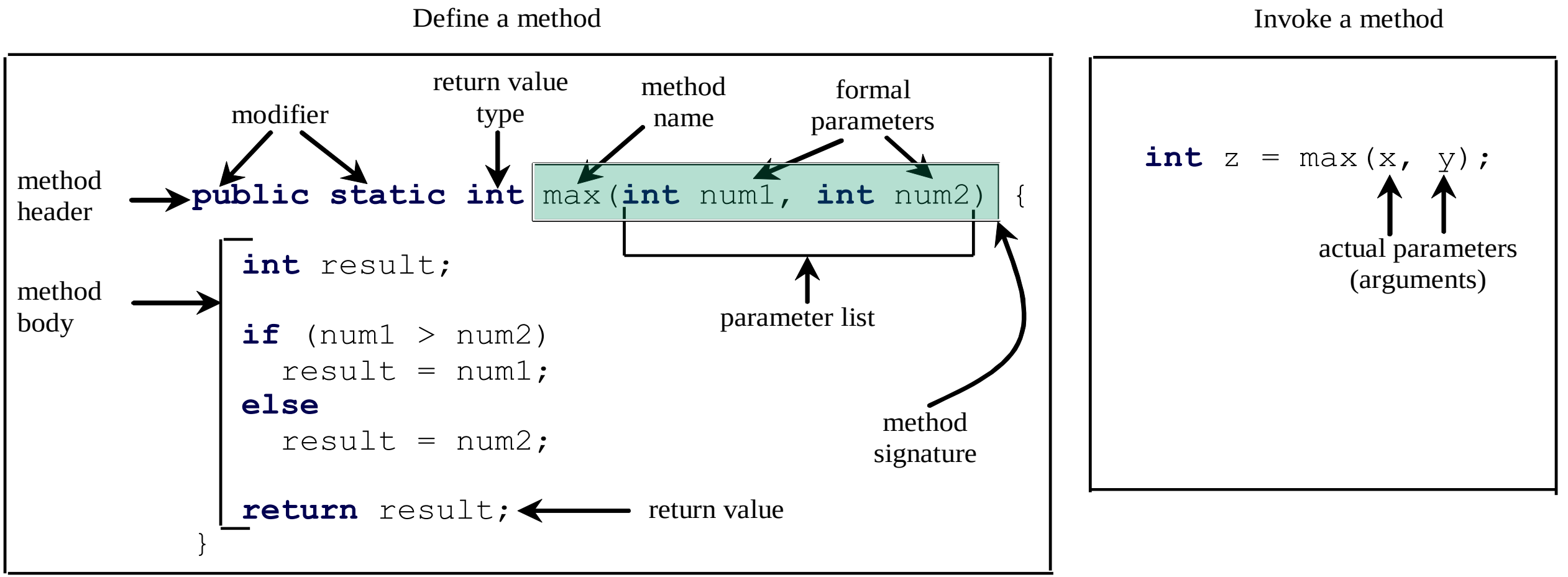
Invoke a method

```
int z = max(x, y);
```

actual parameters (arguments)

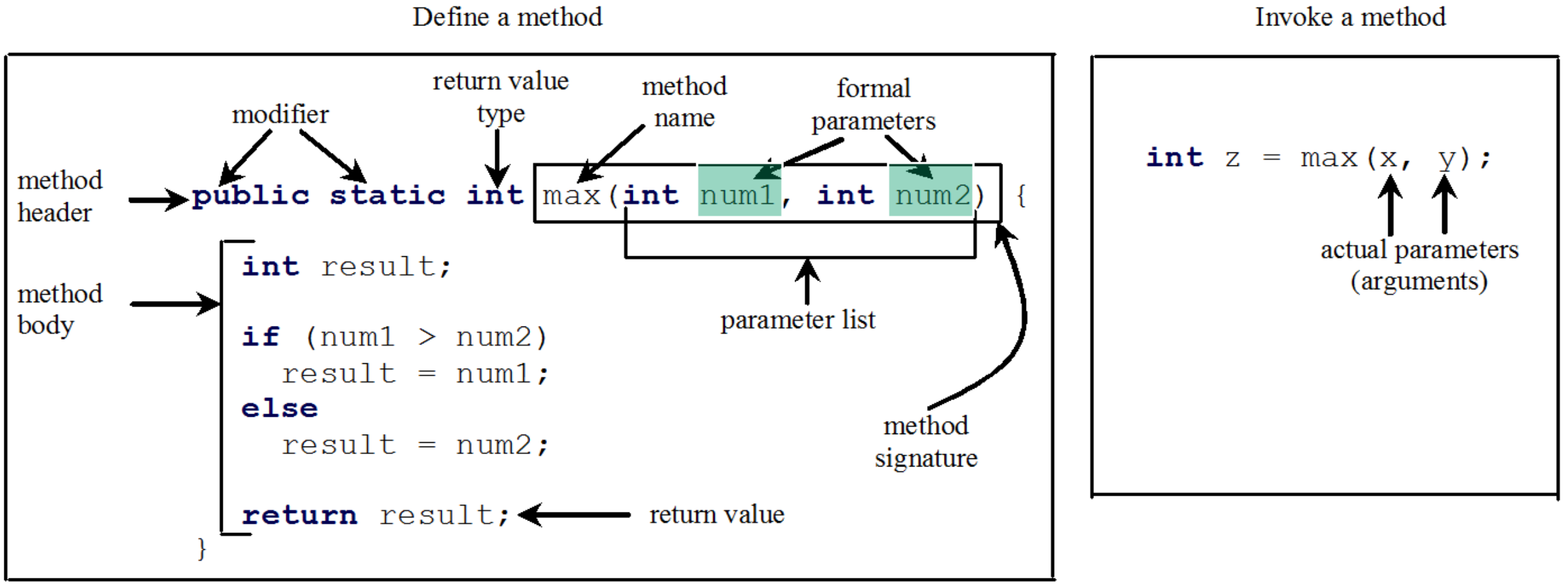
Method Signature

Method signature is the combination of the method name and the parameter list.



Formal Parameters

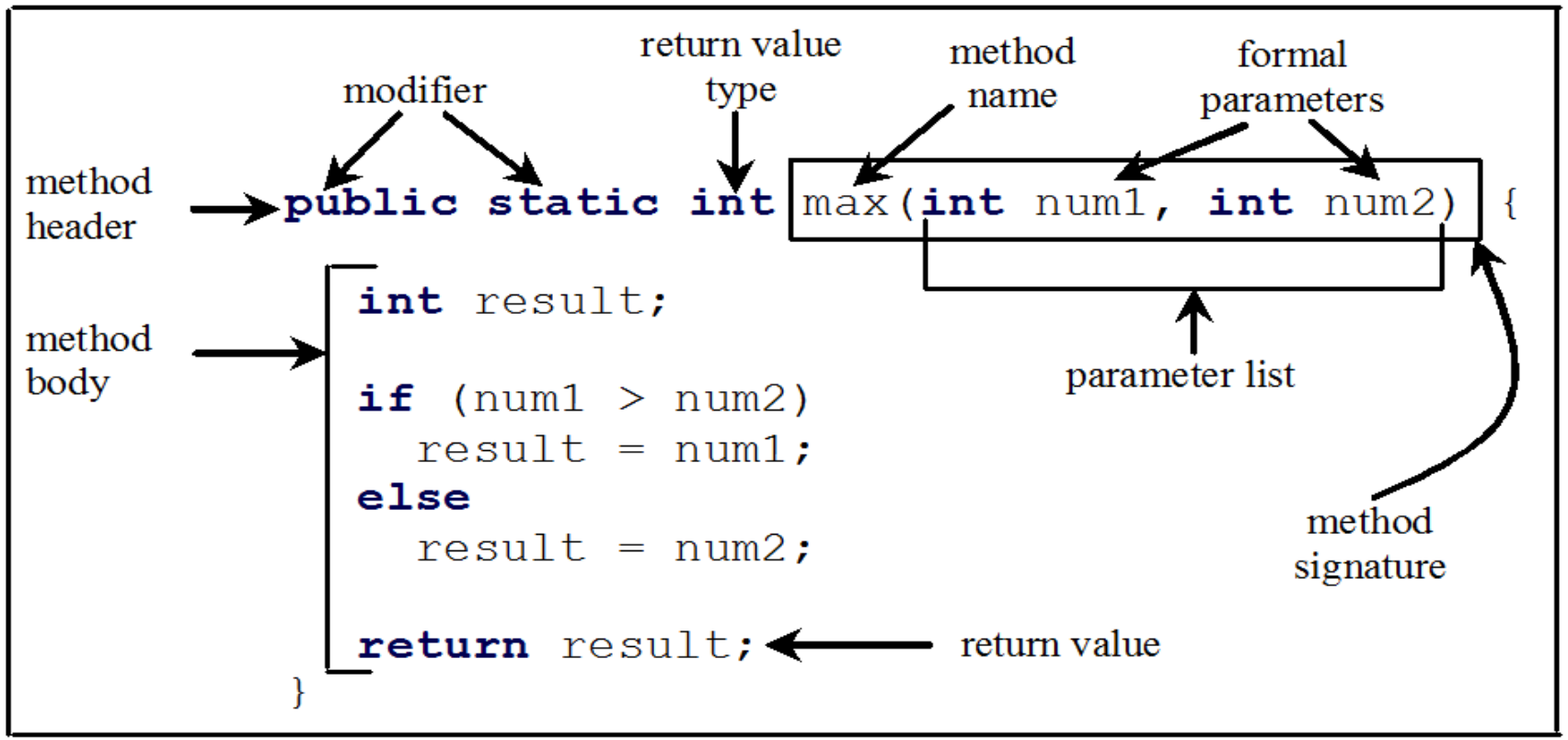
The variables defined in the method header are known as *formal parameters*.



Actual Parameters

When a method is invoked, you pass a value to the parameter. This value is referred to as *actual parameter or argument*.

Define a method



Invoke a method

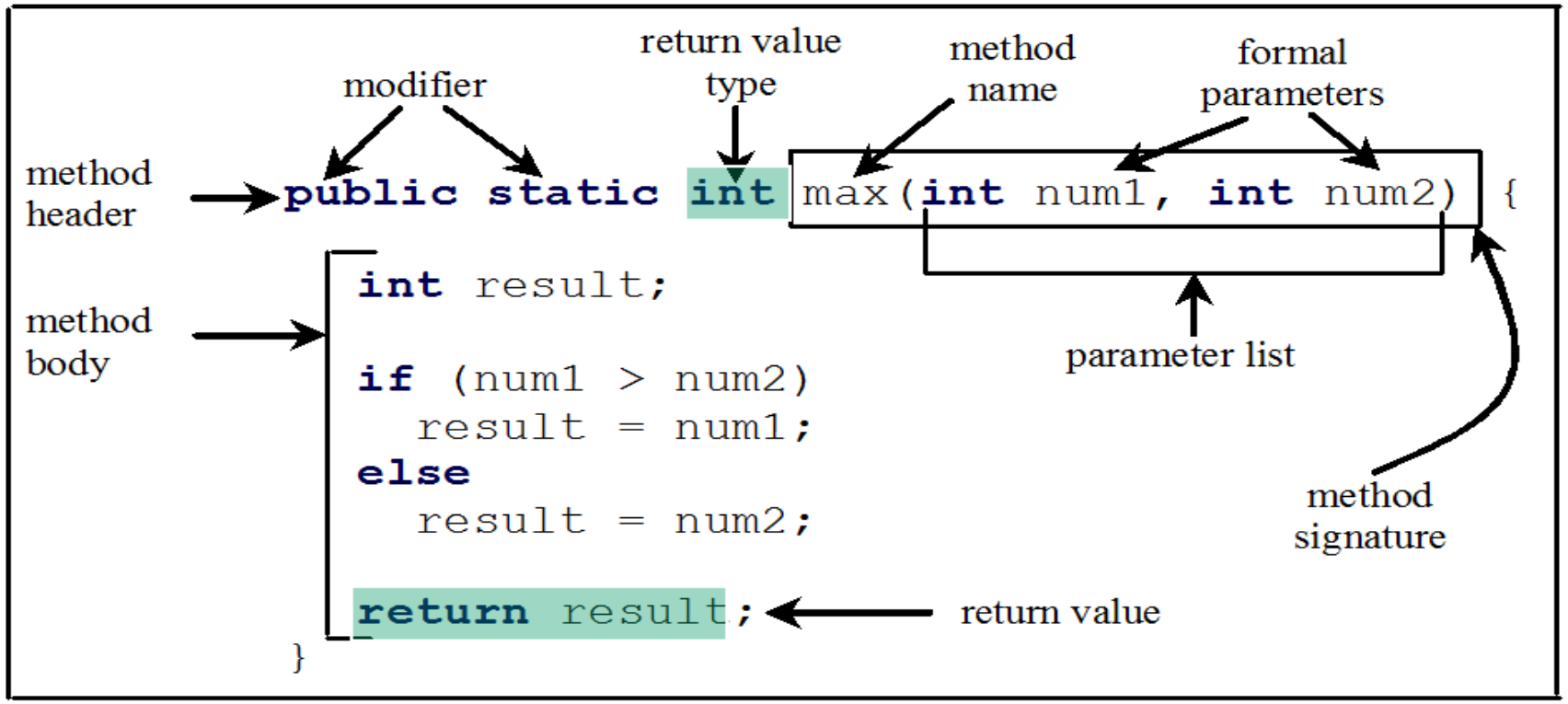
```
int z = max(x, y);
```

actual parameters (arguments)

Return Value Type

A method may return a value. The returnValueType is the data type of the value the method returns. If the method does not return a value, the returnValueType is the keyword void. For example, the returnValueType in the main method is void.

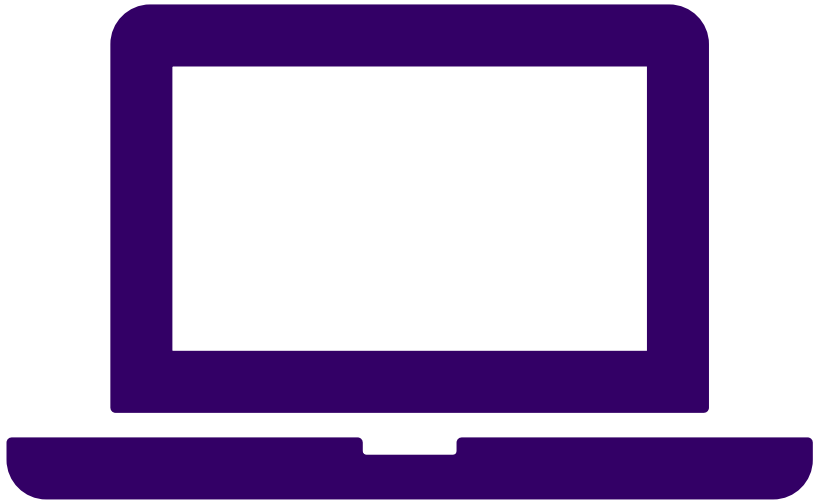
Define a method



Invoke a method

```
int z = max(x, y);
```

actual parameters (arguments)



Demonstration Program

STUDENTGPAMETHOD.JAVA

BlueJ: Terminal Window - Chapter06

Options

```
Enter Student's Name (First, Last): Eric, Chou
Enter Student's SSN (XXX-XX-XXXX): 605-05-3333
Enter Student's Date of Birth (MM/DD/YYYY): 01/02/1999
Enter Student's Address: 7 A Street
                    Washington High School
                  Semester Score Report Card
Name: Eric, Chou                          SSN: XXX-XX-3333  DOB: 01/02/1999
Address: 7 A Street
=============================================================================
Math Score:       63 English Score:    38 Physics Score:    23
Math Grade:       D  English Grade:    F  Physics Grade:    F
Chem. Score:      42 P.E. Score:       0  U.S.Hist. Score:  17
Chem. Grade:      F  P.E. Grade:       F  U.S.Hist. Grade:  F
Student's GPA: 0.17
```

Calling a Method

LECTURE 3

Similarity between Mathematical Function and Java Methods

Mathematical function:

$y = f(x) = x^2 + x + 1$

*x: independent variable over **domain***

y: dependent variable over field

Use of function:

$y = f(3) = 3^2 + 3 + 1$

Java Method:

```
public double f(double x){
  double y = x * x + x + 1;
  return y;
}
```

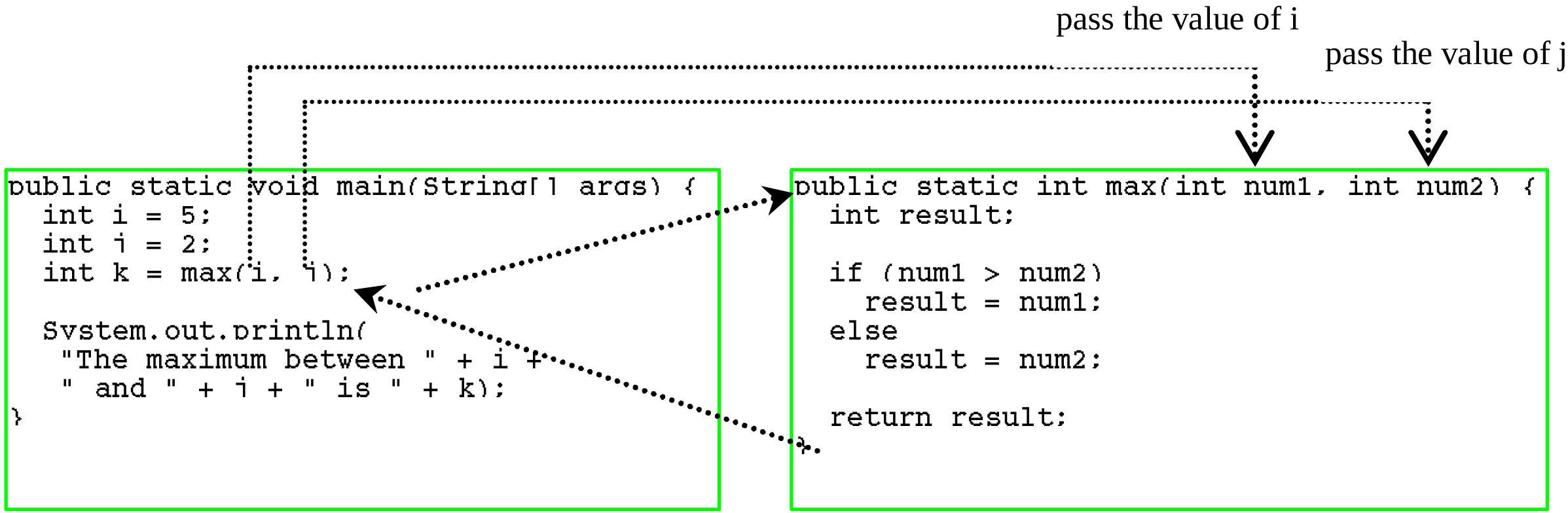
Calling f(x):

```
y = f(3);  // one sample point.
```

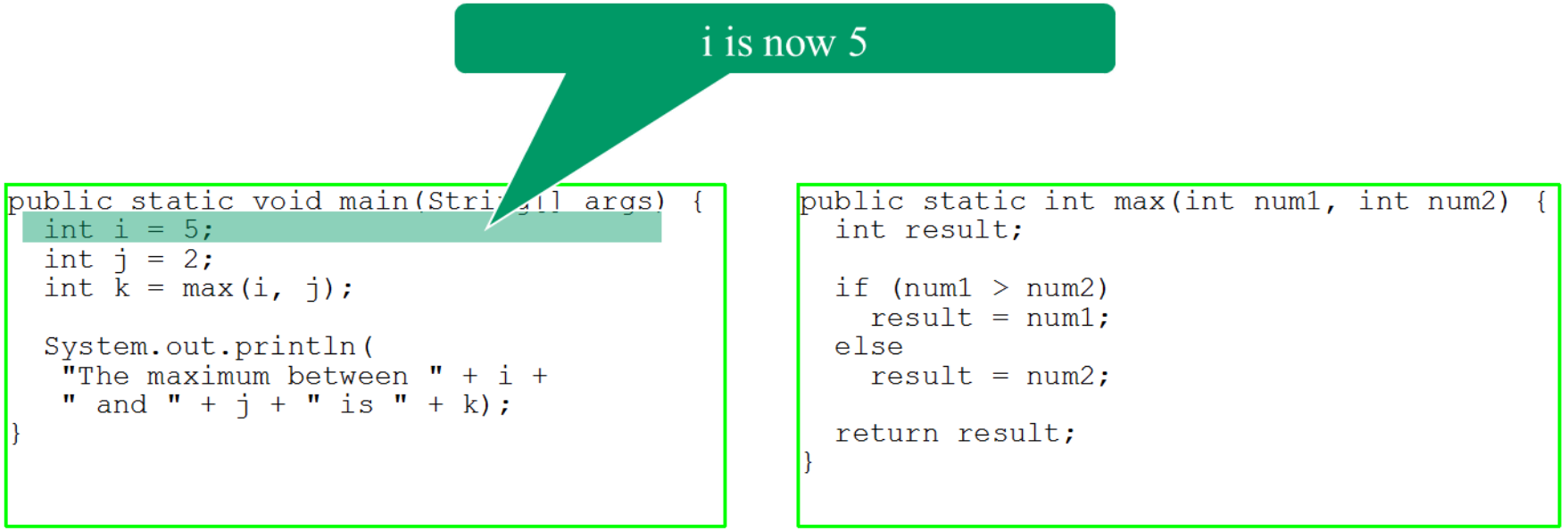
Calling Methods

- Testing the max method
- This lecture illustrates calling a method max to return the largest of the int values.

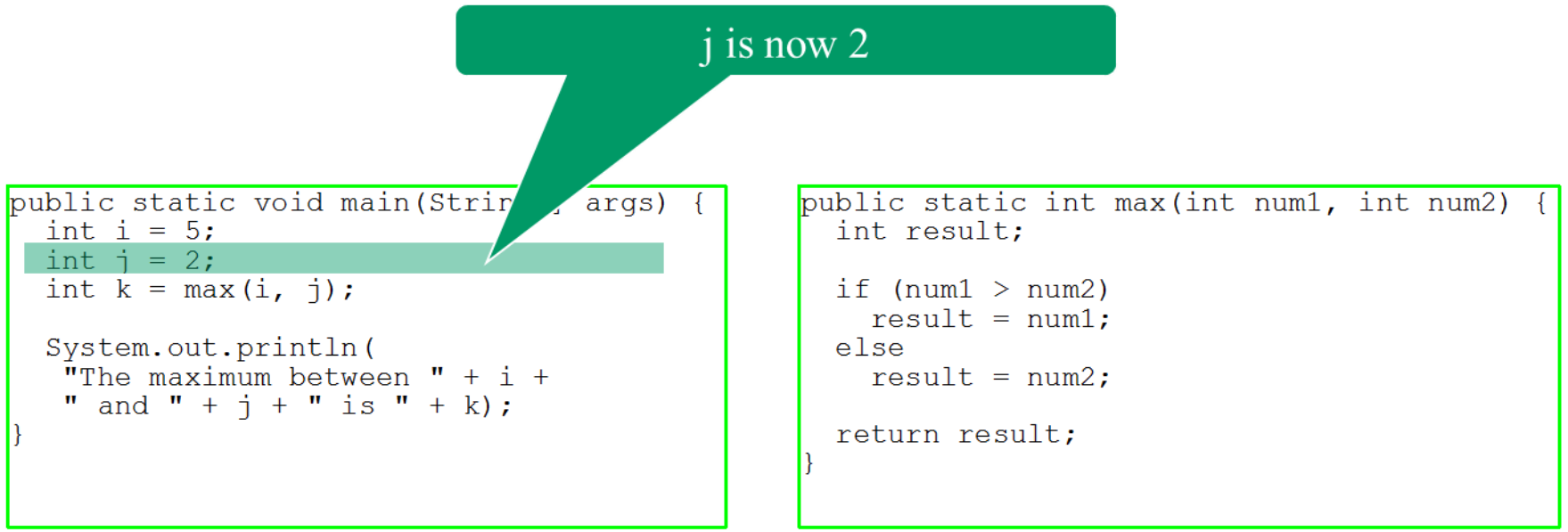
Calling Methods, cont.



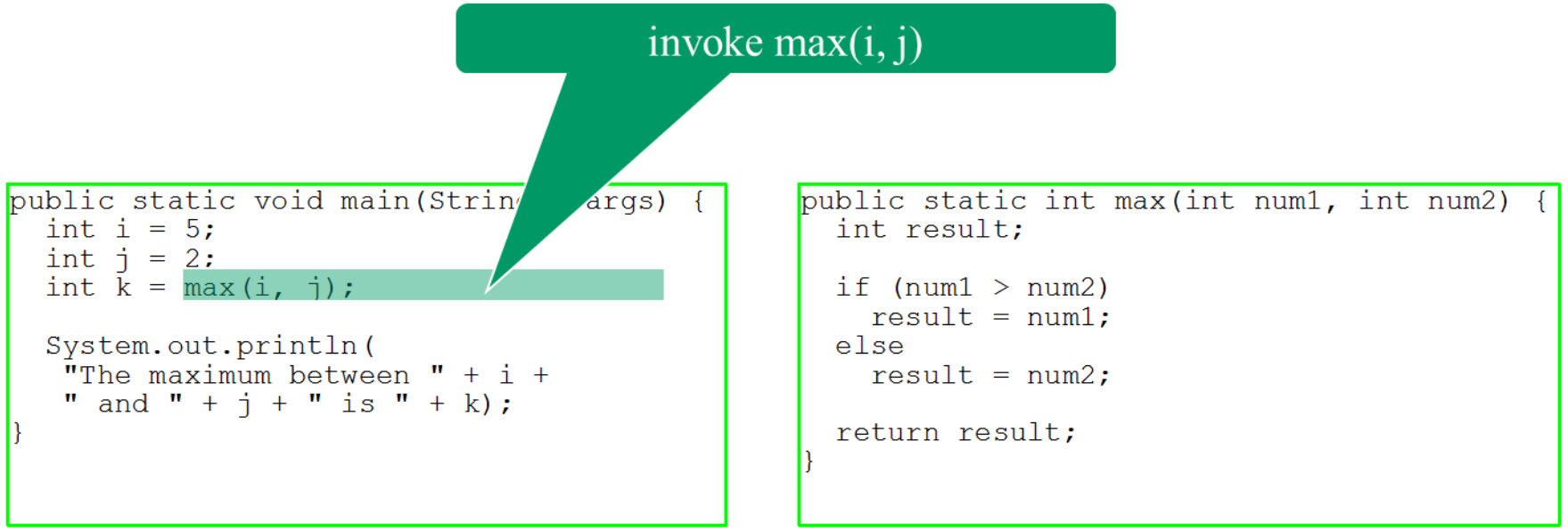
Trace Method Invocation



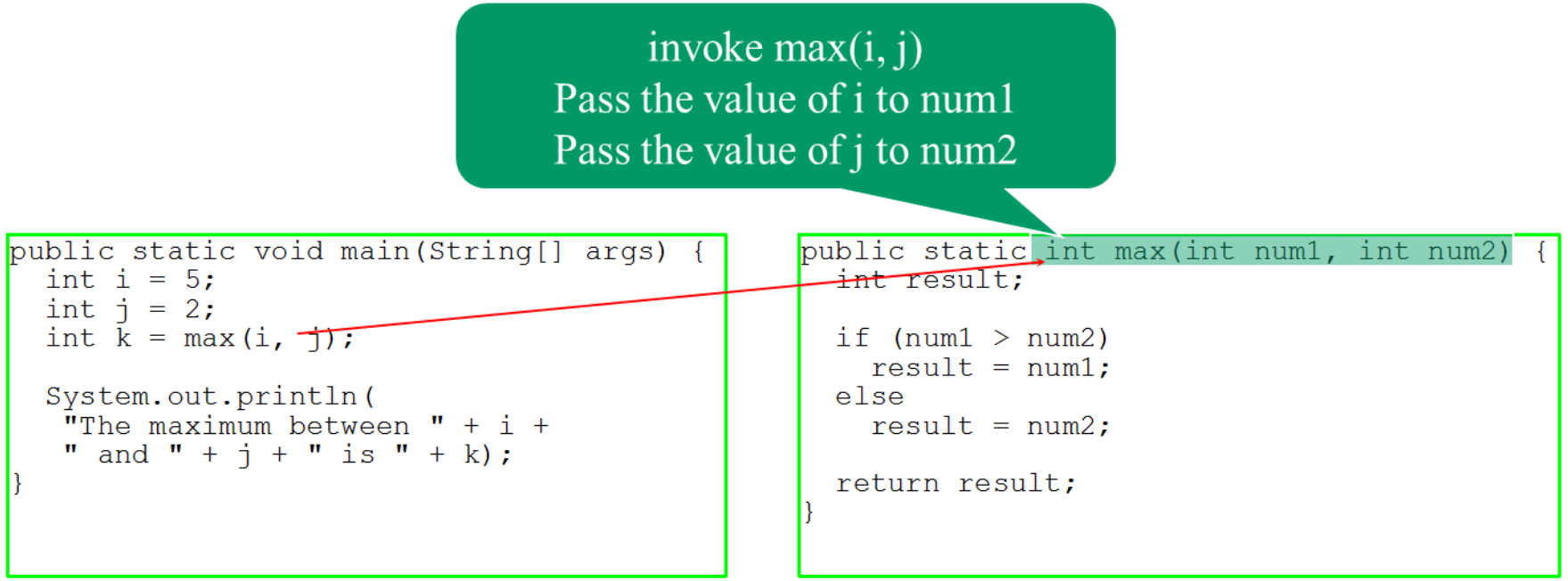
Trace Method Invocation



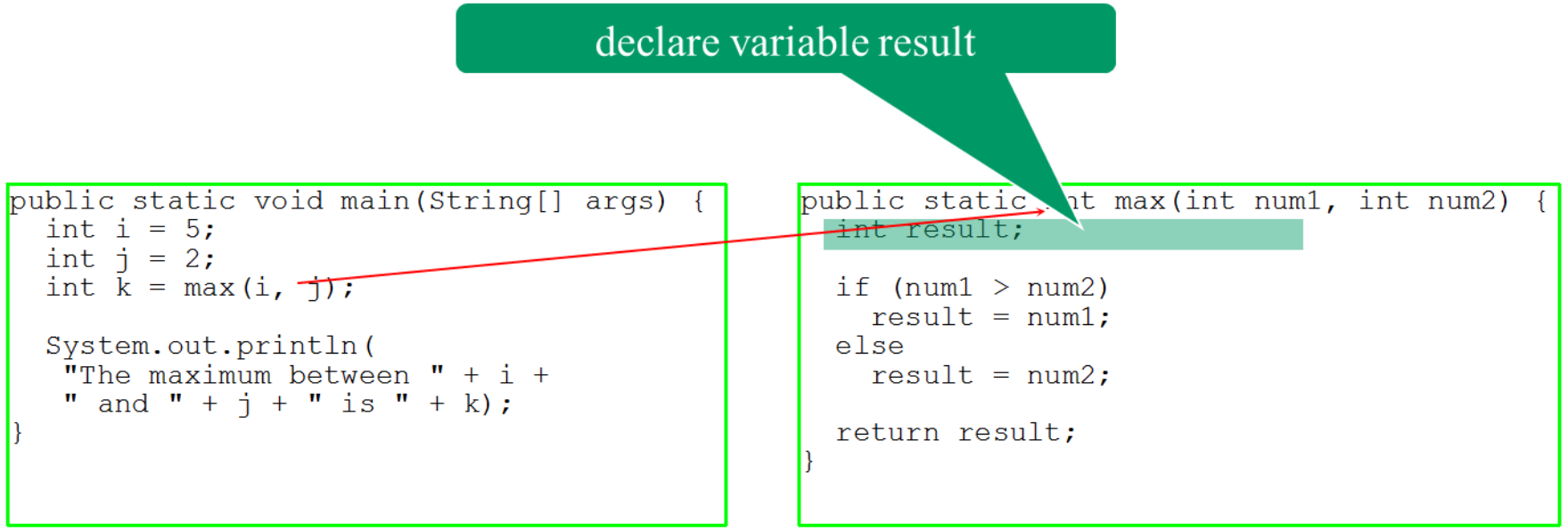
Trace Method Invocation



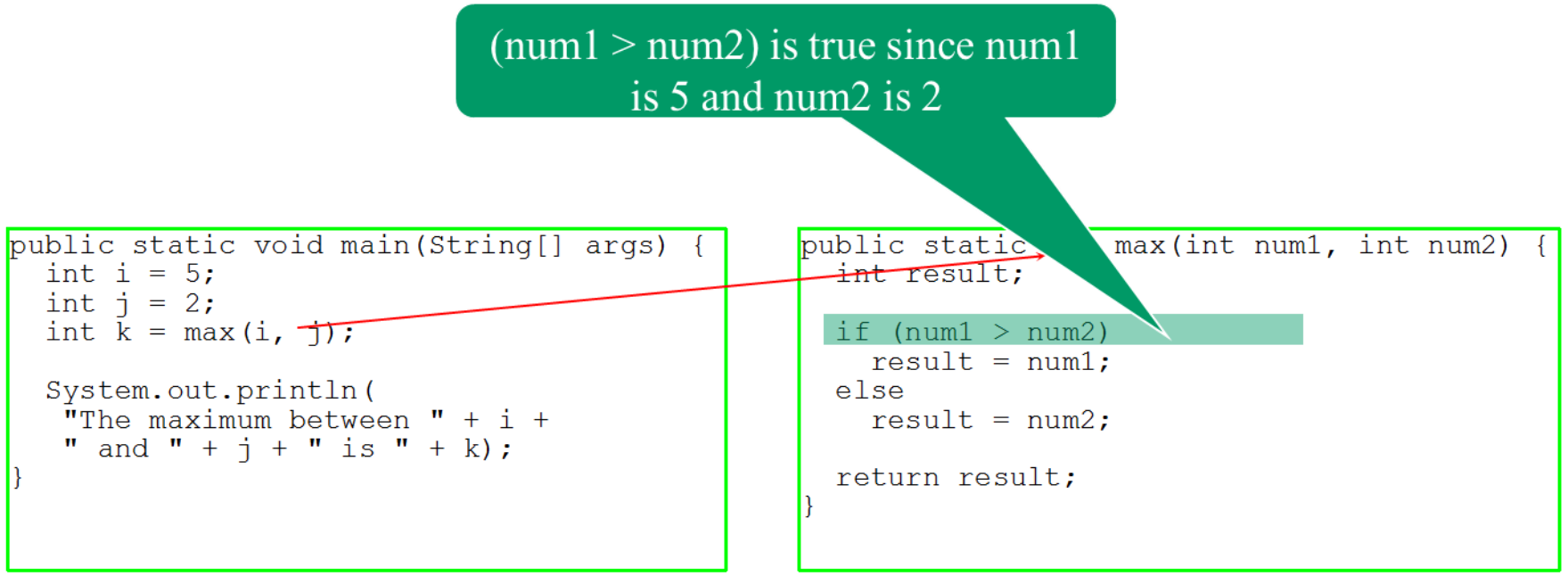
Trace Method Invocation



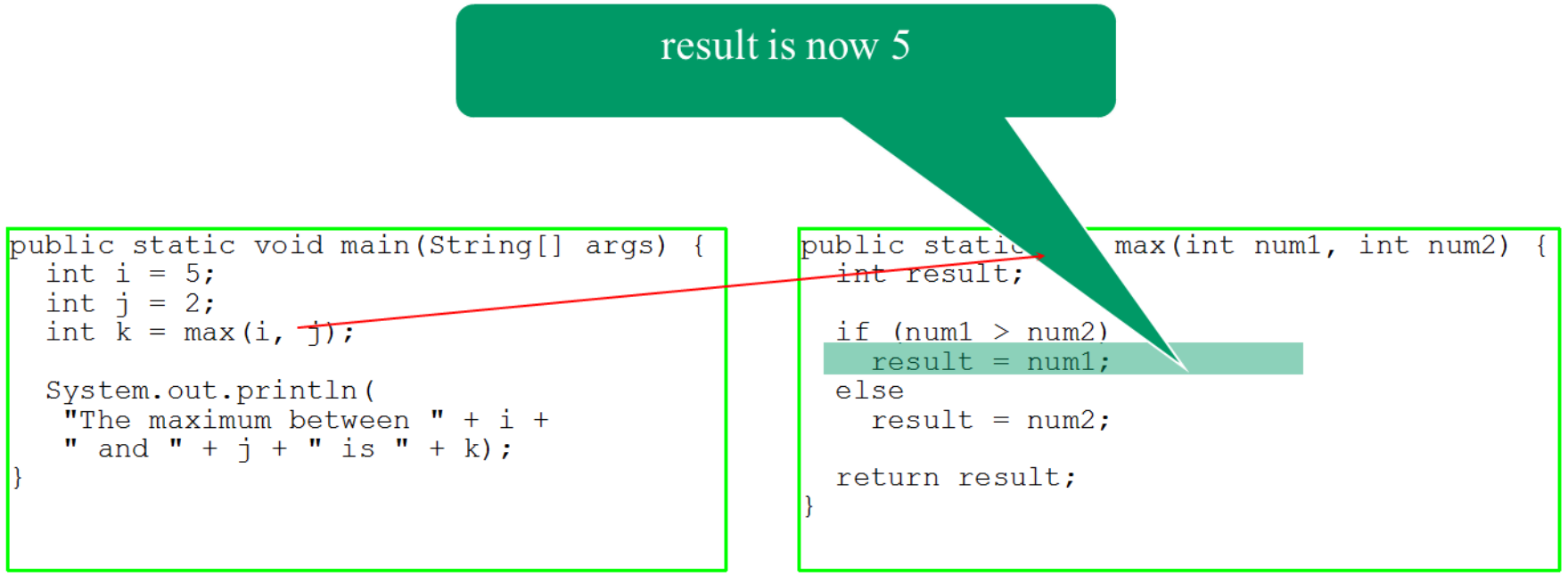
Trace Method Invocation



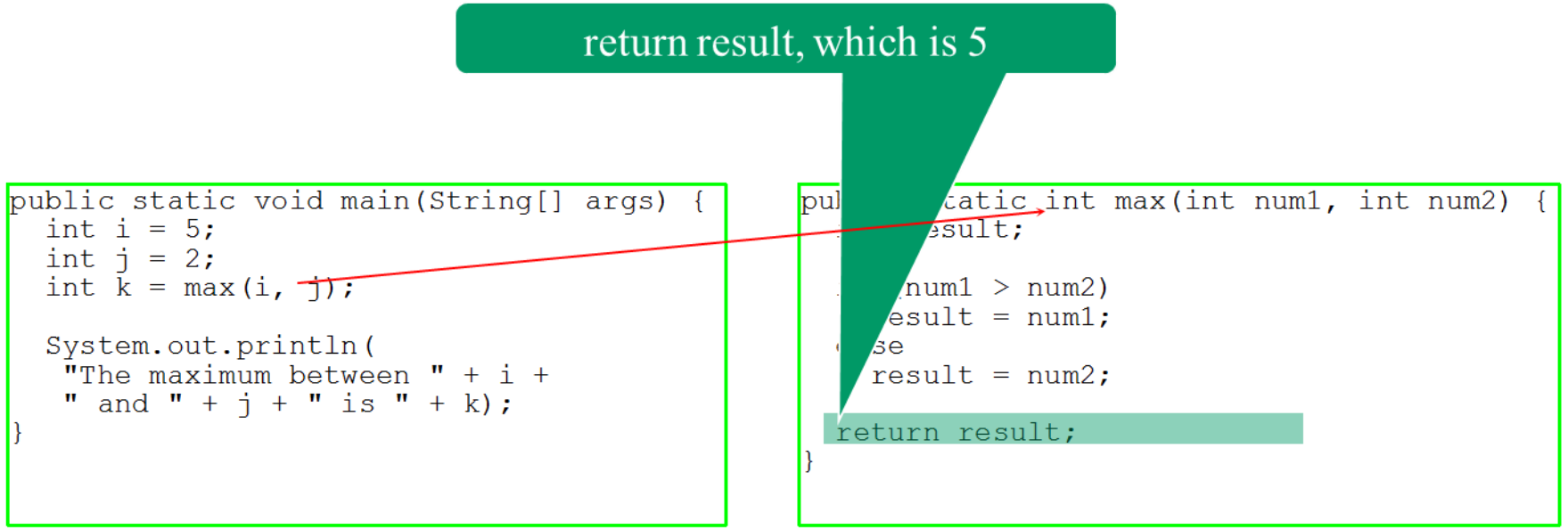
Trace Method Invocation



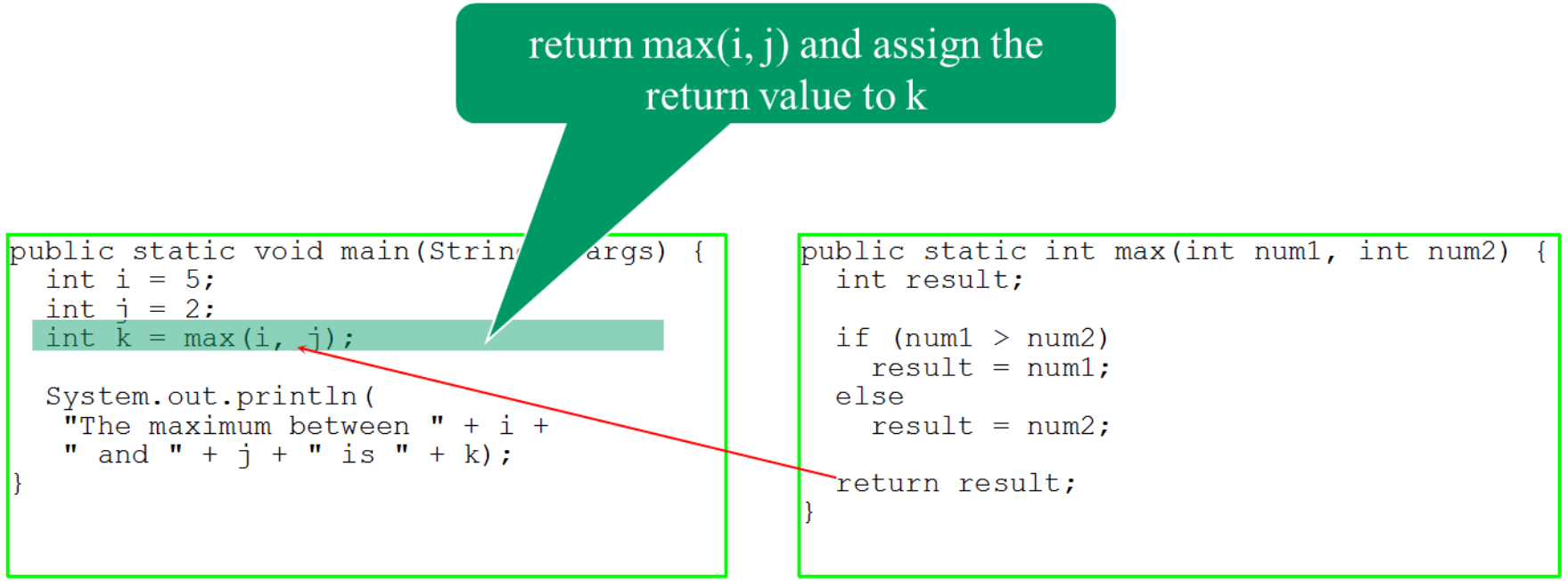
Trace Method Invocation



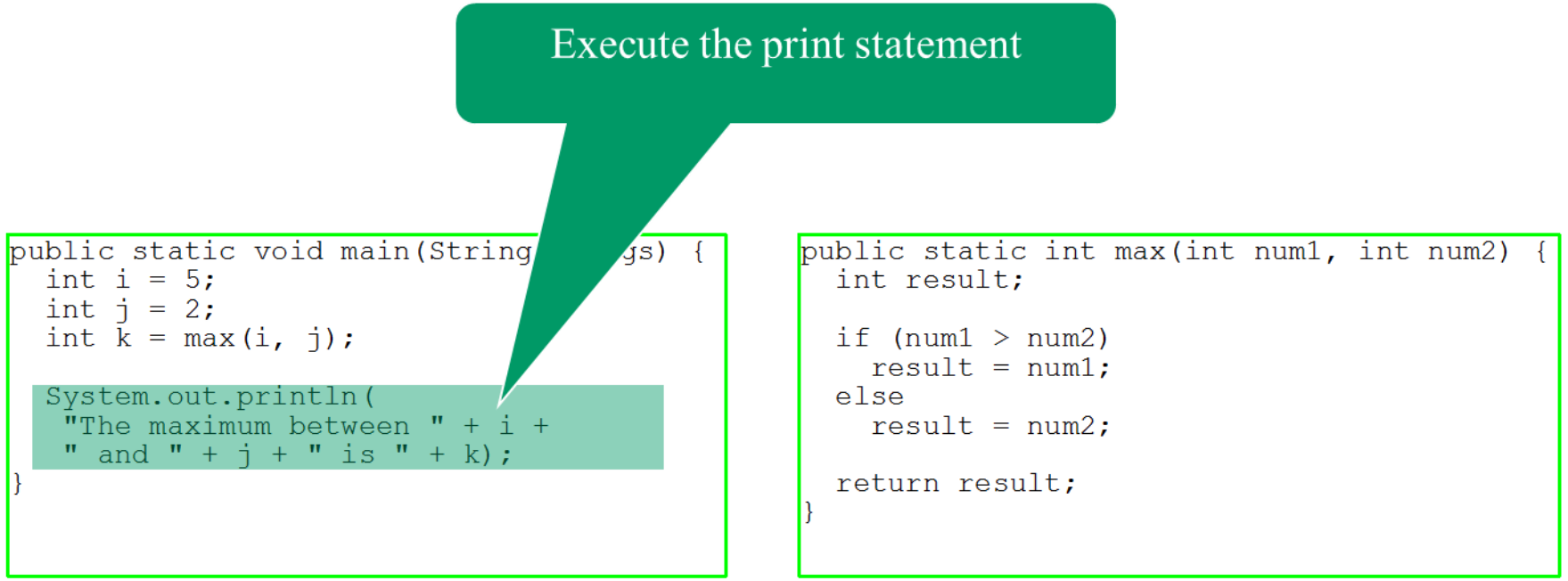
Trace Method Invocation



Trace Method Invocation



Trace Method Invocation



CAUTION

- A return statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

CAUTION

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else if (n < 0)
    return -1;
}
```

(a)

Should be

```
public static int sign(int n) {
  if (n > 0)
    return 1;
  else if (n == 0)
    return 0;
  else
    return -1;
}
```

(b)

To fix this problem, delete *if (n < 0)* in (a), so that the compiler will see a return statement to be reached regardless of how the if statement is evaluated.

Reuse Methods from Other Classes

NOTE: One of the benefits of methods is for reuse. The max method can be invoked from any class besides TestMax. If you create a new class Test, you can invoke the max method using ClassName.methodName (e.g., TestMax.max).

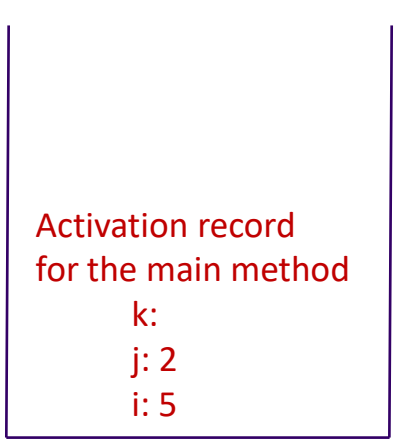
Activation Record

LECTURE 4

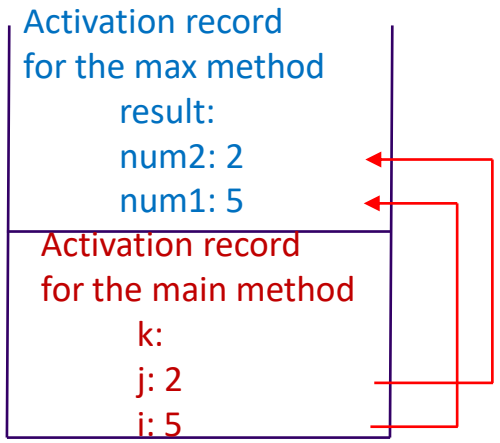
Activation Record

- Each time a method is invoked, the system creates an activation record (also called an **activation frame**) that stores parameters and variables for the method and places the activation record in an area of memory known as a call stack.
- A **call stack** is also known as *an execution stack, runtime*, or *machine stack*, and it is often shortened to just "*the stack*."
- A call stack stores the activation records in a last-in, first-out fashion: The activation record for the method that is invoked last is removed first from the stack.

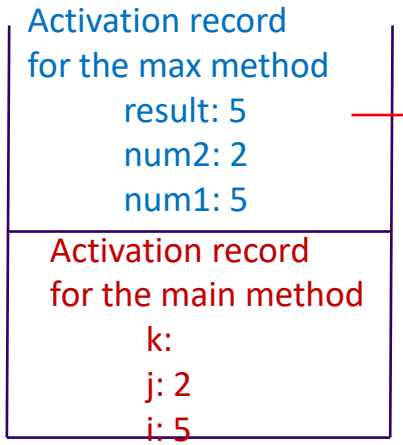
Activation Record



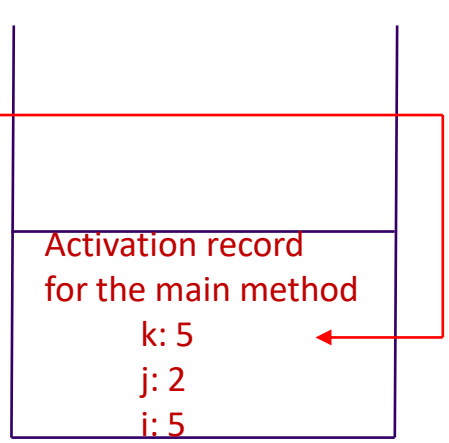
(a) The main method is invoked.



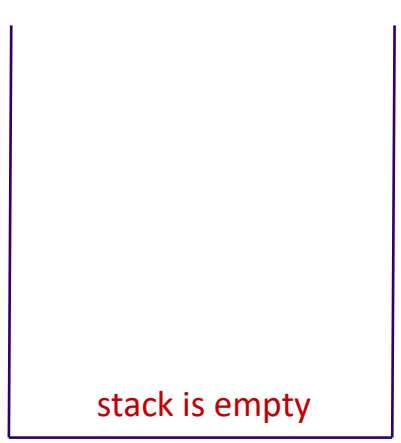
(b) The max method is invoked.



(c) The max method is being executed.



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



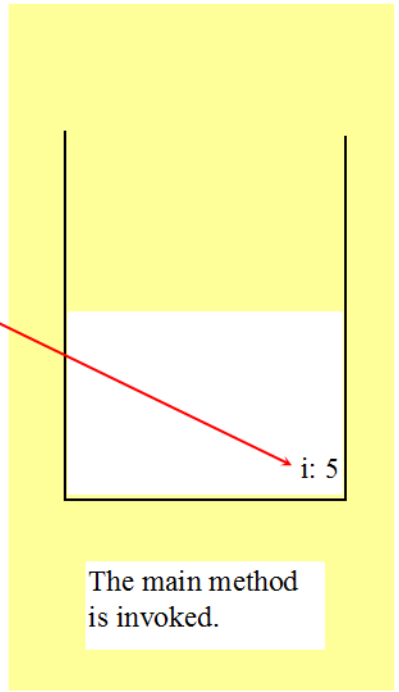
(e) The main method is finished.

Trace Call Stack

i is declared and initialized

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

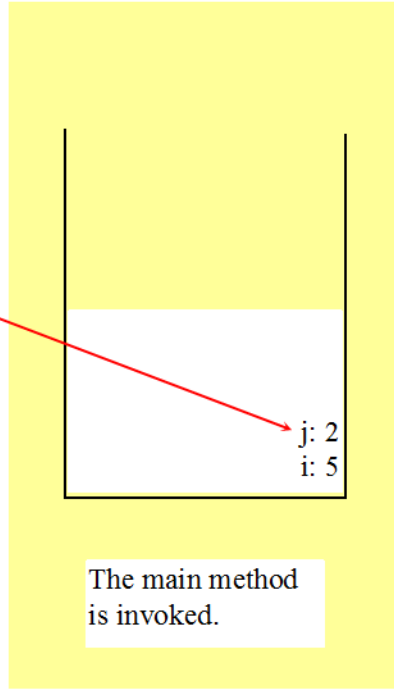


Trace Call Stack

j is declared and initialized

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

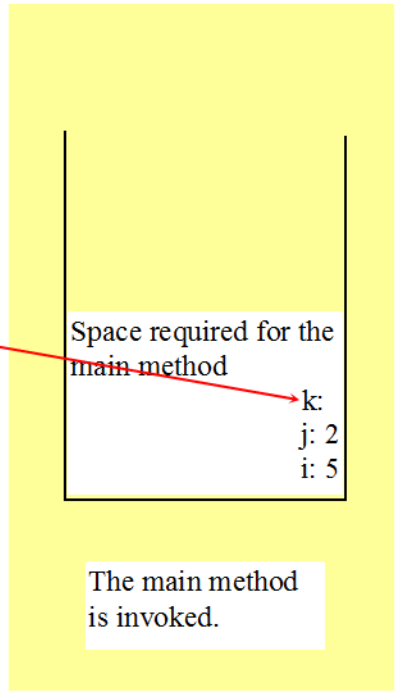


Trace Call Stack

Declare k

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

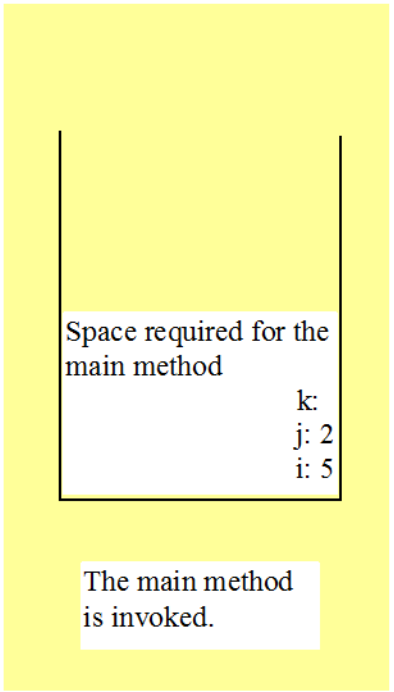


Trace Call Stack

Invoke max(i, j)

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

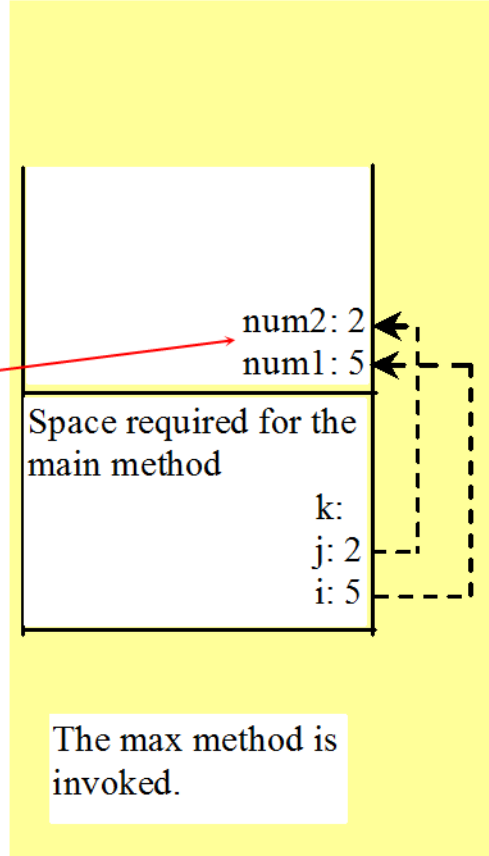


Trace Call Stack

pass the values of i and j to num1 and num2

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

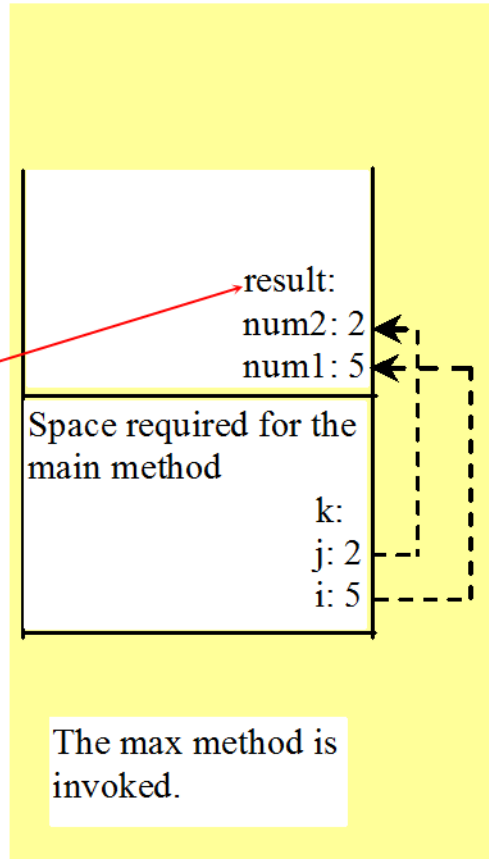


Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

Declare result

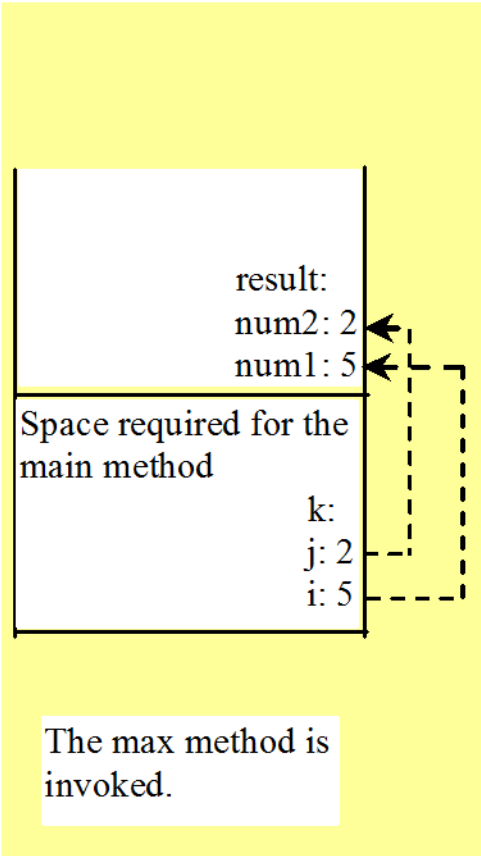


Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

(num1 > num2) is true

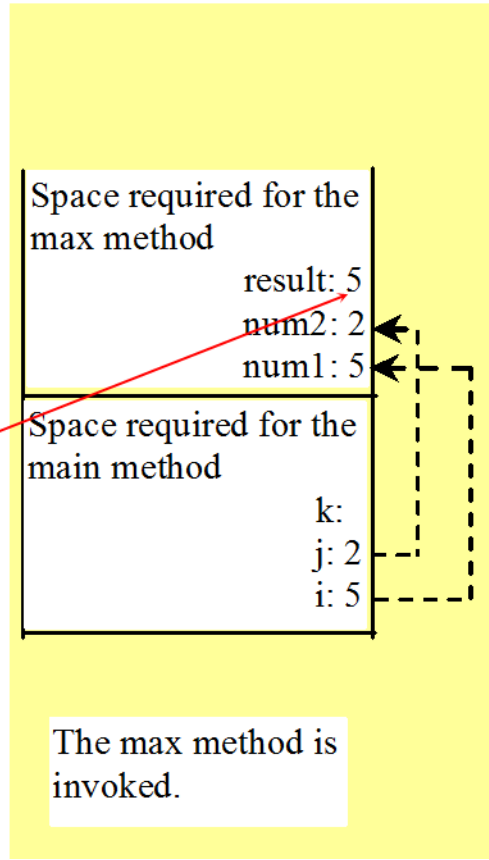


Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2)
  int result;

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

  return result;
}
```

Assign num1 to result

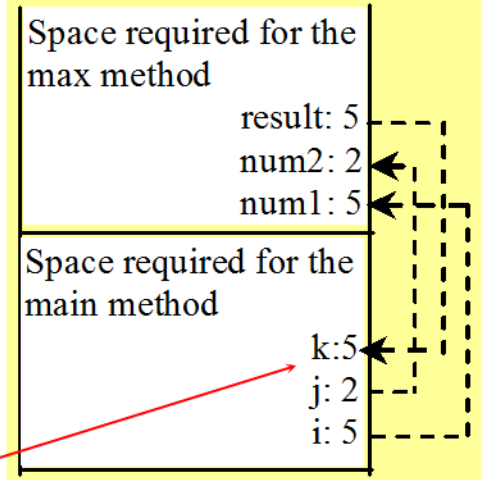


Trace Call Stack

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2
  int result;

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

  return result;
}
```

Return result and assign it to k

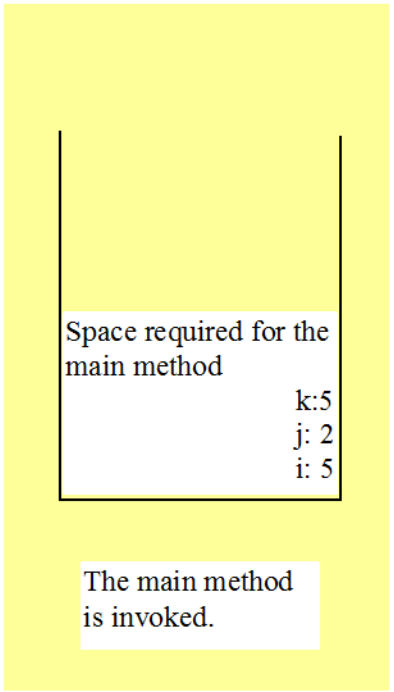


Trace Call Stack

Execute print statement

```
public static void main(String[] args) {
  int i = 5;
  int j = 2;
  int k = max(i, j);

  System.out.println(
   "The maximum between " + i +
   " and " + j + " is " + k);
}
```

```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```



Void Method

LECTURE 5

void Method Example

methods do not pass value

- This type of method does not return a value. The method performs some actions.
- Sometimes, this kind of method is also called as **procedure**.

(In **Pascal** language, **procedure** and **function** are different.)

Statistics Methods

LECTURE 6

Demonstration Program

STATS01.JAVA

Demo Program: Stats01.java

- The simplest user-defined statistics methods for only two inputs.
- As we move on to the next chapters, more robust version will roll out.

Summary:

1. As the parameter list grows, the maximum and minimum methods become very unsystematic.

2. When the length of the parameter list grows, we need to rewrite the method definition.

3. Some data structure may help. (See next Chapter for another version) **Use loop and array combination.**

Call by Value

LECTURE 7

Passing Parameters

- Suppose you invoke the method using nPrintln("Welcome to Java", 5);
- What is the output?
- Suppose you invoke the method using nPrintln("Computer Science", 15);
- What is the output?
- Can you invoke the method using nPrintln(15, "Computer Science");

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

Pass by Value

This program demonstrates passing values to the methods.

Increase.java

Demonstration Program

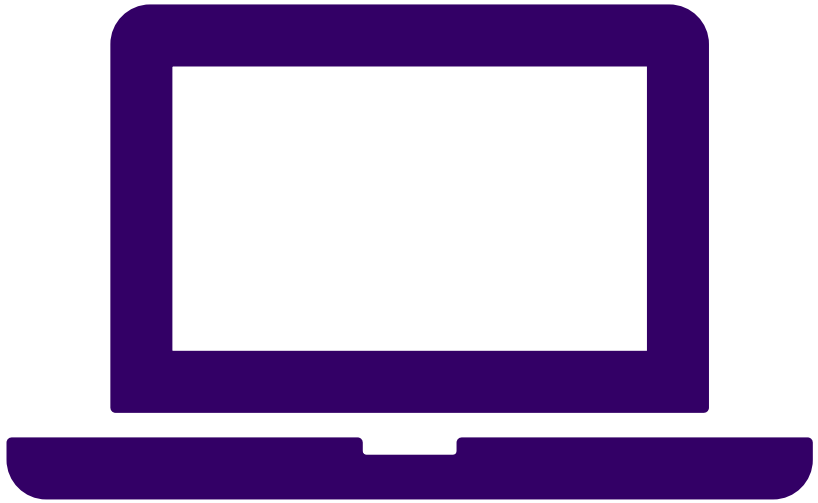
INCREASE.JAVA

Pass by Value

Testing Pass by value

This program demonstrates passing values to the methods.

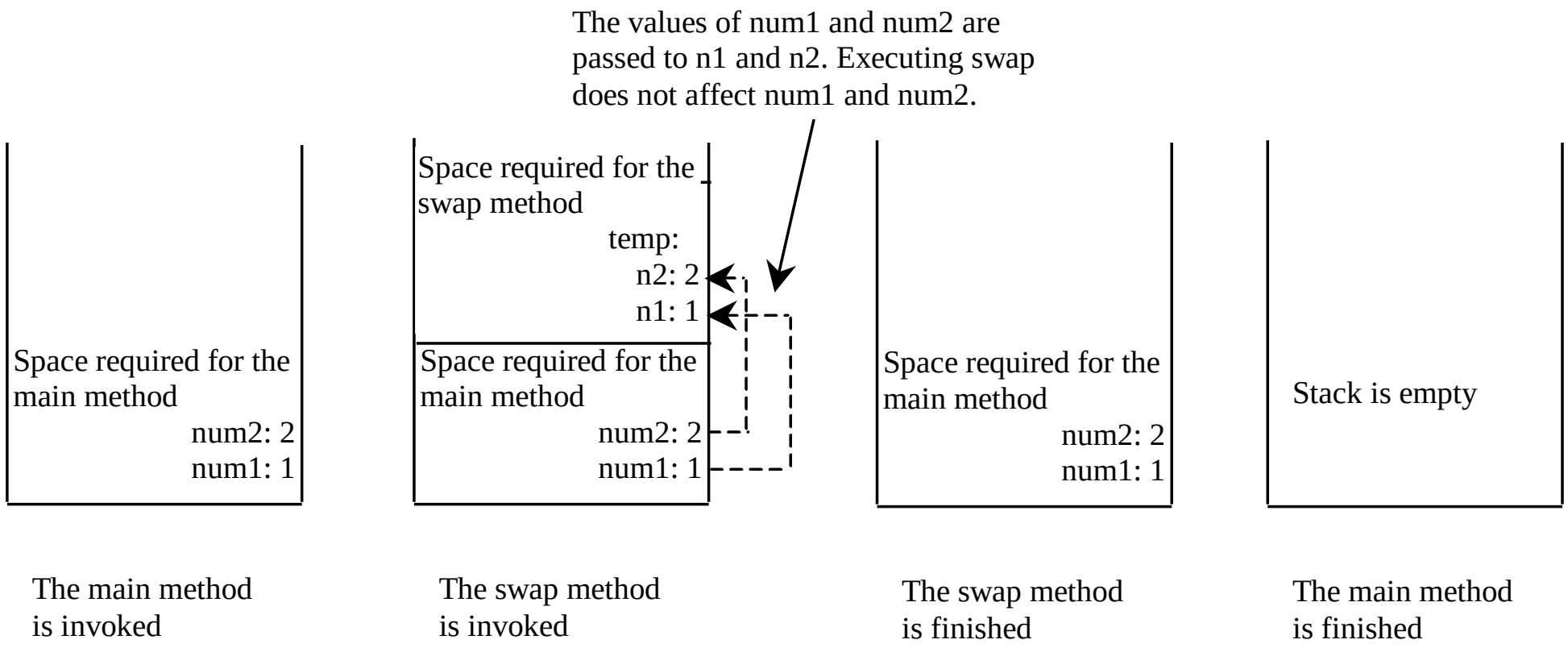
TestPassByValue.java



Demonstration Program

TESTPASSBYVALUE.JAVA

Pass by Value, cont.

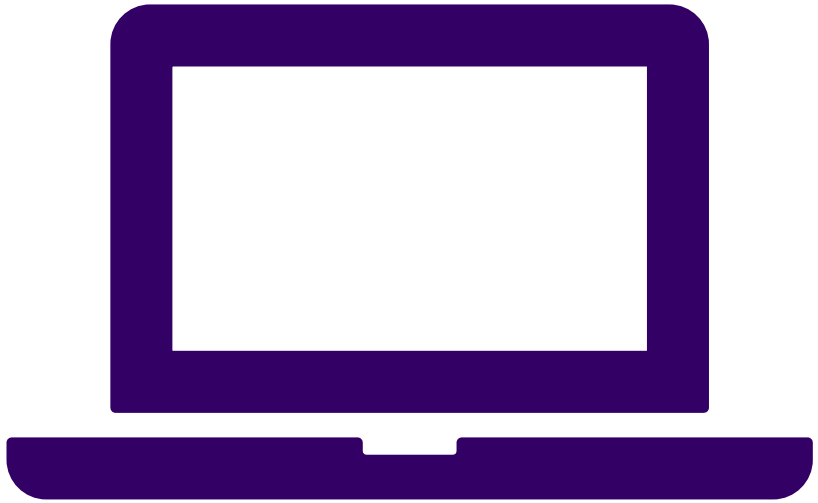


Four types of variables
(declaration location, modifiers)

1. Variables in a class (object) (alive with the class or instance (object))
2. Variable in a method. (alive only with the method)
3. Variable in the parameter list. (alive only with the method)
4. Variable in a loop. (alive only with the loop)

Java is pass-by-value only. Even reference data type is passing reference VALUE

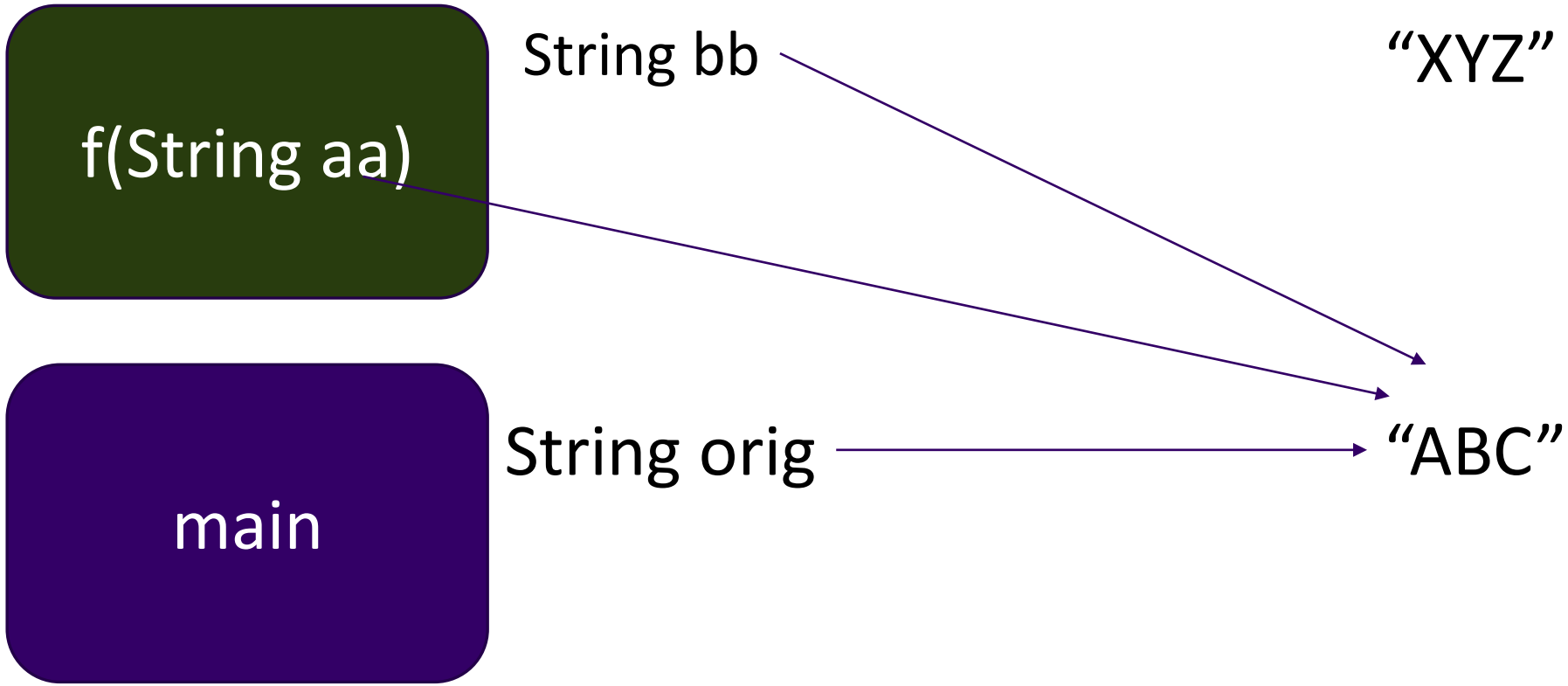
String is passing the reference value (still pass-by-value, in this case the address of string body)



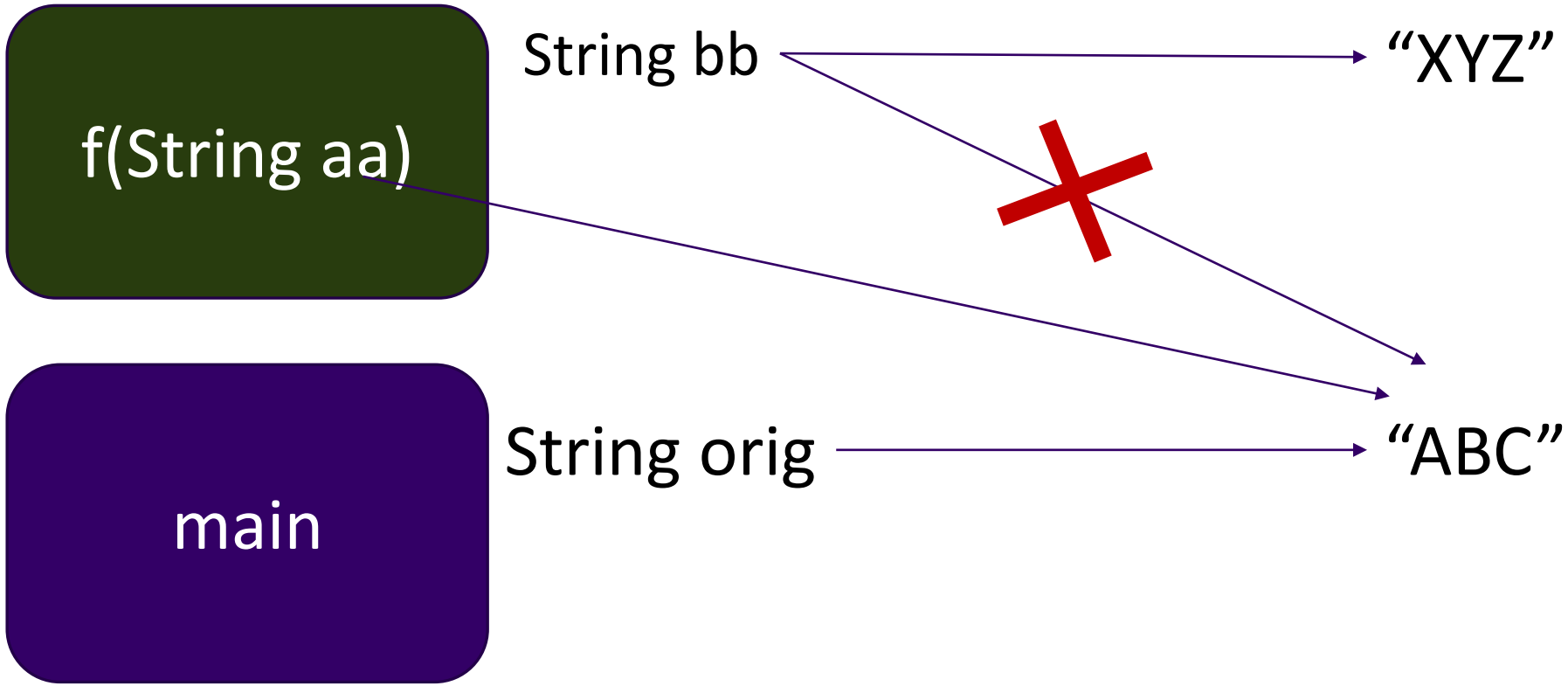
Demonstration Program

TESTPASSSTRING.JAVA

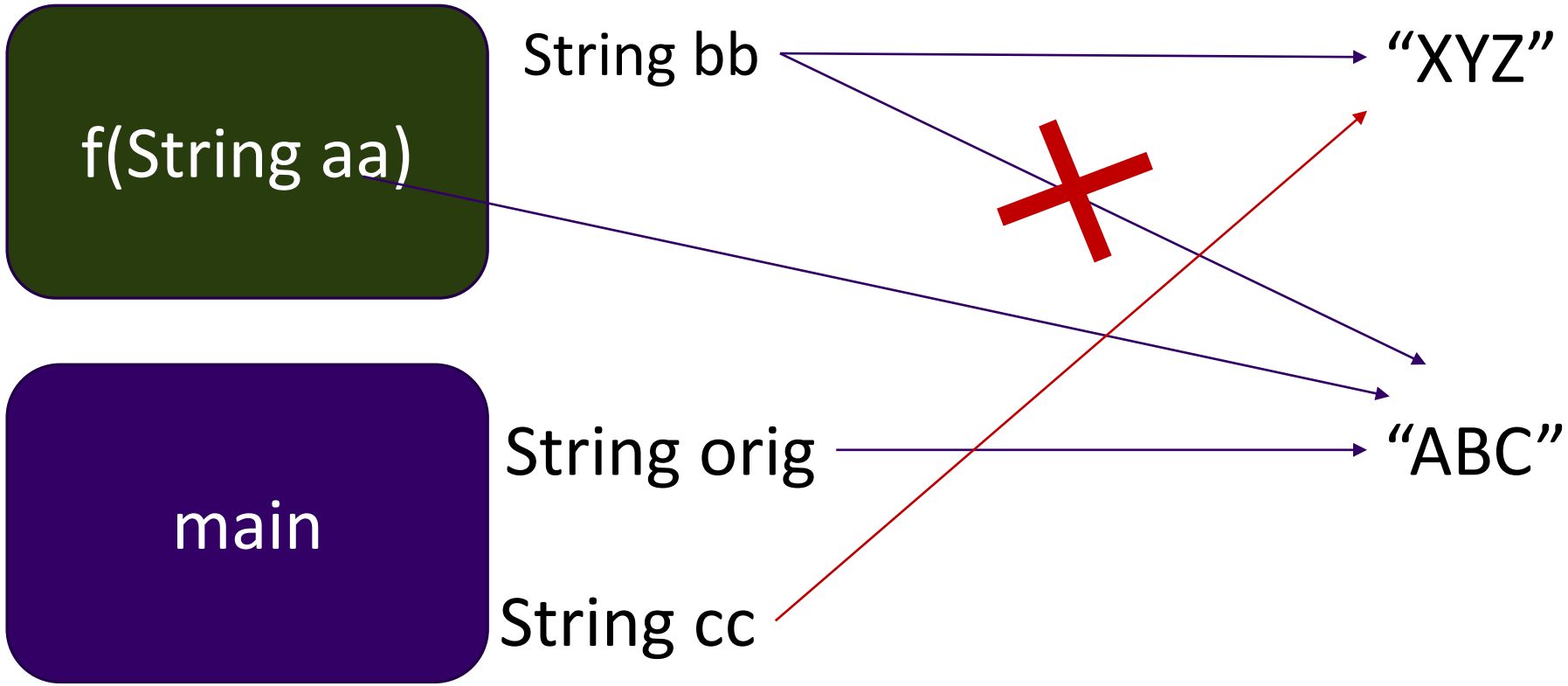
Variable value



Variable value



Variable value

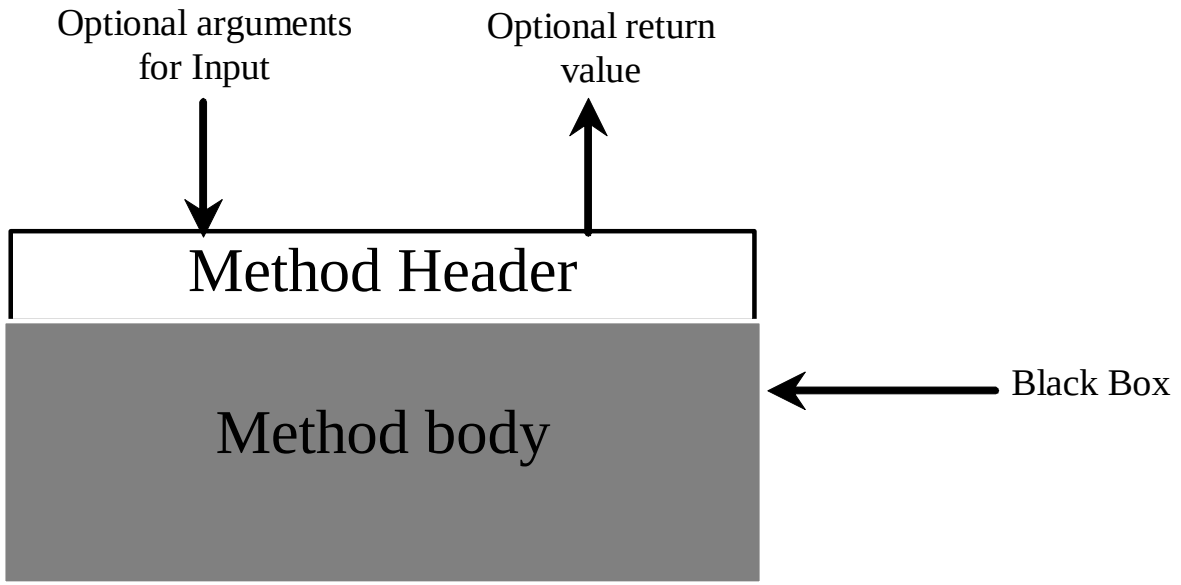


Modularizing Code

LECTURE 8

Method Abstraction

You can think of the method body as a black box that contains the detailed implementation for the method.



Benefits of Methods

Write a method once and reuse it anywhere.

Information hiding. Hide the implementation from the user.

- Reduce complexity
- Information hiding
- Abstraction
- Encapsulation

Modularizing Code

Methods can be used to reduce redundant coding and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

[1] GreatestCommonDivisorMethod.java

run from 1 to min(num1, num2) to check if it is GCD

[2] PrimeNumberMethod.java

run from 2 to n/2 to check if a number is prime (sqrt(n) will be enough.)

Converting Hexadecimal to Decimal

How do you convert a hexadecimal number to a decimal number? To convert a hexadecimal number to a decimal a is to find the decimal value from $h_n, h_{n-1}, h_{n-2}, \ldots, h_2, h_1, h_0 \rightarrow d$

$$d = h_n \times 16^n + h_{n-1} \times 16^{n-1} + \cdots + h_2 \times 16^2 + h_1 \times 16^1 + h_0 \times 16^0$$

$$= (((h_n \times 16^1 + h_{n-1}) \times 16^1 + \cdots + h_2) \times 16^1 + h_1) \times 16^1 + h_0 \times 16^0$$

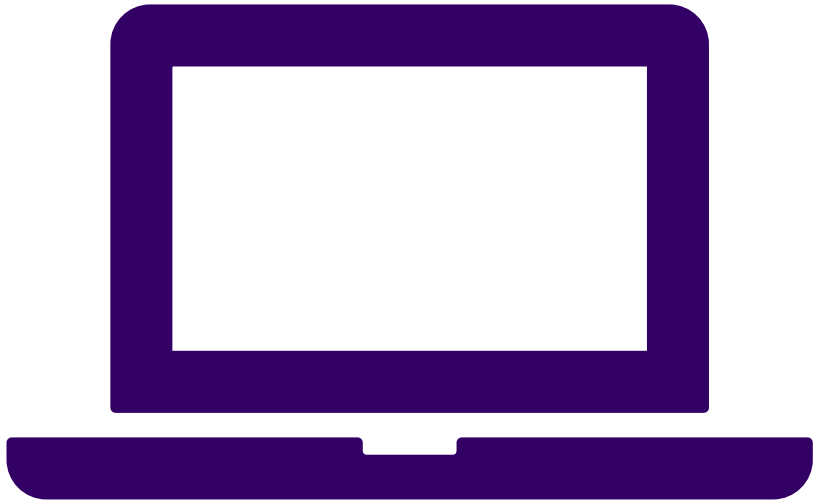
This called Horner's Algorithm.

```
int decimalValue = 0;
for (int I = 0; I < hex.length(); i++) {
   char hexChar = hex.charAt(i);
   decimalValue = decimalValue * 16 + hexCharToDecimal(hexChar);
}
```

Problem: Converting Hexadecimals to Decimals

Write a method that converts a hexadecimal integer to a decimal.

[3] Hex2Dec.java



Demonstration Program

[1] GREATESTCOMMONDIVISORMETHOD.JAVA

[2] PRIMENUMBERMETHOD.JAVA

[3] HEX2DEC.JAVA

Modularize

RandomCharacter()

LECTURE 9

Static Methods

- Java static method program: static methods in Java can be called **without creating an object of class**.
- Have you noticed why we write static keyword when defining main it's because program execution begins from main and no object has been created yet.
- Consider the example below to improve your understanding of static methods.

Static Variables

There are several kinds of variables:

Member variables in a class—these are called fields.

Variables in a method or block of code—these are called local variables (method level).

Variables in a block of code – these are also local variables (block level).

Variables in method declarations—these are called parameters.

Static variables in a class is a **class variable** which does not need to declare any instance to access. The value of the static variable will also live as long as the class is **loaded**. No instantiation needed.

Static Methods and Static Variables

```
class Languages {
  static String message = "Java is my favorite programming language.";
  public static void main(String[] args) {
    display();
  }

  static void display() {
    System.out.println(message);
  }
}
```

Use of Math class's staic variables:

Math.PI;

Math.E;

Java static method vs instance method

Instance method requires an object of its class to be created before it can be called while static method doesn't require object creation.

```
class Difference {
  public static void main(String[] args) {
    display();  //calling without object
    Difference t = new Difference();
    t.show();  //calling using object
  }
  static void display() {
    System.out.println("Programming is
amazing.");
  }
  void show(){
    System.out.println("Java is awesome.");
  }
}
```

Using static method of another classes

- If you wish to call static method of another class then you have to write class name while calling static method as shown in example below.

```
import java.lang.Math;
class Another {
 public static void main(String[] args) {
    int result;
    result = Math.min(10, 20);
    //calling static method min by writing class name
    System.out.println(result);
    System.out.println(Math.max(100, 200));
 }
}
```

Output of program:

```
10
200
```

Static Methods can only Access Static Methods and Static Variables.

Because they are class variables and class method, they do **not** belong to any instance. So, a instance won't be able to access them. Non-static methods are not class methods and are not supposed to access them as well.

The random Method

Generates a random double value greater than or equal to 0.0 and less than 1.0 (0 <= Math.random() < 1.0).

Examples:

```
(int)(Math.random() * 10)
```
→ Returns a random integer between 0 and 9.

```
50 + (int)(Math.random() * 50)
```
→ Returns a random integer between 50 and 99.

In general,

```
a + Math.random() * b
```
→ Returns a random number between a and a + b, excluding a + b.

Case Study: Generating Random Characters

- Computer programs process numerical data and characters. You have seen many examples that involve numerical data. It is also important to understand characters and how to process them.
- Each character has a unique Unicode between 0 and FFFF in hexadecimal (65535 in decimal). To generate a random character is to generate a random integer between 0 and 65535 using the following expression: (note that since 0 <= Math.random() < 1.0, you have to add 1 to 65535.)

(int)(Math.random() * (65535 + 1))

Case Study: Generating Random Characters

- Now let us consider how to generate a random lowercase letter. The Unicode for lowercase letters are consecutive integers starting from the Unicode for 'a', then for 'b', 'c', ..., and 'z'. The Unicode for 'a' is

```
(int) 'a'
```

- So, a random integer between (int)'a' and (int)'z' is

```
(int)(Math.random() * ((int)'z' - (int)'a' + 1) + (int) ’a’)
```

Case Study: Generating Random Characters

- Now let us consider how to generate a random lowercase letter. The Unicode for lowercase letters are consecutive integers starting from the Unicode for 'a', then for 'b', 'c', ..., and 'z'.
- The Unicode for 'a' is (int) 'a'
- So, a random integer between (int)'a' and (int)'z' is

(int)(Math.random() * ((int)'z' - (int)'a' + 1) + 'a')

Case Study: Generating Random Characters

- All numeric operators can be applied to the char operands. The char operand is cast into a number if the other operand is a number or a character. So, the preceding expression can be simplified as follows:

  Math.random() * ('z' - 'a' + 1) + ‘a’

- So a random lowercase letter is

  (char)(Math.random() * ('z' - 'a' + 1) + ‘a’)

Case Study: Generating Random Characters

- To generalize the foregoing discussion, a random character between any two characters ch1 and ch2 with ch1 < ch2 can be generated as follows:

(char)(Math.random() * (ch2 – ch1 + 1) + ch1)

Demonstration Program The RandomCharacter Class

[1] RANDOMCHARACTER.JAVA

[2] TESTRANDOMCHARACTER.JAVA

[3] STRONGPASSWORDMETHOD.JAVA

Overloading of Methods

LECTURE 10

Overloading Methods

Overloading the max Method

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

Demonstration Program Overloading

[1] TESTOVERLOADING.JAVA

Ambiguous Invocation

- Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match.
- This is referred to as **ambiguous invocation**. Ambiguous invocation is a **compilation** error.

Ambiguous Invocation

(Compilation Error: Neither is better. max(int num1, int num2) will win.)

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
  }

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

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

Scope of Variables (Local Variables)

LECTURE 11

Scope of Local Variables (method/block)

A local variable: a variable defined inside a method or block.

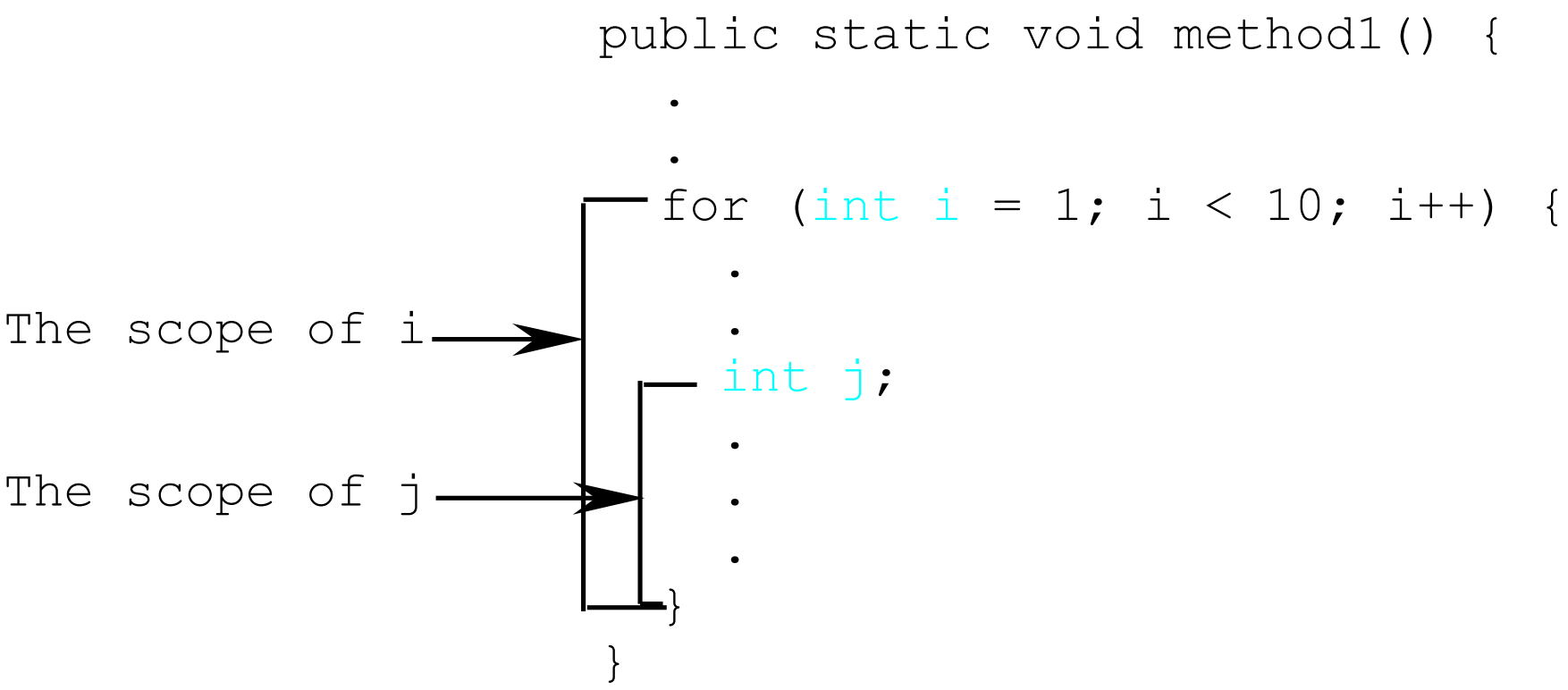
Scope: the part of the program where the variable can be referenced.

- 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 before it can be used.

Scope of Local Variables (method/block)

- You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.
- A variable declared in the initial action part of a for loop header has its scope in the entire loop. But a variable declared inside a for loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.

Scope of Local Variables



Scope of Local Variables

It is fine to declare i in two non-nesting 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 nesting blocks

```
public static void method2() {

  int i = 1;
  int sum = 0;

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

Scope of Local Variables

```
// Fine with no errors
public static void correctMethod() {
  int x = 1;
  int y = 1;
  // i is declared
  for (int i = 1; i < 10; i++) {
    x += i;
  }
  // i is declared again
  for (int i = 1; i < 10; i++) {
    y += i;
  }
}
```

Scope of Local Variables

```
// With errors
public static void incorrectMethod() {
   int x = 1;
   int y = 1;
   for (int i = 1; i < 10; i++) {
      int x = 0;
      x += i;
   }
}
```

Recursion

LECTURE 12

Recursion

- A function which calls itself is named recursive function call.
- In Chapter 17, we will explain recursive function call in details.

Demo Program: Part1

DNA Study and Computing Theory

LECTURE 13

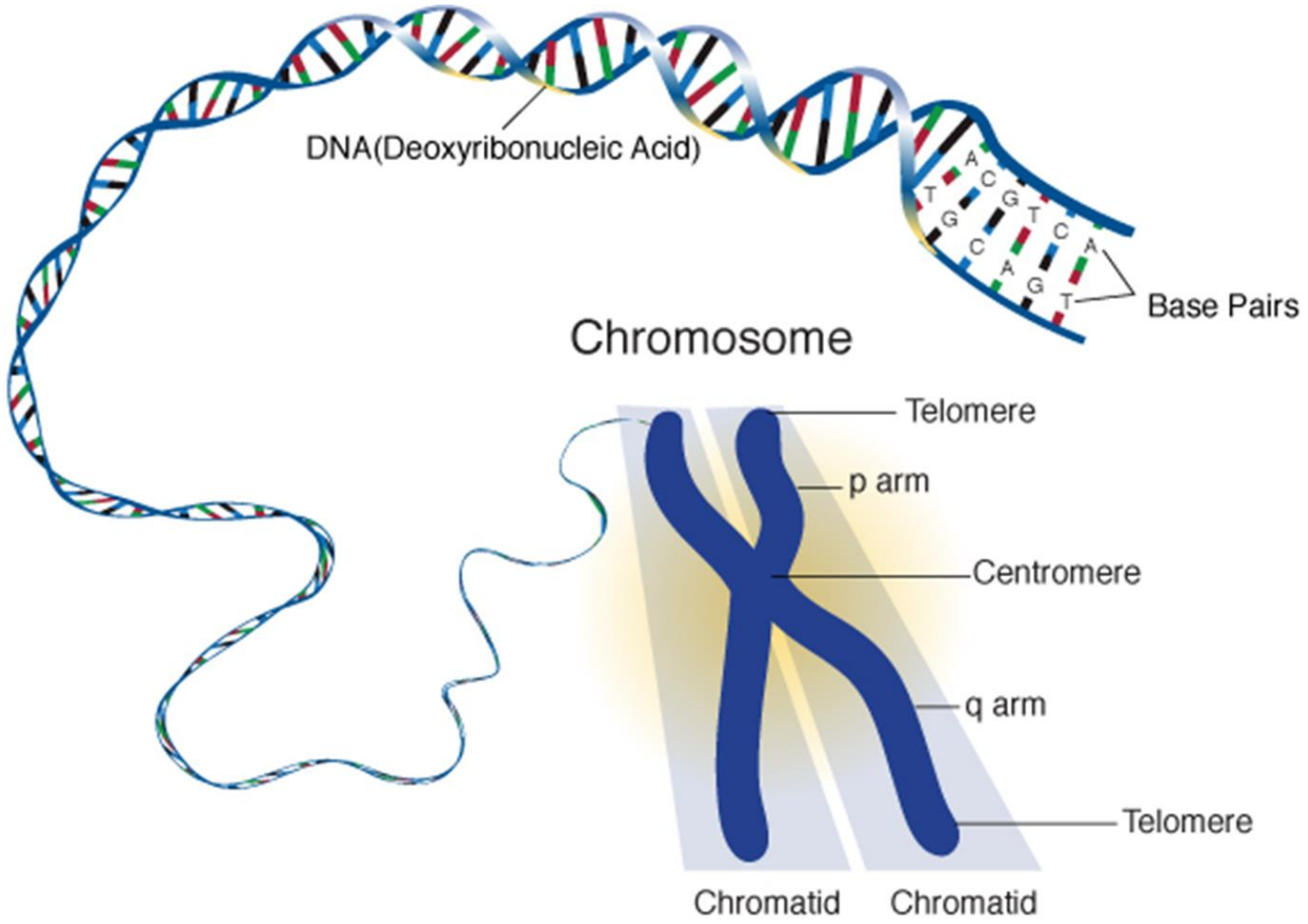
Purpose of this Demo Program:

(1) Get exposure to a good application field for **Computer Science**.

(2) Demonstrate **string** manipulation methods (beyond Java API) for some application field.

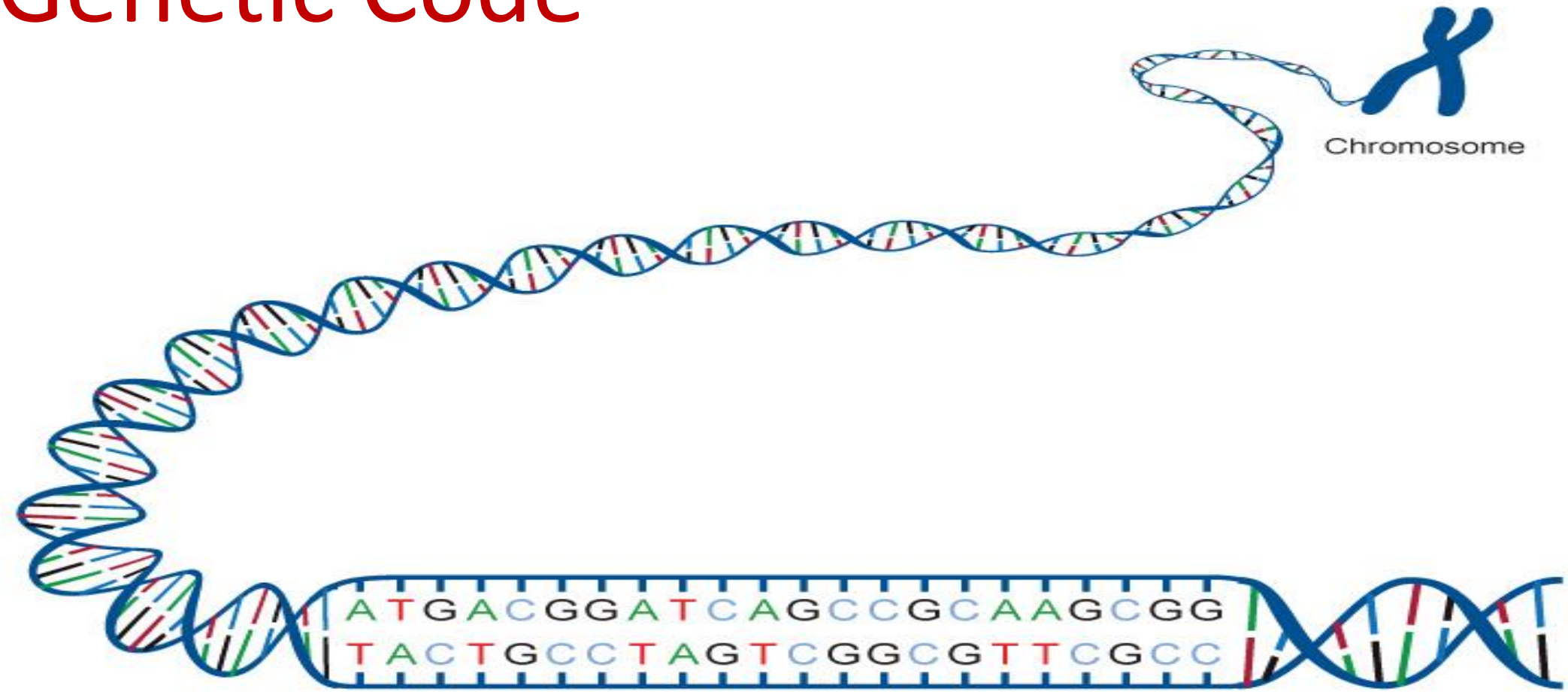
(3) know more about methods and calling methods.

(4) dual I/O using a **log** method.

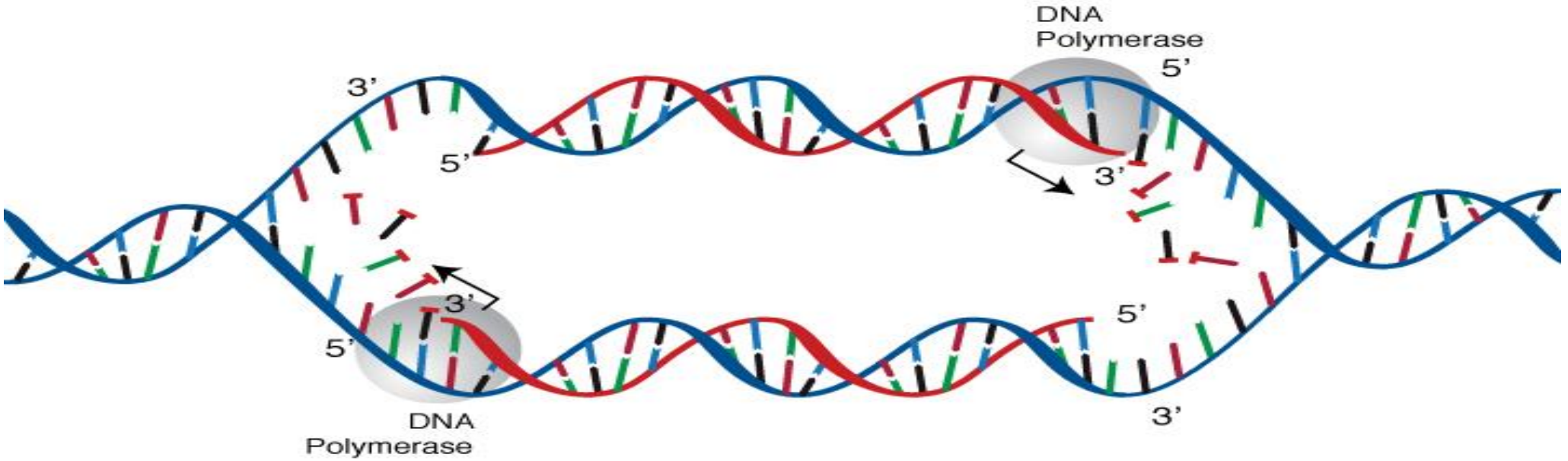


http://geneed.nlm.nih.gov/topic_subtopic.php?tid=15&sid=19

Genetic Code



Replication



DNA replication is the process by which a molecule of DNA is duplicated. When a cell divides, it must first duplicate its genome so that each daughter cell winds up with a complete set of chromosomes.

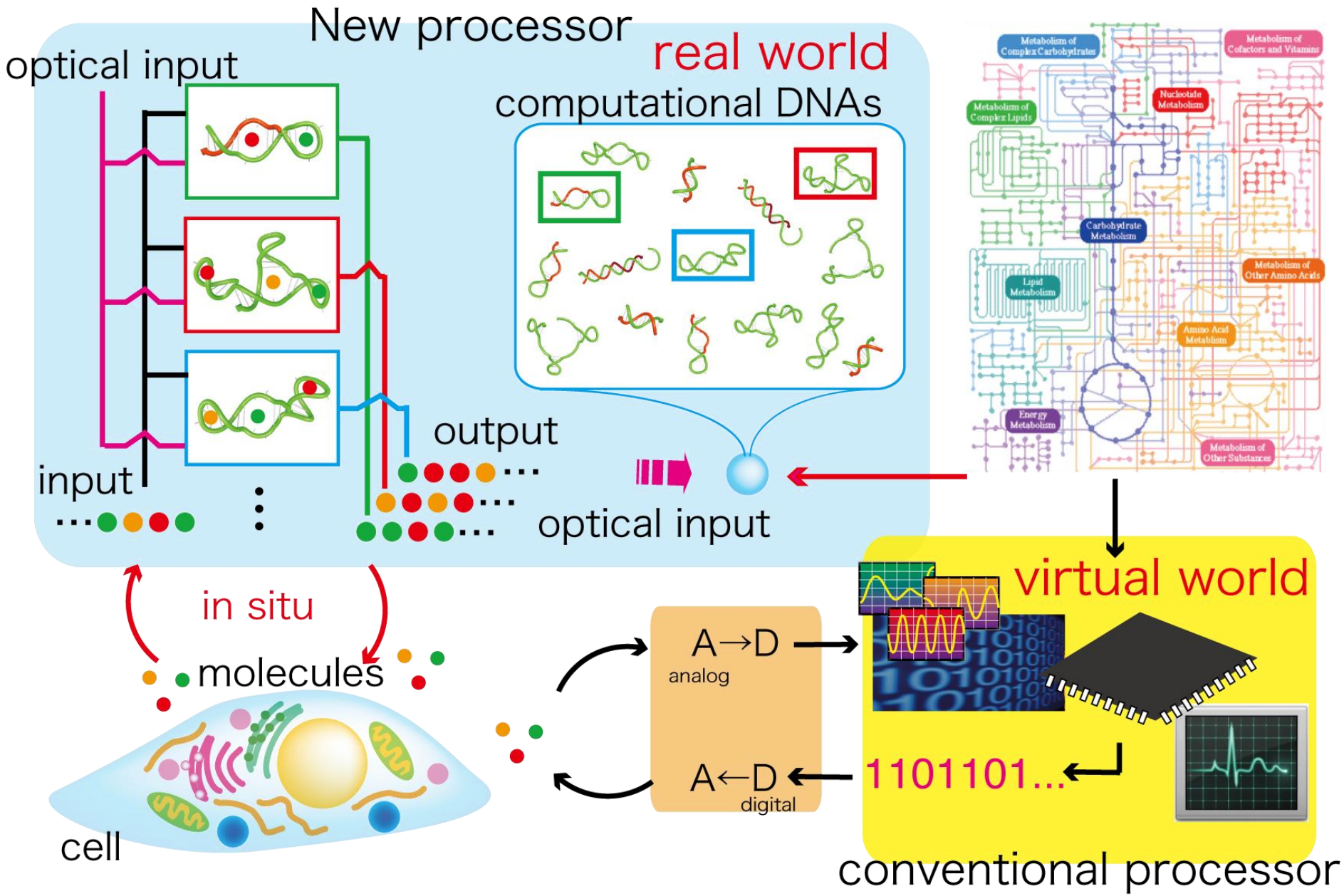
Purpose of Studying DNA

(1) DNA modeling and computer simulation for DNA repair, DNA replication (birth control), DNA therapy, DNA analysis, DNA matching and DNA engineering.

(2) DNA reading (understand all its purpose)

(3) DNA computing. (Photonic Computing, Neural computing, and …)

(Legal issues?)


New processor
real world
optical input
computational DNAs
output
input
optical input
in situ
molecules
cell
A→D
analog
A←D
digital
virtual world
1101101...
conventional processor
Metabolism of Complex Carbohydrates
Metabolism of Cofactors and Vitamins
Metabolism of Complex Lipids
Nucleotide Metabolism
Carbohydrate Metabolism
Lipid Metabolism
Metabolism of Other Amino Acids
Amino Acid Metabolism
Energy Metabolism
Metabolism of Other Substances

Advantage of DNA Computing

Massively parallel computing.

Low Cost. (Cost of protein)

Fast computing speed. (due to randomness)

(1 sec DNA computing can be equivalent to 1 year simulation time)

Disadvantage:

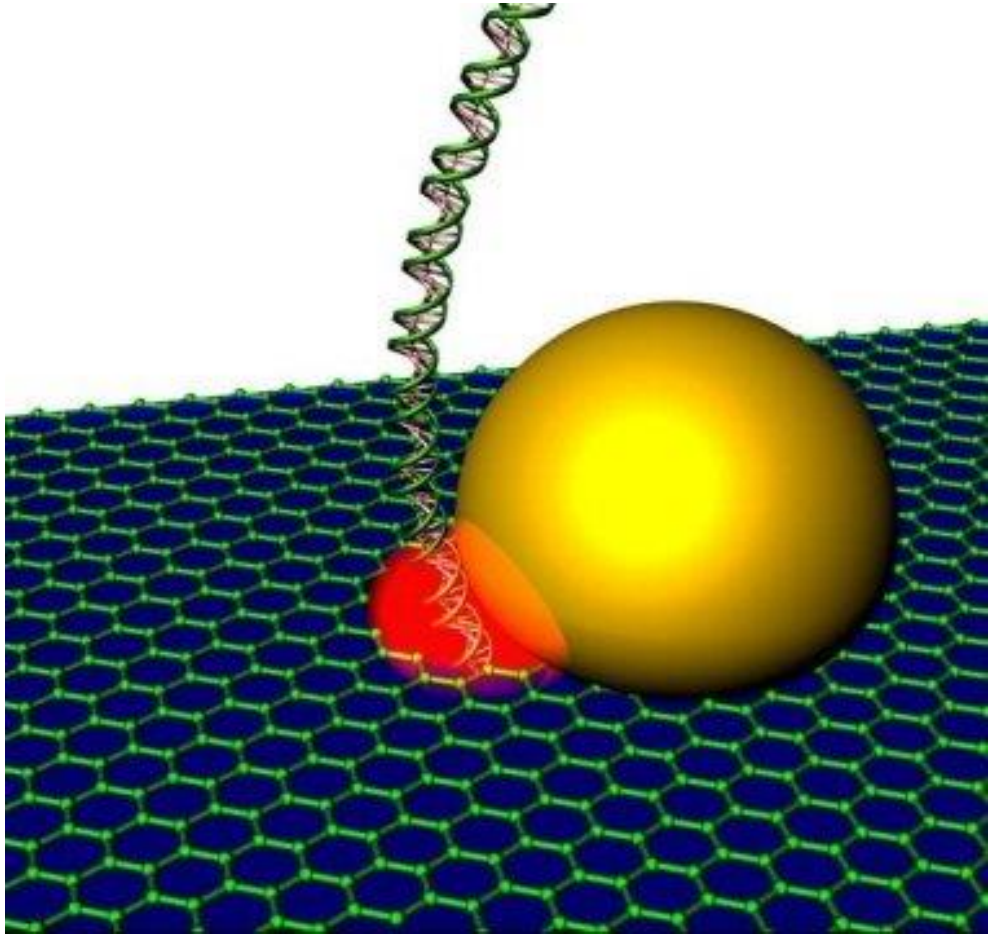
Hard to guarantee results so far.

Not practical yet.

So far, still mostly used for DNA Study.

DNA Reader

Source: DOE/Lawrence Berkeley National Laboratory



High-speed reading of the genetic code should get a boost with the creation of the world's first graphene nanopores -- pores measuring approximately 2 nanometers in diameter -- that feature a "built-in" optical antenna. Researchers have invented a simple, one-step process for producing these nanopores in a graphene membrane using the photothermal properties of gold nanorods.

DNA representation

DNA representation

The complementarity between base pairs (**A** = **T** and **G** ≡ **C**) implies that if you know one sequence you can deduce the complementary sequence.

It is common to represent DNA sequences by 4-letter strings:

TGCTAATGCCGCTACTCTATCTGC

By convention, we write sequences **from 5' to 3'** end.

5' – TGCTAATGCCGCTACTCTATCTGC – 3'

Source: Jacques van Helden

DNA representation

Don't forget the second strand!

When we analyze a DNA sequence represented by

```
ATGCGCGGATG
```

we should keep in mind that the corresponding molecule is a double strand helix with the following base pairs:

```
5' - ATGCGCGGATG - 3'        (upper strand)
     ||||||||||||
3' - TACGCGCCTAC - 5'        (lower strand)
```

Note that the concept of *upper strand* and *lower strand* are purely artificial. A DNA molecules is a 3D structure and there is no reason to consider preferentially one or the other strand.

This does not mean however that the two strands are functionally equivalent: in coding regions for example, only one strand will serve as a template for the synthesis of RNA.

Source: Jacques van Helden

The **5' and 3'** mean "five prime" and "three prime", which indicate the carbon numbers in the **DNA's** sugar backbone. The **5'** carbon has a phosphate group attached to it and the **3'** carbon a hydroxyl group. This asymmetry gives a **DNA** strand a "direction".

5’ and 3’ means directions

DNA representation

Reverse complementarity

Reverse complementary sequences represent the two strands of the same DNA molecule.

The reverse complement is obtained by transposing each nucleotide into its complementary nucleotide (**A → T, T → A, C → G, G → C**), and then reversing the string.

For example the sequences **ATGCGCGGATG** and **CATCCGCGCAT** are mutually reverse complementary. These strings describe the two strands of the same DNA molecule. Consequently, the two following double strand schemes represent the same molecule:

```
5'- ATGCGCGGATG - 3'        5'- CATCCGCGCAT - 3'
    ||||||||||||                ||||||||||||
3'- TACGCGCCTAC - 5'        3'- GTAGGCGCGTA - 5'
```

Source: Jacques van Helden

Symmetries in DNA sequences

Symmetries in DNA sequences

Tandem repeat

```
GATAAGATAAGATAAGATAA = 2 x GATAAGATAA = 4 x GATAA

GATAAGATAAatgtagGATAAGATAA = 2 x GATAAGATAA separated by
                             a non repeated sequence.
```

Tandem repeats are presumed to occur frequently in genomic sequences, comprising perhaps **10%** or more of the human genome (Benson, *NAR* 27:573,1999).

Tandem repeat are sometimes associated to a **repeated structure of a protein** (Ex: some ABC transporters have been shown to contain a tandem repeat of six transmembrane helices, Tusnady et al, *FEBS Lett* 402:1-3,1997).

In recents years, the discovery of **short tandem repeat polymorphisms** are involved in various diseases (e.g. Cancer, Huntington, Parkinson,..., Zhang & Yu, *Eur J Surg Oncol*,33:529-34,2007).

ATP-binding Cassette (ABC) transporters

Source: Jacques van Helden

Symmetries in DNA sequences

Symmetries in DNA sequences

Textual palindromes

ATGGCCGGTA = ATGGC|CGGTA

Note that the corresponding DNA molecule does not contain any axis of symmetry since in 3D space a nucleotide cannot be superimposed on its own image. Therefore, searching for palindromes is not relevant for detecting biological features.

5'- ATGGC - 3' ≠ 5'- CGGTA - 3'

Source: Jacques van Helden

Symmetries in DNA sequences

Symmetries in DNA sequences

Reverse complementary palindromes

A reverse complementary palindrome is a sequence identical to its reverse complement.

Example: `ATGGGGCCCCAT`

Reverse complementary palindromes correspond to 3D symmetries in DNA molecules. In the following 2-strand representation, a 180° rotation around the center would swap the two strands, and each letter would take place of an identical letter on the complementary strand.

```
5'- ATGGGG CCCCAT - 3'
    ||||||.||||||
3'- TACCCC GGGGTA - 5'
```

Note that this sequence is not a textual palindrome.

Note that reverse complementary palindromes can be separated by a stretch of non-symmetrical nucleotides.

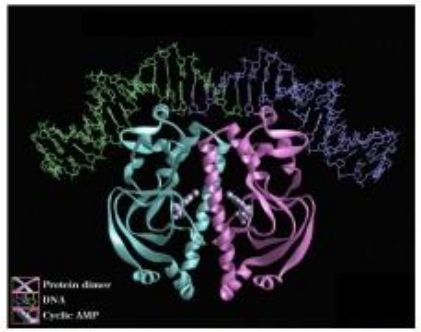
Source: Jacques van Helden

Symmetries in DNA sequences

Symmetries in DNA sequences

Reverse complementary motifs play important roles in biological mechanisms.

Example 1: some classes of transcription factors (e.g. *helix-turn-helix*) typically form homodimers whose tridimensional structure is symmetrical. These protein complexes specifically recognize reverse complementary motifs in gene promoters.



cAMP Receptor Protein (CRP)
TGTGA-N_6-TCACA

Example 2: In bacteria, hexamers with reverse complementary palindromic structure also play an essential role as recognition sites for *restriction enzymes*.

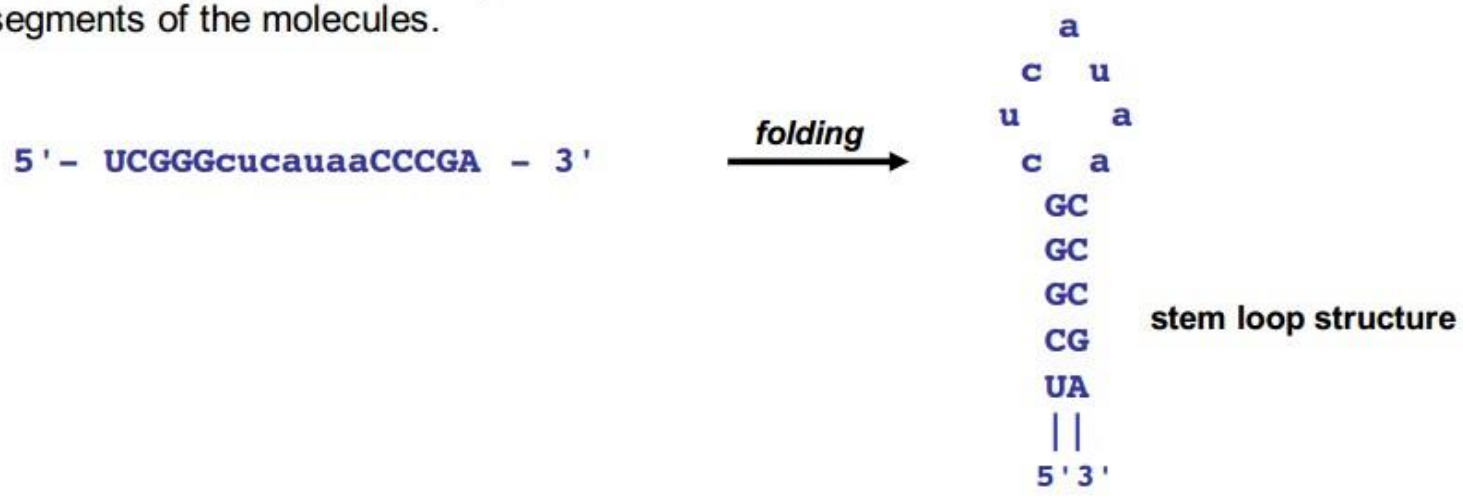
The restriction enzyme HindIII specifically cuts DNA at instance of AAGCTT

Symmetries in DNA sequences

Symmetries in DNA sequences

Reverse complementary motifs play important roles in biological mechanisms.

Example 3: Reverse complementary motifs separated by a stretch are frequent in RNA, where they mediate the pairing between distant segments of the molecules.



Source: Jacques van Helden

Demo Program: Part2

DNA Encoding (Gene.java)

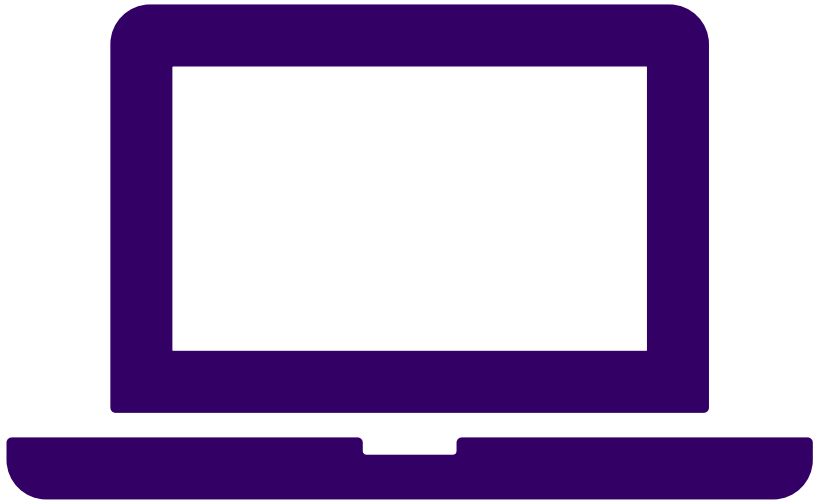
LECTURE 14

Genetic code

ACGT is an acronym for the four types of bases found in a DNA molecule:

adenine (**A**), cytosine (**C**), guanine (**G**), and thymine (**T**).

```
final char A = 'A';  final char C ='C';  final char G = 'G';   // A offset 0, D offset 3
final char T = 'T';  final char D = '3'; final char F = '5';    // F offset 5 (personal preference)
public static char complement(char code){ char comCode = A;
    if (code == A) comCode = T;         else if (code == T) comCode = A;
    else if (code == C) comCode = G; else if (code == G) comCode = C;
    else if (code == D) comCode = F; else if (code == F) comCode = D;
    return comCode;
};
```



Demonstration Program

GENE.JAVA

toComplementary()

```
public static String toComplementary(String inStr){
 String str = “”;
    for (int i = 0; i < inStr.length();  i++)
            str += complement(inStr.charAt(i));
    return str;
}
```

String Representation for DNA

Upper Strand: “**5**ACTGATCGGACT**3**”;

Lower Strand: “**3**ACTGATCGGACT**5**”;

```
public static String reverseComplementary(String inStr){
    String str = “”;
    for (int i = inStr.length()-1 ; i>=0; i--)
            str += complement(inStr.charAt(i));
    return str;
}
```

Tandem Repeat

```
public static String tandemRepeat(String inStr, int repeat){
    String str = “”;
    for (int i = 0;  i<repeat; i++) str += inStr;
    return str;
}
```

Tandem Repeat With Non-Repeat

```
public static String tandemRepeatWithNonRepeat
(String inStr, String nonRepeat){
    return inStr + nonRepeat + inStr;
}
```

isPalindrome() check?

Why stop at low < high ?

```
public static boolean isPalindrome(String s){
    int low = 0;
    // The index of the last character in the string
    int high = s.length() - 1;
    boolean isPalindrome1 = true;
    while (low < high) {
        if (s.charAt(low) != s.charAt(high)) {
          isPalindrome1 = false;
          break;
        }
        low++;
        high--;
    }
    return isPalindrome1;
}
```

isReverseComplementPalindrome() check?

```
public static boolean isReverseComplementaryPalindrome(String s){
    int low = 0;
    int high = s.length() - 1;
    boolean isRCPalindrome1 = true;
    while (low < high) {
     if (!isComplement(s.charAt(low), s.charAt(high))) {
            isRCPalindrome1 = false;
            break;
     }
     low++;
     high--;
    }
    return isRCPalindrome1;
  }
```

String Methods (toolbox) for DNA Simulation

final char A = ‘A’; final char C =‘C’; final char G = ‘G’; final char T = ‘T’;

String duplicate(String inDNAString);

String getDNA(String inDNAString); // get only the DNA part, skipp the 5 and 3

String cutOutS(String inDNAString, String pattern);

String cutOutAll(String inDNAString, String pattern);

String reverse(String inDNAString);

String replace(String pattern, String replacePattern); // Use String standard library Method

String replaceAll(String pattern, String replaceAllPattern); // Use String standard library Method

String swap(int i int j); String randomSwap();

String randomShuffle(String s, int steps);

String randomDNA(int len);

Dual Output: **log** method

Programming Technique

In main():

```
File oFile = new File("Gene.txt");
PrintWriter out = new PrintWriter(oFile);
log(out, "Basic DNA Operations: ");
```

In class Method Definition:

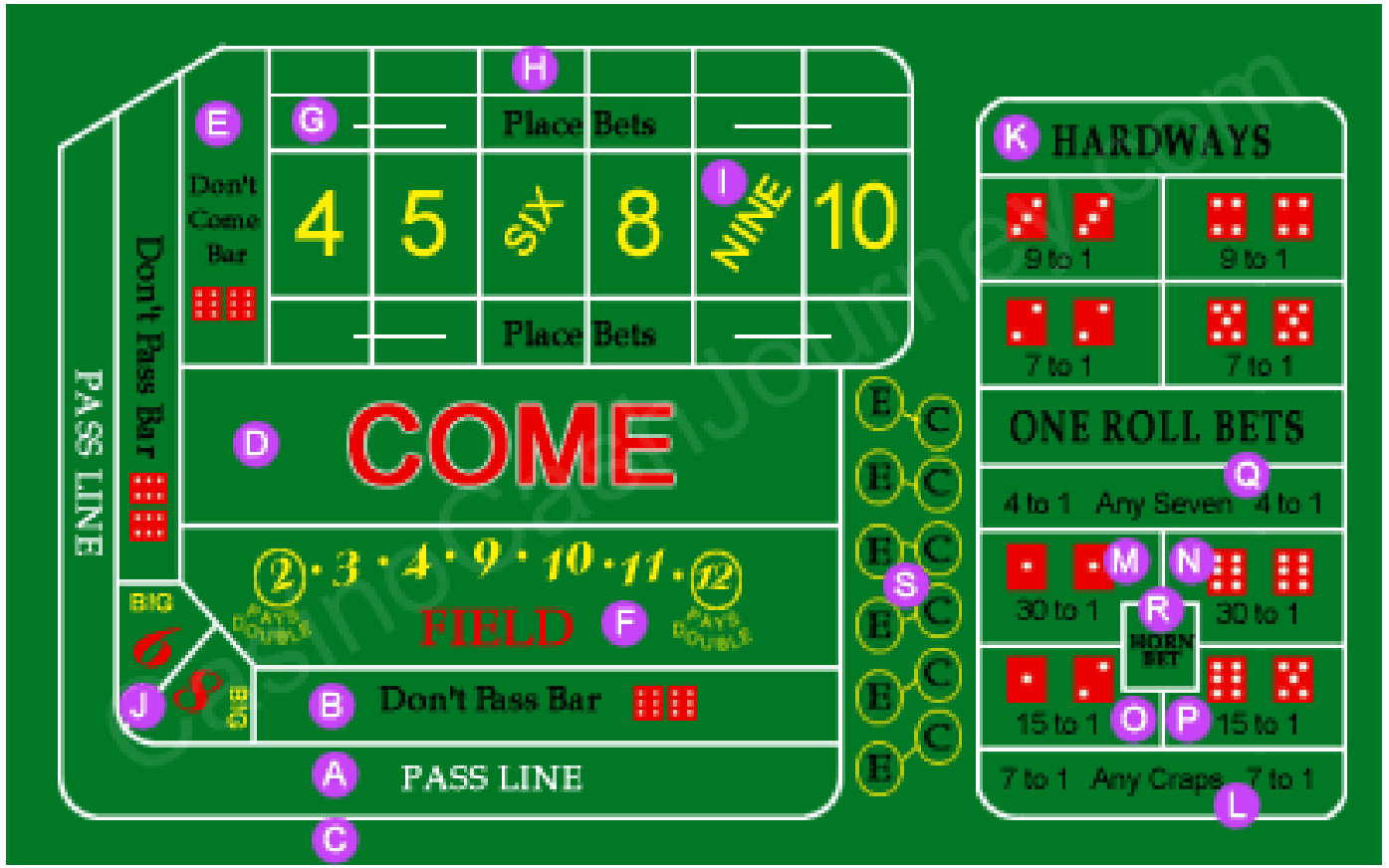
```
public static void log(PrintWriter out, String message)
throws IOException{
    System.out.println(message);     // print to screen
    out.println(message);            // print to file
}
```

Chapter Project: Craps.java

LECTURE 15

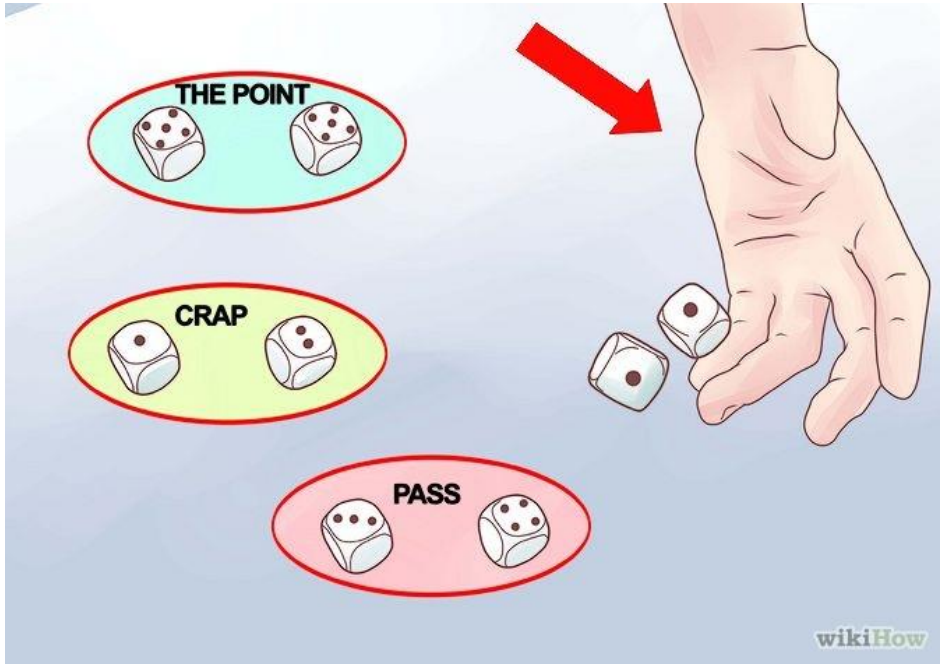
Background Information:

Craps Game Casino Style



Street Craps Game

The Street Craps, sometimes called Shooting Dice or Ghetto Craps is similar to casino craps but is played **without a craps table**.



Street Craps Rules

1. Players must first identify the player who will be shooting dice – the **shooter**. The shooter will then need to make a bet followed by the rest of the group in the clockwise direction. Each player can cover a portion of or all of the shooter's bet. Betting continues until the shooter's wager is matched.

2. The come out roll comes next. This is the game's first roll and it could end the game if it is a 7, 11, 2, 3 or 12. The shooter and any other player who bet in favor of the shooter win the game if a 7 or 11 is rolled. If a 2, 3 or 12 come up when the dice are rolled the shooter and other players who bet for him lose.

Street Craps Rules

3. A Point number, which is a number other than those mentioned above, must be set up. So if the come out roll is not any of those numbers listed above that number will be designated as the point number.

4. The roll is next and the goal is for the shooter to roll the number identified as the point before he rolls a 7. The 7 is referred to an “Out 7” and once the shooter gets this before rolling the point he loses the game.

5.Rolling dice proceeds until a 7 or the Point is rolled. The shooter loses if the 7 comes up and wins if the Point is rolled. If other numbers are rolled the shooter continues rolling the dice. The round ends only after a 7 or the point is rolled.

Chapter Project: Craps.java

(Simulate Street Craps Game for N times)

- Write a program to simulate the Street Craps Dice Game for N times, then sum up the total count for the shooter's winning percentage based on the count/N ratio .
- In each single game, shooter rolls the two dice once to get a **number**. If the number is **7** or **11**, then the shooter has the instant win and the current game is done.

Chapter Project: Craps.java

(Simulate Street Craps Game for N times)

- If the number is **2**, **3**, or **12**, then the shooter has the instant lose and the current game is also done. If the number is none of above, enter into a rolls out stage. Shooter repeat to rolls the dice to get a new number, if he gets a 7, he lose (we call it **out 7**).
- Otherwise, if this new number is the same as the **first rolled number** (at the 7-11 win stage), then shooter gets a win. For other numbers, shooter will repeat to rolls until there is a **out-7** or **a win-number-hit**.

Chapter Project: Craps.java

(Simulate Street Craps Game for N times)

Guidelines:

(1) You may write this program into 4 methods: main, winPassBet, sumOfTwoDice, and dieRoll.

(2) ***dieRoll():*** return the result for a single die roll.

sumOfTwoDice(): return the sum of two dice rolls.

winPassBet(): return the win/lose (win for true and **lose** for **false**) of a single game following the street craps game rules.

main(): write a loop to iterate for N times. Each time play a street craps game and determine whether the shooter is win or lose. Then, get the count number for total wins and print out the winning percentage over N games.

Pseudo Code

```
Integer dieRoll begin
        return /* random integer 1 to 6*/
 end

 // return sum of two dice
 Integer sumOfTwoDice() begin
     /* Generate x y two random integer number
         between 1 and 6 using dieRoll
    */
     return x + y;
 end
```

Pseudo Code

```
Boolean winsPassBet() begin // one game
      // 1. rolls two dice to get a random variable x for sum
       if (x is 7 or 11)       shooter wins and return true;
       if (x is 2, 3, or 12)  shooter lose and return false;
      repeat forever begin
          // get a new sum y for two dice
          if (y is 7) shooter lose and return false;
          if (y == x) shooter win and return true;
       end
   end
```

Pseudo Code

```
Void Main begin
   Assign 0 to Count
   Assign 1000000 to N
   Repeat for N times begin
      play a game if shooter win then count increases
   end
   print out count/N for winning percentage
End
```

Expected Results: