CEN 419
Introduction to Java Programming

Dr. H. Esin ÜNAL

FALL 2021

Slides are modified from original slides of Y. Daniel Liang

Opening Problem

Find the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.

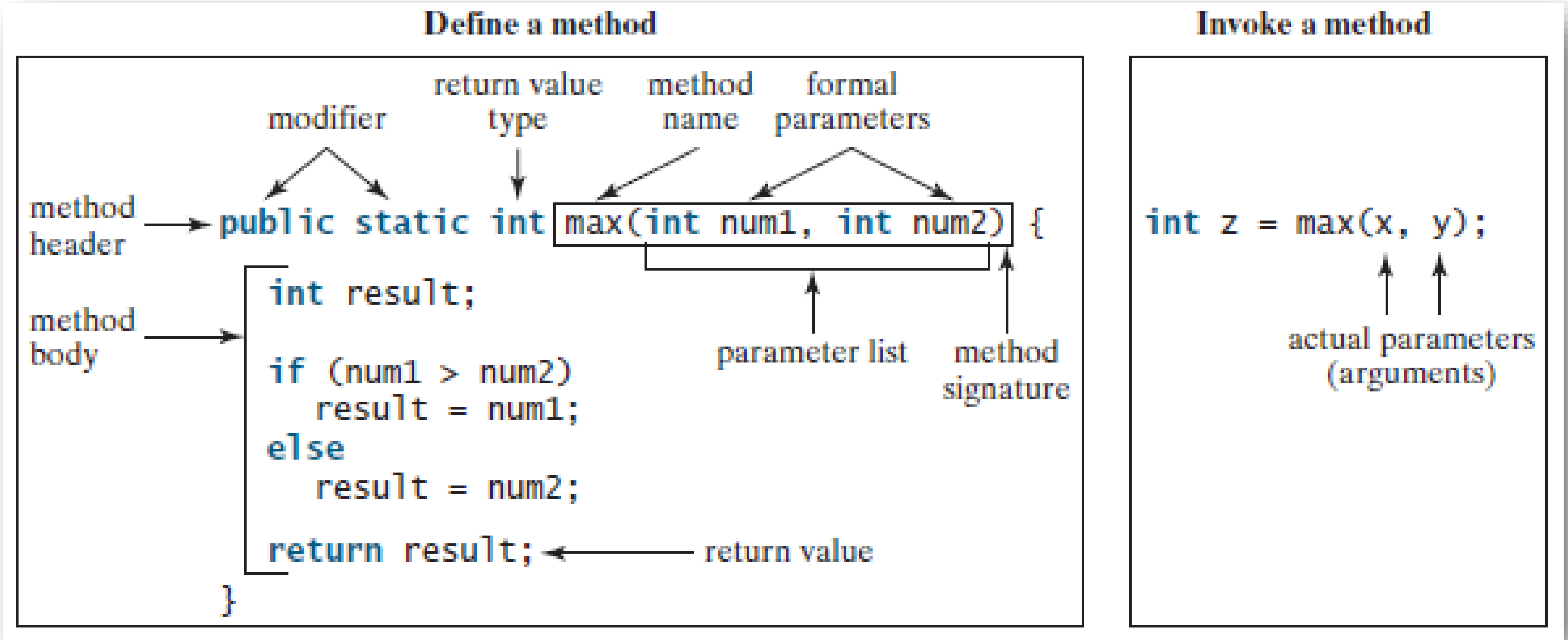
```
int sum = 0;
for (int i = 1; i <= 10; i++)
  sum += i;
System.out.println("Sum from 1 to 10 is " + sum);

sum = 0;
for (int i = 20; i <= 30; i++)
  sum += i;
System.out.println("Sum from 20 to 30 is " + sum);

sum = 0;
for (int i = 35; i <= 45; i++)
  sum += i;
System.out.println("Sum from 35 to 45 is " + sum);
```

Solution

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

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 30 is " + sum(20, 30));
  System.out.println("Sum from 35 to 45 is " + sum(35, 45));
}
```

Defining Methods

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

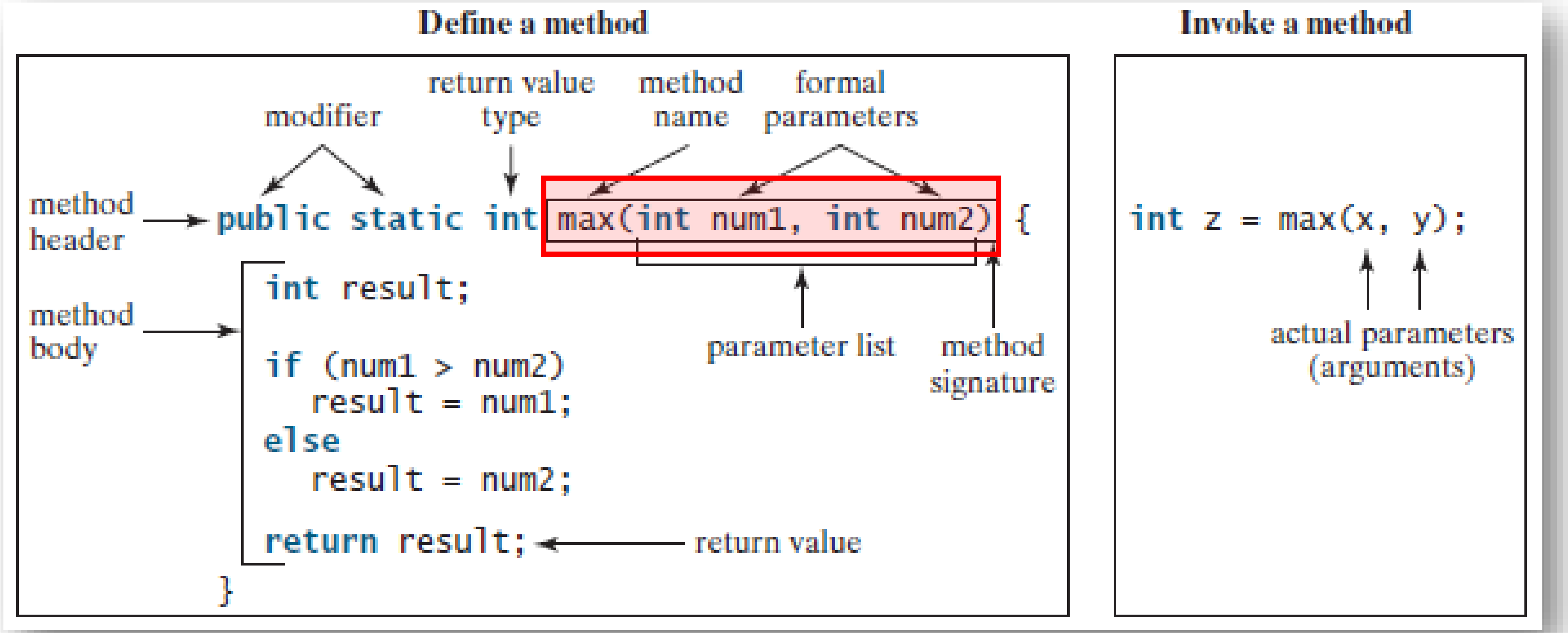


```
public static int max(int num1, int num2) {
   int result;

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

   return result;
}
```

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

Method Signature

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

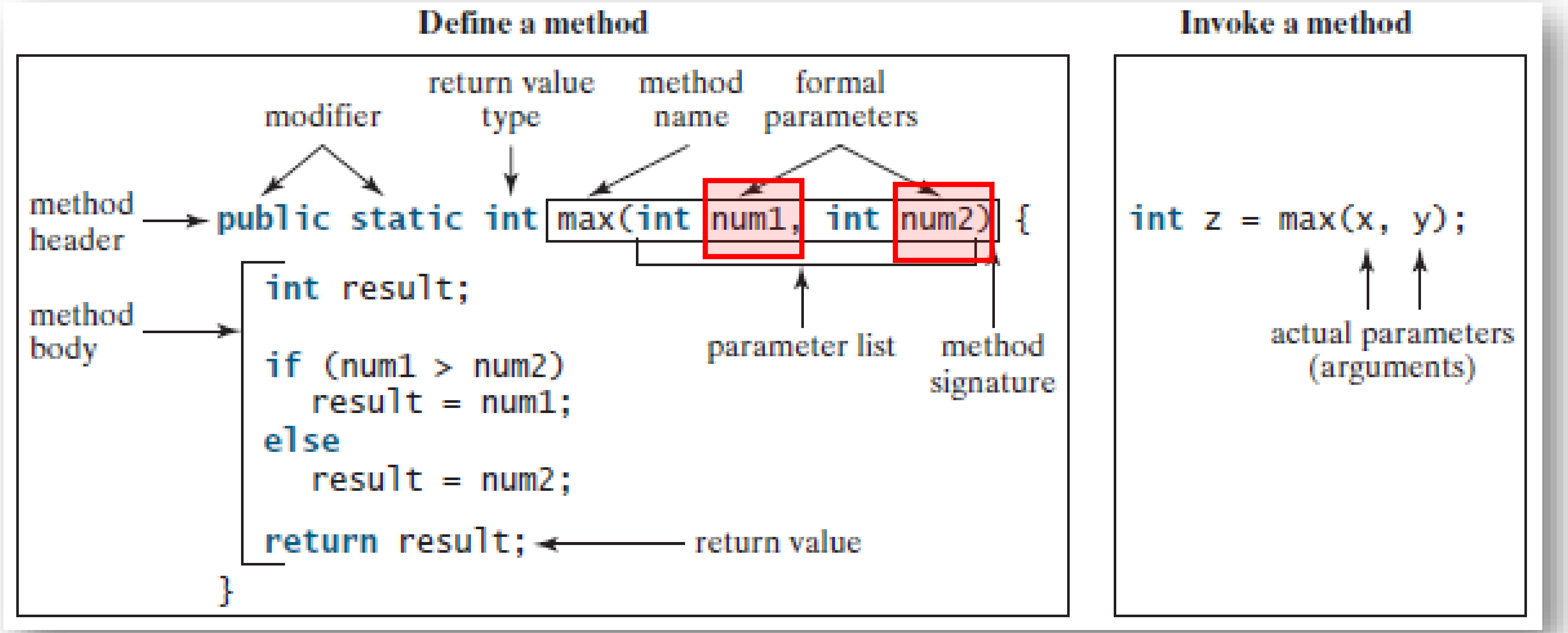


```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

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

Formal Parameters

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

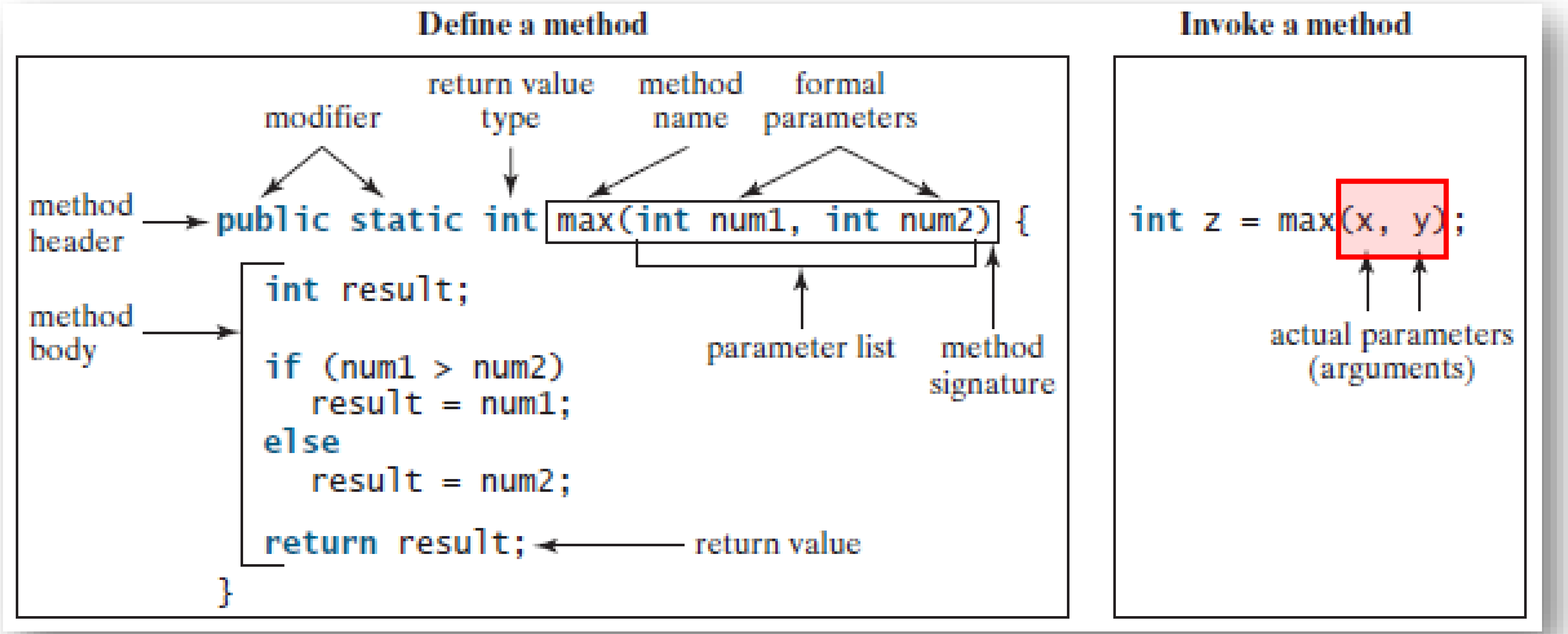


```
public static int max(int num1, int num2) {
    int result;

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

    return result;
}
```

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

Actual Parameters

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

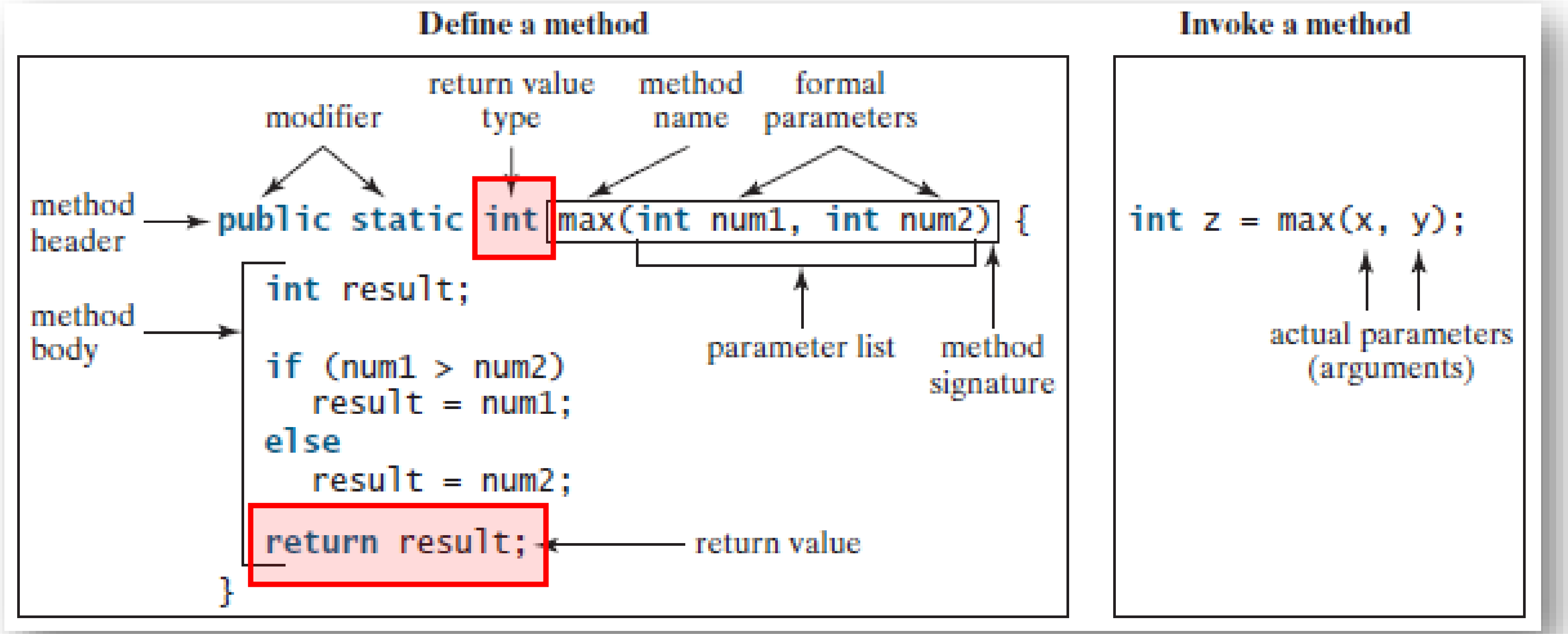


```
public static int max(int num1, int num2) {
  int result;

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

  return result;
}
```

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

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*.



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

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

  return result;
}
```

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

Calling Methods

- 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);`
    - `System.out.println(max(3, 4));`
  - ✓ If a method returns **void**, a call to the method must be a statement:
    - `System.out.println("Welcome to Java!");`

      (method **println** returns **void**)

Calling Methods

Testing the `max` method:

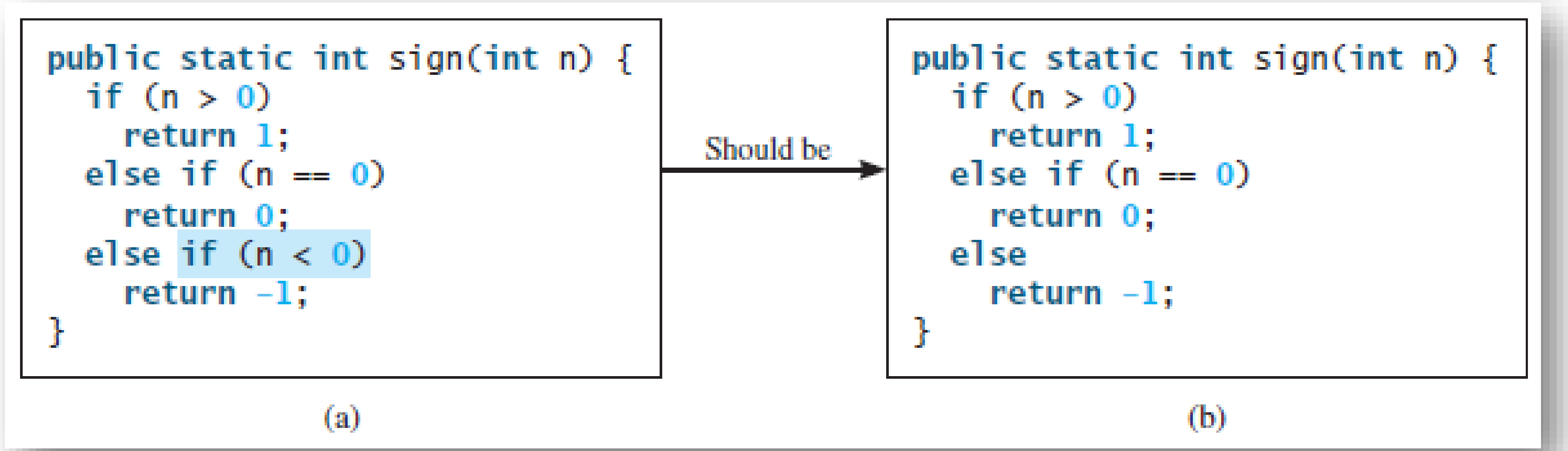
✓This program demonstrates calling a method `max` to return the largest of the `int` values.

Test Maximum Number

Intro to Java Programming, Y. Daniel Liang - TestMax.java (pearsoncmg.com)

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 that this method might not return a value.



```
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

- **One of the benefits of methods is for reuse.**
- The max method can be invoked from any class not just TestMax.
- If you create a new class Test, you can invoke the max method using:

```
ClassName.methodName(arguments)
```

(e.g., `TestMax.max(i,j)`)

Examples

- To see the differences between a void and value-returning method same problem is re-designed for both types

Void Method

[Intro to Java Programming, Y. Daniel Liang - TestVoidMethod.java (pearsoncmg.com)]

Return Method

[Intro to Java Programming, Y. Daniel Liang - TestReturnGradeMethod.java (pearsoncmg.com)]

NOTE

- A **return** statement is not needed for a **void** method, but it can be used for **terminating** the method and returning to the method's caller. The syntax is simply:

```
return;
```

Example

```
public static void printGrade(double score) {
   if (score < 0 || score > 100) {
      System.out.println("Invalid score");
      return;
   }
   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');
   }
}
```

Passing Arguments by Values

The arguments are passed by value to parameters when invoking a method.

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

Suppose you invoke the method using

```
      nPrintln(“Welcome to Java”, 5);
```

What is the output?

Can you invoke the method using

```
      nPrintln(15, “Computer Science”);
```

Passing Arguments by Values

- The arguments that are passed to a method **should have the same** number, type, and order as the parameters in the method signature.

- This program demonstrates passing values to the methods.

Pass by Value

Intro to Java Programming, Y. Daniel Liang - TestPassByValue.java (pearsoncmg.com)

Modularizing Code

Methods can be used to;

- ✓ reduce redundant coding
- ✓ enable code reuse
- ✓ improve the quality of the program.
- ✓ the logic becomes clear and the program is easier to read
- ✓ narrow the scope of debugging

GCD by Method

Intro to Java Programming, Y. Daniel Liang - GreatestCommonDivisorMethod.java (pearsoncmg.com)

Overloading Methods

- Overloading methods enables you to define the methods with the same name as long as their ***signatures are different***.
- ***Method overloading***; two methods have the same name but different parameter lists within one class.

Method Overloading

Intro to Java Programming, Y. Daniel Liang - TestMethodOverloading.java (pearsoncmg.com)

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 compile error.

Ambiguous Invocation

```
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 Local Variables

A ***local variable*** is a variable defined inside a method.

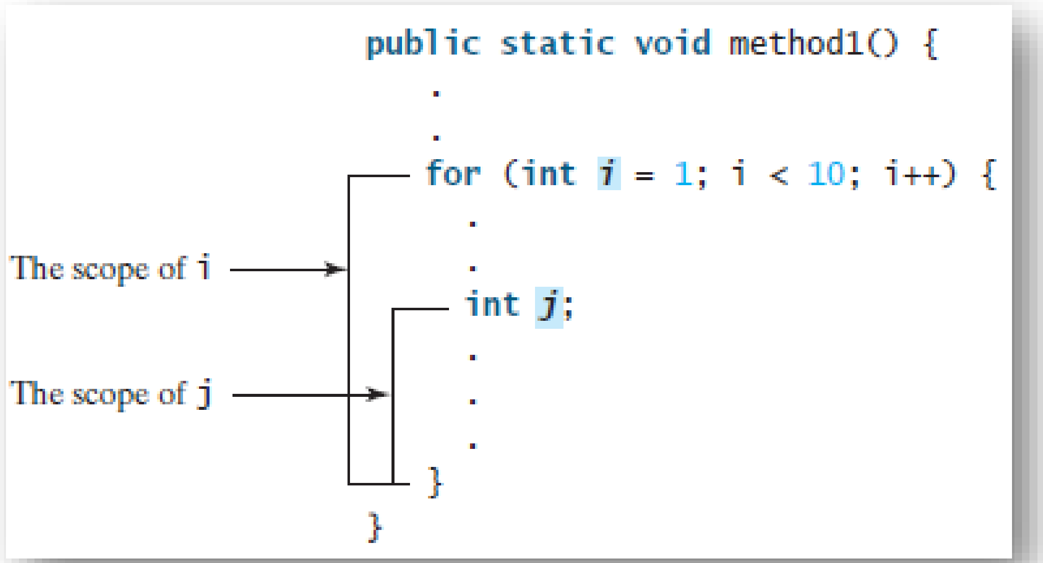
- ***Scope***: the part of the program where the variable can be referenced.

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

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

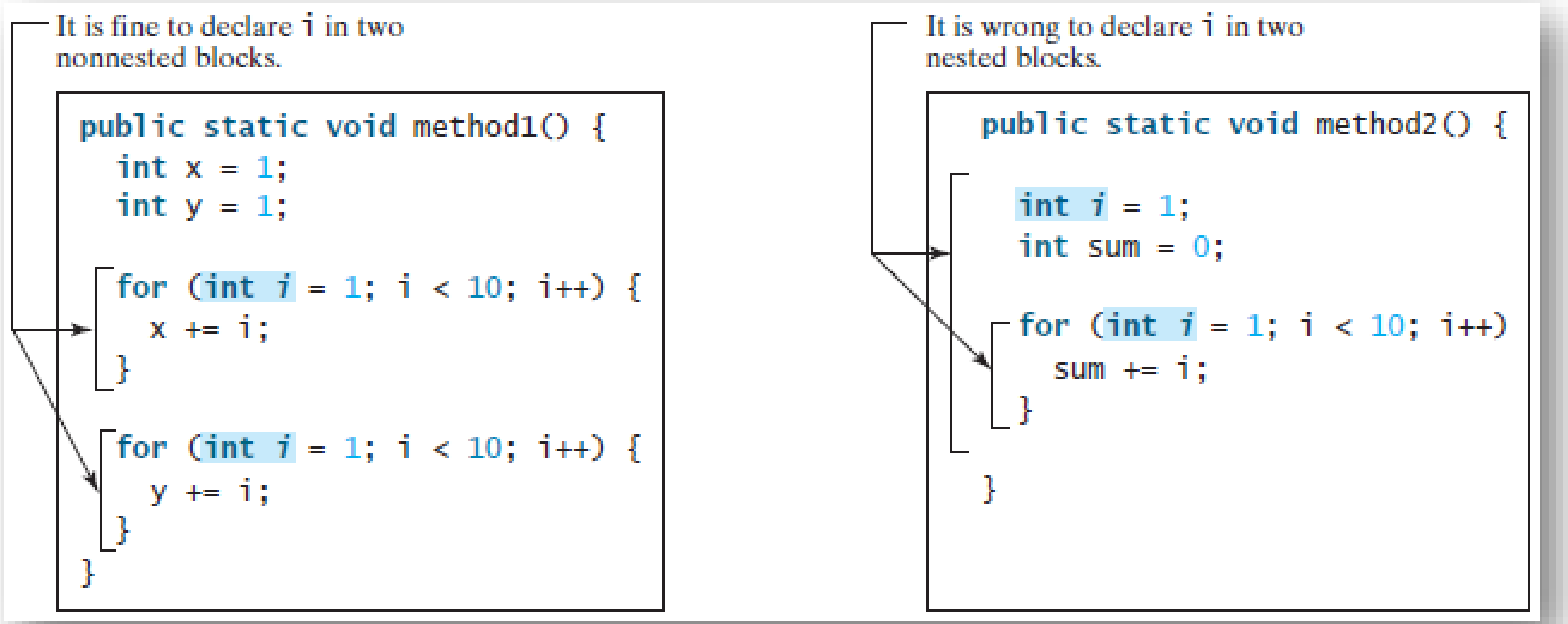
But a variable declared inside a for loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.



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

Scope of Local Variables

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.


It is fine to declare i in two nonnested blocks.
It is wrong to declare i in two nested 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;
  }
}
```

```
public static void method2() {

  int i = 1;
  int sum = 0;

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

}
```

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.

As you know, 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)((int)'a' + Math.random() * ((int)'z' - (int)'a' + 1))
```

Case Study: Generating Random Characters

As discussed in Chapter 2, 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:

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

So a random lowercase letter is

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

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

The RandomCharacter Class

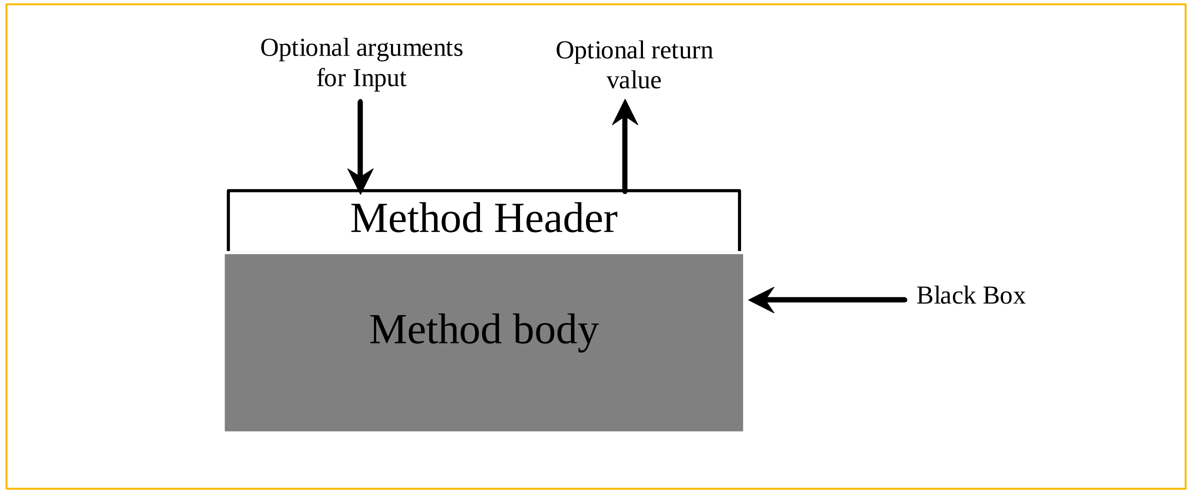
```
// RandomCharacter.java: Generate random characters
public class RandomCharacter {
  /** Generate a random character between ch1 and ch2 */
  public static char getRandomCharacter(char ch1, char ch2) {
    return (char)(ch1 + Math.random() * (ch2 - ch1 + 1));
  }

  /** Generate a random lowercase letter */
  public static char getRandomLowerCaseLetter() {
    return getRandomCharacter('a', 'z');
  }

  /** Generate a random uppercase letter */
  public static char getRandomUpperCaseLetter() {
    return getRandomCharacter('A', 'Z');
  }

  /** Generate a random digit character */
  public static char getRandomDigitCharacter() {
    return getRandomCharacter('0', '9');
  }

  /** Generate a random character */
  public static char getRandomCharacter() {
    return getRandomCharacter('\u0000', '\uFFFF');
  }
}
```

Test Program of RandomCharacter Class

```
public class TestRandomCharacter {
  /** Main method */
  public static void main(String[] args) {
    final int NUMBER_OF_CHARS = 175;
    final int CHARS_PER_LINE = 25;

    //Print random characters between 'a' and 'z', 25 chars per line
    for (int i = 0; i < NUMBER_OF_CHARS; i++) {
       char ch = RandomCharacter.getRandomLowerCaseLetter();
       if ((i + 1) % CHARS_PER_LINE == 0)
           System.out.println(ch);
       else
           System.out.print(ch);
     }
  }
}
```

Method Abstraction

- You can think of the method body as a black box that contains the detailed implementation for the method.
- The implementation of the method is hidden from the client in a “black box”.



Stepwise Refinement

- The concept of method abstraction can be applied to the process of developing programs.
- When writing a large program, you can use the "divide and conquer" strategy, also known as ***stepwise refinement***, to decompose it into subproblems.
- The subproblems can be further decomposed into smaller, more manageable problems.

Implementation Top-Down

- ✓ Top-down approach is to implement one method in the structure chart at a time from the top to the bottom.
- ✓ Stubs can be used for the methods waiting to be implemented.
- ✓ A stub is a simple but incomplete version of a method.
- ✓ The use of stubs enables you to test invoking the method from a caller.

A Skeleton with Stubs

Intro to Java Programming, Y. Daniel Liang - PrintCalendarSkeleton.java (pearsoncmg.com)

Implementation Bottom-Up

- ✓ Bottom-up approach is to implement one method in the structure chart at a time from the bottom to the top.
- ✓ For each method implemented, write a test program to test it.
- ✓ Both top-down and bottom-up methods are fine.
- ✓ Both approaches implement the methods incrementally and help to isolate programming errors and makes debugging easy. Sometimes, they can be used together.

Printing Calendar

Intro to Java Programming, Y. Daniel Liang - PrintCalendar.java (pearsoncmg.com)

Benefits of Stepwise Refinement

- ✓ Simpler Program
- ✓ Reusing Methods
- ✓ Easier Developing, Debugging, and Testing
- ✓ Better Facilitating Teamwork