CS1150
Principles of Computer Science
Methods (Part II)

Yanyan Zhuang

Department of Computer Science

http://www.cs.uccs.edu/~yzhuang

Passing Parameters

```
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?

Suppose you invoke the method using

nPrintln("Computer Science", 15);

What is the output?

Can you invoke the method using

nPrintln(15, "Computer Science");

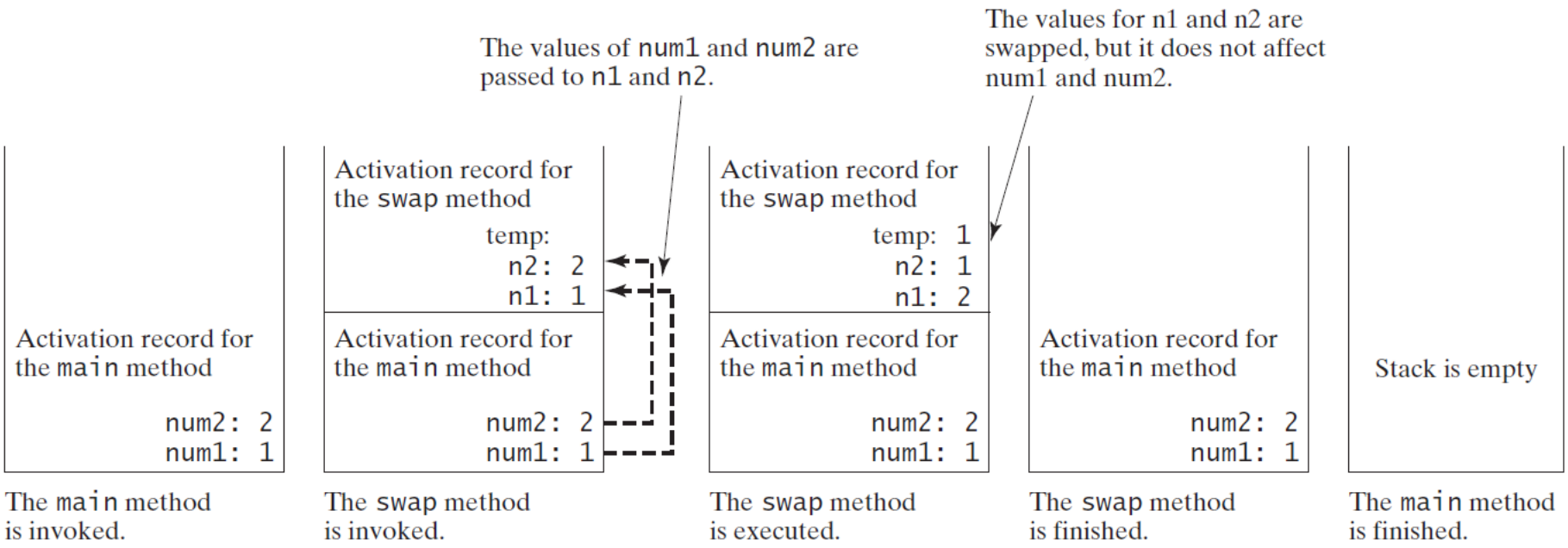
Pass by Value

- Swap.java: Caller **passes** arguments (actual parameters)
  - swap(number1, number2);
- Method **uses** parameters (formal parameters)
  - public static void swap (int num1, int num2)
- Actual and formal parameters match in **order**, **number** and **type**

Pass by Value

- Pass by value means
  - The actual parameter is fully evaluated
  - A copy of that value is placed into the formal parameter variable
  - Because only a copy is sent, the actual value of the argument is NOT changed by the method!
- Values of number1 and number2 are NOT changed
  - Only a copy of the variables sent to the method
  - Whatever happens to variables inside the method does not affect variables outside the method
- But, what happens if we make the formal parameter name match actual parameter name?
  - Will still only change the values of variables outside the method

Pass by Value



Converting Hexadecimals to Decimals

Write a method that converts a hexadecimal number into a decimal number.

ABCD =>

A*16^3 + B*16^2 + C*16^1+ D*16^0

= ((A*16 + B)*16 + C)*16+D

= ((10*16 + 11)*16 + 12)*16+13 = ?

Converting Hexadecimals to Decimals

User input is a String:

```
int decimalValue = 0;

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

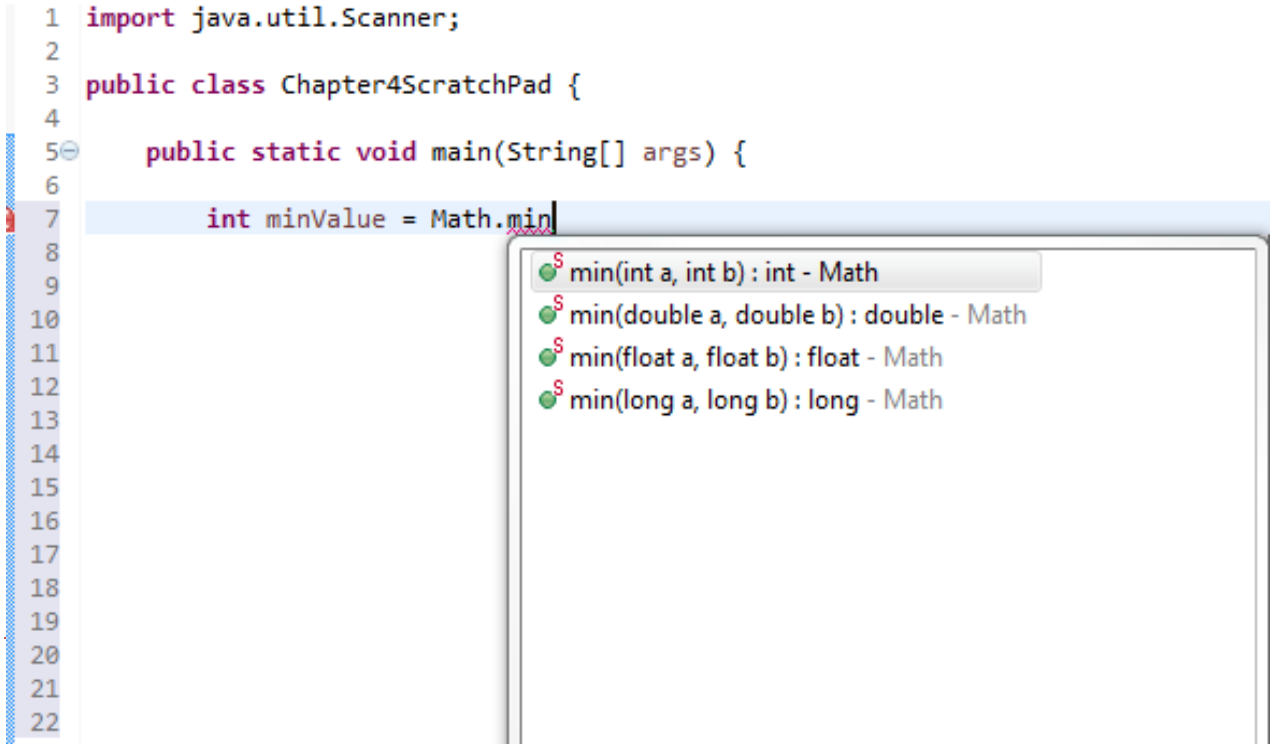
Another example Circle.java

Overloading Methods

- Two or more methods with the **same name** but **different formal parameters** (signature)
  - Gives the ability to create multiple versions of a method
- Why?
  - Because methods that perform the same task but on different data should be named the same (max, min)

Overloading Methods

- In the Math class the **min** and **max** methods are overloaded
- In the example, **min** can take
  - 2 ints
  - 2 doubles
  - 2 floats
  - 2 longs



Overloading Methods

Overloading the `min` Method

```
public static double min(double num1, double
  num2) {
   if (num1 < num2)
     return num1;
   else
     return num2;
}
```

Overloading Methods -- Rules

- To be considered an overloaded method
  - Name - must be the same
  - Return type - can be different - but you cannot change ***only*** the return type
  - Formal parameters - must be different

    Example: OverloadingMax.java
- Java will determine which method to call based on the parameter list
  - Sometimes there could be several possibilities: will pick the "best match"
  - Example: Perimeter.java
- It is possible that the methods are written in way that the complier cannot decide best match

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 an error**.

Ambiguous Invocation

```
public class AmbiguousOverloading {
  public static void main(String[] args) {
    System.out.println(max(1, 2));
    // Error here! The method max(int,double) is ambiguous
  }

  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: a variable defined inside a method/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.

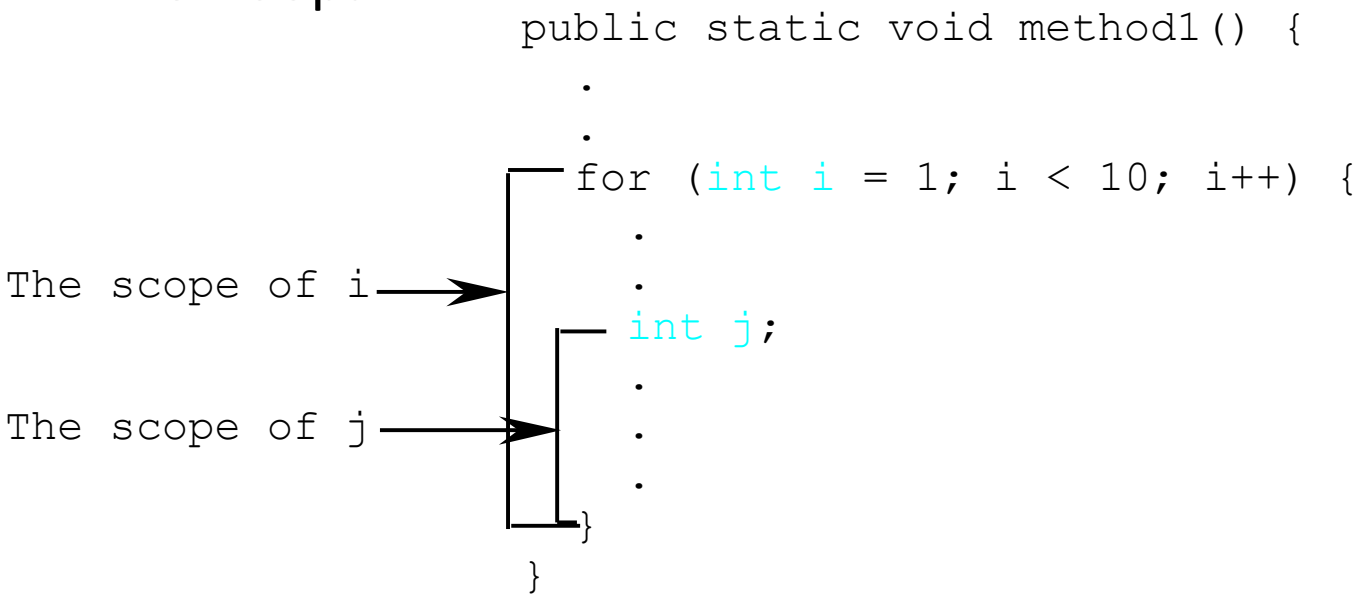
Scope of Local Variables, cont.

- **Can** declare a local variable with the same name multiple times in **different non-nesting** blocks in a method
- **Cannot** declare a local variable twice in **nested** blocks
- Formal parameters are considered local variables

Scope of Local Variables, cont.

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.

Question: inner loops?



Scope of Local Variables, cont.

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, cont.

```
// 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, cont.

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

Case Study: Generating Random Characters

Computer programs process numerical data and characters.

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, cont.

Now 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, cont.

As discussed in Chapter 4, all numeric operators can be applied to the char operands. 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, cont.

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');
  }
}
```

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 FindingMax.

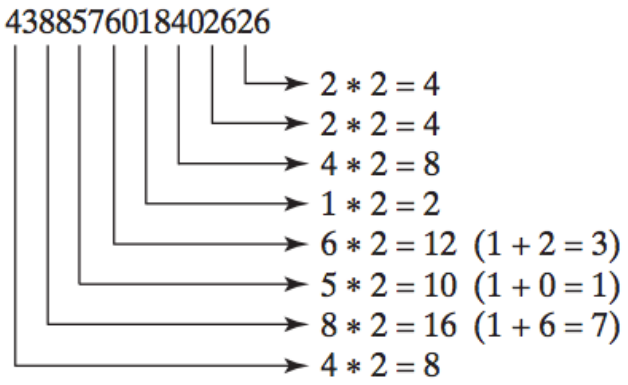
If you create a new class Test, you can invoke the max method using ClassName.methodName (e.g., FindingMax.max) in Test.

Practice: Even and Prime

- Write a method isEven that accepts an int argument
  - Return true if the argument is even, and false otherwise
  - public static boolean isEven(int number)
- Write a second method isPrime to see if an int is a prime number
  - For each integer that is prime return true otherwise return false
  - public static boolean isPrime(int number)
- Prompt a user input (assume to be int), and use the two methods to test the input

Practice: Validating Credit Card

- A credit card number must be between 13 and 16 digits
  1. Double *every second digit from right to left*. If doubling results in a two-digit number, add up the two digits to get a single digit (see 12, 10, and 16 in the figure)
  2. Add all single-digit numbers from Step 1 (4+4+8+2+3+1+7+8 = 37)
  3. Add all digits in odd places from right to left (6+6+0+8+0+7+8+3=38)
  4. Sum the results from Step 2 and Step 3
  5. If the result from Step 4 is divisible by 10, the card number is valid; otherwise, it is invalid



Practice: Validating Credit Card

- Implement the following methods

```
/** Return true if the card number is valid */
public static boolean isValid(String number)

/** Get the result from Step 2 */
public static int sumOfDoubleEvenPlace(String number)

/** Return the number itself if it is a single digit, otherwise,
   * return the sum of the two digits */
public static int getDigit(int number)

/** Return sum of odd-place digits in number (Step 3) */
public static int sumOfOddPlace(String number)

/** Return the number of digits in number */
public static int getLength(String number)
```

Examples:

Input : 379354508162306
Output : Card number is valid

Input : 4388576018402626
Output : Card number is invalid

Summary

- Value-returning methods
- Void methods
- Call stack
- Overloading methods
- Scope of local variables