Methods

CSE 114: Introduction to Object-Oriented Programming
Paul Fodor
Stony Brook University
http://www.cs.stonybrook.edu/~cse114
Contents

• Motivation: Opening Problem
• Why write methods?
• Defining Methods:
  • Method Signature: modifiers,
    • Formal and Actual Parameters
    • Return Value Type
  • Calling Methods
• Call Stacks
• Benefits of Methods: Reuse, Method Abstraction and Information hiding
• Call-by-value
• Overloading
• CAUTION: all execution paths need returns (if a value is returned)
• Scope of Local Variables

Stepwise Refinement: Top-Down and Bottom-Up Implementation
Motivation: Opening Problem

Find multiple sums of integers:
- from 1 to 10,
- from 20 to 30,
- from 35 to 45,
...

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
Opening Problem

- Repeat/copy code:

```java
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);
```
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));
}
Why write methods?

- To shorten your programs
  - avoid writing identical code twice or more
- To modularize your programs
  - fully tested methods can be trusted
- To make your programs more:
  - readable
  - reusable
  - testable
  - debugable
  - extensible and adaptable
Rule of thumb

- If you have to perform some operation in more than one place in your program, write a method to implement this operation and have other parts of the program use it.
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;
}
```

Define a method

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

Invoke a method

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
Method Signature

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

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

Define a method

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

Invoke a method

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
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 values to the formal parameter with *actual parameters* or *arguments*.

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

```java
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**.
Calling Methods

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;
}

pass the value of i

pass the value of j
Trace Method Invocation

public static void main(String[] args) {
    int i = 5;
    int i = 2;
    int k = max(i, i);
    System.out.println("The maximum between " + i + " and " + i + " is " + k);
}

public static int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
Trace Method Invocation

```
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;
}
```

j is now 2
Trace Method Invocation

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

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

public static int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;

    return result;
}
```

Trace Method Invocation

invoke max(i, j)
Pass the value of i to num1
Pass the value of j to num2

```java
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 " + i + " is " + k);
}
```

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

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

    return result;
}
```
Trace Method Invocation

```java
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);
    System.out.println("The maximum between " + i + ", " + j + " is " + k);
}

public static int max(int num1, int num2) {
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```
Trace Method Invocation

**(num1 > num2) is true since num1 is 5 and num2 is 2**

```java
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;
}
```
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 Method Invocation

```java
public static void main(String[] args) {
    int i = 5;
    int i = 2;
    int k = max(i, i);
    System.out.println(
        "The maximum between " + i + " and " + i + " is " + k);
}
```

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

return result, which is 5
Trace Method Invocation

return max(i, j) and assign the return value to k

defined to return max(i, j) and assign the return value to 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 " + i + " is " + k);
}

public static int max(int num1, int num2) {
    int result:
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result:
}
Trace Method Invocation

Execute the print statement

```java
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, j);
    System.out.println("The maximum between " + i + ", " + j + " is " + k);
}

class Solution {
    public static int max(int num1, int num2) {
        int result;
        if (num1 > num2)
            result = num1;
        else
            result = num2;
        return result;
    }
}
```
**Call Stacks**

Methods are executed using a *stack* data structure.

<table>
<thead>
<tr>
<th>(a) The main method is invoked.</th>
<th>(b) The max method is invoked.</th>
<th>(c) The max method is being executed.</th>
<th>(d) The max method is finished and the return value is sent to k.</th>
<th>(e) The main method is finished.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space required for the main method</td>
<td>Space required for the max method</td>
<td>Space required for the main method</td>
<td>Space required for the main method</td>
<td>Stack is empty</td>
</tr>
<tr>
<td>k:</td>
<td>num2: 2</td>
<td>result: 5</td>
<td>k:</td>
<td></td>
</tr>
<tr>
<td>j: 2</td>
<td>num1: 5</td>
<td>num2: 2</td>
<td>j: 2</td>
<td></td>
</tr>
<tr>
<td>i: 5</td>
<td></td>
<td>num1: 5</td>
<td>i: 5</td>
<td></td>
</tr>
</tbody>
</table>

Space required for the main method:
- k: 2
- j: 5
- i: 5

Space required for the max method:
- num2: 2
- num1: 5

Stack is empty:
- k: 5
- j: 2
- i: 5

Methods are executed using a stack data structure.
Trace Call Stack

The main method is invoked.

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;
}
The main method is invoked.

Space required for the main method

k: j: 2
i: 5

Declare k

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

    System.out.println(
        "The maximum between " + i + " and " + i + " 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

The main method is invoked. Space required for the main method:
- k:
- j: 2
- i: 5

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

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 " + i + " is " + k);
}
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, i):

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

public static int max(int num1, int num2) {
    int result:

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

    return result:  
}
public static void main(String[] args) {
    int i = 5;
    int j = 2;
    int k = max(i, i):
    System.out.println(
        "The maximum between " + i + 
        " and " + i + " 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

```java
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 " + i + " 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

Space required for the main method:
- k: 2
- j: 2
- i: 5

The max method is invoked.

result:
- num2: 2
- num1: 5
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 " + i + " is " + k);
}

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

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

    return result;
}
The max method is invoked.

Space required for the max method
result: 5
num2: 2
num1: 5

Space required for the main method
k: 5
j: 2
i: 5

The max method is invoked.

public static int max(int num1, int num2)
public static void main(String[] args) {
Trace Call Stack

```java
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;
}
```

Execute print statement

Space required for the main method
k: 5
j: 2
i: 5

The main method is invoked.
Benefits of Methods

1. Reuse: Write a method once and reuse it anywhere.

2. Method Abstraction and Information hiding:
   - Hide the implementation from the user.
   - Reduces complexity of the program.
Method Abstraction

Application Programming Interface (API) = the method body is a black box that contains the detailed implementation for the method.
Javadoc

- The API for a class is documented using the Javadoc.
- Generate Javadoc for your project in Eclipse with:
  1. Project -> Generate Javadoc
  2. Check the box next to the project/package/file for which you are creating the javadoc
  3. In the "Destination" field browse to find the desired destination (for example, the doc directory of the current project).
  4. Leave everything else as it is.
  5. Click "Finish" and open "index.html"
Class pattern

java.lang.Object
pattern

public class pattern
extends java.lang.Object

Constructor Summary

Constructors
Constructor and Description
pattern()
Call-by-value

• Method formal parameters are *copies of the original data*.

• Consequence?
  • methods cannot assign ("=") new values to primitive type formal arguments and affect the original passed variables.

• Why?
  • changing argument values changes the copy, not the original.
public class Test1 {
    public static void main(String[] args) {
        int num = 1;
        m(num);
        System.out.println(num); // still 1
    }
    public static void m(int n) {
        n = 2;
    }
}
public class Test {
    public static void main(String[] args) {
        int num1 = 1;
        int num2 = 2;
        swap(num1, num2);
        System.out.println(num1 + " " + num2);//still 1 2
    }
    public static void swap(int n1, int n2) {
        int temp = n1;
        n1 = n2;
        n2 = temp;
        System.out.println(n1 + " " + n2); // 2 1
    }
}
The main method is invoked

The values of num1 and num2 are passed to n1 and n2. Executing swap does not affect num1 and num2.

Space required for the main method
num2: 2
num1: 1

The swap method is invoked

Space required for the swap method
temp:
n2: 2
n1: 1

Space required for the main method
num2: 2
num1: 1

The swap method is finished

Stack is empty

num2: 2
num1: 1

The main method is finished
Method overloading is the ability to create multiple methods of the same name with different signatures and implementations:

```java
public class Overloading {
    public static int max(int num1, int num2) {
        if (num1 > num2)
            return num1;
        return num2;
    }
    public static double max(double num1, double num2) {
        if (num1 > num2)
            return num1;
        return num2;
    }
    public static void main(String[] args) {
        System.out.println(max(1, 2)); // 2 (as an int)
        System.out.println(max(1, 2.3)); // 2.3 (as a double)
    }
}
```
Overloading & Ambiguous Invocation

- Overloaded methods must differ either by the types of their parameters or by arity (i.e., number of arguments)
- Method/Call matching is the process to find the method implementation for the call:
  - it uses a "best match" algorithm to cast the actual parameters' types to the formal parameter types
- For example:
  ```java
  System.out.println(max(1.5, 2)); // 2.0 (as a double)
  System.out.println(max(1, 2.5)); // 2.5 (as a double)
  ```
Overloading & 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**. and it is a compilation error.
Overloading & Ambiguous Invocation

public class AmbiguousOverloading {
    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;
    }
    public static void main(String[] args) {
        System.out.println(max(1, 2)); // compiler error here
    }
}
CAUTION: all execution paths

- A return statement is required for a value-returning method. The method shown below has a compilation error because the Java compiler thinks it possible that this method does not return any value if the condition is false in the last if statement.

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

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.

```java
public static int sign(int n) {
    if (n > 0)
        return 1;
    else if (n == 0)
        return 0;
    else
        return -1;
}
```
Scope of Local Variables

• Remember that a local variable is a variable defined inside a method.
  • The scope of a variable the part of the program where the variable can be referenced.
    • In Java, the scope of a local variable starts from its declaration and continues to the end of the block that contains the variable
    • A nested block cannot redefine a local variable:

```java
public static void correctMethod() {
    int x = 1;
    int y = 1;
    for (int i = 1; i < 10; i++) {
        // int x = 0; // Syntax error
        x += i;
    }
}
```
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.
- For example, consider a PrintCalendar program:

```
C:\book>java PrintCalendar
Enter full year (e.g., 2001): 2009
Enter month in number between 1 and 12: 4
April 2009

Sun Mon Tue Wed Thu Fri Sat
1 2 3 4
5 6 7 8 9 10 11
12 13 14 15 16 17 18
19 20 21 22 23 24 25
26 27 28 29 30
C:\book>
```
Design Diagram

- printCalendar (main)
  - readInput
  - printMonth
    - printMonthTitle
      - getMonthName
    - printMonthBody
      - getStartDay
      - getNumOfDaysInMonth
        - isLeapYear
The \textit{top-down} approach is to implement one method in the structure chart at a time \textbf{from the top to the bottom}

- Stubs can be used for the methods waiting to be implemented
  - A \textit{stub} is a simple but incomplete version of a method.

  ```java
  /** A stub for \texttt{getStartDay} may look like this */
  public static int \texttt{getStartDay}(int \texttt{year}, int \texttt{month}) {
    return 1; // A dummy value
  }
  ```

- The use of stubs enables you to test invoking the method from a caller.

- Implement the main method first and then use a stub for the \texttt{printMonth} method.
  - Then implement the methods one by one starting from the top
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 only that method

- 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.
  - Most of the time, they are used together
Benefits of Stepwise Refinement

• Simpler Program

• Reusing Methods

• Easier Developing, Debugging, and Testing

• Better Facilitating Teamwork