Methods

CSE 114, Computer Science 1
Stony Brook University

http://www.cs.stonybrook.edu/~cse114
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 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);
```
• Use a method!

```java
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
  - debuggable
  - extensible
  - adaptable
Rule of thumb

• If you have to perform some operation in more than one place inside your program, make a new 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.

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

---

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
• *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*.
Actual Parameters

- When a method is invoked, you pass a value to the parameter: *actual parameter or argument*.

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

```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);
}
```

```java
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 + ", " + j + " 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

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

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

```java
class Main {
    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
class MaxCalculator {
    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) {
        if (num1 > num2) {
            return num1;
        } else {
            return num2;
        }
    }
}
```

return result, which is 5

return result:
Trace Method Invocation

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

```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

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 + " and " + j + " is " + k);
}
```

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

• Write a method once and reuse it anywhere.
• Information hiding:
  • Hide the implementation from the user.
• Reduces complexity.
Method Abstraction

API = the method body is a black box that contains the detailed implementation for the method.
Javadoc

• Generate Javadoc for your project in Eclipse with:
  1. Project -> Generate Javadoc
  2. In the "Javadoc command" field - browse to find javadoc.exe
     • On the computers in the lab that is
       C:\Program Files\Java\jdk1.8.0\bin\javadoc.exe
     • On other computers it would be
       <path_to_jdk_directory>\bin\javadoc.exe
  3. Check the box next to the project/package/file for which you are creating the javadoc
  4. In the "Destination" field browse to find the desired destination (for example, the doc directory of the current project).
  5. Leave everything else as it is.
  6. Click "Finish" and open "index.html"
public class pattern
extends java.lang.Object

class pattern
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.

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.
Call Stacks

Methods are executed using a stack data structure

(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

```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;
}
```

- `i` is declared and initialized.
- The main method is invoked.
- `i: 5`
Trace Call Stack

```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;
}
```

- **j** is declared and initialized
- The main method is invoked.
Trace Call Stack

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 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 " + 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 values of i and j to num1 and num2

The max method is invoked.

Space required for the main method

num1: 5
num2: 2
i: 5
j: 2
k: 

The max 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:
}
```

(num1 > num2) is true

result:  
num2: 2
num1: 5

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

The max method is invoked.
The max method is invoked.

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

Assign num1 to result

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

The max method is invoked.

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

```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 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;
}
Call-by-value

• Method formal arguments 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.
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.

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

The swap method is finished
Space required for the swap method
Stack is empty

The main method is finished
Space required for the main method
num2: 2
num1: 1

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

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

(c) Pearson Education, Inc. & Paul Fodor (CS Stony Brook)
Method overloading is the ability to create multiple methods of the same name with different implementations.

```java
// Overload the name max for different invocations
public static int max(int x, int y) {
    return (x > y) ? x : y;
}
public static double max(double x, double y) {
    return (x > y) ? x : y;
}

public static void main(String[] args) {
    System.out.println(max(1, 2)); // will call max(int, int)
    System.out.println(max(3.5, 4.7)); // will call max(double, 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*.
- Ambiguous invocation is a compilation error.
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*: a variable defined inside a method.
- *Scope*: 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.
Scope of Local Variables

- A nested block cannot redefine a local variable:

```java
public static void incorrectMethod() {
    int x = 1;
    int y = 1;
    for (int i = 1; i < 10; i++) {
        int x = 0;  // 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
```
Design Diagram

- printCalendar
  (main)
  - readInput
  - printMonth
Implementation: Top-Down

• The *top-down* approach is to implement one method in the structure chart at a time **from the top to the bottom**
  • A *stub* is a simple but incomplete version of a method. Stubs can be used for the methods waiting to be implemented.
    /** A stub for getStartDay may look like this */
    public static int getStartDay(int year, int 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 printMonth method
  • Then implement the complete methods
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
- Easier Developing, Debugging, and Testing
- Better Facilitating Teamwork
- Reusing Methods