Lecture 09: Methods

Sierra College
CSCI-12
Spring 2015
Weds 02/25/15

Announcements

- **Schedule**
  - **Spring Pass/No-Pass deadline is Monday 3/2**
- **Past due assignments**
  - HW06: Template, accepted thru Weds 2/25 @ 11pm
  - HW07: Variables, accepted thru Tues 3/3 @ 11pm
- **Current assignments**
  - HW08: Operators, due Fri 2/27 @ 11pm (*lab time today*)
- **New assignments**
  - HW09: External Input, due Tues 3/3 @ 11pm (*lab time today*)
    - Repeat the Operators assignment, except add external user inputs using the Scanner class (see the posted app note and example code)
    - BOTH assignments must be turned in (no “double-dipping”!)

Lecture Topics

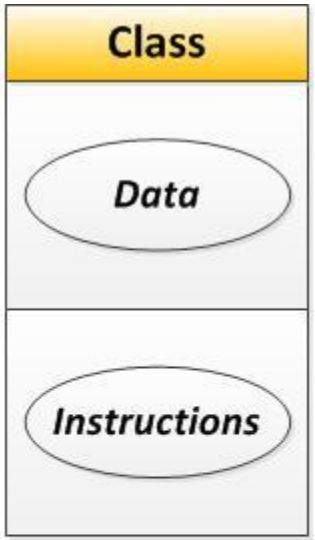
- **Last time**:
  - OPERATORS in Java
  - Demo: using the jGRASP debugger to look at code
- **Today**:
  - Tail end of last lecture: mixed-type expressions, casting
  - METHODS in Java

What Are Methods?

- **Methods** are named software containers for some group of logically-related statements, which together serve some specific purpose
  - "You give me this data, I'll do such-and-such for you"
- **Methods** are used for isolating repeated instructions into one self-contained, reusable software element
  - A named set of instructions which can be executed by that one name
  - Instructions which are the same can be separated from data which varies
  - Methods hide the underlying details from the user or caller
- **Methods** represent "atomic" elements of reusable software, and they:
  - accept 0 or more input values
  - perform some software action(s)
  - return 0 or more output values
- Method black box model:

Methods in OOP

- **Variables** are containers for **data**
- **Methods** are containers for **instructions** (statements)
- **Classes** are containers for related **data and instructions** together



- **Objects** are instances of a class, using the class as a blueprint or template

Methods In Other Languages

- The concept of “methods” is universal across high-level programming languages
- Some other terms you may see in other software languages for methods:
  - Reusable code
  - Functions, user-defined functions
  - Procedures, user-defined procedures
  - Subroutines, subprograms
  - Macros
  - Scripts, script files
- All of these terms are merely dancing around the same fundamental idea!

Why Use Methods?

- Modularity
  - Methods let us organize and compartment our code
  - Methods allow us to hide away all the details of our code (encapsulation)
  - Offloading computational details into methods makes our application code leaner and easier to follow
  - ***"Don't get buried by the details, focus on the bigger picture***"
- Efficiency
  - Problem decomposition helps us tackle a large problem using lots of little problems
  - It's easier to write smaller, focused units of software than it is to write one huge program
  - It's easier to debug and maintain smaller elements of software
  - ***"Divide and conquer"***
- Reusability
  - It's more efficient to carve out repeated operations and place them into methods
  - Placing common, repeated operations into methods facilitates their reuse or on other tasks
  - Build larger software out of smaller, well-tested pieces
  - ***"Write it once, then just reuse it"***

Where Do Methods Appear?

- Some places we have seen, or will see, methods:
  - All the code you've written so far has had a **main()** method
    - At least one main() per application is required by Java
  - Your first Hello World program had a dedicated **printGreeting()** method
  - Your second Hello Again program moved the **printGreeting()** method into the class
  - **System.out.println()** is itself just a pre-defined print method
  - Most classes contain one or more **constructors**
    - These are just special types of methods
  - Almost all classes allow their objects to "do" something
    - Those actions are described in their methods

Recent Method Examples

```
public class HelloWorldRL {

    public static void main(String [] args) {

        printGreeting("Rob", "Lapkass");

    }

    private static void printGreeting(String firstName, String lastName) {

        System.out.println("Hello " + firstName +  " " + lastName +
                           ", good to have you in class");

    }

}
```

Recent Method Examples

```
13 public class HelloRL {
14
15     // instance variables: what the class IS
16     private String firstName;
17     private String lastName;
18
19     // constructor: initializes the class
20     public HelloRL() {
21         firstName = "Anonymous";
22         lastName = "Student";
23     }
24
25     // methods: what a class DOES
26
27     // prints a greeting given first and last names
28     public void printGreeting() {
29         System.out.println("Hello " + firstName +
30                            " " + lastName +
31                            ", good to have you in class.");
32     }
33
34     // updates the first name
35     public void setFirstName(String first) {
36         firstName = first;
37     }
38
39     // updates the last name
40     public void setLastName(String last) {
41         lastName = last;
42     }
43
44 }
```

Methods Motivation

```
public class MethodsInefficient {

    public static void main(String [] args) {

        // data declarations
        final double PI = 3.1415927;
        double r1, area1;
        double r2, area2;
        double r3, area3;
        double r4, area4;
        double r5, area5;

        // data initializations
        r1 = 1.0;
        r2 = 2.0;
        r3 = 3.0;
        r4 = 4.6;
        r5 = 5.8;

        // computations, algorithms
        area1 = PI * r1 * r1;
        area2 = PI * r2 * r2;
        area3 = PI * r3 * r3;
        area4 = PI * r4 * r4;
        area5 = PI * r5 * r5;

        // outputs, display
        System.out.println("r = " + r1 + ", area = " + area1);
        System.out.println("r = " + r2 + ", area = " + area2);
        System.out.println("r = " + r3 + ", area = " + area3);
        System.out.println("r = " + r4 + ", area = " + area4);
        System.out.println("r = " + r5 + ", area = " + area5);

    }

}
```

- Consider the code at left to calculate 5 circle areas from 5 radii
- There are lots of **repetitive operations** in this code:
  - Same actions are performed
  - But, using different data
- This is easy to write using cut-and-paste in jGRASP
- But, can we make this more efficient??
- See ***MethodsInefficient.java*** in **Example Source code**

Improved Method Example

```java
public class MethodsMoreEfficient {

    public static void main(String [] args) {

        // data declarations
        double r1, area1;
        double r2, area2;
        double r3, area3;
        double r4, area4;
        double r5, area5;

        // data initializations
        r1 = 1.0;
        r2 = 2.0;
        r3 = 3.0;
        r4 = 4.6;
        r5 = 5.8;

        // computations, algorithms
        area1 = calculateArea(r1);
        area2 = calculateArea(r2);
        area3 = calculateArea(r3);
        area4 = calculateArea(r4);
        area5 = calculateArea(r5);

        // outputs, display
        System.out.println("r = " + r1 + ", area = " + area1);
        System.out.println("r = " + r2 + ", area = " + area2);
        System.out.println("r = " + r3 + ", area = " + area3);
        System.out.println("r = " + r4 + ", area = " + area4);
        System.out.println("r = " + r5 + ", area = " + area5);

    } // end main
```

```java
    // calculate the area of a circle, given the radius
    public static double calculateArea(double radius) {

        // data declarations
        final double PI = 3.1415927;
        double area;

        // computations
        area = PI * radius * radius;

        // outputs
        return area;

    } // end calculateArea

} // end class
```

- Here, we have encapsulated the area calculation into a new utility method, inside the same file
- There is still repetition here, which we can improve even more
- See ***MethodsMoreEfficient.java*** in **Example Source Code**

More Improved Method Example

```
public static void main(String [] args) {

    // data declarations
    double r;

    // data initialization, computation, display
    // (same variables get reused, by reordering statements)
    r = 1.0;
    displayCircle(r);

    r = 2.0;
    displayCircle(r);

    r = 3.0;
    displayCircle(r);

    r = 4.6;
    displayCircle(r);

    r = 5.8;
    displayCircle(r);

} // end main
```

```
//----------------- utility methods -------------------

// calculate the area of a circle, given the radius
public static double calculateArea(double radius) {

    // data declarations
    final double PI = 3.1415927;
    double area;

    // computations
    area = PI * radius * radius;

    // outputs
    return area;

} // end calculateArea

// display one circle's properties (radius and area)
public static void displayCircle(double radius) {

    // data declaration and calculation
    double area = calculateArea(radius);

    // output display, nothing gets returned
    System.out.println("r = " + radius + ", area = " + area);

    return;

} // end displayCircle
```

- **Avoid the temptation to do TOO much in a method**
  - Keep them short and specific
- Methods can be combined or nested, and you can always create new ones
  - Here, one method just calculates area
  - Here, one method just displays circle parameters (but uses area calculation)
- See ***MethodsStillMoreEfficient.java*** in **Example Source Code**

Method Structure

- A **method** is a **container**
  - For executable instructions (statements)
  - All the statements are enclosed within a **block** (a pair of curly braces)
- A method consists of two main parts:
  - The method **interface**
  - The method **implementation**

method interface {

method implementation

}

Method Interface

```
accessSpecifier(s)  returnType  methodName( argumentList ) {
    // code for the method implementation goes here
    return returnValue;
}

public int addTwoNumbers(int number1, int number2) {
    // code goes here
    return result;
}
public void displayMessage(String messageText) {
    // code goes here
    return;
}
```

accessSpecifier(s): specifies the visibility to outside programs; can it be used/seen by outside code?
(for us, usually ***public*** or ***private***; can also be ***protected***)
[also add ***static*** specifier, if the method is being directly called by another static method like main()]

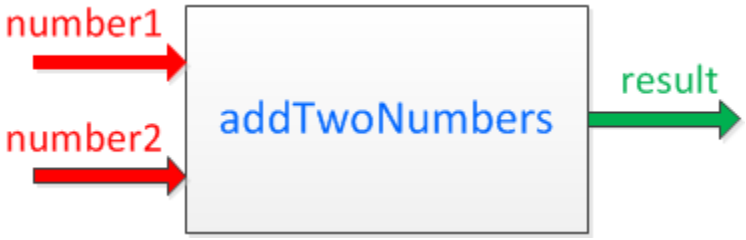
returnType: datatype that the method returns to the caller
(any primitive datatype, a class type, or else ***void***)

methodName the name of the method
(should describe what it DOES, using camelCase naming)

argumentList list of input data to the method
(comma-separated list datatype-variable name pairs)

returnValue variable of type ***returnType*** (primitive or class) returned by method
(if ***returnType*** is ***void***, then this can be left blank)
if a *return* statement is present, it must be the LAST statement of method!

Method Mapping From A Black Box




```
// this method just computes the sum of two numbers
public int addTwoNumbers(int number1, int number2) {

        int result;
        result = number1 + number2;

        return result;

}
```

Method Interface = Contract

- We often speak of the input/output boundaries of a method as its **interface**
  - The **interface** is the first line of a method
  - The rest of a method is its **implementation**
- An **interface** can be viewed as a **software contract**
  - Something that we can count on or rely upon
  - We can **program to** an interface, without worrying about the internal details
  - "If you give me these things, I will give you this"
- An interface is very specific
  - Number, order, and datatypes of inputs
  - However, the actual names of the parameters don't matter
- Familiar real-world examples:
  - A 3-prong electrical outlet
  - A USB port

Method Naming Conventions

- Method names are **identifiers**
  - They must follow all the usual rules for Java identifiers.
- Methods names should observe good Java conventions
  - Method names begin with lower case letters
  - Method names use camelCase naming (same as variables)
  - Methods describe actions, so descriptive verb-centric names are preferred
- Examples:
  - setFirstName(), getFirstName()
  - add(), subtract(), initialize(), shutdown(), withdraw(), query(), stop()
  - turnOffHeater(), startEngine(), serveFood(), pedalBike()
  - solveProblem1(), displayProblem2()

Method Implementation

- A method is just a container for any valid Java code
- Inside the enclosing container {…} braces:
  - The contained code is executed sequentially
  - The method can use the passed-in arguments as variables
    - The interface already takes care of declaring these variables
    - They do NOT have to be declared again inside the method
  - Additional variables can be declared and used within the method
  - Any valid Java statements may be used
  - Class instances (objects) may be instantiated and used
  - Other methods may be called (nested methods)

Variable Scope

- **Variable scope** refers to the extent within which a variable is “visible”, or able to be seen and used by other code
- In general:
  - Variables declared inside a method are only visible within the method in which they are declared
  - Variables used as method arguments can be used only within the method for which they are defined
    - AND, they do not need to be declared again internally
  - Variables declared within a class, but outside any method are visible to ALL methods within that class
    - AND, they should almost always be declared ***private*** to the class

Variable Scope Examples

- ***result*** variables
  - Each is local to its own method
  - Neither knows anything about the other
  - When method completes, it vanishes
- ***numOne***, ***numTwo*** variables
  - Each are local to their own method interfaces
  - Neither knows anything about the one in the other method
  - They can be used internally without needing to declare them again
- ***scaleFactor*** variable
  - Visible throughout entire class ("global"), but not outside it
  - Each method can see and use it
  - Again, no need to redeclare it

```
public class Scope {

   private double scaleFactor = 0.50;

   public double addTwo(int numOne, int numTwo) {
      double result;
      result = (numOne + numTwo) * scaleFactor;
      return result;
   }

    public double subTwo(int numOne, int numTwo) {
      double result;
      result = (numOne – numTwo) * scaleFactor;
      return result;
   }
} // end class
```

See ***Scope.java*** and ***ScopeClient.java*** in **Example Source Code**

Using a Method

- **Using a method** within some application code means providing that method with all the parameters it expects to receive
  - Other names: “calling a method”, “invoking a method”, “method call”, etc.
  - Copies of the parameters are provided, original values are unchanged (“call by value”)
- The parameters provided must match the interface exactly, in terms of:
  - **Number** of arguments
  - **Order** of arguments
  - **Datatypes** of arguments
  - The variable names of the argument DO NOT have to match
- Calling a method differs slightly, depending upon whether you are calling it from within or external to its class
  - Within class: direct method call
  - External to class: must be called in context of an object, with **dot notation**

Method Parameters

- The parameters that are passed to a method may be any valid expression that evaluates to the required datatype
- Examples:
  - Variables
  - Literal values
  - Constants
  - Other method calls
  - Expressions involving any combination of the above
- See examples on next slide…
- See also: ***MethodExamples.java*** and ***MethodExamplesClient.java*** in **Example Source Code**

Using a Method

Internal to the same class:

```
int num1 = 10;
int num2 = 20;
int result;


// all these are valid method calls
result = addTwoNumbers(3, 4);
result = addTwoNumbers(num1, num2);
result = addTwoNumbers(2+5, num1+num2);

// these are identical statements
result = 10 + 20;
result = 10 + addTwoNumbers(5, 15);

//  a void method simply stands alone
restartOverheadProjector();
printGreeting("Joe", "Student");
```

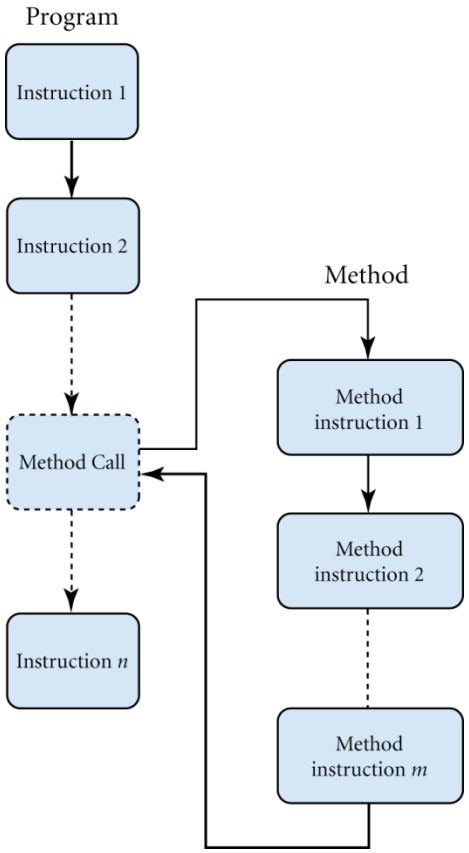
Within external application code:

```
int num1 = 10;
int num2 = 20;
int result;
SomeClass  obj = new SomeClass();

// all these are valid method calls
result = obj.addTwoNumbers(3, 4);
result = obj.addTwoNumbers(num1, num2);
result = obj.addTwoNumbers(2+5, num1+num2);

// these are identical statements
result = 10 + 20;
result = 10 + obj.addTwoNumbers(5, 15);

//  a void method simply stands alone
obj.restartOverheadProjector();
obj.printGreeting("Joe", "Student");
```

Flows of Control

- There are 4 basic flows of control in programming:
  - **Sequential execution**
    - Instructions are performed in linear order, one after another
    - Most of our code to date has been of this form
  - **Method call**
    - A method is encountered
    - Program control shifts to the method, which is evaluated
    - A value may or may not be returned from the method
    - Control resumes at the point the original method was encountered
  - **Selection** (later…)
  - **Looping** (later…)
- Flows of control in a program are very nicely visualized using a **debugger**, such as the one in jGRASP

Method Call Control Flow



Procedure for Creating Methods

1) Start with a problem description
2) Translate the problem into a black-box model
3) Select a descriptive method name
4) Specify inputs – the KNOWNS (#, names, datatypes, order)
5) Specify output – the UNKNOWN (name, datatype)
6) Translate steps 2-6 into a method **interface** and code skeleton
   - Write **pseudocode**, to rough out the method implementation
   - Specify a "**stubbed**" output, so method can at least compile
7) Provide the needed method **implementation**
8) **Course coding standard: comment EACH method briefly**

Pseudocode

- **Pseudocode** is a means of sketching out your program flow in a free-form language, without actually writing any code
  - Consider it the “brainstorming” or “rough draft” of your code
  - You can start writing a program by writing pseudocode as comments inside your method body
  - Then, gradually flesh out the methods with actual Java code
  - Leave some of the pseudocode in place as comments!

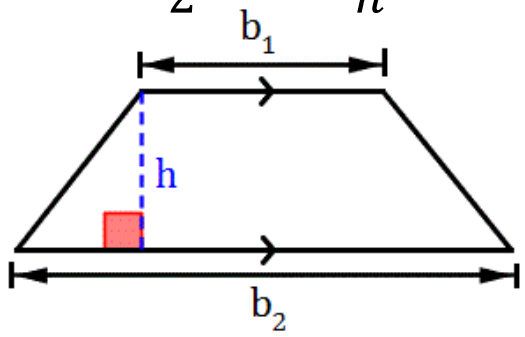
Stubbing Methods

- **Stubbing** methods refers to creating method interfaces and shells which (for now) return bogus values
- A common refrain in software engineering is: **"Nail down your interfaces first"**
  - Specify what to do, not how to do it yet
  - Then, methods/classes can be doled out to team members to implement
- Stubbed methods can be used by their calling applications
  - The results are not correct (yet)
  - But the overall flow of execution begins to take shape
- Examples:
  - For methods which return a numerical value, simply return a 0 or 0.0, or some other recognizable value (i.e. -1, 999)
  - For methods which return a String, return "<TBD>" or similar
  - For methods which return a boolean, return a fixed *true*/*false*

Example of Creating a Method

- 1) Find the area of an isoceles trapezoid, using:

$$A = \frac{1}{2} * \frac{(b1 + b2)}{h}$$



- 2) Black box model:

- 3) Method name is:
  calculateTrapezoidArea()
- 4) 3 inputs/datatypes are:
  double b1, double b2, double h
- 5) 1 output/datatype is:
  double area

Example of Creating a Method (cont.)

- 6) Method interface only, pseudocode, stubbed output:

```
public static double calculateTrapezoidArea(double b1, double b2, double h) {

    // details of this are TBD...

    return 0.0;

}
```

- 7, 8) Method implementation + commenting:

```
// this method calculates the area of a trapezoid, given its dimensions
public static double calculateTrapezoidArea(double b1, double b2, double h) {

    double area;
    area = ((b1 + b2) * h) / 2;

    return area;

} // end method
```

See ***MethodExampleTrapezoid.java*** in **Example Source Code**

For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Assignments**
  - See slide 2 for new/current/past due assignments