Introduction to Methods and Interfaces

CS1: Java Programming
Colorado State University

Kris Brown, Wim Bohm and Ben Say
Methods - motivation

- We want to write a program that manipulates areas of certain 2D shapes
  - rectangles, squares
  - circles, and spheres
- We do not want to write the expression for these areas every time we need to compute one
  - Similarly, we do not want to write one monster main method to do all the work!
  - We want to divide and conquer: separate logical groups of statements together in one construct
Methods

- A **method** allows us to group a set of statements together into a logical operation.
- There are two aspects to methods:
  - The method **definition**
    - A method is a collection of statements that are grouped together to perform an operation.
  - The method **call**
    - Another method can now use the defined method to perform the operation.
Method definition

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

```java
public int areaRec(int length, int width) {
    // compute area of Rectangle
    int area = length * width;
    return area;
}
```

- **Modifier**: determines access level (public, protected, private, etc.)
- **Return**: type and name of the value returned (if any)
- **Method**: name of the method
- **Formal parameters**: variables that represent the arguments passed to the method

Method body, ending with return value;
Calling a Method

A method is called in another piece of code (main or another method). Calling a method:

```java
// definition
class AreaCalculator {
    public int areaRec(int length, int width) {
        // compute area of Rectangle
        int area = length * width;
        return area;
    }
}
```

```java
// example usage
int area = areaRec(5, 7);
```

The **Method signature** is the combination of the method name and the formal parameter list.
Method call: parameter passing

- When a method is called, the values of the **actual** parameters of the caller are passed (copied) to the **formal** parameters of the definition.

  - `areaRec(5, 7)` (in our example) passes 5 to `length` and 7 to `width`
Method return

- 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`.
- When a method call is finished it returns the `returnValue` to the caller. In our example code `int area = areaRec(5, 7)`

  `areaRec(5, 7)` returns 35

Let’s go check out the code . . .
Call Stack

In our example code
main called doRectangularShapes()
and
doRectangularShapes called areaRec(9,5)

When our program gets executed, a run time stack allows records called stack-frames to be stacked up and removed, thereby keeping track of the call history.
main starts

main
args: ....
main calls doRectangularShapes()
doRectangularShapes calls areaRec(9,5)

<table>
<thead>
<tr>
<th>areaRec</th>
</tr>
</thead>
<tbody>
<tr>
<td>length: 9</td>
</tr>
<tr>
<td>width: 5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>doRectangularShapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>area:</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>args: ....</td>
</tr>
</tbody>
</table>
areaRec(9,5) returns 45

doRectangularShapes prints

---

doRectangularShapes

area: 45

main

args: ....

output:

9 by 5 rectangle has area 45
doRectangularShapes calls areaRec(12)

<table>
<thead>
<tr>
<th>areaRec</th>
</tr>
</thead>
<tbody>
<tr>
<td>length:</td>
</tr>
<tr>
<td>width: 12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>doRectangularShapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>area: 45</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>main</th>
</tr>
</thead>
<tbody>
<tr>
<td>args: ....</td>
</tr>
</tbody>
</table>
areaRec calls areaRec(12,12)

<table>
<thead>
<tr>
<th>method</th>
<th>length</th>
<th>width</th>
</tr>
</thead>
<tbody>
<tr>
<td>areaRec</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>areaRec</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

doRectangularShapes

<table>
<thead>
<tr>
<th>method</th>
<th>area</th>
</tr>
</thead>
<tbody>
<tr>
<td>area:</td>
<td>45</td>
</tr>
</tbody>
</table>

main

<table>
<thead>
<tr>
<th>method</th>
<th>args</th>
</tr>
</thead>
<tbody>
<tr>
<td>args:</td>
<td>.....</td>
</tr>
</tbody>
</table>
areaRec(12, 12) returns 144
areaRec(12) returns 144
doRectangularShapes prints

<table>
<thead>
<tr>
<th>doRectangularShapes</th>
</tr>
</thead>
<tbody>
<tr>
<td>area: 144</td>
</tr>
</tbody>
</table>

output:
  square with width 12 has area 144

main
  args: ....
doRectangularShapes returns

main
   args: ....
Your turn!

- Read the program and trace what happens next
- Draw the run time stack with its stack frames for all the call / return events
Pass by Value

The call

\texttt{volumeBlck(10,12,6)}

in

\texttt{doRectangularShapes()}

passes the \texttt{integer values} 10, 12, and 6 to \texttt{volumeBlck}.

This will become relevant later in the course
Overloading

Notice that there are e.g. two methods volumeBlck, with two different method signatures:

public int volumeBlck(int length, int width, int height)
and

public static int volumeBlck(int width)

We call this method overloading. A call will check the number and types of the parameters and select the method with the matching method signature.

E.g. volumeBlck(11) will select

public static int volumeBlck(int width)
Method Abstraction

You can think of the method body as a black box that contains the detailed implementation for the method.
Benefits of Methods

• Write a method once and reuse it anywhere.
• Hide the implementation from the user.
• Reduce complexity (e.g. of main), thereby increasing the readability of your program.
• Simplify maintenance: if the method needs to change, you only change it in one place.
  (and the user does not need to know about it)
Your Turn!

Write two methods that will calculate the perimeter of a rectangle and of a square

```java
public int perimRec(int length, int width)
```

and

```java
public int perimRec(int width)
```
Introduction to Interfaces
Consider the task of writing classes to represent 2D shapes such as Ellipse, Circle, Rectangle and Square. There are certain attributes or operations that are common to all shapes: e.g. their area.

Idea of interface: contract:
"I'm certified as a 2D shape. That means you can be sure that my area can be computed."
Interfaces

- **interface**: A list of methods that a class promises to implement.
- Only method **stubs** (method without a body) and constant declarations in the interface, e.g.
  
  ```java
  public double PI = 3.14159;
  public int areaRec(int length, int width);
  ```
- A class can **implement** an interface
  - A rectangle has an area that can be computed by the method **AreaRec**
  - If a class implements an interface, it must have methods for all methods stubs in the interface.
Implementing an interface

- A class can declare that it implements an interface:

  ```java
  public class <name> implements <interface name> {
    ...
  }
  ```

- This means the class needs to contain an implementation for each of the methods in that interface.
  
  (Otherwise, the class will fail to compile.)

Let’s go look at some code . . .
Your Turn!

You wrote two methods that calculate the perimeter of a rectangle and of a square

    public int perimRec(int length, int width)

and

    public int perimRec(int width)

How does the Interface now change?