Introduction to Methods and Interfaces

CS1: Java Programming
Colorado State University

Kris Brown, Wim Bohm and Ben Say

Methods - motivation

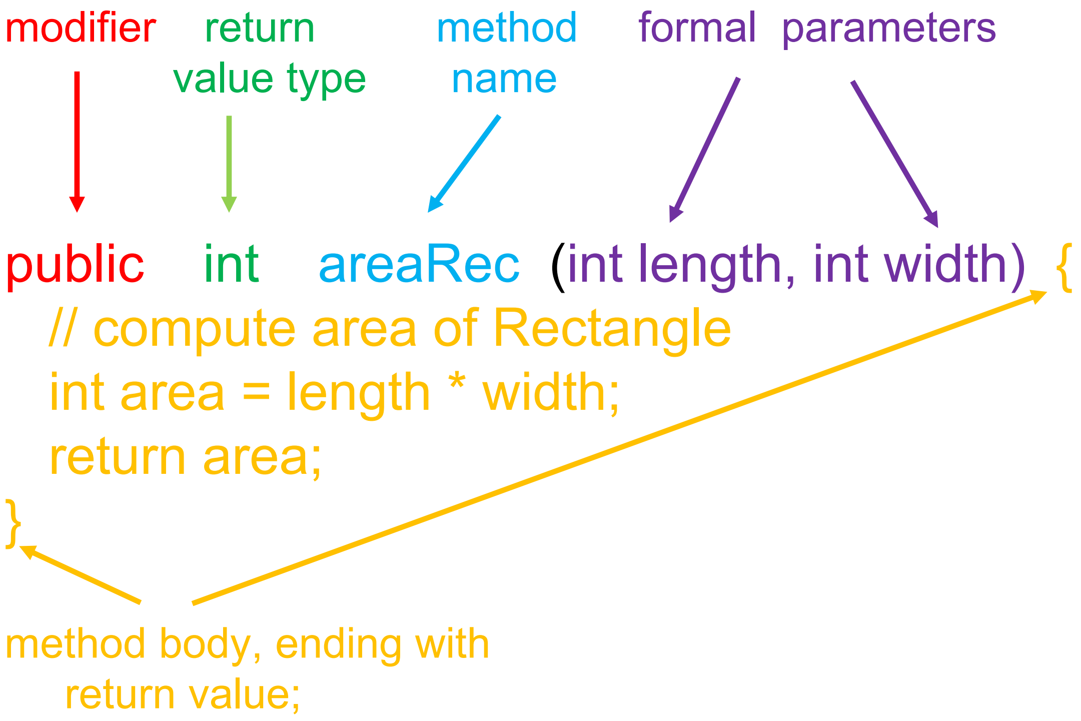
- We want to write a program that manipulates areas of certain 2D shapes
  - rectangles, squares
  - circles, and spheres
- We do not want to write the expression for these areas every time we need to compute one
  - Similarly, we do not want to write one monster main method to do all the work!
  - We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



modifier

return value type

method name

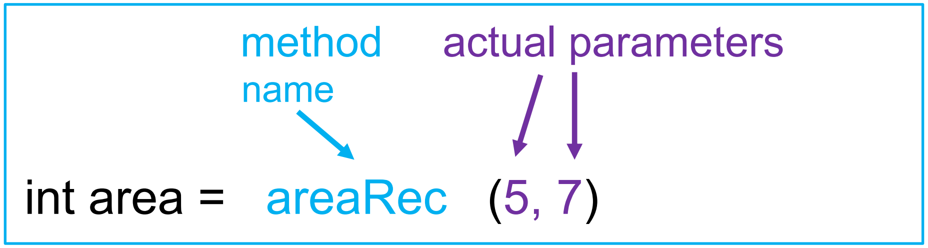
formal parameters

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

method body, ending with return value;

Calling a Method

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



```
int area =   areaRec   (5, 7)
```

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

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

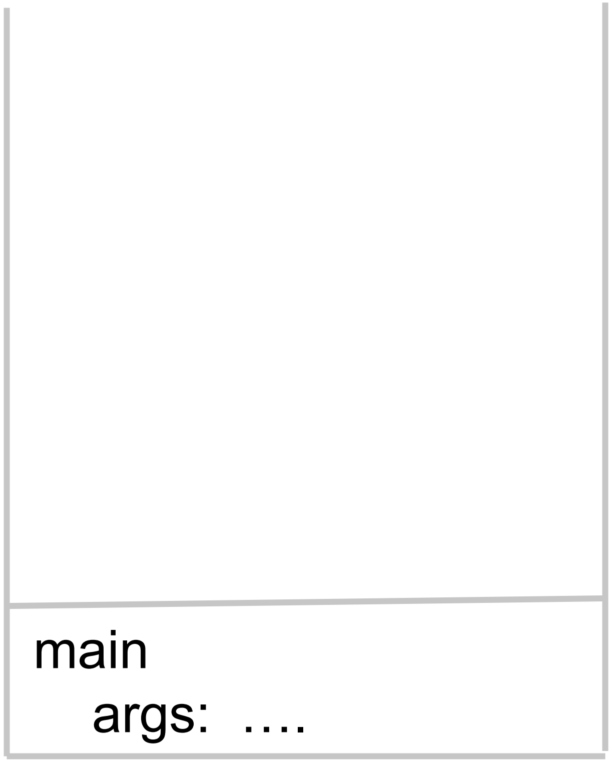
main called doRectangularShapes()

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.

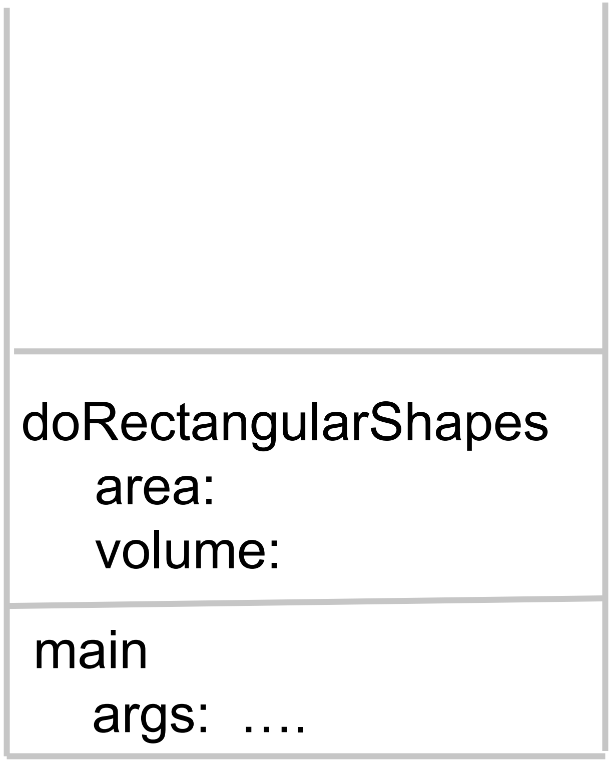
main starts



main

args: ….

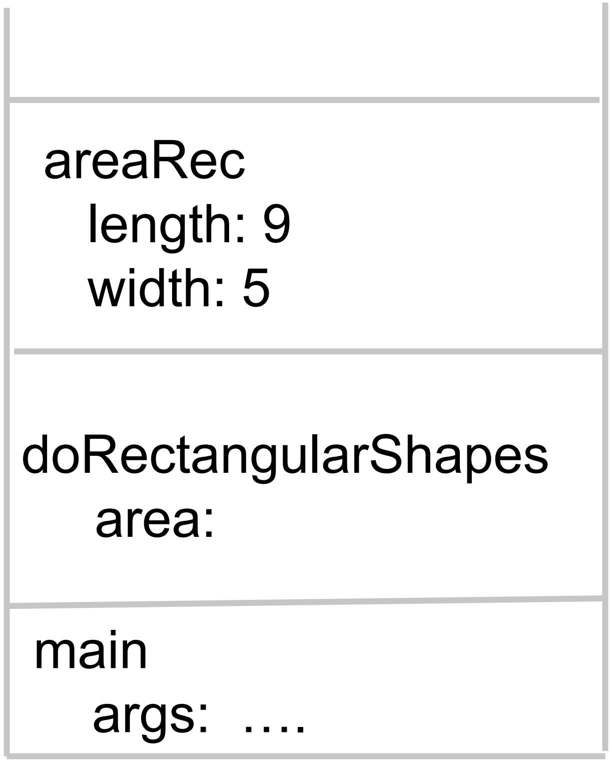
main calls doRectangularShapes()



doRectangularShapes
    area:
    volume:

main
    args: ….

doRectangularShapes calls areaRec(9,5)

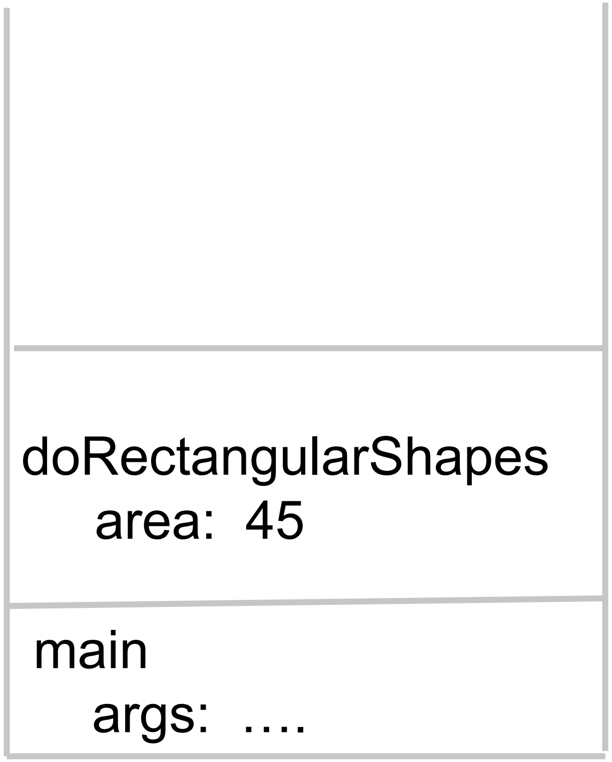


areaRec
  length: 9
  width: 5

doRectangularShapes
  area:

main
  args: ….

areaRec(9,5) returns 45 doRectangularShapes prints

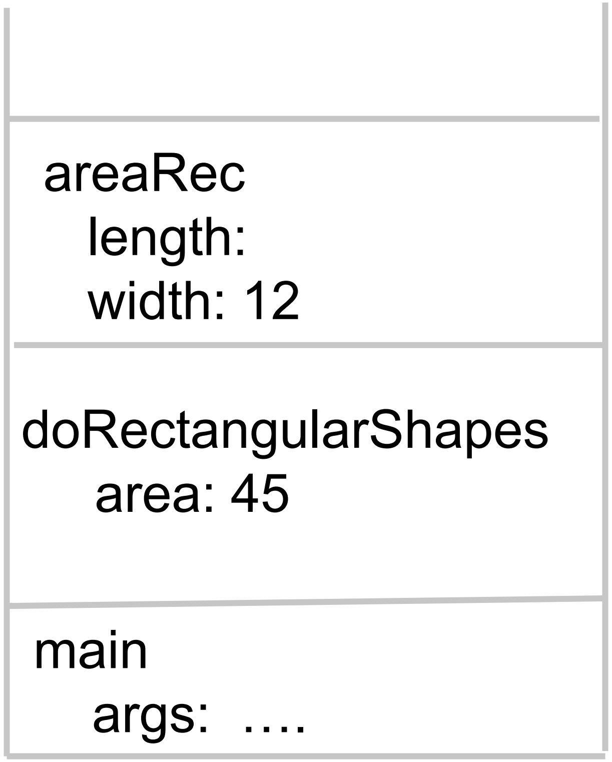


doRectangularShapes
area: 45

main
args: ….

```
output:
   9 by 5 rectangle has area 45
```

doRectangularShapes calls areaRec(12)

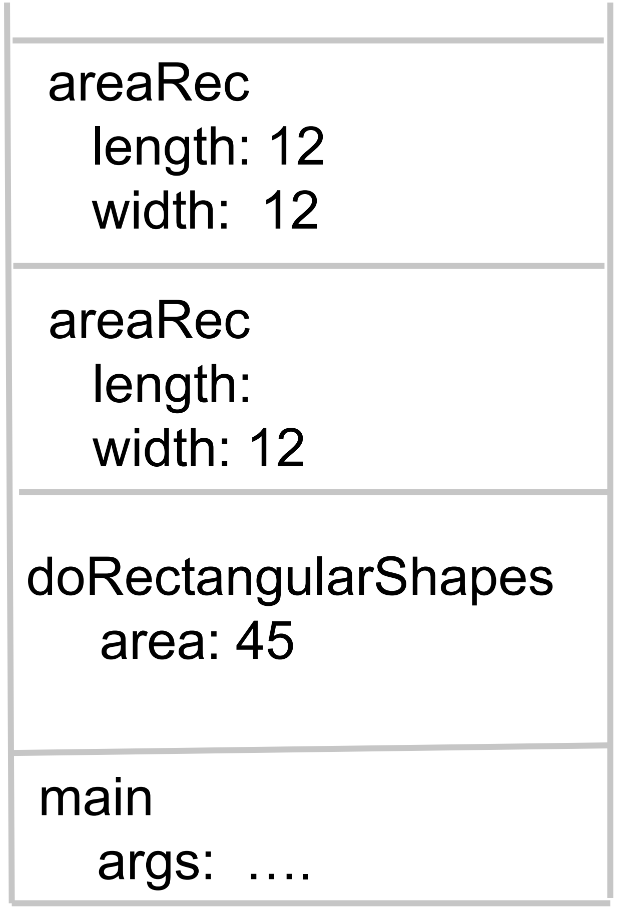


areaRec
  length:
  width: 12

doRectangularShapes
  area: 45

main
  args: ….

areaRec calls areaRec(12,12)



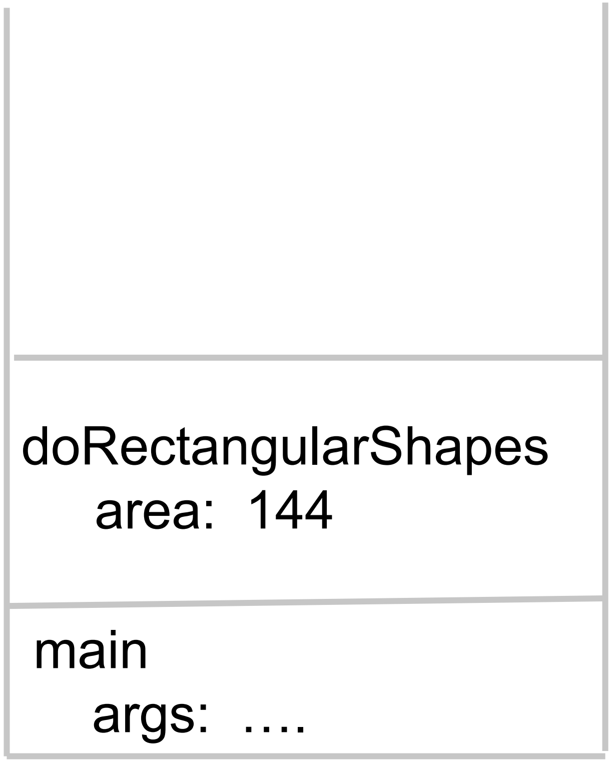
areaRec
length: 12
width: 12

areaRec
length:
width: 12

doRectangularShapes
area: 45

main
args: ….

areaRec(12,12) returns 144
areaRec(12) returns 144
doRectangularShapes prints

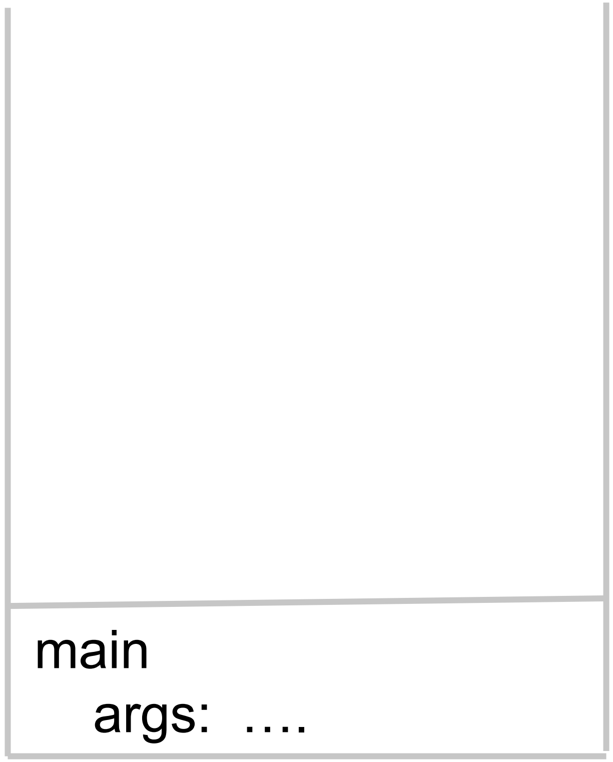


doRectangularShapes
area: 144

main
args: ….

```
output:
   square with width 12 has area 144
```

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

```
     volumeBlck(10,12,6)
```

in

```
    doRectangularShapes()
```

passes the integer values 10, 12, and 6 to 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)
```

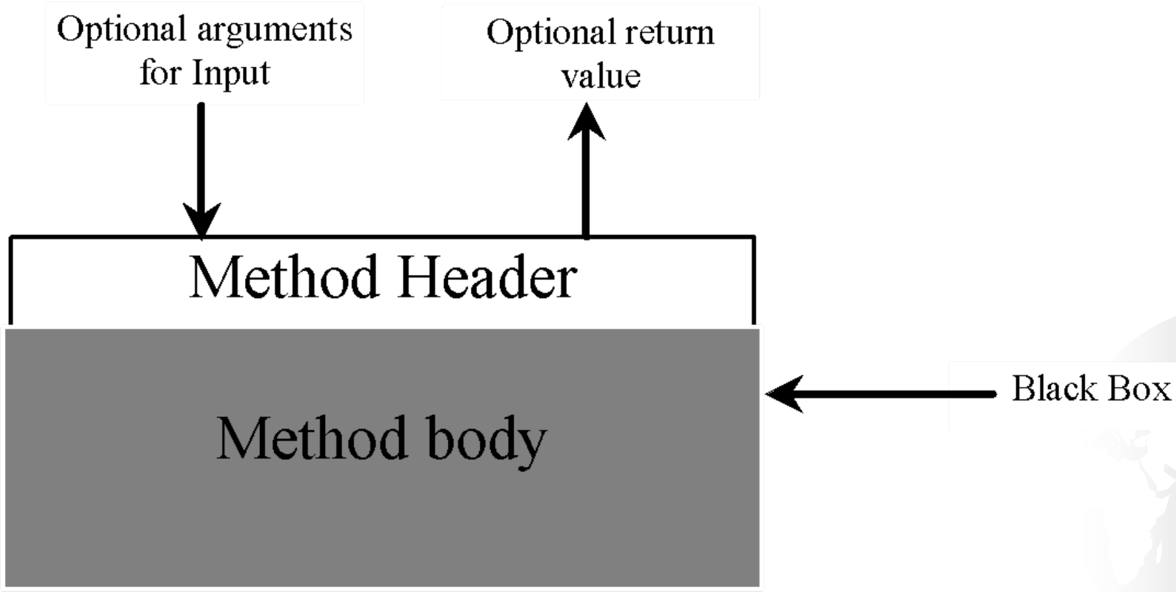
We call this method overloading. A call will check the number and types of the parameters and select the method with the matching method signature.

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), therby 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

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

and

```
public int perimRec(int width)
```

Introduction to Interfaces

Interfaces - motivation

- 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.

    ```
    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:

```
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?