CalPolyPomona

Methods

Lecture 5a

Topics

- Introduction to Methods
- Passing Arguments to a Method
- More About Local Variables
- Returning a Value from a Method

Why Write Methods? (1 of 3)

- Methods are commonly used to break a problem down into small manageable pieces (*divide and conquer)*.
- Also, by simplifying the programs in multiple methods we allow code reuse, a very desired characteristic of Software Engineering.

- Point to ponder #1:

  What means code reuse?

  If a specific task is performed in several places in the program, a method can be written once to perform that task, and then be executed anytime it is needed

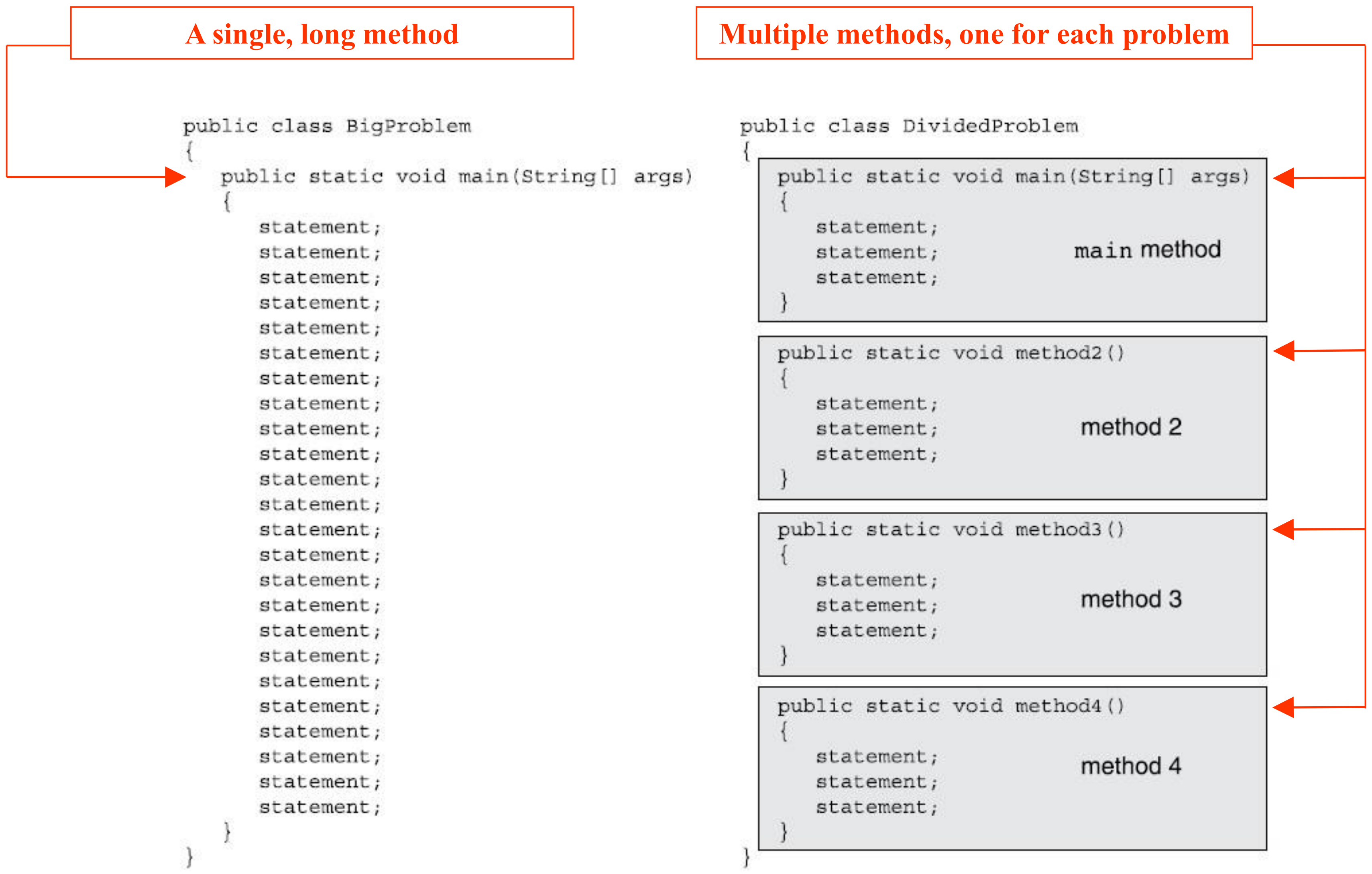
Why Write Methods? (2 of 3)

- Point to ponder #2:

  List relevant benefits of code reuse …

Time, money, security, simplification, organization …

Why Write Methods? (3 of 3)



```
public class BigProblem
{
   public static void main(String[] args)
   {
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
      statement;
   }
}
```

```
public class DividedProblem
{
   public static void main(String[] args)
   {
      statement;
      statement;
      statement;
   }
   public static void method2()
   {
      statement;
      statement;
      statement;
   }
   public static void method3()
   {
      statement;
      statement;
      statement;
   }
   public static void method4()
   {
      statement;
      statement;
      statement;
   }
}
```

`void` Methods and Value-Returning Methods

- A `void` method is one that simply performs a task and then terminates.

```
System.out.println("Hi!");
```

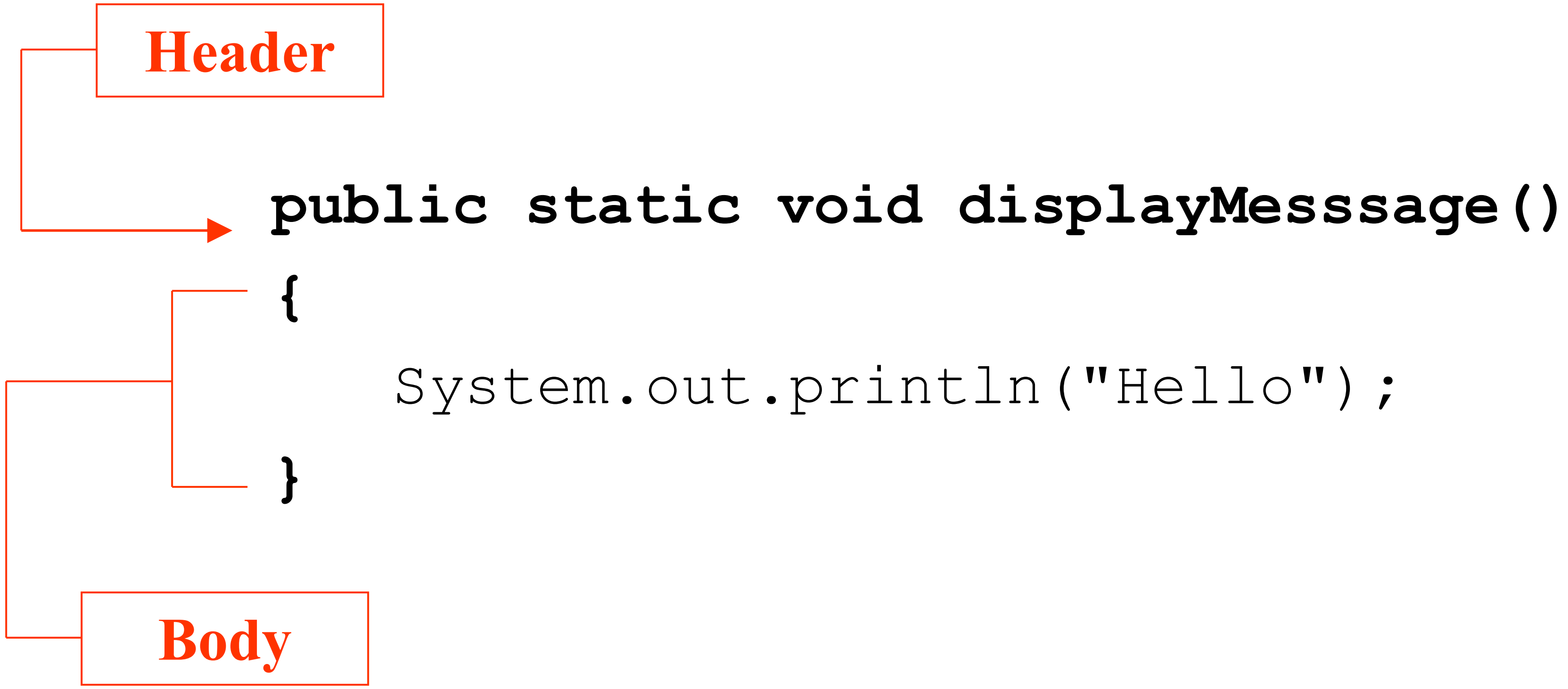
- A value-returning method not only performs a task, but also sends a value back to the code that called it.

```
String text = String.valueOf("700");
char character = "Hello".charAt(0);
```

Defining a `void` Method

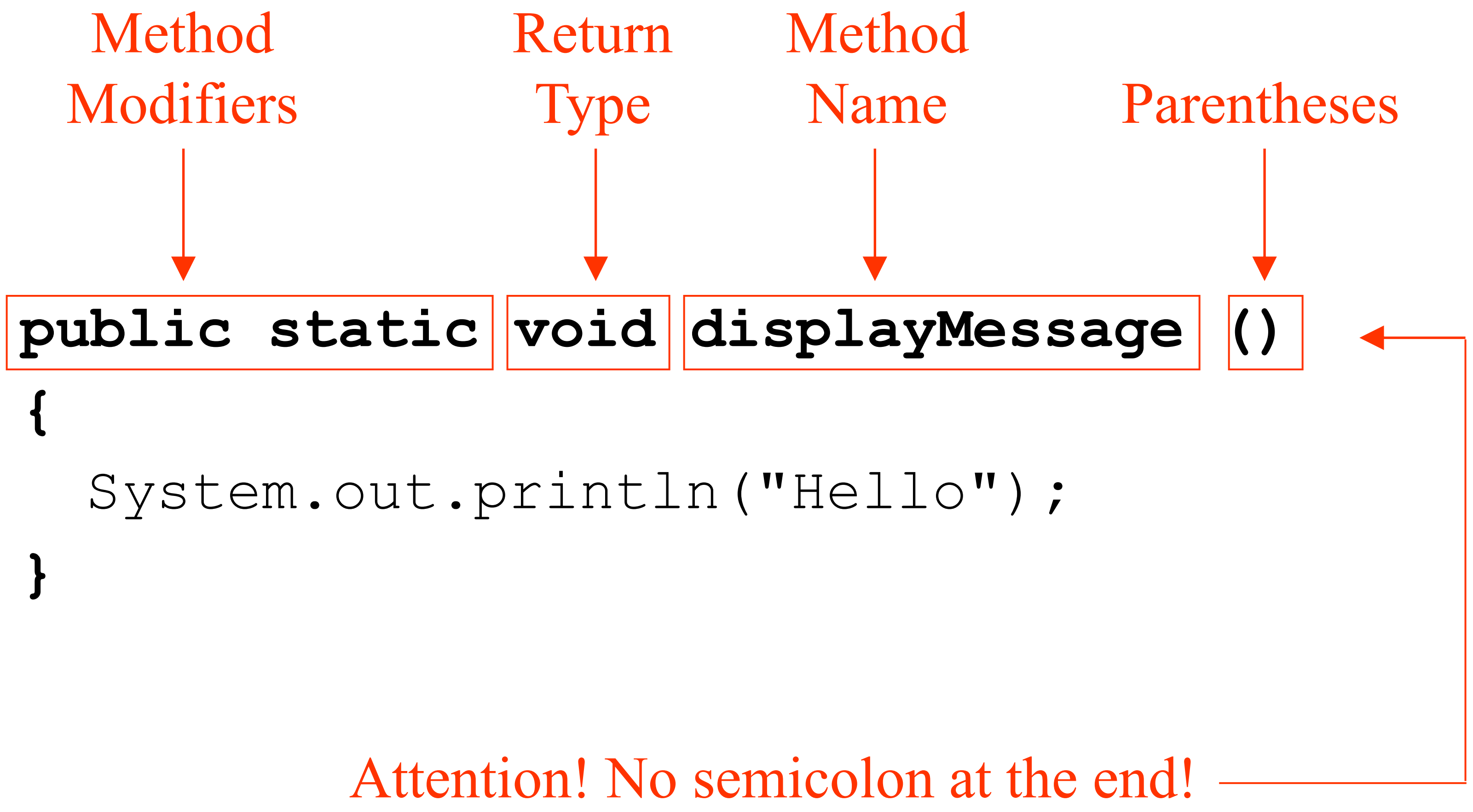
- To create a method, you must write a definition, which consists of a *header* and a *body*.
- The method header, which appears at the beginning of a method definition, lists several important things about the method, including the method's name.
- The method body is a collection of statements that are performed when the method is executed. These statements are enclosed inside a set of curly braces.

Two Parts of Method Declaration



```
public static void displayMesssage()
{
    System.out.println("Hello");
}
```

Parts of a Method Header (1 of 3)



```
public static void displayMessage ()
{
   System.out.println("Hello");
}
```

Parts of a Method Header (2 of 3)

- Method modifiers
  - `public`—method is publicly available to code outside the class
  - `static`—method belongs to a class, not a specific object.
- Return type—`void` (does not return a value) or the data type from a value-returning method
- Method name—name that is descriptive of what the method does
- Parentheses—contain nothing or a list of one or more variable declarations (parameters) if the method is capable of receiving arguments.

Parts of a Method Header (3 of 3)

- Point to ponder #3:
  How many arguments those methods receive?

`println("Hello");`	1
`pow(2,3);`	2
`toUpperCase();`	0

Calling a Method (1 of 5)

- A method executes when it is called.
- The `main` method is automatically called when a program starts, but other methods are executed by method call statements.

```
displayMessage();
```

- Notice that the method modifiers and the `void` return type are not written in the method call statement. Those are only written in the method header.

Calling a Method (2 of 5)

Example: SimpleMethod.java

```
1  /**
2  This program defines and calls a simple method.
3  */
4
5  public class SimpleMethod
6  {
7     public static void main(String[] args)
8     {
9       System.out.println("Hello from the main method.");
10      displayMessage();
11      System.out.println("Back in the main method.");
12    }
13
14    /**
15    The displayMessage method displays a greeting.
16    */
17
18    public static void displayMessage()
19    {
20      System.out.println("Hello from the displayMessage method.");
21    }
22 }
```

The JVM branches to the displayMessage method and executes the statements in its body

Once the displayMessage method has finished executing, the JVM branches back to the main method and resumes at line 11

Calling a Method (3 of 5)

- Point to ponder #4:

  Can we call a method inside loops? ✓

  What about inside if statements? ✓

  What about inside switch statements? ✓

  What about inside another method? ✓

Calling a Method (4 of 5)

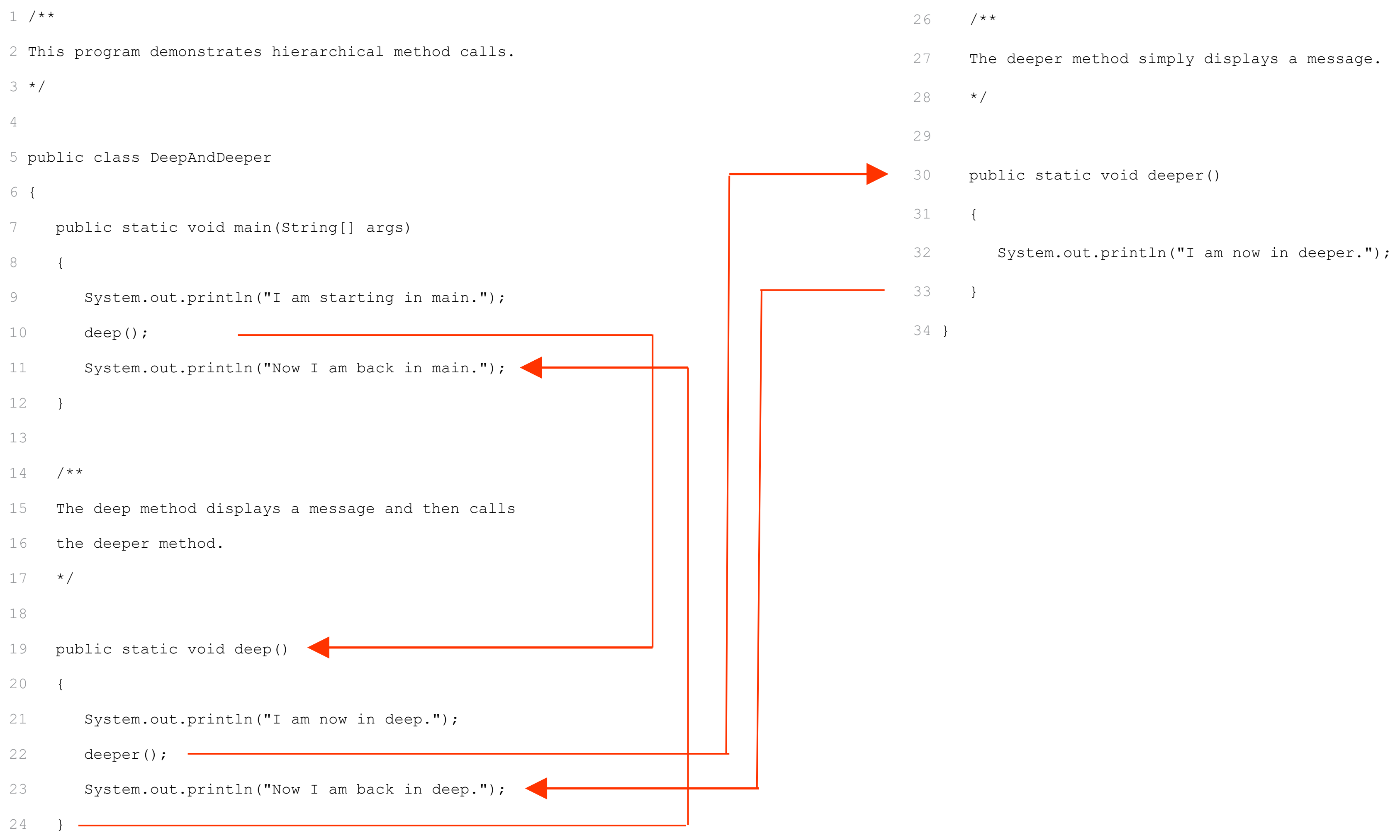
Example: `LoopCall.java`

```java
/**
This program defines and calls a simple method.
*/

public class LoopCall
{
   public static void main(String[] args)
   {
      System.out.println("Hello from the main method.");
      for (int i = 0; i < 5; i++)
           displayMessage();
      System.out.println("Back in the main method.");
   }

   /**
   The displayMessage method displays a greeting.
   */

   public static void displayMessage()
   {
      System.out.println("Hello from the displayMessage method.");
   }
}
```

See: CreditCard.java

Calling a Method (5 of 5)

`Example: DeepAndDeeper.java`



```java
/**
This program demonstrates hierarchical method calls.
*/

public class DeepAndDeeper
{
   public static void main(String[] args)
   {
      System.out.println("I am starting in main.");
      deep();
      System.out.println("Now I am back in main.");
   }

   /**
   The deep method displays a message and then calls
   the deeper method.
   */

   public static void deep()
   {
      System.out.println("I am now in deep.");
      deeper();
      System.out.println("Now I am back in deep.");
   }
   /**
   The deeper method simply displays a message.
   */

   public static void deeper()
   {
      System.out.println("I am now in deeper.");
   }
}
```

Documenting Methods

- A method should always be documented by writing comments that appear just before the method's definition.
- The comments should provide a brief explanation of the method's purpose.
- The documentation comments begin with `/**` and end with `*/`.
- These types of comments can be read and processed by a program named javadoc, which produces attractive HTML documentation.

Passing Arguments to a Method

- Values that are sent into a method are called arguments.

```
System.out.println("Hello");
number = Integer.parseInt(str);
```

- The data type of an argument in a method call must correspond to the variable declaration in the parentheses of the method declaration. The parameter is the variable that holds the value being passed into a method.
- By using parameter variables in your method declarations, you can design your own methods that accept data this way.

Passing Arguments to a Method

- You may pass literals as arguments.

```
displayValue(5);
```

The argument 5 is copied into the parameter variable `num`.

```
public static void displayValue(int num)
{
 System.out.println("The value is " + num);
}
```

The method will display the value 5

Passing Arguments to a Method

- You may also pass the contents of variables and the values of expressions as arguments.

```
int x = 3;
displayValue(x);
displayValue(2+4);


public static void displayValue(int num)
{
 System.out.println("The value is " + num);
}
```

The method will display the values 3 and 6

See: PassArg.java

Argument and Parameter Data Type Compatibility

- When you pass an argument to a method, be sure that the argument's data type is compatible with the parameter variable's data type.
- Java will automatically perform widening conversions but narrowing conversions will cause a compiler error.

```
double d = 1.0;
displayValue(d);
```

Error! Can't convert `double` to `int`

Parameter Variable Scope

- Point to ponder #5:

  Where is the problem here?

```
public static void main(String[] args) {
     showSum(5, 10);
     System.out.println(num1 + num2);
}
public static void showSum(double num1, double num2)
{
     System.out.print("The sum is ");
}
```

A parameter variable's scope is the method in which the parameter is declared. No statement outside the method can access the parameter variable by its name.

Passing Multiple Arguments

- Often it is useful to pass more than one argument to a method (parameter list)

The argument 5 is copied into the `num1` parameter.

The argument 10 is copied into the `num2` parameter.

```
showSum(5, 10);
```

NOTE: Order matters!

```
public static void showSum(double num1, double num2)
{
   double sum;     //to hold the sum
   sum = num1 + num2;
   System.out.println("The sum is " + sum);
}
```

Passing Multiple Arguments

- Point to ponder #6:

  Where is the problem here?

```
divide(10, 5); //the method should return 10 divided by 5
```

Wrong order of arguments!

```
public static void divide(int divisor, int dividend)
{
   double quotient = dividend / divisor;
   System.out.println("The quotient of this division is
     " + quotient);
}
```

Output: 0.0

Methods

Lecture 5b

Topics

- Introduction to Methods
- Passing Arguments to a Method
- More About Local Variables
- Returning a Value from a Method

Arguments are Passed by Value

- In Java, all arguments of the primitive data types are *passed by value*, which means that only a copy of an argument's value is passed into a parameter variable.
- A method's parameter variables are separate and distinct from the arguments that are listed inside the parentheses of a method call.
- If a parameter variable is changed inside a method, it has no affect on the original argument.
- See example: PassByValue.java

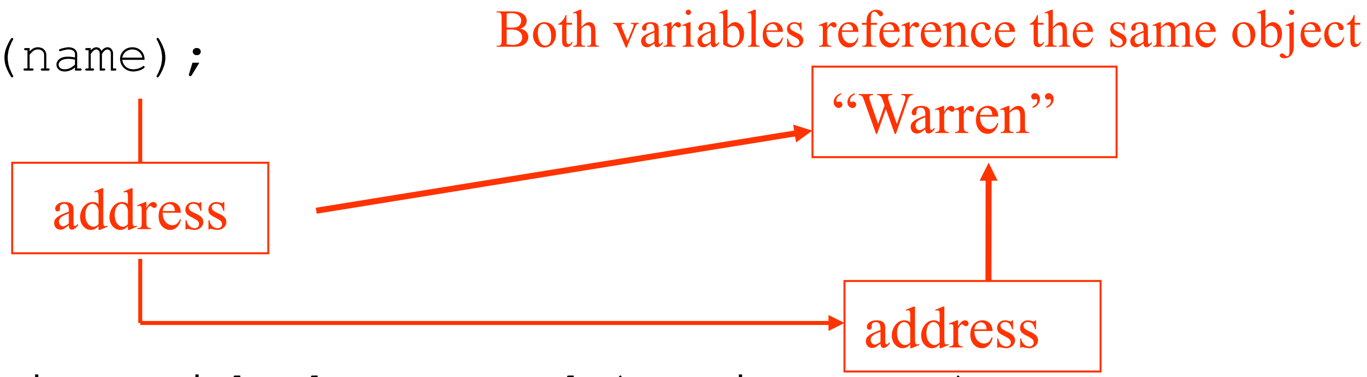
Passing Object References to a Method

- Recall that a class type variable does not hold the actual data item that is associated with it but holds the memory address of the object. A variable associated with an object is called a reference variable.

- When an object such as a `String` is passed as an argument, it is actually a reference to the object that is passed.

Passing a Reference as an Argument

```
showLength(name);
```



```
public static void showLength(String str)
{
 System.out.println(str + " is " + str.length()
          + " characters long.");
 str = "Joe" // see next slide
}
```

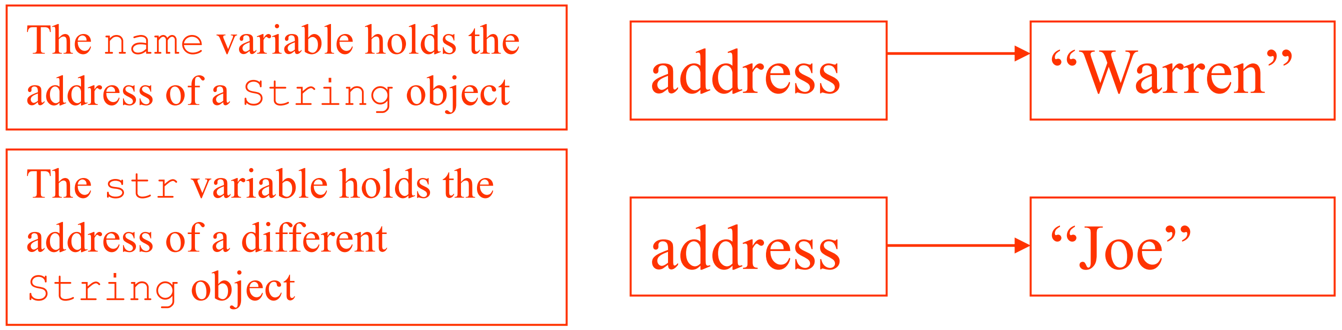
The address of the object is copied into the **`str`** parameter.

`String`s are Immutable Objects

- `String`s are immutable objects, which means that they cannot be changed. When the line

  ```
  str = "Joe";
  ```

  is executed, it cannot change an immutable object, so creates a new object.



The name variable holds the address of a String object
address
"Warren"
The str variable holds the address of a different String object
address
"Joe"

- See example: PassString.java

`@param` Tag in Documentation Comments

- You can provide a description of each parameter in your documentation comments by using the `@param` tag.
- General format

```
@param parameterName Description
```

- See example: [TwoArgs2.java]()
- All `@param` tags in a method's documentation comment must appear after the general description. The description can span several lines.

More About Local Variables

- A local variable is declared inside a method and is not accessible to statements outside the method.
- Different methods can have local variables with the same names because the methods cannot see each other’s local variables.
- A method’s local variables exist only while the method is executing. When the method ends, the local variables and parameter variables are destroyed, and any values stored are lost.
- Local variables are not automatically initialized with a default value and must be given a value before they can be used.
- See example: LocalVars.java

Returning a Value from a Method

- Data can be passed into a method by way of the parameter variables. Data may also be returned from a method, back to the statement that called it.

```
int num = Integer.parseInt("700");
```

- The string “700” is passed into the `parseInt` method.
- The `int` value 700 is returned from the method and assigned to the `num` variable.

Defining a Value-Returning Method

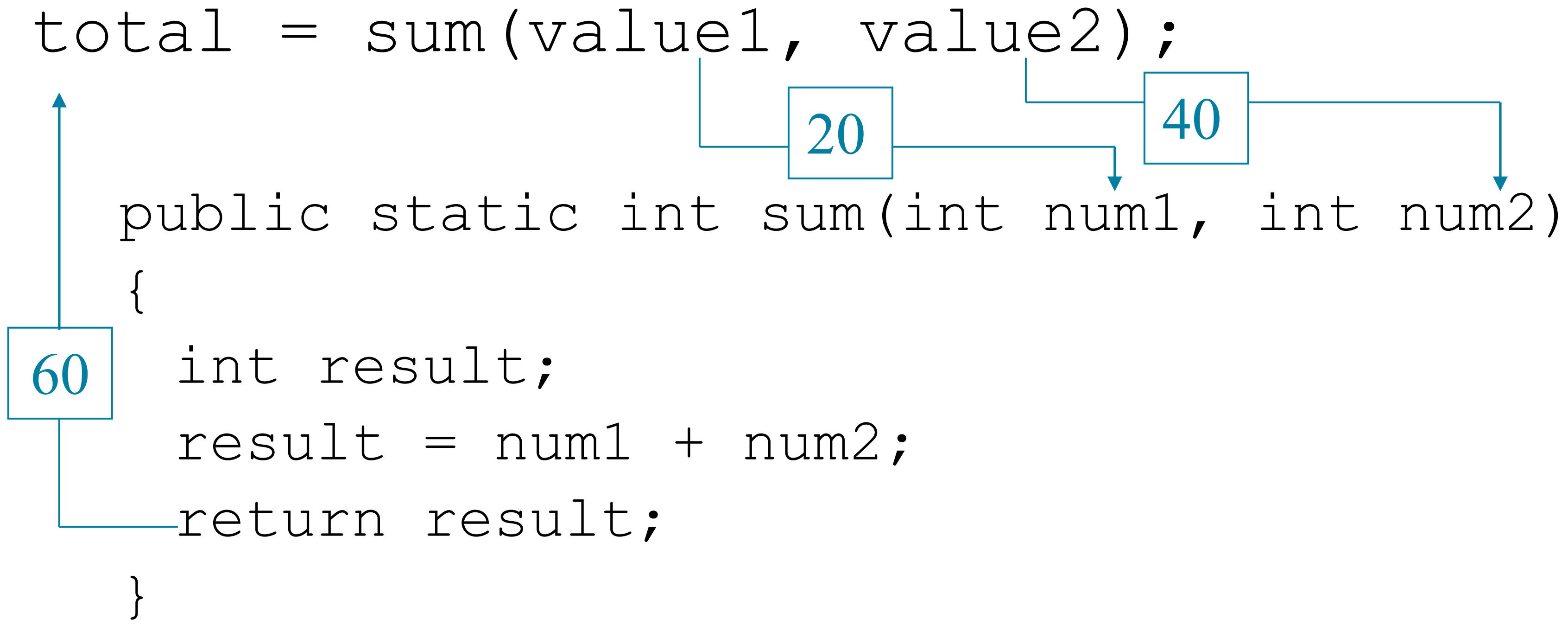
```
public static int sum(int num1, int num2)
{
 int result;
 result = num1 + num2;
 return result;
}
```

Return type

This expression must be of the same data type as the return type

The `return` statement causes the method to end execution and it returns a value back to the statement that called the method.

Calling a Value-Returning Method



```
total = sum(value1, value2);

  public static int sum(int num1, int num2)
  {
    int result;
    result = num1 + num2;
    return result;
  }
```

`@return` Tag in Documentation Comments

- You can provide a description of the return value in your documentation comments by using the `@return` tag.
- General format

```
    @return Description
```

- See example: ValueReturn.java
- The `@return` tag in a method's documentation comment must appear after the general description. The description can span several lines.

Returning a `boolean` Value

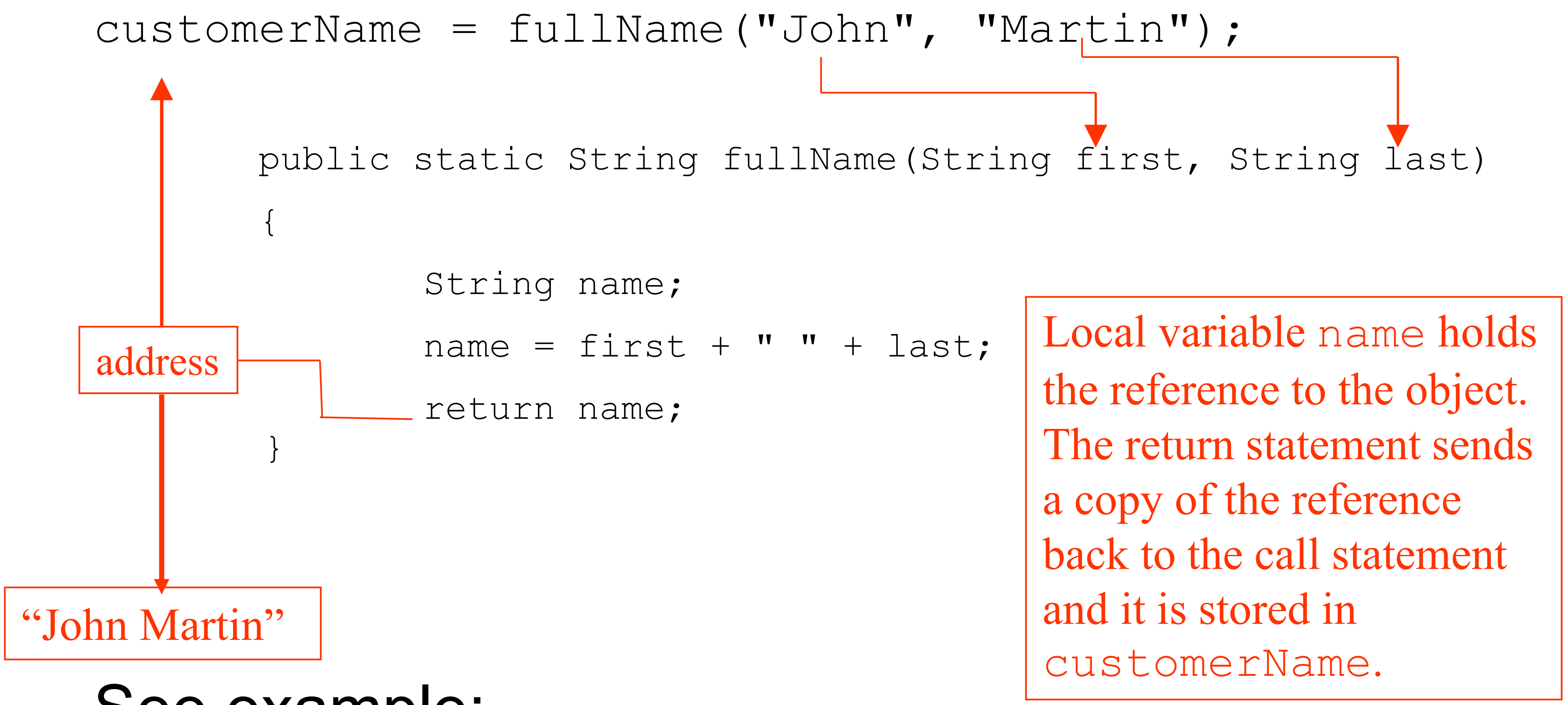
- Sometimes we need to write methods to test arguments for validity and return true or false

```
public static boolean isValid(int number)
{
   boolean status;
   if(number >= 1 && number <= 100)
      status = true;
   else
      status = false;
   return status;
}
```

Calling code:

```
int value = 20;
if(isValid(value))
   System.out.println("The value is within range");
else
   System.out.println("The value is out of range");
```

Returning a Reference to a `String` Object



```
customerName = fullName("John", "Martin");
```

```
public static String fullName(String first, String last)
{
        String name;
        name = first + " " + last;
        return name;
}
```

Local variable `name` holds the reference to the object. The return statement sends a copy of the reference back to the call statement and it is stored in `customerName`.

See example:

[ReturnString.java]()