

# Programming 1

Week 09 – Methods

# Why Write Methods?

- Methods are commonly used to break a problem down into small manageable pieces.
- Methods simplify programs. 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. This is known as **code reuse**.

# Why Write Methods?

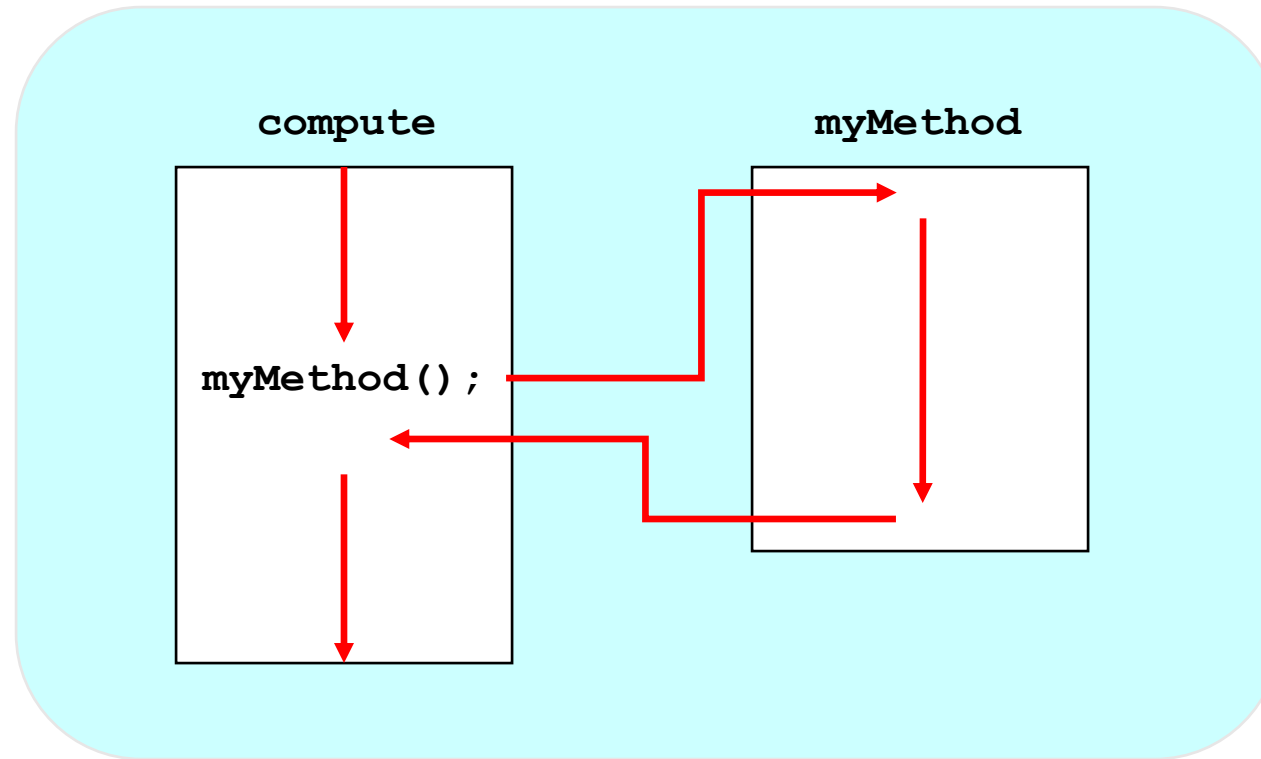
- A method is a collection of statements that performs a specific task. So far you have experienced methods in two ways:
  1. You have created a method named **main** in every program you've written
  2. you have executed predefined methods from the Java API, such as `System.out.println()`, and `Math.pow()`.
- Today, you will learn how to create your own methods, other than **main**, that can be executed just as you execute the API methods

# Method Declarations

- Let's now examine methods in more detail
- A *method declaration* specifies the code that will be executed when the method is invoked (called)
- When a method is invoked, the flow of control jumps to the method and executes its code
- When complete, the flow returns to the place where the method was called and continues
- The invocation may or may not return a value, depending on how the method is defined

# Method Control Flow

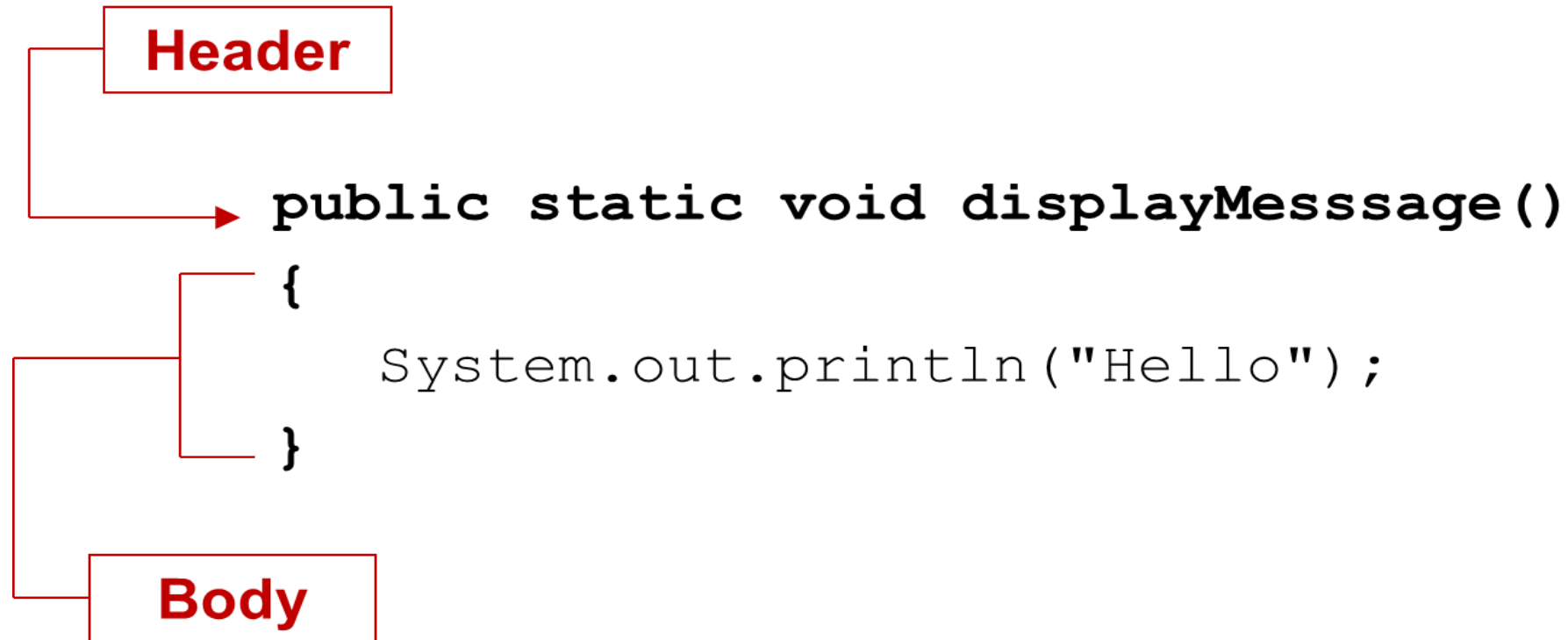
- If the called method is in the same class, only the method name is needed



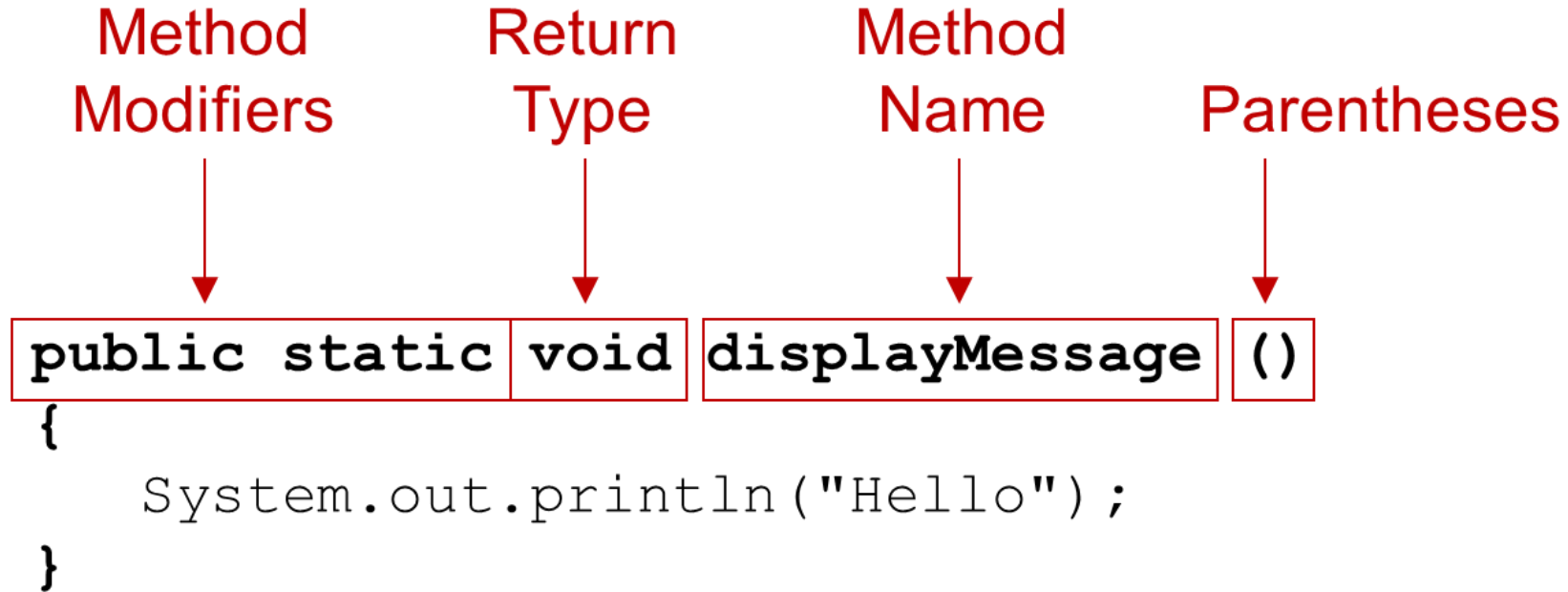
# Method Declarations

- 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 between a pair of curly braces that are performed when the method is executed. Also called **implementation**

# Two Parts of Method Declaration



# Parts of a Method Header (1 of 2)





# Parts of a Method Header (2 of 2)

- Method modifiers
  - `public`—method is publicly available to code outside the class. Everyone has access to the method.
  - `static`—method belongs to a class, not a specific object.
- Return type—`void` 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 if the method is capable of receiving arguments.

# Calling a Method

- 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.

# Demo

```
public static void main(String[] args)
{
    System.out.println("Hello from the main method.");
    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.");
}
```

# Demo..

- Method call statements may be used in control structures like loops, if statements, and switch statements.

```
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.");
}
```

# Passing Arguments to a Method

- A method may be written so it accepts arguments. Data can then be passed into the method when it is called.
- Values that are sent into a method are called arguments.

```
System.out.println("Hello");
```

- 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 5 to the `displayValue` Method

`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 is 5

# 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**

# Demo

```
int x = 10;
System.out.println("I am passing values to displayValue.");
displayValue(5);           // Pass 5
displayValue(x);           // Pass 10
displayValue(x * 4);       // Pass 40
displayValue(Integer.parseInt("700")); // Pass 700
System.out.println("Now I am back in main.");
}

/**
    The displayValue method displays the value
    of its integer parameter.
*/

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



# Passing Multiple Arguments


Argument  
order  
matters!

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

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

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

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

A diagram consisting of a horizontal red line with two vertical red lines extending downwards from it, ending in red arrows. The first arrow points to the 'num1' parameter in the method signature, and the second arrow points to the 'num2' parameter. This diagram illustrates how the arguments '5' and '10' from the function call are mapped to the parameters 'num1' and 'num2' respectively.

# 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.

# 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.

# 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.

# Returning a Value from a Method

- The *return type* of a method indicates the type of value that the method sends back to the calling location
- A method that does not return a value has a `void` return type
- A *return statement* specifies the value that will be returned  
`return expression;`
- Its expression must conform to the return type

# Returning a Value from a Method

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

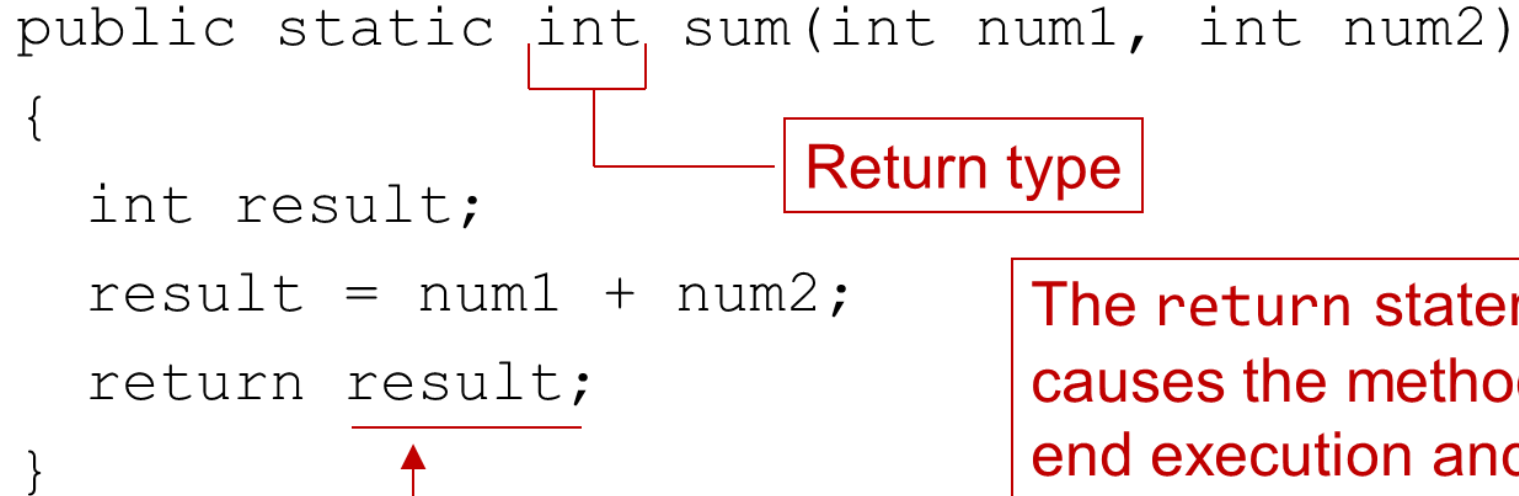
```
System.out.println("Hi!");
```

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

```
int number = input.nextInt();
```

# Defining a Value-Returning Method

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

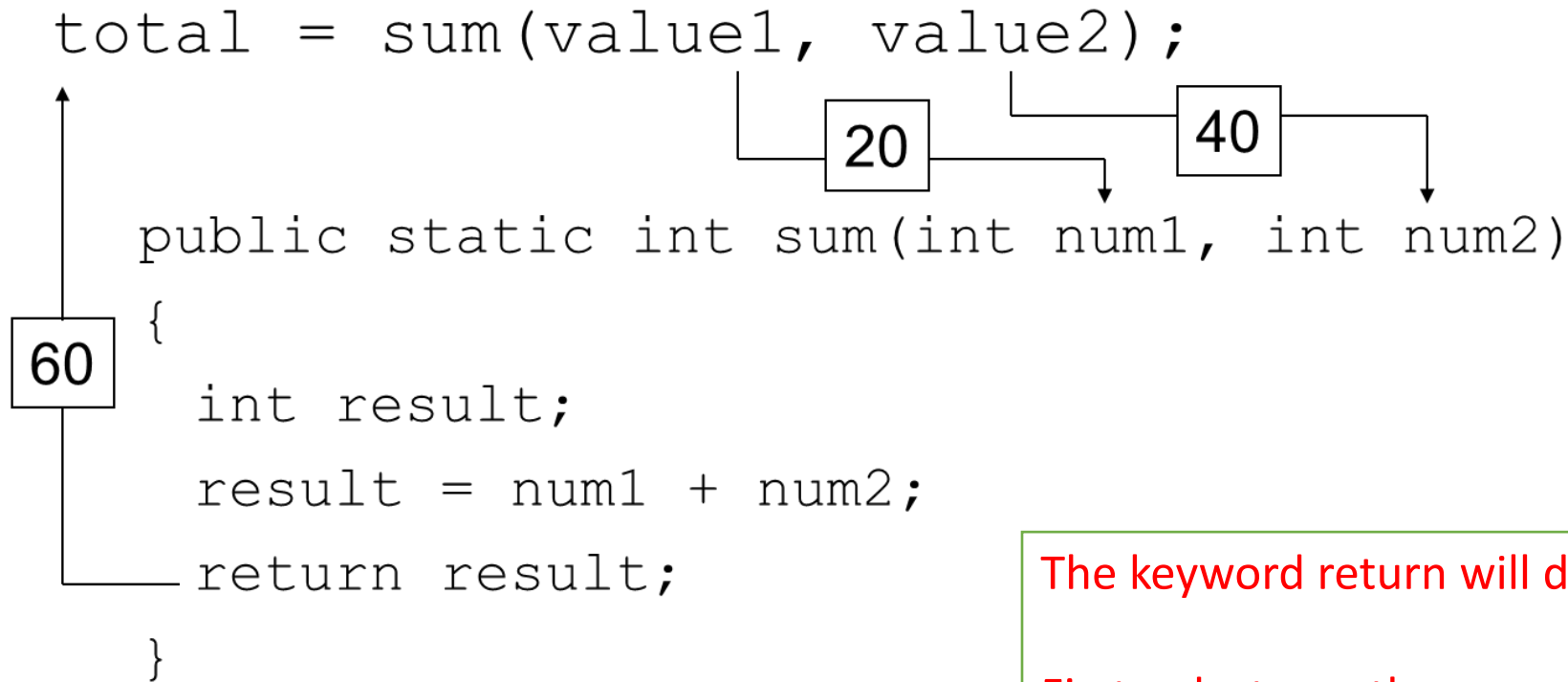


Return type

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



The keyword `return` will do two things:

First, destroy the current method, so all variables declared inside (`num1`, `num2`) will be destroyed.

Second, bring the value behind it back to where the method is called (line 2), and use



## Another Value-Returning Methods Example

- A method declaration begins with a *method header*

```
char calc(int num1, int num2, String message)
```

The diagram shows the method declaration `char calc(int num1, int num2, String message)`. A red arrow points from the text **return type** to the `char` keyword. Another red arrow points from the text **method name** to the `calc` identifier. A red curly brace spans the entire parameter list `(int num1, int num2, String message)`, with the text **parameter list** centered below it.

**return type**

**method name**

**parameter list**

**The parameter list specifies the type and name of each parameter**


**The name of a parameter in the method declaration is called a *formal parameter***

## Another Value-Returning Methods Example

- The method header is followed by the *method body*

```
char calc(int num1, int num2, String message)
{
    int sum = num1 + num2;
    char result = message.charAt(sum);

    return result;
}
```

The return expression  
must be consistent with  
the return type

sum **and** result  
are local data

They are created  
each time the  
method is called, and  
are destroyed when  
it finishes executing

# Demo

```
/**
    The sum method returns the sum of its two parameters.
    @param num1 The first number to be added.
    @param num2 The second number to be added.
    @return The sum of num1 and num2.
*/

public static int sum(int num1, int num2)
{
    int result;    // result is a local variable

    // Assign the value of num1 + num2 to result.
    result = num1 + num2;

    // Return the value in the result variable.
    return result;
}
```

# Returning a `boolean` Value (1 of 2)

- 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;
}
```

# Returning a `boolean` Value (2 of 2)

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");
```

# Overload Methods

- Java does not allow to have two methods with the same name and the same formal parameter list in a class since this will create ambiguous for Java.
- In the String class, we have seen three different versions of the `indexOf()` method.
  - `indexOf(char c)`: takes a char and then return the index of it in a string, the searching starts at the first character of the string.
  - `indexOf(String str)`: takes a String and then return the index of it in a string , the searching starts at the first character of the string.
  - `indexOf(char c, int startIdx)`: takes a char and then return the index of it in a string, the searching starts at the startIdx.

# Overload Methods

- All the three methods belongs to the String class. (they are at the same class)
- The three methods have exactly the same name. (`indexOf`)
- But the formal parameter list of the three methods are different
  - `char`
  - `String`
  - `char, int`

# Overload Methods

- Java allows different methods with the same name but different parameter list in the same class. These methods are called "overload" methods.
- When a method is overloaded, it means that multiple methods in the same class have the same name but use different types of parameters. Method overloading is important because sometimes you need several different ways to perform the same operation
- Java uses a method's signature to distinguish it from other methods of the same name. A method's signature consists of the method's name and the data types of the method's parameters, in the order that they appear.



# Demo

```
public static double max(double num1, double num2) {  
    return Math.max(num1, num2);  
}  
  
public static char max(char c1, char c2) {  
    return max(c1, c2);  
}  
  
public static double max(double num1, double num2, double num3) {  
    return Math.max(Math.max(num1, num2), num3);  
}
```

# Demo

- Write 2 methods named square where the first method accepts an integer argument and returns the square of that argument, and the second method accepts a double argument and returns the square of that argument.

# Passing Object References to a Method

- Recall that a reference variable does not hold the actual data item that is associated with it but holds the memory address of the object.
- 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) ;
```

address

Both variables reference the same object

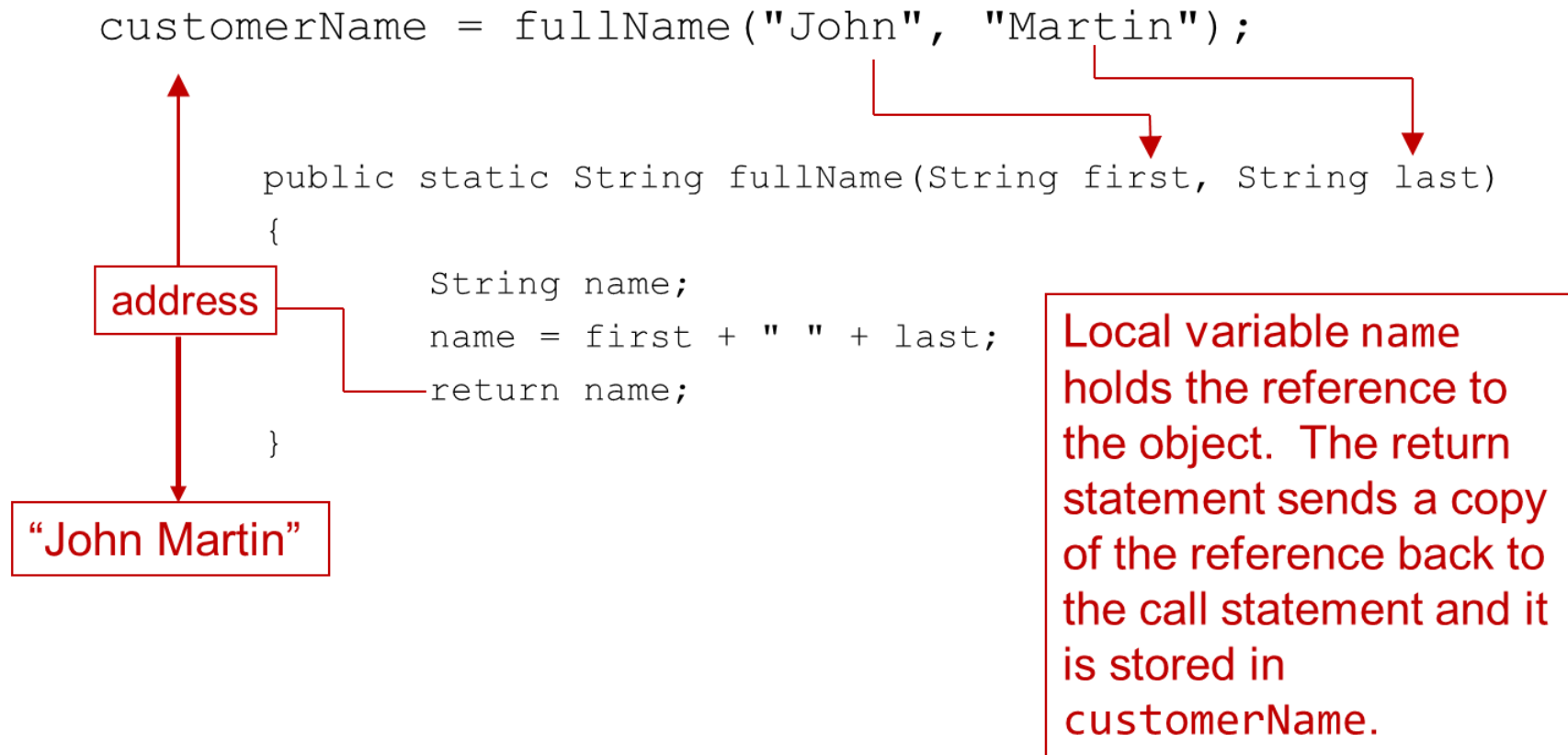
"Warren"

address

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

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

# Returning a Reference to a String Object

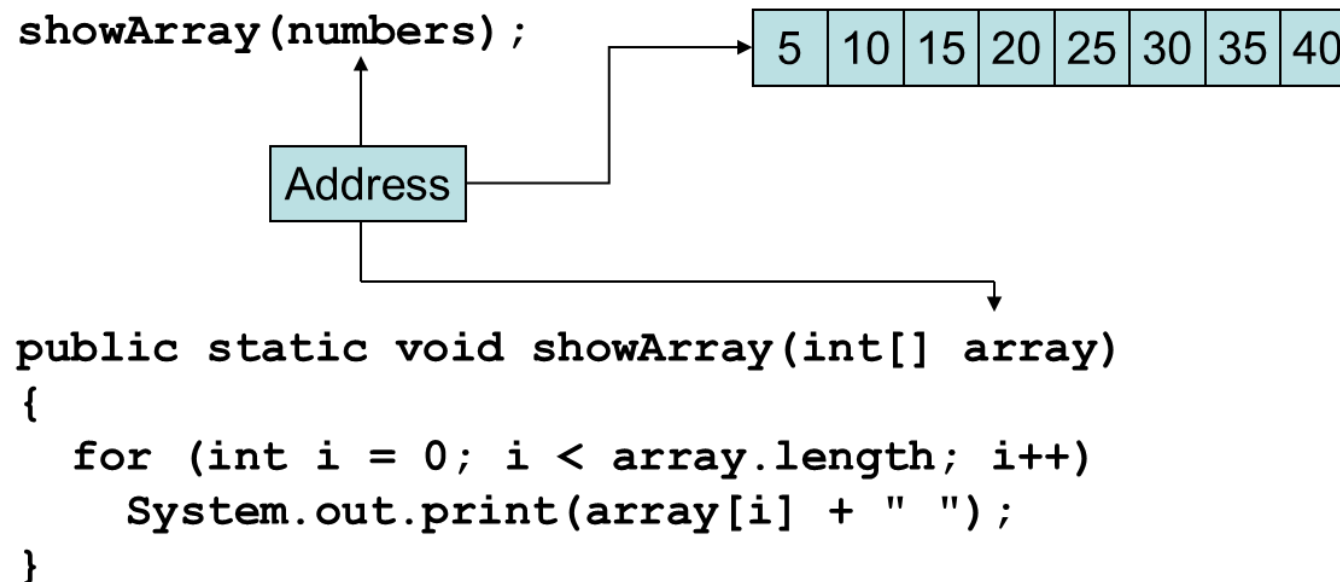


# Passing Array Elements to a Method

- When a single element of an array is passed to a method it is handled like any other variable.
- More often you want to write methods to process array data by passing the entire array, not just one element at a time.

# Passing Arrays as Arguments

- Arrays are objects.
- Their references can be passed to methods like any other object reference variable.



# Returning Arrays

```
class ArrayDemo{
public static void main(String[] args)
{
    int[] storage = methodReturningArray();
    // Printing the elements of the array
    for (int i = 0; i < storage.length; i++)
        System.out.print(storage[i] + " ");
}

public static int[] methodReturningArray()
{
    int[] sample = { 1, 2, 3, 4 };
    return sample;
}
}
```



# Demo

- Write a method name `getIndexOfMax( int[ ] a )` that returns the index of the maximum element of `a`.

# 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 `*/`.

# @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**

- All @param tags in a method's documentation comment must appear after the general description. The description can span several lines.
- If you have more than one parameter, you need @param for each of them. Remember, in the documentation, you should explain what exactly are the parameters and return value. (e.g.: @param num1 the first input number)

# @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**

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