CHAPTER 5

Methods

Why Write Methods?

- Methods are commonly used to break a problem down into small manageable pieces. This is called *divide and conquer*.
- 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*.

Chapter Topics

Chapter 5 discusses the following main topics:

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

Why Write Methods?

- Methods are commonly used to break a problem down into small manageable pieces. This is called *divide and conquer*.
- 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*.

`void` Methods and Value-Returning Methods

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

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

class object method argument

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

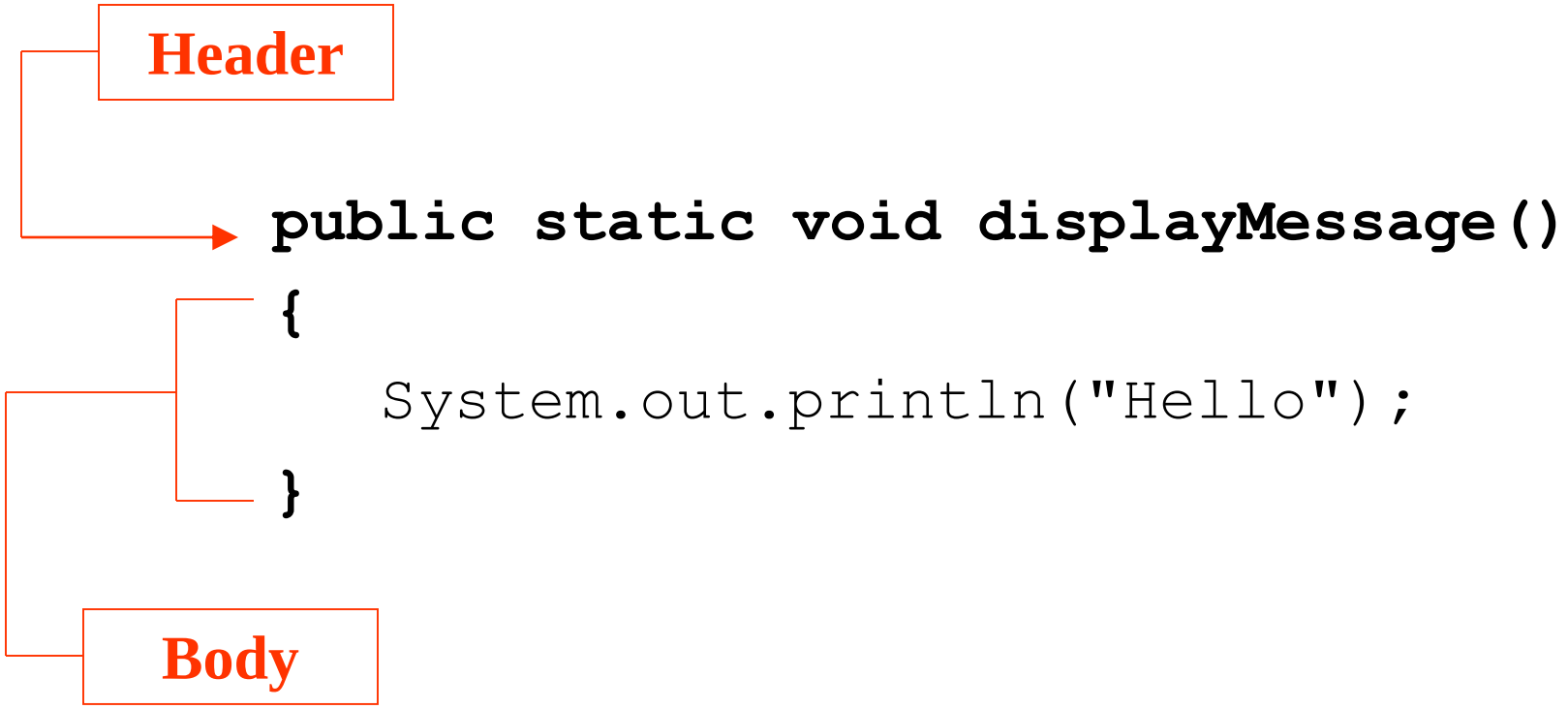
```
int number =Integer.parseInt("700");
```

method for converting a string to number

Defining a `void` Method

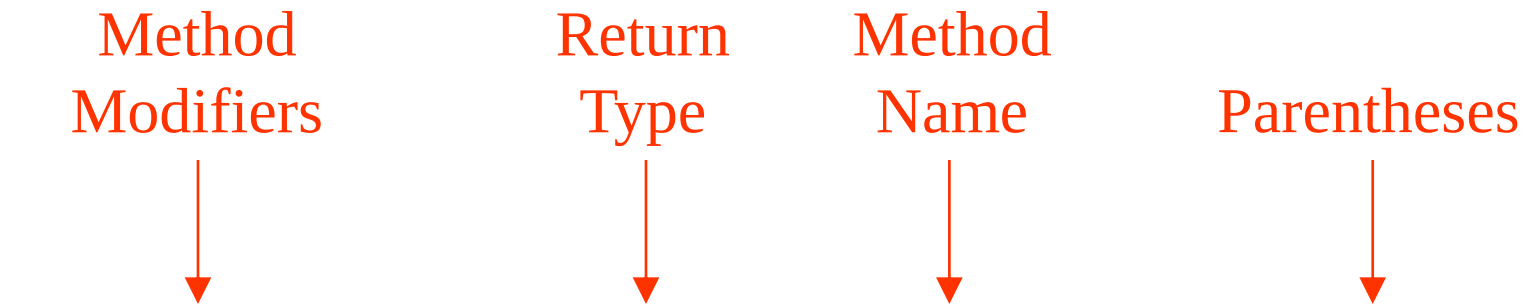
- To create a method, you must write a definition, which consists of a *header* and a *body*.
- The method header, which appears at the beginning of a method definition, lists several important things about the method, including the method's name.
- The method body is a collection of statements that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

Two Parts of Method Declaration



A `void` method

Parts of a Method Header




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

Parts of a Method Header

- 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` 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. (Formal parameters list?)

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.
- Examples: SimpleMethod.java, LoopCall.java, CreditCard.java, DeepAndDeeper.java

```java
package methodCall05_01PK;

public class MethodCallEx01 {

    public static void main(String[] args) {
        String str = "Pls. complete this task.";
        double costPerUnit = 3106.75;
        displayMessage(str, costPerUnit);

    }
    public static void displayMessage(String strLine,
                                      double cost) {
        System.out.printf(strLine +
                "\nThe unit cost is %,.2f.\n", cost);
    }//end of displayMessage
}
```

```
Pls. complete this task.
The unit cost is 3,106.75.
```

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 `*/`.

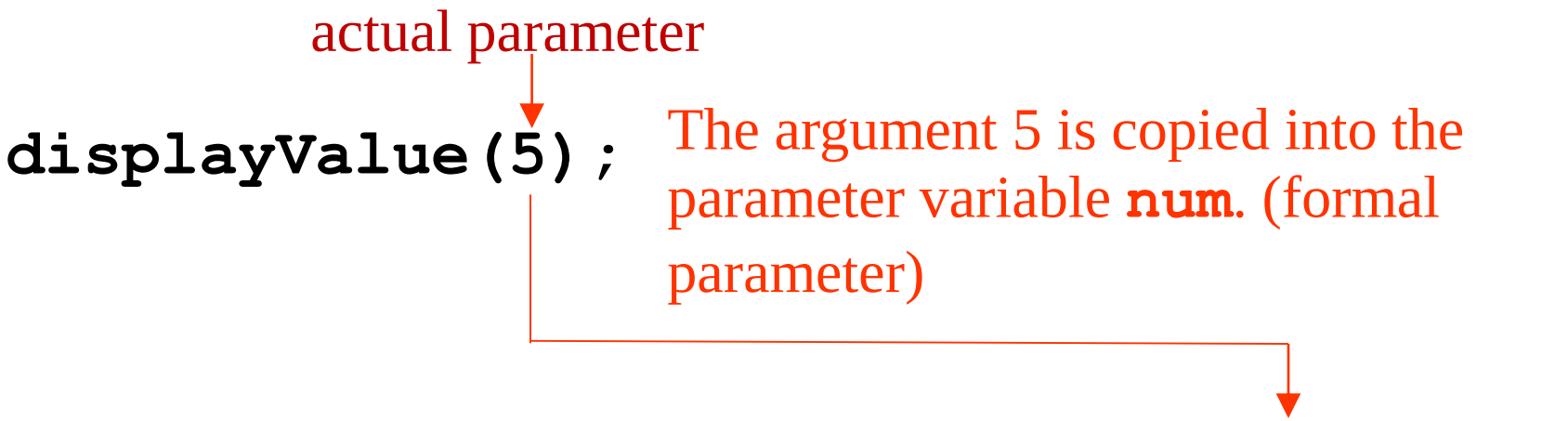
Passing Arguments to a Method

- 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.
- See example: PassArg.java

Passing 5 to the `displayValue` Method




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

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

Passing Multiple Arguments

```
double x = 5;
double y = 10;
showSum(x, y);
```

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

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

NOTE: Order matters!

```
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);
}
```

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

```java
package methodCall05_01PK;
public class MethodCallEx01 {
    public static void main(String[] args) {
        String str = "Pls. complete this task.";
        final double costPerUnit = 3106.75;
        int nosUnits = 10;
        double totalCost;
        totalCost = computeTCost(costPerUnit, 10);
//or     totalCost = computeTCost(costPerUnit, nosUnits);
        displayMessage(str, costPerUnit, totalCost);
    }
    public static void displayMessage(String strLine,
                                      double cost, double tCost) {
        System.out.printf(strLine +
             "\nThe unit cost is %,.2f.\n" +
             "The total cost is %,.2f.\n", cost, tCost);
    }//you can change cost(or replaced by costPerUnit) although final costPerUnit.

    public static double computeTCost(double uCost,
                                                int uNos) {
         return uCost * uNos;
    }
}
```

```
package methodCall05_01PK;
import java.util.Random;
public class MethodCallEx01 {
    public static void main(String[] args) {
        int rNumber;
        rNumber = RandomNosGen(100, 50);
        System.out.println("The random number is " +
                                            rNumber);
    }
/**
 *
 * @param range
 * @param between
 * @return
 */
    public static int RandomNosGen(int range,
                                   int between) {
        Random rand = new Random();
        int rNumber = rand.nextInt(range) - between;
        return rNumber;
    }
}
```

```
The random number is 34
```

```java
package chapter05_demos;
import java.util.Random;
public class Chapter_05_Demons {
    public static void main(String[] args) {
        int rNumber;
        Random rand = new Random();
        rNumber = RandomNosGen(rand, 100, 50);
        System.out.println("The random number is "
        + rNumber);
    } //end of main

    public static int RandomNosGen(Random rand, int
range, int between) {
        //Random rand = new Random();
        int rNumber = rand.nextInt(range)-between;
        return rNumber;
    } //end of RandomNosGen
} //end of class
```

```java
import java.util.Random;

public class RandomNosCh05 {

public static void main(String[] args) {
// TODO Auto-generated method stub
        int rNumber;
        Random rand = new Random();
        rNumber = RandomNosGen(100, 50);
        System.out.println("The random number is " + rNumber);
        System.out.printf("The 2nd random number is: %.4f\n",
        RandomNosGen(rand, 100, 50));
    }// end of main

    public static int RandomNosGen(int range, int between) {
        Random rand = new Random();
        int rNumber = rand.nextInt(range) - between;//between through
        range-(between+1)
        return rNumber;
    }
    public static double RandomNosGen(Random rand, int range, int
    between) {
        //Random rand = new Random();
        double rNumber = rand.nextInt(range) + between;//(range -
        between +1) through range +(between)
        return rNumber;
    }
}
```

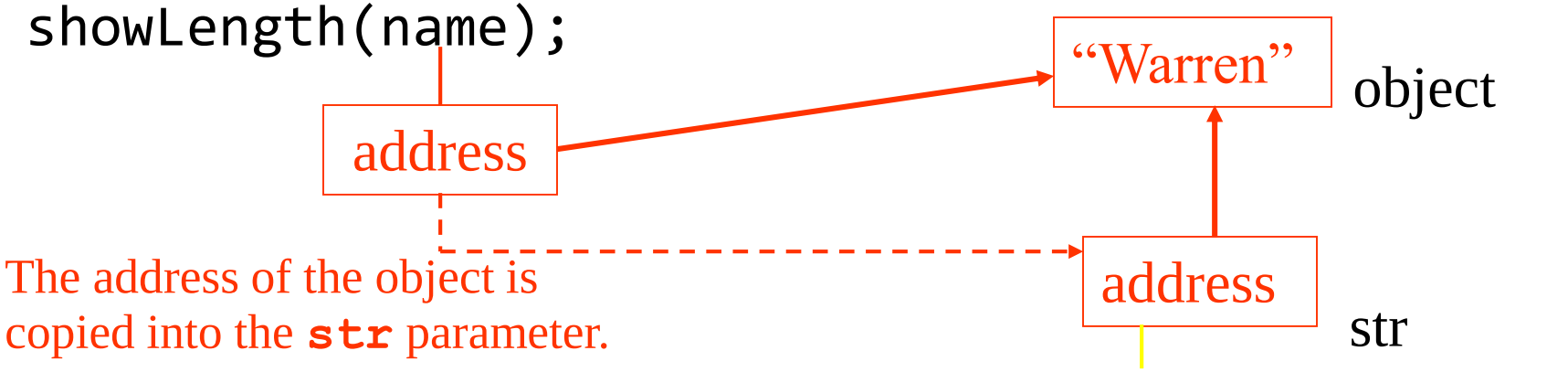
Passing Object References to a Method

- 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

```
String name = “Warren”;
showLength(name);
```




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

`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.

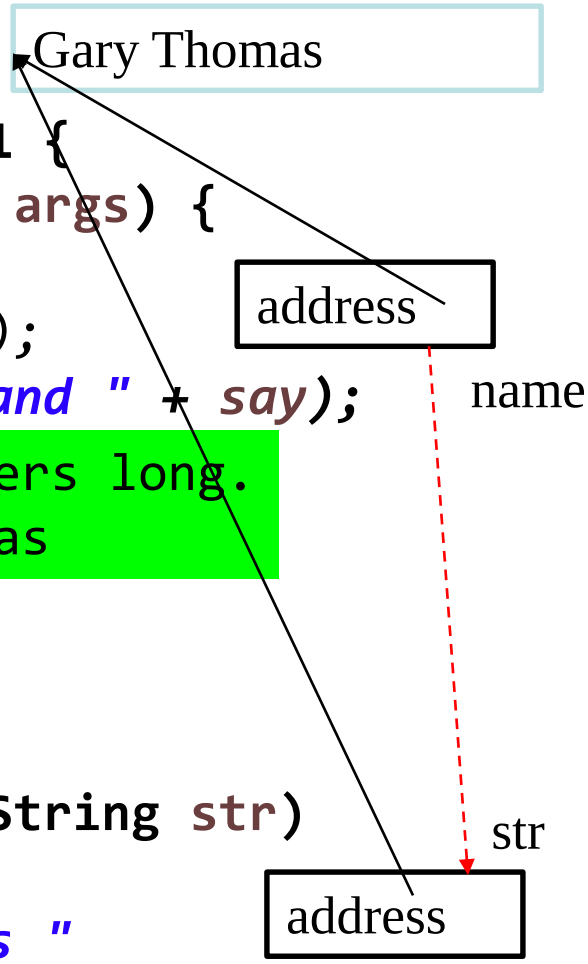
- See example: PassString.java

```java
package methodCall05_02StringPK;

public class MethodCallString05_02_01 {
    public static void main(String[] args) {
        String name = "Gary Thomas";
        String say = showLength(name);
        System.out.println(name + " and " + say);
    }
```

```
Gary Thomas is 11 characters long.
Gary Thomas and Gary Thomas
```

```java
    /**
     *
     * @param str
     * @return
     */
    public static String showLength(String str)
    {
        System.out.println(str + " is "
                           + str.length()
                           + " characters long.");
        //str = " joe " ;
        return str;
    }
}
```



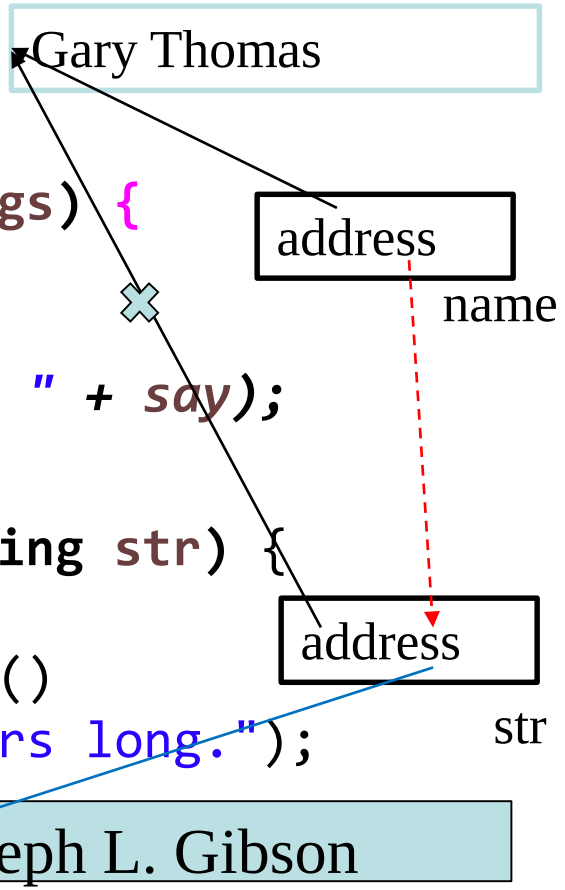
```java
package methodCall05_02StringPK;

public class MethodCallString05_02_01 {
    public static void main(String[] args) {
        String name = "Gary Thomas";
        String say = showLength(name);
        System.out.println(name + " and " + say);
    }

    public static String showLength(String str) {
        System.out.println(str + " is "
                           + str.length()
                           + " characters long.");

        str = "Joseph L. Gibson";
        System.out.println(str + " is "
                           + str.length()
                           + " characters long.");

    return str;
    }
}
```




```
Gary Thomas is 11 characters long.
Joseph L. Gibson is 16 characters long.
Gary Thomas and Joseph L. Gibson
```

`@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

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

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

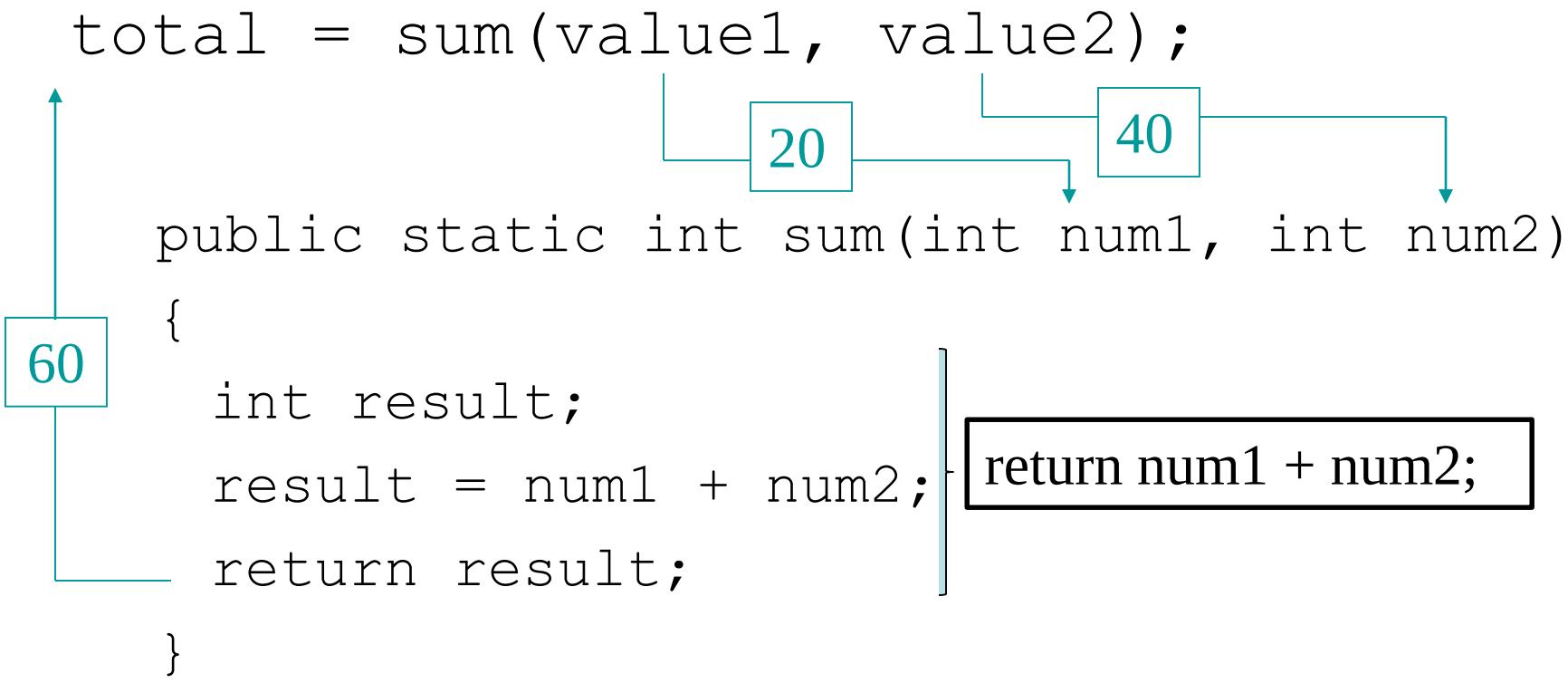
The call statement can be:

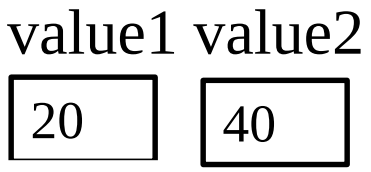
```
int value2 = 40, result = 100;
result = sum(20, value2);//???
```

The call statement can be:

```
int value2 = 40, result = 100;
result = result + sum(20, value2);
```

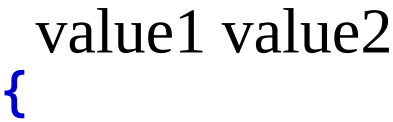
Calling a Value-Returning Method

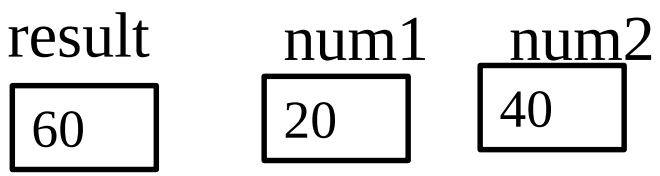





```
package methodCallValueRetn05_02_02PK;

public class MethodCallValueReturn05_02_02 {
    public static void main(String[] args) {
        int value1 = 20, value2 = 40;
        System.out.println("sum is " +
                                    sum(value1, value2));
        int total = sum(value1, value2);
        System.out.println("total is " + total);
}
    public static int sum(int num1, int num2) {
        int result;
        result = num1 + num2;
        return result;
    }
}
```






```
sum is 60
total is 60
```

```java
package methodCallValueRetn05_02_02PK;

public class MethodCallValueReturn05_02_02 {
    public static void main(String[] args) {
        int value1 = 20, value2 = 40, value3 = 30, value4 = 50;
        System.out.println("sum is " + sum(value1, value2));
        int total = sum(value1, value2);
        System.out.println("total is " + total);
        sumTotal(value3, value4);
        System.out.println("T02: result is " + result);

    }
    public static int sum(int num1, int num2) {
        int result;
        result = num1 + num2;
        return result;
    }
    public static void sumTotal(int num1, int nu
        int result;
        result = num1 + num2;
        System.out.println("T01: result is " + result);
    }
}
```

```
sum is 60
total is 60
T01: result is 80
```

`@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;

}
```

address

"John Martin"

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

```
public class MethodCallRetString05_02_03 {

    public static void main(String[] args) {
        String customerName;
        customerName = fullName("John", "Martin");
        String str = "The customer's name is %s.\n";
        System.out.printf(str, customerName);
}

        public static String fullName(String first,
                                      String last)
        {
            String name;
            name = first + " " + last;
            System.out.printf("T01: Customer's name " +
                              "is  %s.\n", name);
            return name;
        }
}
```

```
T01: Customer's name is John Martin.
The customer's name is John Martin.
```

Problem Solving with Methods

- A large, complex problem can be solved a piece at a time by methods.
- The process of breaking a problem down into smaller pieces is called *functional decomposition*.
- See example: SalesReport.java
- If a method calls another method that has a `throws` clause in its header, then the calling method should have the same `throws` clause.

Adding a throws Clause to the Method Header

Suppose we create a PrintWriter object and pass the name of a file to its constructor.

```
import java.util.Scanner;
import java.io*’
public class FileWriteDemo {
   public static void main(String[ ] args) throws IOException {
            Scanner kb = new Scanner(System.in);
            System.out.print( “Enter the filename: ”);
            String filename = kb.nextLine();

            //open the file.
            PrinterWriter outputFile = New PrintWriter(filename);
            //write the name to the file
            outputFile.println(“Thomas Jefferson.”);
            //close the file
            outputFile.close();
   }
}
```

Adding a throws Clause to the Method Header

Suppose we create a PrintWriter object and pass the name of a file to its constructor.

The PrintWriter objects attempts to create the file, but unexpectedly the disk is full and the file cannot be created.

Obviously, the program cannot continue until this situation has been dealt with, so an exception is thrown, which causes the program to suspend normal execution.

When an unexpected event occurs in a Java program, it is said that the program *throws* an exception.

We can think of an *exception* as a signal indicating that the program cannot continue until the unexpected even has been dealt with.

Adding a throws Clause to the Method Header

When an exception is thrown, the method that is executing must either deal with the exception, or throw it again.

If the main method throws an exception, the program halts and an error message is displayed.

Because PrintWriter objects are capable of throwing exceptions, we must either write code that deals with the possible exceptions (in Chapter 10), or we simply allow our methods to rethrow the exceptions when they occur.

To allow a method to rethrow an exception that has not been dealt with, we simply write a throws clause in the method header.

Adding a throws Clause to the Method Header

To allow a method to rethrow an exception that has not been dealt with, we simply write a *throws* clause in the method header. An example is:

```
public static void main(String[ ] args) throws IOException
{
      ….
 }
```

This header indicates that the *main* method is capable of throwing an exception of the IOException type. This is the type of exception that PrintWriter objects are capable of throwing. So, any method that uses PrintWriter objects, and does not respond to their exceptions, must have this throws clause listed in its header.

Adding a throws Clause to the Method Header

In addition, any method that calls a method that uses a PrintWriter object should have a *throws IOException* clause in its header.

For example, suppose the *main* method does not perform any file operations, but calls a method named *buildFile* that opens a file and write data to it.

Both the *buildFile* and *main* methods should have a *throws IOException* clause in their headers. Otherwise, a compiler error will occur.

Calling Methods that Throw Exceptions

- Note that the `main` and `getTotalSales` methods in *SalesReport.java* have a `throws IOException` clause.
- All methods that use a `Scanner` object to open a file must throw or handle `IOException`.
- You will learn how to handle exceptions in Chapter 12.
- For now, understand that Java required any method that interacts with an external entity, such as the file system to either throw an exception to be handles elsewhere in your application or to handle the exception locally.

```
import java.util.Scanner;
import javax.swing.JOptionPane;
/**
 * A program prompts user to enter their first and last name.
 * Then display the user's name in question.
 * @author apeng
 */
public class methodCall {
    public static void main(String[] args) {
        String name, task, sFormat;
        int length;
        String gtitle = "Your First and Last Name";
        task = "Enter first and last name: ";

        name = inputName(task, gtitle); //a method call
        length = name.length() - 1;
        sFormat = String.format("Is your name, %s of length %d?\n",  name, length);
        displayName(sFormat, gtitle, length);  //a method call
        System.exit(0);
    } //end of main() method
    public static String inputName(String func, String title) { … } //end of inputName
    public static void displayName(String str, String title) { … } //end of displayName
    }//end of class methodCall
```

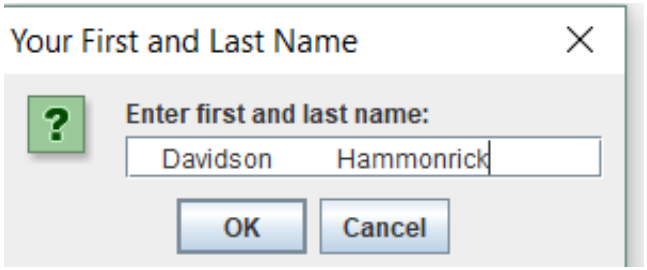
```
/**
 *
 * @param func
 * @param title
 * @return
 */
public static String inputName(String func, String title)
{
      String nStr, fName, sName;
      nStr = JOptionPane.showInputDialog(null, func, title,
                                    JOptionPane.QUESTION_MESSAGE);
      Scanner splitter = new Scanner(nStr);
      fName = splitter.next();
      sName = splitter.next();
      nStr = fName + " " + sName;
      return nStr;
}//end of inputName
```

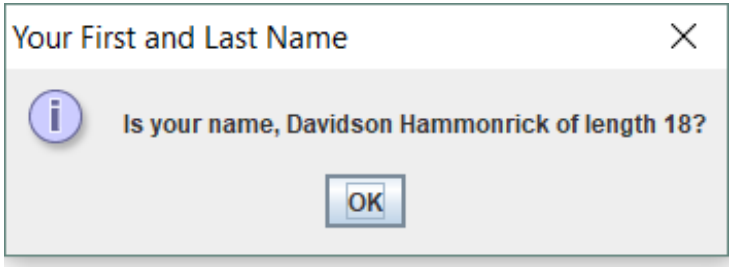
```
/**
 *
 * @param str
 * @param title
 */
public static void displayName(String str, String title, int len)
{

      System.out.println("From displayinConsole:");
      System.out.print(str);
      System.out.print(str + "Confirmed name's length is " + len);
      JOptionPane.showMessageDialog(null, str, title,
                  JOptionPane.INFORMATION_MESSAGE);

} //end displayName
```

The program outputs:






```
From displayinConsole:
Is your name, Davidson Hammonrick of length 18?
Confirmed name's length is 18
```