



# JAVA Textbook

## *Chapter 9*

# Advanced Modularization Techniques



# Objectives

In this chapter, you will learn about:

- Writing methods that require no parameters
- Writing methods that require a single parameter
- Writing methods that require multiple parameters
- Writing methods that return values
- Passing entire arrays or single elements of an array to a method
- Overloading methods in Java

# Writing Methods with No Parameters

```
import javax.swing.JOptionPane;
public class CustomerBill
{
    public static void main(String args[])
    {
        // Declare variables local to main()
        String name;
        String balanceString;
        double balance;

        // Get interactive input
        name = JOptionPane.showInputDialog(
            "Enter customer's name: ");
        balanceString = JOptionPane.showInputDialog(
            "Enter customer's balance: ");

        // Convert String to double
        balance = Double.parseDouble(balanceString);

        // Call nameAndAddress() method
    }
}
```

```
    nameAndAddress();

    // Output customer name and address
    System.out.println("Customer Name: " + name);
    System.out.println("Customer Balance: " + balance);
}
public static void nameAndAddress()
{
    // Declare and initialize local, constant Strings
    final String ADDRESS_LINE1 = "ABC Manufacturing";
    final String ADDRESS_LINE2 = "47 Industrial Lane";
    final String ADDRESS_LINE3 = "Wild Rose, WI 54984";

    // Output
    System.out.println(ADDRESS_LINE1);
    System.out.println(ADDRESS_LINE2);
    System.out.println(ADDRESS_LINE3);
} // End of nameAndAddress() method
}
```

In the header of the method:

- The *public* keyword makes this method available for execution.
- The *static* keyword means the method can be called without having to create a *CustomerBill* object.
- The *void* keyword indicates that the method does not return a

# Methods Requiring a Single Parameter

- Arguments
  - Sometimes called actual parameters.
  - Are data items sent to methods.
- Passing an argument by value
  - A copy of the value of the argument is passed to the method.
  - Within the method, the value is stored in the formal parameter at a different memory location, and is considered local to that method.
  - In this example, even though the name of the parameter, *number*, is the

```
// EvenOrOdd.java - This program determines if a number
// input by the user is an even number or an odd number.
// Input: Interactive.
// Output: The number entered and whether it is even or odd.
```

```
import javax.swing.*;
```

```
public class EvenOrOdd
```

```
{
    public static void main(String args[])
    {
```

```
        int number;
        String numberString;
```

The variable named *number* is local to the *main()* method. Its value is stored at one memory location. For example, it may be stored at memory location 2000.

```
        numberString = JOptionPane.showInputDialog(
            "Enter a number or -999 to quit: ");
        number = Integer.parseInt(numberString);
```

```
        while(number != -999)
        {
```

```
            even_or_odd(number);
```

```
            numberString = JOptionPane.showInputDialog(
                "Enter a number or -999 to quit: ");
            number = Integer.parseInt(numberString);
        }
```

```
        System.exit(0);
```

```
    } // End of main() method.
```

```
    public static void even_or_odd(int number)
    {
```

```
        if((number % 2) == 0)
            System.out.println("Number: " + number +
                               " is even.");
```

```
        else
            System.out.println("Number: " + number +
                               " is odd.");
```

```
    } // End of even_or_odd method.
```

```
} // End of EvenOrOdd class.
```

The value of the formal parameter, *number*, is stored at a different memory location and is local to the *even\_or\_odd()* method. For example, it may be stored at memory location 7800.

# Methods Requiring Multiple Parameters

- To specify a method requiring multiple parameters, include a list of data types and local identifiers separated by commas as part of the method's header.
- To call a method that expects multiple parameters, list the actual

```
// ComputeTax.java - This program computes tax given a  
// balance and a rate  
// Input: Interactive.  
// Output: The balance, tax rate, and computed tax.
```

```
import javax.swing.*;
```

```
public class ComputeTax  
{
```

```
    public static void main(String args[])  
    {
```

```
        double balance; _____ Memory address 1000  
        String balanceString;  
        double rate; _____ Memory address 1008  
        String rateString;
```

```
        balanceString = JOptionPane.showInputDialog(  
            "Enter balance: ");  
        balance = Double.parseDouble(balanceString);  
        rateString = JOptionPane.showInputDialog(  
            "Enter rate: ");  
        rate = Double.parseDouble(rateString);
```

```
        computeTax(balance, rate);
```

```
        System.exit(0);
```

```
    } // End of main() method.
```

```
    public static void computeTax(double amount, double rate)  
    {
```

```
        double tax; _____ Memory address 9008  
  
        tax = amount * rate;  
        System.out.println("Amount: " + amount + " Rate: " +  
            rate + " Tax: " + tax);
```

```
    } // End of computeTax method
```

```
} // End of ComputeTax class.
```

# Methods Returning a Value

- In Java, a method can only return a single value.
  - Must indicate the return type (data type of the return value).
  - Can be any of Java's built-in data types, as well as a class type, such as String.
- The method returns a copy of the value (stored in *workHours*) to the location in the calling method where *getHoursWorked()* is called (the right side of the assignment statement).
- Can also use the return value directly without storing in a variable, such as:

*gross = getHoursWorked() \**

```
// GrossPay.java - This program computes an employee's
// gross pay.
// Input: Interactive.
// Output: The employee's hours worked and their gross pay.

import javax.swing.*;

public class GrossPay
{
    public static void main(String args[])
    {
        double hours;
        final double PAY_RATE = 12.00;
        double gross;

        hours = getHoursWorked();
        gross = hours * PAY_RATE;

        System.out.println("Hours worked: " + hours);
        System.out.println("Gross pay is: " + gross);

        System.exit(0);
    } // End of main() method.

    public static double getHoursWorked()
    {
        double workHours;
        String workHoursString;

        workHoursString = JOptionPane.showInputDialog(
            "Please enter hours worked: ");
        workHours = Double.parseDouble(workHoursString);

        return workHours;
    } // End of getHoursWorked method
} // End of GrossPay class.
```

# Passing an Entire Array to a Method

```
import javax.swing.*;
public class PassEntireArray
{
    public static void main(String args[])
    {
        // Declare variables
        final int LENGTH = 4;
        int someNums[] = {10, 12, 22, 35};
        int x;
        System.out.println("At beginning of main method...");
        x = 0;
        while (x < LENGTH) // Print initial array values
        {
            System.out.println(someNums[x]);
            x++;
        }
        // Call method, pass array
        quadrupleTheValues(someNums);
        System.out.println("At the end of main method...");
        x = 0;
        // Print changed array values
        while (x < someNums.length)
        {
            System.out.println(someNums[x]);
            x++;
        }
        System.exit(0);
    } // End of main() method.
}
```

```
public static void quadrupleTheValues(int [] vals)
{
    final int LENGTH = 4;
    int x;
    x = 0;
    // Print array values before they are changed
    while(x < LENGTH)
    {
        System.out.println(
            " In quadrupleTheValues() method, value is " +
            vals[x]);
        x++;
    }
    x = 0;
    while(x < LENGTH) // This loop changes array values
    {
        vals[x] = vals[x] * 4;
        x++;
    }
    x = 0;
    // Print array values after they are changed
    while(x < LENGTH)
    {
        System.out.println(" After change, value is " +
            vals[x]);
        x++;
    }
} // End of quadrupleTheValues method
} // End of PassEntireArray class.
```

- The array is passed by reference, i.e., the array's memory address is passed.
  - The method has access to that memory location, thus being<sup>7</sup>



# Passing an Entire Array to a Method

```
C:\Java>java PassEntireArray
At beginning of main method...
10
12
22
35
In quadrupleTheValues() method, value is 10
In quadrupleTheValues() method, value is 12
In quadrupleTheValues() method, value is 22
In quadrupleTheValues() method, value is 35
After change, value is 40
After change, value is 48
After change, value is 88
After change, value is 140
At the end of main method...
40
48
88
140
C:\Java>
```

Figure 9-10 Output from the Pass Entire Array program

- The array displayed at the end of the main method shows values changed from inside the *quadrupleTheValues()* method



# Passing an Array Element to a Method

```
int someNums[] = {10, 12, 22, 35};  
int newNum;  
newNum = tripleTheNumber(someNums[1]);
```

```
public static int tripleTheNumber(int num)  
{  
    int result;  
    result = num * 3;  
    return result;  
}
```

- Passing a single array element to a method is just the same as passing a variable or constant.



# Overloading Methods

- Overload methods by giving the same name to more than one method.
  - Useful to perform the same action on different types of inputs.
- Overloaded methods must either have a different number of arguments or the arguments must be of different data types.
- Java figures out which method to call based on the method's name and its arguments, the combination of which is known as the method's signature.
  - The signature of an overloaded method does not include the method's return type.
- Overloading methods is a form of

# Overloading Methods

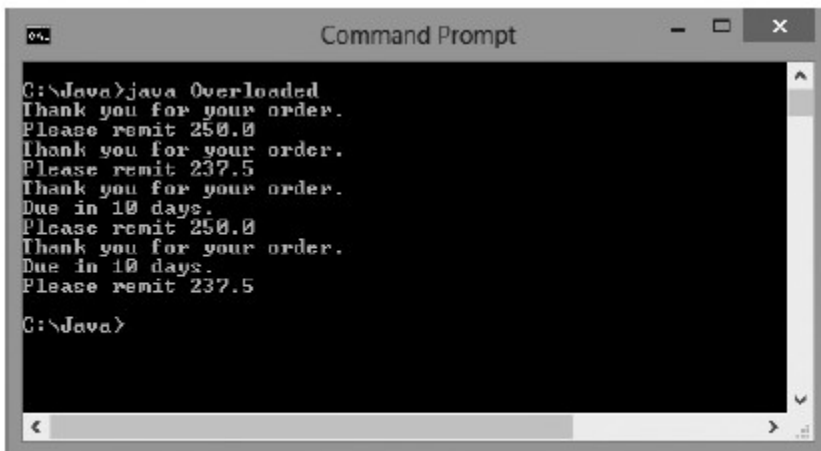
```
// Overloaded.java - This program illustrates overloaded
// methods.
// Input: None.
// Output: Bill printed in various ways.
import javax.swing.*;
public class Overloaded
{
    public static void main(String args[])
    {
        double bal = 250.00, discountRate = .05;
        String msg = "Due in 10 days.";
        printBill(bal); // Call version #1.
        printBill(bal, discountRate); // Call version #2.
        printBill(bal, msg); // Call version #3.
        printBill(bal, discountRate, msg); // Call version #4.
        System.exit(0);
    } // End of main() method.
}
```

```
// printBill() method version #1.
public static void printBill(double balance)
{
    System.out.println("Thank you for your order.");
    System.out.println("Please remit " + balance);
} // End of printBill version #1 method.

// printBill() method version #2.
public static void printBill(double balance,
                             double discount)
{
    double newBal;
    newBal = balance - (balance * discount);
    System.out.println("Thank you for your order.");
    System.out.println("Please remit " + newBal);
} // End of printBill version #2 method.

// printBill() method version #3.
public static void printBill(double balance,
                             String message)
{
    System.out.println("Thank you for your order.");
    System.out.println(message);
    System.out.println("Please remit " + balance);
} // End of printBill version #3 method.

// printBill() method version #4.
public static void printBill(double balance,
                             double discount,
                             String message)
{
    double newBal;
    newBal = balance - (balance * discount);
    System.out.println("Thank you for your order.");
    System.out.println(message);
    System.out.println("Please remit " + newBal);
} // End of printBill version #4 method.
} // End of Overloaded class.
```



```
Command Prompt
C:\Java>java Overloaded
Thank you for your order.
Please remit 250.0
Thank you for your order.
Please remit 237.5
Thank you for your order.
Due in 10 days.
Please remit 250.0
Thank you for your order.
Due in 10 days.
Please remit 237.5
C:\Java>
```

Figure 9-12 Output from the Overloaded program



# Using Java's Built-in Methods

- Java's built-in methods learned so far:
  - *println()*, *showInputDialog()*, *parseInt()*, *parseDouble()*, etc.
- Another built-in method: *format()*
  - Can be used to control the number of places displayed after the decimal point when printing a value of data type double.
  - One of several ways to control the number of places displayed after a decimal point.

# Using Java's Built-in Methods

```
double valToFormat = 1234.12;  
System.out.format("%.3f%n", valToFormat);
```

- Format specifier: begins with a percent sign (%) and end with a converter.
  - May include optional flags and specifiers in between.
- `%.3f`: display three places after the decimal points for a floating-point value.
- `%n`: display a newline character.
- Output from this code sample: 1234.120



Thank You!