Java Methods

Subroutines, Functions --> Methods
Java Math Library
User Defined Methods

Dan McElroy

Oct 2018

Topics Covered

- Introduction to Java Methods - Some Definitions
- Parameters and Return Data Type
- Predefined and User Defined Methods
- Methods as Part of Objects
- Overloading and Overriding

Functions and Subroutines Serve Two Main Purposes

- They allow a programmer to say: `this piece of code does a specific job which stands by itself and should not be mixed up with anything else',
- Second they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

Functions and Subroutines

Think about a company that has a boss and several employees who each specialize in one task. The boss can say to one person, "Build a thinga-ma-bob and report back to me when you are finished." This would be like a subroutine. A numeric return value is not returned to the boss by the employee.

To another employee, the boss says, "Count the number of hammers in the tool bin and tell me how many there are." This is like a function because there is a return value, the count of items.

Subroutine or Function ?

Depending on the computer language that you are using, there are different names that define the same or almost the same thing.

Subroutine: A set of statements that perform a specific task but does not return a value. It can not be used in an arithmetic expression. In some languages, a subroutine is called a procedure.

Function: A set of statements that perform a specific task and return a value that can be used in an arithmetic expression or part of an assignment statement. x = Math.sqrt(25);

(NOTE: some functions return non-numeric data)

Subroutine or Function

In Visual Basic, the keywords **Sub** and **Function** are used when declaring these pieces of code:

```
Private Sub clearData( txtPaycheck As TextBox )
Private Function calculatePaycheck (Hours As Double) As Double
```

Parameters — Return Data Type

In C, C++ and Java, a subroutine is just a function with a **void** return data type

```
private static void clearData( textField txtPaycheck)
private static double calculatePaycheck (double hours)
```

Return Data Type — Parameters

What is a **Method**?

When a subroutine or function becomes part of a class definition in Object Oriented Programming, it is usually referred to as a **method**. In Java, everything is part of a class, so technically, all functions and subroutines are methods.

Standard Library Methods

The Java language already has many predefined methods that are already provided in its libraries:

- Math and other methods that work with numbers
- Strings Methods - working with character strings
- Character Methods - working with individual characters
- Array Methods
- Print Methods - print, println, printf, etc.
- and many others

```
public class MyClass {

    public static void main(String[] args) {

        double result = Math.sqrt(25); // sqrt as part of an assignment statement
        System.out.println ("The square root of 25 is " + result);

        // put everything in the println statement
        System.out.println ("The square root of 25 is " + Math.sqrt(25));

    } // end of main
} // end of class definition
```

When you run the program, the output is

```
The square root of 25 is 5.0
The square root of 25 is 5.0
```

- print(), println() and printf() are methods of java.io.PrintStream
- Math.sqrt() is a method of the Math class

Math.sqrt() can be used as part of an expression or part of the println argument. Both **result** and **Math.sqrt( )** are of type double and are automatically cast into strings when used as part of the println statement.

Commonly Used Math Methods and Constants

Math.sqrt(x)	$\sqrt{x}$	Math.log(x)	ln x	Math.ceil(x)	next largest int value if there are digits past the decimal as double
Math.pow(x,y)	x^y	Math.log10(x)	$\log_{10} x$		
Math.sin(x)	sin x	Math.abs(x)	|x|	Math.floor(x)	next smallest int value if there are digits past the decimal as double
Math.cos(x)	cos x	Math.max(x,y)	maximum of the two values		
Math.tan(x)	tan x	Math.min(x,y)	minimum of the two values	**CONSTANTS**	
Math.asin(x)	$\sin^{-1} x$	Math.round(x)	closest integer to x, as a long	Math.PI 3.14159265358979323846	
Math.atan(x)	$\tan^{-1} x$			Math.E 2.7182818284590452354	

```
double a = ceil(3.0);  // a becomes 3.0
double b = ceil(3.4);  // b becomes 4.0
```

```
double x = floor(3.0);  // x becomes 3.0
double y = floor(3.4);  // y becomes 3.0
```

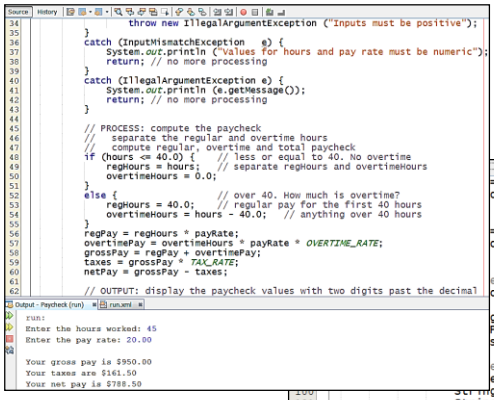
For more informatio see: https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html

User Defined Methods

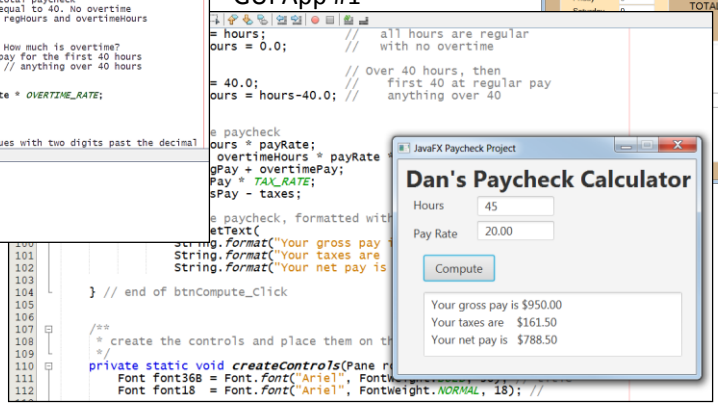
Several different versions of the paycheck program have been created. Each of these has the same way of separating overtime and regular hours from the total hours worked. As programs grow, it is better to break the code into separate sections so that each part does only one thing and does it well.

computePay

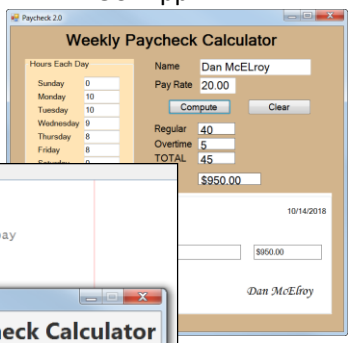
Console App



GUI App #1



GUI App #2



Define the computeGrossPay method

```
private static double computeGrossPay(double hours, double payRate) {
    // compute the regHours and overtimeHours
    double regHours, overtimeHours;

    if (hours <= 40) {                  // Less or = to 40 hours, then
        regHours = hours;               //   all hours are regular
        overtimeHours = 0.0;            //   with no overtime
    }
    else {                              // Over 40 hours, then
        regHours = 40.0;                //    first 40 at regular pay
        overtimeHours = hours-40.0; //    anything over 40
    }

    // compute the paycheck
    double regPay = regHours * payRate;
    double overtimePay = overtimeHours * payRate * OVERTIME_RATE;

    return regPay + overtimePay;
} // end of computeGrossPay( ) method
```

'Call' the *computeGrossPay* method

```
    // compute the paycheck
    grossPay = computeGrossPay (hours, payRate);
    taxes = grossPay * TAX_RATE;
    netPay = grossPay - taxes;

    // display the paycheck, formatted with 2 digits past the decimal
    txtPaycheck.setText(
            String.format("Your gross pay is $%.2f\n", grossPay) +
            String.format("Your taxes are    $%.2f\n", taxes) +
            String.format("Your net pay is   $%.2f", netPay) );

} // end of btnCompute_Click
```

Creating a user-defined method

```
private static double computeGrossPay(double hours, double payRate) {
    //
    // Body of th function
    //
    return 0.0;    A return value is required unless it is a void function
} // end of compu GrossPay( ) method
```

OUTPUT data type Method name INPUT arguments

Creating a user-defined method

```
private static double computeGrossPay(double hours, double payRate) {
    //
    // Body of the function
    //
    return 0.0;  // A return value is required unless it is a void function
} // end of computeGrossPay( ) method
```

Access Modifier, sometimes referred to as "Access Specifier" - specifies who can access the class, method, variable, etc. The access modifier has four types:
1) public - visible to any part of the program
2) private - visible only within the class
3) protected - visible to anything in the package and any subclass
4) default - visible to the package
For programs that only have one class defined, it does not make any difference whether the public or private access modifier is used.

Creating a user-defined method

```
private static double computeGrossPay(double hours, double payRate) {

    //
    // Body of the function
    //
    return 0.0;  // A return value is required unless it is a void function

} // end of computeGrossPay( ) method
```

static - indicates that the method can be accessed without defining an object of the class. Math.sqrt( ) does not need an object to be created for it to be used. Math.sqrt( ) is a static function

Creating a user-defined method

```
private static double computeGrossPay(double hours, double payRate) {

    //
    // Body of the function
    //
    return 0.0;  // A return value is required unless it is a void function

} // end of computeGrossPay( ) method
```

return data type - Specifies the type of data **OUTPUT. I**n this example, the method needs to return a piece of data of type double to the main program. Any valid data type, or even an object can be returned to the main program. There can only be ONE piece of data returned from a method or function.

If no data is to be returned by the method, then **void** is specified as the return data type. In this case, the **return;** statement can be used to end the function, but nothing can be placed after **return;**

Creating a user-defined method

```
private static double computeGrossPay(double hours, double payRate) {

    //
    // Body of the function
    //
    return 0.0;  // A return value is required unless it is a void function

} // end of computeGrossPay( ) method
```

Argument List - When defining a function, a list of inputs to the method can be specified. Specify a data type and a name for each argument. Each time the method is activated, a new set of variables is created using the names from the argument list. Although many times, programmers use the same name in the main program as the argument list, actually they are totally unrelated. It is the actual value of the data that gets passed from the main program to the method.

Creating a user-defined method

```
private static double computeGrossPay(double hours, double payRate) {

    //
    // Body of the function
    //
    return 0.0;  // A return value is required unless it is a void function

} // end of computeGrossPay( ) method
```

Body of the method/function - is all of the code within the open and close curly braces **{ }.** It contains the code for doing the work of the method, including processing any of the data received in the arguments and sending any data back to the main program through the **return** statement.

The part of the program that 'calls' the computeGrossPay method

```
double grossPay;
double hours = 40.0;
double payRate = 17.50;
grossPay = computeGrossPay (hours, payRate);
```

❶ 40.0 17.5

The computeGrossPay method definition

```
// compute gross pay without considering overtime
private static double computeGrossPay(double hr, double rate) {
    double pay = hr * rate;  // without computing overtime

    return pay;
}
```

❶ In the main program, values are placed in variables named hours and payRate. These numeric values are used as the parameters when calling the **computeGrossPay** method.

The part of the program that 'calls' the computeGrossPay method

```
double grossPay;
double hours = 40.0;
double payRate = 17.50;
grossPay = computeGrossPay (hours, payRate);
```

❶ 40.0 17.5

❷

The computeGrossPay method definition

```
// compute gross pay without considering overtime
private static double computeGrossPay(double hr, double rate) {
    double pay = hr * rate;  // without computing overtime

    return pay;
}
```

❷ The **computeGrossPay** method receives 40.0 into its own internal variable named hr, and receives 17.5 into its own internal variable named rate.

The part of the program that 'calls' the computeGrossPay method

```
double grossPay;
double hours = 40.0;
double payRate = 17.50;
grossPay = computeGrossPay (hours, payRate);
                            40.0    17.5
```

❶ ❷

The computeGrossPay method definition

```
// compute gross pay without considering overtime
private static double computeGrossPay(double hr, double rate) {
    double pay = hr * rate;  // without computing overtime
        700.0 ←40.0 * 17.5
                ❸
    return pay;
}
```

❸ The **computeGrossPay** method multiplies **hr** (40.0) times **rate** (17.5) which equals 700.0 The 700.0 is stored into its internal variable named **pay**.

The part of the program that 'calls' the computeGrossPay method

```
double grossPay;
double hours = 40.0;
double payRate = 17.50;
grossPay = computeGrossPay (hours, payRate);
                            40.0    17.5
```

❶ ❷

❹ 700.0

The computeGrossPay method definition

```
// compute gross pay without considering overtime
private static double computeGrossPay(double hr, double rate) {
    double pay = hr * rate;  // without computing overtime
        700.0 ←40.0 * 17.5
                ❸
    return pay;
}
```

❹ The value 700.0 is passed from the **return** statement back to the main program. Now, computeGrossPay(hours, payRate) can be treated just as though it were any double.

The part of the program that 'calls' the computeGrossPay method

```
double grossPay;
double hours = 40.0;
double payRate = 17.50;
grossPay = computeGrossPay (hours, payRate);
```

❶ ❷ 40.0 17.5 ❺ 700.0 ❹ 700.0

The computeGrossPay method definition

```
// compute gross pay without considering overtime
private static double computeGrossPay(double hr, double rate) {
   double pay = hr * rate;  // without computing overtime
      700.0 ←40.0 * 17.5 ❸
   return pay;
}
```

❺ The result of executing the **computeGrossPay** method gives the value 700.0 as a double which is then assigned (stored) into the double variable **grossPay** that is declared in the main program.

Parameter List, Argument List ???

The part of the program that 'calls' the computeGrossPay method

```
double grossPay;
double hours = 40.0;
double payRate = 17.50;
grossPay = computeGrossPay (hours, payRate);
```

Parameter list is when a method is called

The computeGrossPay method definition

```
private static double computeGrossPay(double hr, double rate) {
   double pay = hr * rate;

   return pay;
}
```

Argument list is when a method is defined

Method Overloading

One of the features of Object Oriented Programming (OOP) is the ability to overload methods/functions. A method is considered to be overloaded within the same class when there are two methods with the same name, but different argument lists. The arguments can be different in number or their data types can be different. Here are some examples:

public static void showPaycheck (String name, double gross, double taxes, double netPay)
public static void showPaycheck (double gross, double taxes, double netPay)
public static void showPaycheck (String name, String department, double netPay)

Overloading does not occur if data types in the argument list are exactly the same, even if different names were used for the arguments. Also overloading does not occur if the argument list is exactly the same but return data type is different.

Method Overloading

If you look closely at Java's Math library, you will see four different versions of the max( ) method, and four versions of the min( ) method.

Math.max(double, double)	Math.min(double, double)
Math.max(float, float)	Math.min(float, float)
Math.max(long,long)	Math.min(long,long)
Math.max(int, int)	Math.min(int, int)

What happens if a program has mixed data types when calling Math.max? Java will promote one of the data types up until a match is found. For example, int can be promoted to a float or a double.

double x = max(42.8, 50); // 42.8 is a double, 50 is an integer

Overloading vs. Overriding

Overriding a method is completely different from overloading a method. The full definition for overriding occurs in a more detailed description of inheritance in OOP.

Once a class has been defined with all of its functionality, it is possible to create a new subclass that inherits the capabilities of the parent class, add more features and replace others. For example, we start by creating a class called **Employee** that has a lot of code and many methods. And one of the methods is called **pay( )** which computes and displays the pay based on hours worked.

Later, we decide that we have a sales department with employees whose pay is based on commissions. Instead of either rewriting or copying all of the code from **Employee** except for the **pay( )** method, we can create a subclass named **CommissionEmployee** based on the **Employee** class that uses everything except **pay( ).** The CommissionEmployee class can override **pay( ).** When an object is created using ComissionEmployee it will use everything from Employee except pay( ) which has been overridden.