**CHAPTER 6: METHODS**

Methods can be used to define reusable code and organize and simplify coding, and make code easy to maintain.

* 1. **DEFINING A METHOD**

A method definition consists of method name, parameters, return value type, and body.

The syntax for defining a method is as follows:

modifier returnValueType methodName(list of parameters) {

// Method body;

}

Let’s suppose thes method, named max, has two int parameters, num1 and num2, the larger of which is returned by the method.

Formal parameters

Method name

Illustration of the components of this method:

Return value type

modifier

Method header

public static int max(int num1, int num2) {

int result;

if (num1 > num2)

Method body

result = num1;

else

result = num2;

return result; return value

}

A method definition consists of a method header and a method body.

The method header specifies the modifiers, return value type, method name, and parameters of the method.

* 1. **CALLING A METHOD**

Calling a method executes the code in the method.

To execute the method, you have to **call** or **invoke** it. The program that calls the function is called a **caller**. There are two ways to call a method, depending on whether the method returns a value or not.

If a method returns a value, a call to the method is usually treated as a value.

For example,

int larger = max(3, 4);

Calls max(3, 4) and assigns the result of the method to the variable larger.

**NOTE:** A **void** method does not return a value.

* 1. **PASSING ARGUMENTS BY VALUES**

The arguments are passed by value to parameters when invoking a method.

The power of a method is its ability to work with parameters.

You can use **println** to print any string, and **max** to find the maximum of any two int values. When calling a method, you need to provide arguments, which must be given in the same order as their respective parameters in the method signature. This is known as **parameter order association**.

For example, the following method prints a message n times:

public static void nPrintln(String message, int n) {

for (int i = 0; i < n; i++)

System.out.println(message);

}

When you invoke a method with an argument, the value of the argument is passed to the parameter.

This is **referred to as pass-by-value**. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter.

The variable is not affected, regardless of the changes made to the parameter inside the method.

* 1. **MODULARIZING CODE**

Modularizing makes the code easy to maintain and debug and enables the code to be reused.

Methods can be used to reduce redundant code and enable code reuse. Methods can also be used to modularize code and improve the quality of the program.

* 1. **OVERLOADING METHODS**

Overloading methods enable you to define the methods with the same name as long as their parameter lists are different.

The **max** method used earlier works only with the **int** data type.

But what if you need to determine which of the two floating-point numbers has the maximum value?

The solution is to create another method with the same name but different parameters.

For example:

public static double max(double num1, double num2) {

if (num1 > num2)

return num1;

else

return num2;

}

If you call max with int parameters, the max method that expects int parameters will be invoked; and if you call max with double parameters, the max method that expects double parameters will be invoked.

This is referred to as **method overloading**; that is, two methods have the same name but different parameter lists within one class.

The Java compiler determines which method to use based on the method signature.

* 1. **THE SCOPE OF VARIABLES**

The scope of a variable is the part of the program where the variable can be referenced.

A variable defined inside a method is referred to as a local variable.

The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.

A local variable must be declared and assigned a value before it can be used.

A parameter is actually a local variable.

The scope of a method parameter covers the entire method.

A variable declared in the initial-action part of a for-loop header has its scope in the entire loop. However, a variable declared inside a **for**-loop body has its scope limited in the loop body from its declaration to the end of the block that contains the variable.

For example:

public static void method() {

for (int i = 1; i < 10; i++) {

.

.

The scope of i

int j;

.

The scope of j

.

}

}

A variable declared in the initial-action part of a **for**-loop header has its scope in the entire loop.

You can declare a local variable with the same name in different blocks in a method, but you cannot declare a local variable twice in the same block or in nested blocks,

for (int i = 0; i < 10; i++) {

}

System.out.println(i);

This statement would cause a syntax error, because variable i is not defined outside of the **for** loop.

* 1. **METHOD ABSTRACTION AND STEPWISE REFINEMENT**

The key to developing software is to apply the concept of abstraction.

Method abstraction is achieved by separating the use of a method from its implementation.

The client can use a method without knowing how it is implemented.

The details of the implementation are encapsulated in the method and hidden from the client who invokes the method.

This is also known as **information hiding or encapsulation**.

If you decide to change the implementation, the client program will not be affected, provided that you do not change the method signature.

The concept of method abstraction can be applied to the process of developing programs.

When writing a large program, you can use the **divide-and-conquer strategy**, also known as **stepwise refinement**, to decompose it into subproblems.

The subproblems can be further decomposed into smaller, more manageable problems.

Example:

Suppose that you write a program that displays the calendar for a given month of the year. The program prompts the user to enter the year and the month, and then displays the entire calendar for the month.

Let us use this example to demonstrate the divide-and-conquer approach.

**Top-Down Design**

printCalendar

(main)

printMonth

printMonth

printMonthBody

readInput

printMonthTitle

**(a) (b)**

The structure chart shows the **printCalendar** problem is divided into two subproblems, readInput and **printMonth** in (a),

and **printMonth** is divided into two smaller subproblems, **printMonthTitle** and **printMonthBody** in (b).

**Benefits of stepwise refinement**

Stepwise refinement breaks a large problem into smaller manageable subproblems.

Each subproblem can be implemented using a method.

This approach makes the program easier to write, reuse, debug, test, modify, and maintain.

* Simpler Program
* Reusing Methods
* Easier Developing, Debugging, and Testing
* Better Facilitating Teamwork

**THE END!**