

A method is a block of code that performs a specific task. The main advantage is code reusability. We can write a method once, and use it multiple times. Methods make code more readable and easier to debug.

In Java, there are two types of methods:

- **Standard Library Methods:** These are built-in methods in Java that are available to use. `print()` method of `System` class, `sqrt()` method of the `Math` class, etc.
- **User-defined Methods:** You can create your own method based on your requirements.

1. Defining a Method

The syntax to define a method is:

```
modifier returnType methodName(parameter1, parameter2, ...) {
    // method body
}
```

Here,

- **returnType** - It specifies what type of value a method returns. For example, if a method has an `int` return type then it returns an integer value.
- If the method does not return a value, its return type is `void`.
- **methodName** - It is an identifier that is used to refer to the particular method in a program.
- **modifier** - It defines access types whether the method is `public`, `static`, and so on.
- **parameter1, parameter2** - These are values passed to a method. You can pass any number of arguments to a method.

Let's look at a method defined to find the larger between two integers. This method, named `max`, has two `int` parameters, `num1` and `num2`, the larger of which is returned by the method. Figure 6.1 illustrates the components of this method.

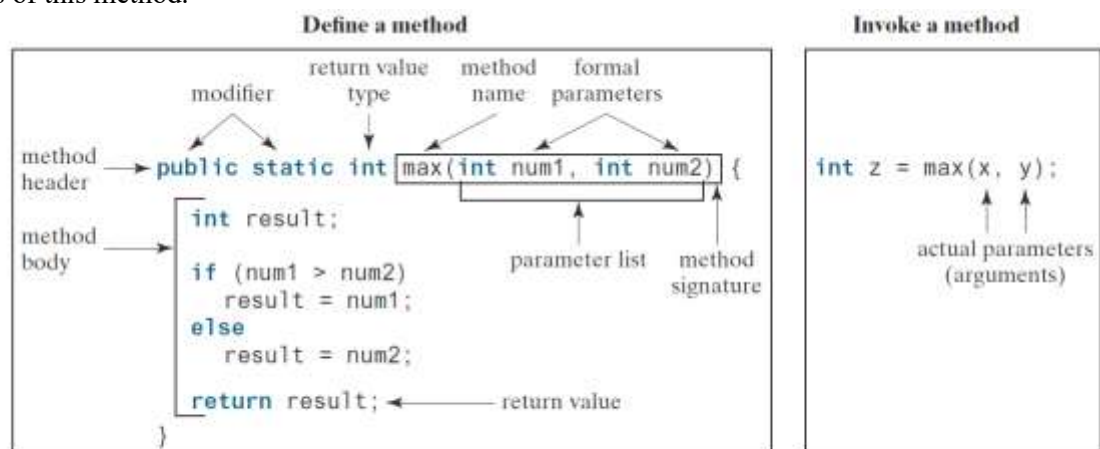


FIGURE 6.1 A method definition consists of a method header and a method body.

Example 01: In this example, we create a method named `addNumbers()` which takes two parameters `a` and `b` and return `sum`.

Program.java

```
class Program {
    int addNumbers(int a, int b) { // a and b are formal parameters
        int sum = a + b;

        return sum; // return statement
    }
    public static void main(String[] args) {
        int num1 = 25;
        int num2 = 15;

        // Create an object of Program
        Program obj = new Program();

        // Calling the method
        int result = obj.addNumbers(num1, num2); // num1 and num2 are actual
                                                // parameters or arguments
        System.out.println("sum is " + result);
    }
}
```

Output

sum is 40

Here, we have called the method by passing two arguments `num1` and `num2`. Since the method is returning some value, we have stored the value in the `result` variable.

Note that the method is not `static`. Hence, we are calling the method using an object of the `Program` class.

2. `static` Keyword

You can apply the `static` keyword with different objects like: variables, methods (excludes constructors), block and nested class.

An **instance variable** or **instance method** is a variable or method that belongs to an instance/object of a class, not to the class i.e., it requires an object of its class to be created before it can be called by the object.

In contrast, a `static` variable or `static` method belongs to a class not to any particular instance/object of the class. It can be accessed directly from the class, without having to create an object of the class. For example, the `sqrt()` method of standard `Math` class is static. Hence, you can directly call `Math.sqrt()` without creating an instance of `Math` class. If you access a static variable or static method through an object, the compiler will show the warning message.

2.1 Instance Methods Vs. **static** Methods

Instance methods are also called non-static methods. To call an instance method, you have to use an object of the class.

Example 02: Using instance method.

```
class Program {  
    int getSquare(int x){  
        return x * x;  
    }  
    public static void main(String[] args) {  
        Program obj = new Program(); // create an object  
  
        System.out.println("Square of 5 is: " + obj.getSquare(5));  
    }  
}
```

Output

```
sum is 40
```

static methods are also called class methods. You can call **static** methods directly using the class name.

Example 03: Using **static** method.

```
class Program {  
    static int getSquare(int x){  
        return x * x;  
    }  
    public static void main(String[] args) {  
        System.out.println("Square of 5 is " + Program.getSquare(5));  
        System.out.println("Square of 5 is " + getSquare(5));  
    }  
}
```

Output

```
Square of 5 is 25  
Square of 5 is 25
```

When you are accessing a static member from another class, you have to use the class name to access it. However, if the static member is accessed from inside the class, it can be accessed directly; the class name becomes optional.

In every Java program, you have to declare the main method **static**. It is because to run the program, the JVM should be able to invoke the `main()` method during the initial phase where no objects exist in the memory.

2.2 Type of Variables

There are three types of variables: local, instance, and **static**.

2.2.1 Local Variables

Local variables are variables declared inside the body of the method. They are declared in methods, constructors, or blocks.

Local variables are created when the method, constructor or block is entered and the variable will be destroyed once it exits the method, constructor, or block. They are visible only within the declared method, constructor, or block.

Access modifiers cannot be used for local variables.

2.2.2 Instance Variables

Instance variables are also called **non-static variables**. To access an instance variable, you have to use an object of the class.

In Java, when you create objects of a class, every object will have its own copy of all the variables of the class.

Example 04: Using instance variables.

```
class Program {  
    int var = 10;  
  
    public static void main(String[] args) {  
        Program obj1 = new Program();  
        Program obj2 = new Program();  
  
        obj1.var = 20;  
  
        System.out.println(obj1.var);  
        System.out.println(obj2.var);  
    }  
}
```

Output

```
20  
10
```

2.2.3 static Variables

static variables are also known as a **class variable**. If you declare a variable **static**, all objects of the class share the same **static** variable. It is because like **static** methods, **static** variables also belong to class not to any specific object of the class.

Example 05: Using **static** variables.

```
class Program {
    static int var = 10;

    public static void main(String[] args) {
        Program obj1 = new Program();
        Program obj2 = new Program();

        Program.var = 20;

        System.out.println(obj1.var); // Warning
        System.out.println(obj2.var); // Warning
        System.out.println(Program.var);
    }
}
```

Output

```
20
20
20
```

static variables are created at the start of program execution and destroyed automatically when execution ends.

static variables are rarely used in Java. Instead, the static constants are used. These static constants are defined by **static final** keyword.

Example 06: User **static** constants.

```
class Program {
    static final int var = 10;

    public static void main(String[] args) {
        //Program.var = 20; //Error

        System.out.println("var = " + Program.var);
    }
}
```

Output

```
var = 10
```

`static` variables cannot be declared within a method, constructor or within a block of code. It can be declared on the class level only.

```
class Program {  
    public static void main(String[] args) {  
  
        static int var = 10; // Error  
    }  
}
```

3. Passing Parameters

There are basically two types of techniques for passing the parameters in some modern programming languages, they are **pass-by-value** and **pass-by-reference**. Basically, pass-by-value means that the actual value of the variable is passed, while pass-by-reference means the reference (address) is passed.

However, in Java, the pass by reference concept is degraded. It supports only the pass by value concept.

In the pass-by-value concept, the called method copies the value of actual parameters. Since the work is done on a copy, the original values are not affected by the changes.

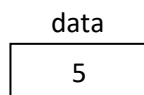
3.1 Primitive Type Vs. Reference Type

- Primitive Type: `boolean`, `char`, `byte`, `int`, `short`, `long`, `float`, `double`.
- Reference Type (or Non- primitive Type): `String`, `Arrays`, etc. Default value is `null`.

Let's see the difference:

Example 1:

```
int data = 5 // The value of the data variable is 5
```



Example 2:

```
Integer data = 5 // The value of the data variable is an address (or reference)  
                // that references to 5
```



Java is pass by value, and it is not possible to pass primitive types by reference in Java.

Example 07: Passing by values in case of primitive types.

```
class Program {
    int data = 10;

    void change(int data){
        data = 50; //changes the local variable only
    }
    public static void main(String args[]){
        Program obj = new Program();

        System.out.println("Before change, data = "+ obj.data);

        obj.change(obj.data);
        System.out.println("After change, data = "+ obj.data);
    }
}
```

Output

Before change, data = 10
After change, data = 10

Example 08: Passing by values in case of reference types.

```
class Program {
    int data = 10;

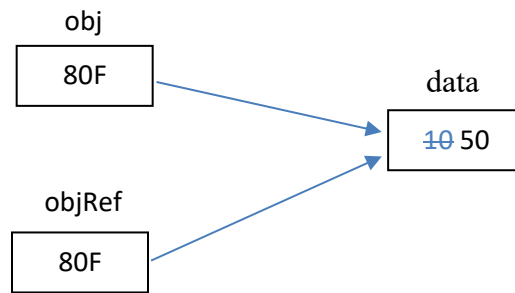
    void change(Program objRef){
        objRef.data = 50;
    }
    public static void main(String[] args) {
        Program obj = new Program();

        System.out.println("Before change, data = "+ obj.data);

        obj.change(obj);
        System.out.println("After change, data = "+ obj.data);
    }
}
```

Output

Before change, data = 10
After change, data = 50



So, in Java, parameters of primitive types are passed-by-value which is same as pass-by-value in some other languages, and parameters of reference types are also passed-by-value (the reference is copied) which is same as pass-by-reference in some other languages. So, in Java all parameters are passed by value only.

3.2 Immutable Classes

An object is known as immutable if its state cannot be changed after the object creation. Immutable Classes are needed because, in current days, most of the applications are running into multi-threading environment which results into concurrent modification problems.

Java provides immutable classes such as all wrapper classes ([Double](#), [Float](#), [Long](#), [Integer](#), [Short](#), [Byte](#), [Character](#), [Boolean](#)), [String](#), and [more](#).

Example 09: Passing by values in case of [String](#).

```

class Program {
    String data = "Hello";

    void change(String data){
        data = "Hello World";
    }
    public static void main(String args[]){
        Program obj = new Program();

        System.out.println("Before change, data = "+ obj.data);

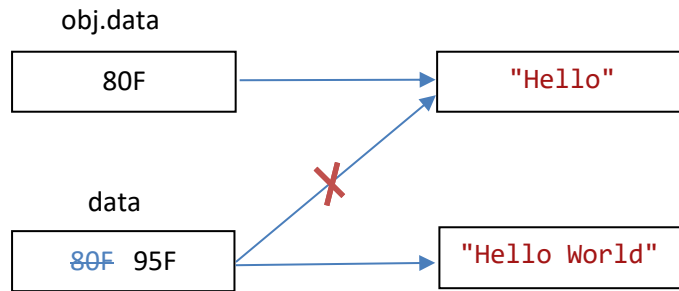
        obj.change(obj.data);
        System.out.println("After change, data = "+ obj.data);
    }
}
  
```

Output

```

Before change, data = Hello
After change, data = Hello
  
```


`String` is a reference type, but it is immutable. The string object `data` in the `change` method at first pointed to the exact same string object `obj.data` as in the caller method, which is the `main` method. Then, the `data` in the `change` method changes its reference to a new string object `"Hello World"`. But nothing has happened to the `"Hello"` string object and `obj.data` is still referring to that.



4. Overloading Methods

Unlike C++ or Python, Java does not support assigning a default value to a method parameter. To simulate default parameters in Java, use of **method overloading**.

Two or more methods can have the same name inside the same class if they accept different parameters. This feature is known as **method overloading**.

To overload the methods, we can either change the number of parameters or change the data type of parameters. It is not method overloading if we only change the return type of methods.

Example 10: Overloading by changing the number of parameters

```

class Program {
    static void display(int a){
        System.out.println("Arguments: " + a);
    }
    static void display(int a, int b){
        System.out.println("Arguments: " + a + " and " + b);
    }
    public static void main(String[] args) {
        display(1);
        display(1, 2);
    }
}
  
```

Output:

```

Arguments: 1
Arguments: 1 and 2
  
```

Example 11: Method Overloading by changing the data type of parameters.

```
class Program {
    static void display(int a){    // This method accepts int
        System.out.println("Got Integer value");
    }
    static void display(String a){ // This method accepts String object
        System.out.println("Got String value");
    }
    public static void main(String[] args) {
        display(1);
        display("Hello");
    }
}
```

Output

```
Got Integer value
Got String value
```

Here, both overloaded methods accept one argument. However, one accepts the argument of type int whereas other accepts String object.

5. Arrays and Methods

5.1 Passing Arrays to Methods

Just as you can pass primitive type values to methods, you can also pass arrays to methods. When passing an array to a method, the reference of the array is passed to the method.

Example 12: The following method displays the elements in an integer array.

```
class Program {
    static void printArray(int[] array) {
        for (int i = 0; i < array.length; i++) {
            System.out.print(array[i] + " ");
        }
    }
    public static void main(String[] args) {
        printArray(new int[]{3, 1, 2, 6, 4, 2});
    }
}
```

Output

```
3 1 2 6 4 2
```

Example 13: The following method updates the elements in an integer array.

```
import java.util.Arrays;

class Program {
    static void updateArray(int[] array) {
        Arrays.fill(array, 8);
    }
    public static void main(String[] args) {
        int[] array = {3, 1, 2, 6, 4, 2};

        updateArray(array);

        System.out.print(Arrays.toString(array));
    }
}
```

Output

```
[8, 8, 8, 8, 8, 8]
```

Example 14: Passing 2D array to a method.

```
import java.util.Arrays;

class Program {
    static void updateArray(int[][] array2D) {
        array2D[1][1] = 100;
    }
    public static void main(String[] args) {
        int[][] array2D = {
            {3, 1, 2, 6, 4, 2},
            {1, 6, 4, 2},
        };

        updateArray(array2D);

        System.out.print(Arrays.deepToString(array2D));
    }
}
```

Output

```
[[3, 1, 2, 6, 4, 2], [1, 100, 4, 2]]
```

5.2 Returning an Array from a Method

You can pass arrays when invoking a method. A method may also return an array. When a method returns an array, the reference of the array is returned.

Example 15: The following method returns an array that is the reversal of another array.

```
import java.util.Arrays;

class Program {
    static int[] reverse(int[] array) {
        int[] result = new int[array.length];

        for (int i = 0, j = result.length - 1; i < array.length; i++, j--) {
            result[j] = array[i];
        }
        return result;
    }
    public static void main(String[] args) {
        int[] array1 = {10, 25, 3, 15, 5, 60};
        int[] array2 = reverse(array1);

        System.out.print(Arrays.toString(array2));
    }
}
```

Output

```
[60, 5, 15, 3, 25, 10]
```

Example 16: Returning 2D array from a method.

```
import java.util.Arrays;

class Program {
    static int[][] getUpdatedArray(int[][] array2D) {
        array2D[1][1] = 100;

        return array2D;
    }
    public static void main(String[] args) {
        int[][] array2D = {
            {3, 1, 2, 6, 4, 2},
            {1, 6, 4, 2},
        };
        int[][] result = getUpdatedArray(array2D);

        System.out.println(Arrays.deepToString(result));
    }
}
```

Output

```
[[3, 1, 2, 6, 4, 2], [1, 100, 4, 2]]
```

6. Variable-Length Argument

A method with variable length arguments **can have zero or multiple arguments**. Variable length arguments are most useful when the number of arguments to be passed to the method is not known beforehand. They also reduce the code as overloaded methods are not required.

The parameter in the method is declared as follows:

`dataType... parameterName`

In the method declaration, you specify the type followed by an ellipsis (...). Only one variable-length parameter may be specified in a method, and this parameter must be the last parameter. Any regular parameters must precede it.

Java treats a variable-length parameter as an array. You can pass an array or a variable number of arguments of the same type to a variable-length parameter. When invoking a method with a variable number of arguments, Java creates an array and passes the arguments to it.

Example 17: The following method prints the maximum value in a list of an unspecified number of values.

```
class Program {
    static void printMax(double... numbers) {
        if (numbers.length == 0) {
            System.out.println("No argument passed");

            return;
        }

        double result = numbers[0];

        for (int i = 1; i < numbers.length; i++)
            if (numbers[i] > result)
                result = numbers[i];

        System.out.println("The max value is " + result);
    }
    public static void main(String[] args) {
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }
}
```

Output:

```
The max value is 56.5
The max value is 3.0
```

Exercises

In all the following exercises, you are asked to write the following methods and test programs that call the methods to show that they work as intended. Note: you can create more methods per your needs.

1. (*Twin primes*) Twin primes are a pair of prime numbers that differ by 2. For example, 3 and 5 are twin primes, 5 and 7 are twin primes, and 11 and 13 are twin primes. Write a method called `displayTwinPrimes` that displays all twin primes less than 1200. Here is a sample run:

```
The twin prime numbers which are less than 1200:
(3, 5)
(5, 7)
...
```

2. (*Even palindromes*) A palindrome is a word, number, or other sequence of characters which reads the same backward as forward. Write a method called `generateEvenNumeralPalindrome` that asks the user how many even palindromes they want, then **returns an array** that contains those palindromes. Write a test program as below:

```
Enter the number of even palindromes: 10 Enter
0 2 4 6 8 22 44 66 88 202
```

3. (*Anagram*) Two words are anagrams if they contain the same letters in different orders, for example, *binary* and *brainy*. Write a method called `isAnagram` that takes two strings and returns `true` if they are anagrams, otherwise, returns `false`.
4. Write a method called `split` that will accept a sentence then split it into words, and returns an array of those words. Note: the input sentence might contain special characters, but words cannot. Also, you are not allowed to use the `split()` method from the `String` class.
5. Write a method called `removeSubstring` that accepts two strings named `substring` and `string`, and removes all the occurrences of the `substring` from the `string`, then return the new string.
6. Write a method called `verbose` that, given an integer number less than 10^9 , returns the number in English. Example: `verbose(987123456)` will return `nine hundred eighty-seven million, one hundred twenty-three thousand, four hundred fifty-six`.
7. (*Shuffle rows*) Write a method that shuffles the column by column in a 2D array of characters using the following header: `public static void shuffleRows(char[][] array2D)`

Write a test program that shuffles the following matrix:

```
char[][] array2D = {
    {'A', 'B', 'C', 'D'},
    {'A', 'B', 'C', 'D'},
    {'A', 'B', 'C', 'D'},
    {'A', 'B', 'C', 'D'}
};
```

8. Write a method that takes in a 2D array of numbers and find the value of the **second largest** number stored in that array. Use the following header:

```
public static int secondLargest(int[][] array2D)
```

9. Write a method called `countOccurrences` that accepts **unspecified** number of integers, then counts the occurrences of each. Here is a sample run:

```
Enter integers: 2 6 3 3 2 3 5 7 7 9 Enter
The occurrences of each number:
2 occurs 2 times
6 occurs 1 time
3 occurs 4 times
...
```

10. Write a method called `displayLeaders` that accepts **unspecified** number of integers, then prints all the LEADERS. An element is a leader if it is greater than all the elements to its right side. And the rightmost element is always a leader. Write a test program that randomly generate an array of 20 integers and displays the leaders.

Reference

- [1] Y. Daniel Liang. 'Introduction to Java Programming', 11e – 2019
[2] <https://www.programiz.com/java-programming>