Methods

30 December 2025 05:04 AM

1. Method:

A method declares executable code that can be invoked, passing a fixed number of values as arguments.

2. The Benefits of the Method:

1. A method is a way of reducing code duplication.
2. A method can be executed many times with potentially different results, by passing data to the method in the form of arguments.

3.

Structure of the Method

One of the simplest ways to declare a method is shown on this slide.

This method has a name, but takes no data in, and returns no data from the method (which is what the special word **void** means in this declaration).

```
public static void methodName() {

    // Method statements form the method body

}
```

4.

Executing a Method as a Statement

To execute a method, we can write a statement in code, which we say is calling, or invoking, the method.

For a simple method like calculateScore, we just use the name of the method, where we want it to be executed, followed by parentheses, and a semi-colon to complete the statement.

So for this example, the calling statement would look like the code shown here:

```
calculateScore();
```

5. Parameters or Arguments?

1. **Parameter:** A parameter is the definition as shown in the method declaration
2. **Argument:** Argument will be the value that's passed to the method when we call it.

6.

Executing a Method with parameters

To execute a method that's defined with parameters, you have to pass variables, values, or expressions that match the type, order and number of the parameters declared.

In the calculateScore example, I declared the method with four parameters, the first; a boolean, and the other three of int data types.

So we have to pass first a boolean, and then 3 int values as shown in this statement:

```
calculateScore(true, 800, 5, 100);
```

I can't pass the boolean type in any place, other than as the first argument, without an error.

7.

Executing a Method with parameters

The statement below would cause an error.

```
calculateScore(800, 5, 100, true);
```

And you can't pass only a partial set of parameters as shown here.

This statement, too, would cause an error.

```
calculateScore(true, 800);
```

8.

Method structure with parameters and return type

```
// Method return type is a declared data type for the data that
// will be returned from the method
public static dataType methodName(p1type p1, p2type p2, {more}) {

    // Method statements
    return value;

}
```

So, similar to declaring a variable with a type, we can declare a method to have a type.

This declared type is placed just before the method name.

In addition, a return statement is required in the code block, as shown on the slide, which returns the result from the method.

9. The return statement

1. Java states that a return statement **returns** control to the invoker of a method.
2. The most common usage of the return statement, is to return a value back from a method.
3. In a method that doesn't return anything, in other words, a method declared with void as the return type, a return statement is not required. It is assumed and execution is returned after the last line of code in the method is executed.
4. But in methods that do return data, a return statement with a value is required.

10. Is the method a statement or an expression?

1. a method can be a statement or an expression in some instances.
2. Any method can be executed as a statement.
3. A method that returns a value can be used as an expression, or as part of any expression.

11. What are functions and procedures?

1. Some programming languages will call a method that returns a value, a **function**, and a method that doesn't return a value, a **procedure**.

12. Method Declaration

1. This consists of:
    1. Declaring Modifiers. These are keywords in Java with special meanings, we've seen public and static as examples, but there are others.
    2. Declaring the return type.
        1. **void** is a Java keyword meaning no data is returned from a method.
        2. Alternatively, the return type can be any **primitive data type or class.**
        3. If a return type is defined, the code block must use at least one return statement, returning a value, of the declared type or comparable type.
2. Lower camel case is recommended for method names.
3. Declaring the method parameters in parentheses. A method is not required to have parameters, so a set of empty parentheses would be declared in that case.
4. Declaring the method block with opening and closing curly braces. This is also called the **method body.**

13. Parameters Declaration

1. Parameters are declared as a list of comma-separated specifiers, each of which has a parameter type and a parameter name (or identifier).
2. Parameter order is important when calling the method.
3. The calling code must pass arguments to the method, with the same or comparable type, and in the same order, as the declaration.
4. The calling code must pass the same number of arguments, as the number of parameters declared.

14. Method Signature

1. A method is uniquely defined in a class by its name, and the number and type of parameters that are declared for it.
2. This is called the method signature.
3. You can have multiple methods with the same method name, as long as the method signature (meaning the parameters declared) are different.

15. Default values for parameters.

1. In many languages, methods can be defined with default values, and you can omit passing values for these when calling the method.
2. But Java doesn't support default values for parameters.

16.

The Return Statement for methods that have a return type

If a method declares a return type, meaning it's not void, then a return type is required at any exit point from the method block.

Consider the method block shown here:

```
public static boolean isTooYoung(int age) {
    if (age < 21 ) {
        return true;
    }
}
```

17.

The Return Statement for methods that have a return type

So in the case of using a return statement in nested code blocks in a method, all possible code segments must result in a value being returned.

The following code demonstrates one way to do this:

```
public static boolean isTooYoung(int age) {
    if (age < 21 ) {
        return true;
    }
    return false;
}
```

18.

The Return Statement for methods that have a return type

One common practice is to declare a default return value at the start of a method, and only have a single return statement from a method, returning that variable, as shown in this example method:

```
public static boolean isTooYoung(int age) {
    boolean result = false;
    if (age < 21 ) {
        result = true;
    }
    return result;
}
```

19.

The Return Statement for methods that have void as the return type

The return statement can return with no value from a method, which is declared with a **void** return type.

In this case, the return statement is optional, but it may be used to terminate execution of the method at some earlier point than the end of the method block, as shown here:

```
public static void methodDoesSomething(int age) {
    if (age > 21) {
        return;
    }
    // Do more stuff here

}
```

20. Method Overloading

1. Method overloading occurs when a class has multiple methods with the same name, but the methods are declared with different parameters.
2. So, you can execute multiple methods with the same name, but call it with different arguments.
3. Java can resolve which method it needs to execute based on the arguments being passed when the method is invoked.

a. More on Method Signature:

1. A method signature consists of the name of the method, and the uniqueness of the declaration of its parameters.
2. In other words, a signature is unique, not just by the method name, but in combination with the number of parameters, their types, and the order in which they are declared.
3. A method's return type is not part of the signature.
4. A parameter name is also not part of the signature..

b. Valid Overloaded Methods

1. The type, order, and number of parameters, in conjunction with the name, make a method signature unique.
2. A unique method signature is the key for the Java compiler, to determine if a method is overloaded correctly.
3. The name of the parameter is not part of the signature, and therefore it doesn't matter, from Java's point of view, what we call our parameters.

4.

Valid Overloaded Methods

This slide demonstrates some valid overloaded methods, for the doSomething method.

```
public static void doSomething(int parameterA) {
    // method body
}

public static void doSomething(float parameterA) {
    // method body
}

public static void doSomething(int parameterA, float parameterB) {
    // method body
}

public static void doSomething(float parameterA, int parameterB) {
    // method body
}

public static void doSomething(int parameterA, int parameterB, float parameterC) {
    // method body
}
```

Invalid Overloaded Methods

Parameter names are not important when determining if a method is overloaded.

Nor are return types used when determining if a method is unique.

```
public static void doSomething(int parameterA) {
    // method body
}

public static void doSomething(int parameterB) {
    // method body
}

public static int doSomething(int parameterA) {
    return 0;
}
```