Theory: Declaring a method

18 minutes 5 / 13 problems solved

Start practicing

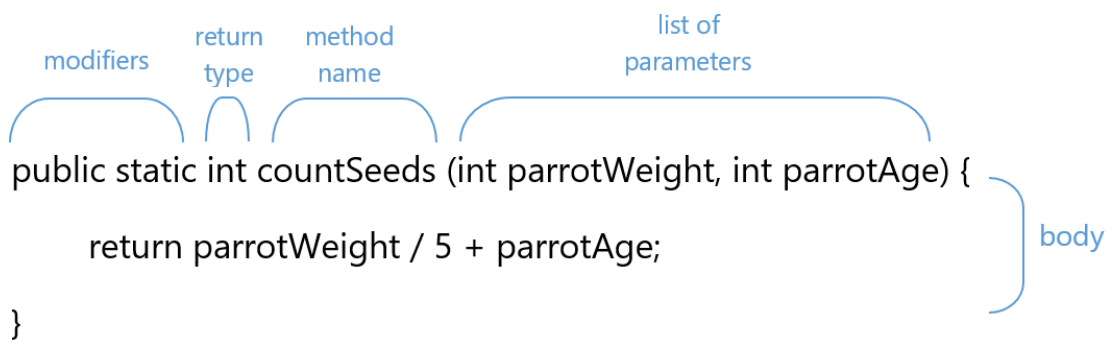
The built-in method is a real deal when you manage routine tasks. Still, it is not a cure-all solution since it's impossible to provide a standard method for each specific task. For example, you may need to convert dollars to euros, count your monthly spendings, or even calculate a daily portion of seeds for your parrot multiple times. That's when you create your own method to avoid repeating tonnes of code!

By contrast with built-in methods, **user-defined** methods are created by the programmer. It is a common practice to create a customized subprogram for a specific purpose.

But how to create it? Let's figure it out.

§1. The syntax of the method

Technically, a method is just a structured part of the code with a few components. In Java, a method is always located inside a **class**. Let's take a closer look at the method that calculates a daily portion of seeds for the parrot:



A method contains a **set of modifiers**, a **type of the return value**, a **name**, a list of **parameters** in parentheses `()` , and a **body** in curly brackets `{}`. The combination of the name of the method and the list of its parameters is known as a method **signature**. In our example, the signature is `countSeeds(int, int)`.

Some methods also have a list of exceptions – they define its behavior in case of some mistake in the program. For now, we'll consider simple methods without exceptions.

Let's focus on the main components of the method that we need to write simple methods from scratch.

§2. Modifiers

First words are so-called modifiers. There are two types of modifiers in Java: **access modifiers** and **non-access modifiers**.

Access modifiers define the visibility of the method. For now, we're using a `public` access modifier, which means there are no restrictions for invoking the method even from other classes.

Non-access modifiers provide information about their behavior to JVM. The modifier `static` means that the method belongs to the class and it can be accessed without creating any object. This type of method is called a **static method**.

If the method is declared without `static` modifier, it means that the method can be invoked only through or with an object of this class, or its instance. Such methods are called **instance methods**.

Current topic:
✓ Declaring a method Stage 3

Topic depends on:
✓ Calling a method Stage 2

Topic is required for:
✓ The main method Stage 3
✓ Overloading
✓ Functional decomposition Stage 3
Arrays as parameters
Call stack
Constructor Stage 6
Instance methods Stage 6

Remember that there is a recommended order for the modifiers that you can find in Java Language Specification. In our case, it is `public static`.

§3. Method parameters

In parentheses after the method name, we define the type, number, and order of the parameters. This reflects how they will be passed to the method when it is invoked. Take a look at these signatures:

```
convertEuroToDollars(double dlrRate, long eur);

countMonthlySpendings(long food, long rent, long fun);

replace(char a, char b);
```

As you know, there are also methods that don't have values passed to them. These methods are known as *non-parameterized*.

§4. Body and return statement

Before a method completes its execution and exits, it returns a value known as a **return value**. The result of your method execution can be a value of primitive types like `int`, `float`, `double`, `boolean`, or the reference types like `String`. Take a look at our `countSeeds` method with a returning type `int`:

```
public static int countSeeds(int parrotWeight, int parrotAge) {
    return parrotWeight / 5 + parrotAge; // it returns an int
}
```

What you see inside the curly brackets is known as the **body** of the method. The body holds the logic we want to implement by our method: a set of statements to perform with the passed values to obtain the result. Our method `countSeeds` takes two provided values, performs certain calculations, and returns the result within a **return statement**.

Methods do not necessarily have to return a value. If we want a method not to return a value but to perform operations only, the keyword `void` is used as a return type:

```
public static void countSeeds(int parrotWeight, int parrotAge) {

System.out.println("give your parrot " + (parrotWeight / 5 + parrotAge) +
    "g of seeds per day");
}
// this method just prints the line, so it returns no value
```

This method prints the line with recommendations on feeding the parrot and does not allow us to keep value. Note that there is no return statement.

Though methods with `void` return type return nothing, you still may use a `return` word to exit the method. Usually, it's applicable to the methods with conditions. Take a look:

```
public static void isPositive(int num) {
    if (num > 0) {
        System.out.println("the number is positive");
    } else {
        return;
    }
}
```

Remember, that if you try to return a value from a method with a **void** return type, a compile error will be thrown.

§5. What happens when we invoke a method

When invoking a method, we can write the returned value to some variable, print it out, or pass on to another method. That's how it looks like in a program:

```
int myParrotWeight = 100;
int myParrotAge = 3;

int myParrotPortion = countSeeds(myParrotWeight, myParrotAge);
// now myParrotPortion equals 23
```

There's another important thing to remember. When you pass a variable of a primitive type to a method, a copy of this variable is created with the same value. Inside a method, only this copy is processed. Any changes to this copy will not affect the variable that was passed.

Take a look:

```
public static void main(String[] args) {
    int portion = 100;
    addSeeds(portion); // try to change portion

    // now let's print a portion
    System.out.println(portion);
    // 100, because the method didn't change portion, only its copy
}


/**

 * The method increases the portion of seeds by 50

 * and prints the resulting value

 */

public static void addSeeds(int portion) {

    portion += 50;

    System.out.println("The increased portion is " + portion);

}
```

The output will look like:

```
The increased portion is 150
100
```

Although the `addSeeds` method changes the passed argument, it happens to a different variable with its own value, leaving the value of the `portion` we've created intact.

§6. Conclusion

As you see, a method from the inside is a block of code that contains a set of modifiers, a return type, a name of a method, a list of its parameters, and a body of a method. A method may return a value or return nothing, which is indicated with the `void` keyword.

If you know the syntax of the method, you can create your own and adjust it for your needs. This practice will make your code more structured and reusable.

Report a typo

184 users liked this theory. **0** didn't like it. **What about you?**

Start practicing

This content was created 4 months ago and updated 10 days ago. Share your feedback below in comments to help us improve it!

Comments (5) Hints (0) Useful links (0) Show discussion