Theory: Functional decomposition

18 minutes 5 / 7 problems solved

Start practicing

You already know how to create simple methods in Java. This is a very useful skill that can help you shorten your code, reuse some operations, and make your program more readable.

Step by step, your programming tasks are becoming more complex, so are your methods. Though you can create a complex program that is wrapped in one solid method or even in a `main` method, it is better to divide a program into a number of more specific methods that are easy to read and understand. The approach of dividing a complex program into subroutines is called **functional decomposition**.

In this topic, we'll see how to decompose the solution of a particular problem into methods.

§1. Solving complex tasks

The idea of simple decomposing some problems into several subproblems is quite intuitive. If you want to cook a pizza, you don't just put all the ingredients in the oven: instead, you break the process up into separate tasks – from making the dough to actual cooking. *Functional* decomposition is not about cooking pizza, but it is based on the same principle of breaking a problem into small pieces called methods.

Let's consider an example. Think of a program that simulates the Smart home app. This app is used to control home devices that can be remotely accessed: wireless speaker systems, lights, home security, door locks, and even robots. Imagine that we have a simple Smart home app that can perform three actions: turn the music on or off, switch the light on and off, and control the door lock. Let's consider these actions as parts of our computer program.

If we decompose this task, that's how its algorithm can be described in general:

1. Parse the input data (entered password);
2. Check that the password is correct;
3. Ask the user what they want to do;
4. If the action is supported, perform it.

Imagine that you wrapped this program in code, but without a single method. That's how its structure would look like:

Current topic:

Functional decomposition Stage 3

Topic depends on:

Switch statement Stage 3

Declaring a method Stage 3

The main method Stage 3

Topic is required for:

Algorithms in Java

Defining classes Stage 3

Lambda expressions

Table of contents:

↑ Functional decomposition

§1. Solving complex tasks

§2. Decomposing a program into methods

§3. Adding new features

§4. Conclusion

Feedback & Comments

```
        // ...
        int password = 76543210;
        String speakersState;
        String lampState;
        String doorState;

        // reading the password
        System.out.println("Enter password: ");
        int passwordInput = scanner.nextInt();



        // checking if the password is correct

        if (passwordInput != password) {

            System.out.println("Incorrect password!");

        } else {

            // asking the user what they want to do

            System.out.println("Choose the object: 1 – speakers, 2 – lamp, 3 – door");

            String action = scanner.next();



            switch (action) {

                case "1":

                    // asking the user about speakers


                    switch (speakersState) {

                        case "on":

                            // ...

                        case "off":

                            // ...

                        default:

                            // ...

                    }

                    break;

                case "2":

                    // asking the user about lights...

                case "3":

                    // asking the user about the door...

                }

        }
```

Though you see just a truncated version of a real program, this code still looks overloaded. At the same time, it works perfectly fine for our problem and we could leave it like that. However, later on we might want to adjust it for our needs or extend its functionality.

What if we want this code to work for many users and not just one, or to expand the number of actions, make them more complex? Some parts of this code would be useful, and some of them would probably be deleted. To make this code less specific and more flexible, we can use *functional* decomposition.

§2. Decomposing a program into methods

Functional decomposition is simply a process of decomposing a problem into several functions or methods. Each method does a particular task so that we can perform these methods in a row to get the results we need. When we look at a problem, we need to think which actions we may want to repeat multiple times or, alternatively, perform separately. This is how we get the desired methods. As a result, these methods are easier to read, understand, reuse, test, and debug.

Let's look at our Smart home app again and figure out which steps can be turned into separate methods. First of all, we can separate our main operations into three methods: one method to control the music, another one to turn the lights on and off, and the third to operate the door lock. Take a look at the method `controlMusic()` that controls the music.

Methods `controlLight()` and `controlDoor()` follow the same algorithm.

```
// method that turns the music on and off

public static void controlMusic() {
    Scanner scanner = new Scanner(System.in);
    System.out.println("on/off?");
    String tumbler = scanner.next();
    if (tumbler.equals("on")) {
        System.out.println("The music is on");
    } else if (tumbler.equals("off")) {

        System.out.println("The music is off");

    } else {

        System.out.println("Invalid operation");

    }

}
```

These controlling methods perform the main actions that our app provides. Of course, these actions are greatly simplified, but the main goal here is to show the process of revising the functionality of our program.

To make things work, we need to create a method that checks the password.

```

// method that verifies the password and gives access to Smart home actions if the
 password is correct
  public static void accessSmartHome() {
      Scanner scanner = new Scanner(System.in);
      final int password = 76543210;
      System.out.println("Enter password: ");
      int passwordInput = scanner.nextInt();
      if (passwordInput == password) {
          chooseAction();
      } else {

          System.out.println("Incorrect password!");

      }

  }
```

Also, we need a method with the main menu where you can choose the action, so we created a method `chooseAction()`. This method asks the user what action they want to perform and gives control to the method that performs the picked action.

Finally, we can run our decomposed program in the `main` method, which is called once our program is started:

```
public static void main(String[] args) {
    accessSmartHome();
}
```

This method calls `accessSmartHome`, which asks to enter a password and, if it is correct, allows us to manage the Smart home.

§3. Adding new features

Now, if we want to add another action, all we have to do is define the method with this action. For example, we've got a new Smart device, an electric kettle. We create a method that switches it on and off. To get access to the new method, we need to modify the `chooseAction()` method by adding a new case statement:

```
// method that controls electric kettle
public static void controlKettle() {
    // ...
}

// method with the main menu for choosing the action
public static void chooseAction() {
    Scanner scanner = new Scanner(System.in);
// adding case 4

    System.out.println("Choose the object: 1 – speakers, 2 – lamp, 3 – door, 4 – kettle");

    // ...

        case 4:

            controlKettle();

            break;

    // ...

    }
```

As you see, we now have a real functioning program that won't fall apart if we decide to change it a bit. We can easily test separate components since they are determined in separate methods. This also makes it easier to support the program in the future.

§4. Conclusion

In this topic, we have learned the importance of functional decomposition. It is used to divide the program into several methods. This approach has a lot of advantages. It helps us to:

- structure the code;
- follow the general logic of the program;
- make changes easily;
- test separate methods.

Functional decomposition is not key to everything, but using this approach will help you create neat and understandable programs that are easy to work with.

Report a typo

241 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

This content was created 6 months ago and updated 5 days ago. Share your feedback below in comments to help us improve it!

Comments (8) Hints (0) Useful links (0) Show discussion