Theory: Scanning the input

24 minutes 5 / 9 problems solved

The **standard input** is a stream of data going into a program. It is supported by the operating system. By default, the standard input obtains data from the keyboard input but it's possible to redirect it to a file.

Actually, not all programs need to use the standard input. But we will often use it here to help you master your programming skills! The typical way to solve programming problems is the following:

1. Read data from the standard input (System.in);
2. Process data to obtain a result;
3. Output the result to the standard output (System.out).

This type of code challenges can be easily tested by different formats of input data, and for this reason, we will use them a lot.

§1. Reading data with a scanner

The simplest way to obtain data from the standard input is to use the standard class `Scanner`. It allows a program to read values of different types (string, numbers, etc) from the standard input. In this topic, we will consider reading data from the input.

To use this class you should add the following import statement to the top of your file with the source code.

```
1 import java.util.Scanner;
```

Then you add the following construction after the import:

```
1 Scanner scanner = new Scanner(System.in);
```

With this line, we create an object of `Scanner` class, that enables us to use its methods. We will learn more about creating objects in other topics. `System.in` indicates that the program will read text that you type in the standard input. For now, you will always require this line exactly.

There are two ways to read strings with a `Scanner` class. If your input is an integer number or a single word, you can read the data using `next()` **method**. As an example, the following code fragment reads the user's name and prints *hello* message:

```
1 String name = scanner.next();
2
3 System.out.println("Hello, " + name +"!");
```

For instance, the user's name is James. The output of the program will be the following:

```
1 Hello, James!
```

If the user's input is an integer number like 123, the program will output this number. Note that `next()` method will store 123 or another integer number as a string, even if we know that this string consists of a number.

```
1 Hello, 123!
```

There are more specialized methods for reading other types of input values. In this topic, we only consider reading strings.

But, if the user prints a compound name like Erich Maria, the program will output only the first word:

Current topic:
Scanning the input Stage 1

Topic depends on:
Types and variables Stage 1
Comments Stage 1

Topic is required for:
Integer types and operations Stage 1
What are streams
Errors in programs Stage 7

Table of contents:
↑ Scanning the input
§1. Reading data with a scanner
§2. Reading a multiline input
§3. Conclusion
Feedback & Comments

```
Hello, Erich!
```

In this case, you'll need another method, a `nextLine()` **method,** which reads and outputs the whole line:

```
Hello, Erich Maria!
```

As you may notice, `next()` method reads one word only and doesn't include any whitespace. By contrast, `nextLine()` method includes all space characters it encounters.

> Note that in Java **whitespace** includes not only space character, but mostly everything that looks empty when printed: tab, the newline character, and other non-printing characters.

In this article, we are dealing with space and newline characters: technically, we produce a corresponding character when pressing *Enter* and starting a new line. The term "whitespace" is used to refer to either of them. The more correct term to refer to what we've called "word" is **token**: it is a piece of text surrounded by whitespace. We can say now that `next()` method finds and returns the next token, while `nextLine()` reads all data till the end of the current line.

Now you can read a whole word and even a whole line invoking these two methods. To invoke both of them correctly, it is important to know the difference between them.

§2. Reading a multiline input

Reading multiline input may still be a bit tricky: you should take into account the position of a cursor and the reading methods behavior.

Let's investigate this process with an example:

```
|This is a simple

multiline input,

that is being read
```

| is a position of a cursor before reading the input.

If we invoke `next()` method, the program will read the input till the whitespace, which is indicated by blue color:

```
This| is a simple

multiline input,

that is being read
```

After invoking `nextLine()` method the program reads the whole line starting from the whitespace. This data is indicated by green color. `nextLine()` places the cursor at the beginning of a new line (if there is such a line in your input):

```
This is a simple

|multiline input,

that is being read
```

Then, let's invoke `next()` method two times. The first input is indicated by orange color. You may see that the position of the cursor is right after the word and before the whitespace:

```
This is a simple

multiline| input,

that is being read
```

Now we invoke `next()` method again. The program outputs the second word in the line ***without*** whitespace. It doesn't even matter how many space characters are there, because `next()` method will skip the whitespace until it finds the next token.

The second input is indicated by light blue color. As you may see, the position of the cursor is still at the current line right before the new line and after the comma:

```
This is a simple

multiline input,

that is being read
```

Here is a tricky thing about the `nextLine()` method that also shows a major difference between `next()` and `nextLine()` methods. As you already know, the program will read input from the position of the cursor till the new line (and again, if there is such a line in your input). In this example the cursor is located before the new line: thus, the `nextLine()` method will return an empty line ("") and place the cursor at the beginning of a new line.

```
This is a simple

multiline input,

that is being read
```

To sum up, let's look at the code as a whole and consider the variables we have just read:

```
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        String word1 = scanner.next(); // "This"
        String line1 = scanner.nextLine(); // " is a simple"

        String word2 = scanner.next(); // "multiline"

        String word3 = scanner.next(); // "input,"

        String line2 = scanner.nextLine(); // ""



    }

}
```

This example may seem artificial, but it should help you to catch the difference between these two methods. Remember that usually the variables are named in a more expressive way.

§3. Conclusion

We can read data from the standard input with a special `Scanner` class. `next()` and `nextLine()` methods will help you to read strings. Both of them are used for getting input, but they act differently. `next()` method can read the input only till the whitespace while the `nextLine()` method reads the input till the end of the whole line.

We recommend you to use the class `Scanner` when solving programming problems. It is one of the simplest ways to get values from the standard input. More complex ways to read data will be discussed in further topics.

Report a typo

1536 users liked this theory. **82** didn't like it. **What about you?**

Start practicing

Comments (31) Hints (1) Useful links (1) Show discussion