7-Loops II

Author: Vincent Lau

Learning Objectives

> Understand and use for-each, while & do-while loops
>
> Understand and use break and continue on the above types of loops

Loops are used to **iterate through multiple values or objects, and repeatedly run specific code blocks**. There are various types of loops in Java:

- ***for-each*** *loop*
- ***while*** *loop*
- ***do-while*** *loop*

For-each Loop

- The Java for-each loop or enhanced for loop has been introduced since J2SE 5.0.

- The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the **for-each loop.**
- It simplifies the process of iterating through the elements of an array without the need for explicit indexing.
- The drawback of the for-each loop is that it cannot traverse the elements in reverse order.

Syntax

```
for(Identifier : Collection){
    // Statements
}
```

Example 1: Arrays

Here's how you can use the for-each loop with int arrays:

```
int[] numbers = {1, 2, 3, 4, 5};

// Using for-each loop to iterate over elements of the int array
for (int number : numbers) {
    System.out.println(number);
}
```

In this example, we have an int array named `numbers` containing integers. The for-each loop iterates over each element in the `numbers` array and assigns it to the variable `number`. Inside the loop, we can perform operations using the current element, such as printing it to the console.

Here's how you can use the for-each loop with String arrays:

```
String[] names = {"Alice", "Bob", "Charlie", "David"};

// Using for-each loop to iterate over elements of the String array
for (String name : names) {
    System.out.println(name);
}
```

In this example, we have a String array named `names` containing names. The for-each loop iterates over each element in the `names` array and assigns it to the variable `name`. Inside the

loop, we can use the current element, such as printing it to the console or performing other operations.

Example 2: String.spilt()

The `split()` method is an method of the `String` class in Java. It is used to split a string into an array of substrings based on a specified delimiter or regular expression pattern. The `split()` method returns an array of strings resulting from the split operation.

Here's an example that demonstrates the usage of the `split()` method with a simple delimiter:

```
String str = "Hello,World!";
String[] parts = str.split(",");

for (String part : parts) {
    System.out.println(part);
}
```

In this example, the `split()` method is called on the `str` string with the delimiter ", ". It splits the string at each occurrence of ", " and returns an array of substrings. The `for` loop is used to iterate over the resulting array and print each substring.

The output will be:

```
Hello
World!
```

In addition to using a simple delimiter, you can also use a regular expression pattern as the argument to the `split()` method. This allows for more advanced splitting based on patterns.

Here's an example:

```
String str = "apple, banana, cherry, date";
String[] fruits = str.split(",\\s*");

for (String fruit : fruits) {
    System.out.println(fruit);
}
```

In this example, the `split()` method is called on the `str` string with the regular expression pattern ",\s*". This pattern matches a comma followed by zero or more whitespace characters. The string is split at each occurrence of this pattern, resulting in an array of substrings. The `for` loop is used to print each substring. The output will be:

```
apple
banana
cherry
date
```

It's important to note that the `split()` method treats the argument as a regular expression. If your delimiter or pattern contains special regex characters, you may need to escape them using a double backslash (`\\`). For example, if your delimiter is a dot ".", you should use `"\\."` as the argument to `split()` .

If you want to limit the number of splits performed, you can use the overloaded version of `split()` that takes an additional `limit` parameter.

```java
String str = "one, two, three, four, five";
String[] parts = str.split(", ", 3);

for (String part : parts) {
    System.out.println(part);
}
```

In this example, the `split()` method is called with a `limit` of 3. This ensures that the string is split into **at most 3 substrings**.

The output will be:

```
// output
one
two
three, four, five
```

The `split()` method is useful when you need to break a string into multiple substrings based on a delimiter or pattern. It is commonly used for parsing CSV files, tokenizing strings, extracting data from structured text, and more. By utilizing the `split()` method, you can split a string into meaningful components and process them individually.

While Loop

The `while` loop is a control flow statement in Java that allows you to repeatedly execute a block of code as long as a specified condition is true.

The *while loop* continues to run until the expression within the parentheses evaluates to *false*.

Here's an introduction to the `while` loop with examples covering various scenarios for beginners, including the use of `break` and `continue` statements:

Simple While Loop

```java
int count = 0;

while (count < 5) {
    System.out.println("Count: " + count);
    count++;
}
// what is the value of count after the while loop?
```

```
// output
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

In this example, the `while` loop iterates as long as the condition `count < 5` is true. It starts with `count` initialized to 0 and prints the value of `count`. After each iteration, the value of `count` is incremented by 1. The loop continues until the `count` becomes 5, at which point the condition becomes false, and the loop terminates.

With logical operator

```java
int num = 1;

while (num >= 1 && num <= 5) {
    System.out.println("Number: " + num);
    num++;
}
```

```
7 // what is the print out?
```

In this example, the `while` loop continues as long as `num` is greater than or equal to 1 AND less than or equal to 5. The loop iterates from 1 to 5, printing the value of `num` in each iteration.

```
1 int num = 1;
2
3 while (num < 3 || num > 5) {
4     System.out.println("Number: " + num);
5     num++;
6 } // what is the print out?
```

In this example, the `while` loop continues as long as the `num` is less than 3 OR greater than 5. The loop iterates until `num` reaches 3 because at that point, the condition becomes false (`num < 3` is false, but `num > 5` is true), and the loop terminates.

```
1 boolean condition = true;
2
3 while (!condition) {
4     System.out.println("This line won't be executed");
5 }
6 // what is the print out?
```

In this example, the `while` loop continues as long as the condition `!condition` is true. However, since `condition` is initially `true` , the logical NOT operator negates it to `false` , making the loop condition false from the start. As a result, the loop body is not executed, and the loop terminates immediately.

With break statement

```
1 int num = 0;
2
3 while (true) {
4     System.out.println("Number: " + num);
5     num++;
6     if (num == 5) {
7         break; // Terminates the loop when num reaches 5
8     }
9 }
```

```
// output
Number: 0
Number: 1
Number: 2
Number: 3
Number: 4
```

In this example, the `while` loop continues indefinitely (`while (true)`) until the `break` statement is encountered when `num` becomes 5. The `break` statement terminates the loop and the program execution continues after the loop.

With continue statement

```
int num = 0;

while (num < 5) {
    num++;
    if (num == 3) {
        continue; // Skips the current iteration when num is 3
    }
    System.out.println("Number: " + num);
}
```

```
// output
Number: 1
Number: 2
Number: 4
Number: 5
```

In this example, the `while` loop iterates until `num` becomes 5. However, when `num` is 3, the `continue` statement is encountered, which skips the remaining code in the current iteration and proceeds to the next iteration. This means that the line `System.out.println("Number: " + num);` is not executed when `num` is 3, resulting in "Number: 3" being skipped in the output.

The `while` loop is useful when you want to **repeat a block of code based on a condition**. It is suitable when the number of iterations is not known in advance or when you need to **continuously monitor a condition** to determine if the loop should continue or terminate.

Remember to ensure that the condition within the `while` loop **eventually evaluates to false**, or you include a mechanism (such as a `break` statement) to exit the loop when necessary. Additionally, be cautious with infinite loops (`while (true)`) and **ensure there is a condition or a mechanism to break out of the loop**.

Do-While Loop

The `do-while` loop is a control flow statement in Java that is similar to the `while` loop. However, the `do-while` loop guarantees that the block of code is executed at least once before checking the loop condition.

Another difference between *do-while loop* and *while loop* is that *do-while* **evaluates its expression at the bottom** instead of the top.

Simple Do-while Loop

```java
int count = 0;

do {
    System.out.println("Count: " + count);
    count++;
} while (count < 5);
```

In this example, the `do-while` loop starts with the `do` keyword followed by a code block that contains the desired operations. The loop block is executed first, and then the loop condition `count < 5` is checked. If the condition is true, the loop iterates again. If the condition is false, the loop terminates.

```
// output
Count: 0
Count: 1
Count: 2
Count: 3
Count: 4
```

With User Input

One advantage of the `do-while` loop is that it is **suitable when you want to ensure that a block of code is executed at least once, regardless of the initial condition**. It is commonly used **when you need to validate user input or perform some operation before checking the loop condition**.

Here's an example that showcases the `do-while` loop for validating user input:

```java
import java.util.Scanner;

int number;
Scanner scanner = new Scanner(System.in);

do {
    System.out.print("Enter a positive number: ");
    number = scanner.nextInt();
} while (number <= 0);

System.out.println("You entered a positive number: " + number);
```

In this example, the **user is prompted to enter a positive number**. If the entered number is less than or equal to 0, the loop continues, and the user is prompted again. **This process repeats until a positive number is entered**. Once a positive number is provided, the loop terminates, and the program continues with the remaining code.

With break statement

```java
int count = 0;

do {
    if (count == 3) {
        break; // Terminates the loop when count is 3
    }
    System.out.println("Count: " + count);
    count++;
} while (count < 5);
```

In this example, the `do-while` loop executes the block of code inside the loop and increments `count`. However, when `count` becomes 3, the `break` statement is encountered, and the loop terminates, even though the condition `count < 5` is still true.

With continue statement

```java
int count = 0;

do {
    count++;
```

```
    if (count % 2 == 0) {
        continue; // Skips the current iteration when count is even
    }
    System.out.println("Count: " + count);
} while (count < 5);
```

In this example, the `do-while` loop continues until `count` reaches 5. However, when `count` is even (divisible by 2), the `continue` statement is encountered, and the remaining code in the current iteration is skipped. The loop proceeds to the next iteration without printing anything.

The `break` and `continue` statements can be used in both `while` and `do-while` loops, but they may be more commonly used in `while` loops compared to `do-while` loops.

Challenge

```
int[] arr = { 32, 87, 3, 589, 12, 1076, 2000, 8, 622, 127 };
int target = 12;

int i = 0;
boolean foundIt = false;

for (i = 0; i < arr.length; i++) {
    if (arr[i] == target) {
        foundIt = true;
        break;
    }
}

if (foundIt) {
    System.out.println("Found " + target + " at index " + i);
} else {
    System.out.println(target + " not in the array");
}
// what is the print out?
```