Part 2

More loops

Learning Objectives

- You're familiar with the condition of the `while` loop condition.
- You know how to use the `for` loop.
- You recognize situations where a `while` loop should be used and those where a `for` loop is more appropriate.

Motivation and study strategy questionnaire

Here, you will answer a questionnaire about motivation and study strategies. The questionnaire is known as "MSLQ" and was created by Paul Pintrich and his colleaques. It contains 81 statements about motivation and study strategies. The results will be used for course development and education research. Open the questionnaire by clicking this link.

Quiz:

Motivation and study strategies questionnaire

Points: 2/2

Above, there is a link to a questionnaire. Once you have completed it, check the box called 'I have answered the questionnaire'. Completing the questionnaire is worth two course points.

- [x] I have answered the questionnaire

Answered The number of tries is not limited

The "while-true" loop we've been using is very handy when the program has to repeat a functionality until the user provides certain input.

Next, we'll come to know a few other ways to implement loops.

While Loop with a Condition

So far we have been using a loop with the boolean `true` in its parenthesis, meaning the loop continues forever (or until the loop is ended with the `break` command).

Actually, the parenthesis of a loop can contain a conditional expression, or a condition, just like the parenthesis of an `if` statement. The `true` value can be replaced with an expression, which is evaluated as the program is executed. The expression is defined the same way as the condition of a conditional statement.

The following code prints the numbers 1,2,...,5. When the value of the variable `number` is more than 5, the `while`-condition evaluates to false and the execution of the loop ends for good.

```
int number = 1;

while (number < 6) {
    System.out.println(number);
    number++;
}
```

The code above can be read "As long as the value of the variable number is less than 6, print the value of the variable number and increase the value of the variable number by one".

```
1 public class Example {
2     public static void main(String[] args) {
3         int number = 1;
4
5         while (number < 6) {
```

main:3

Name Value

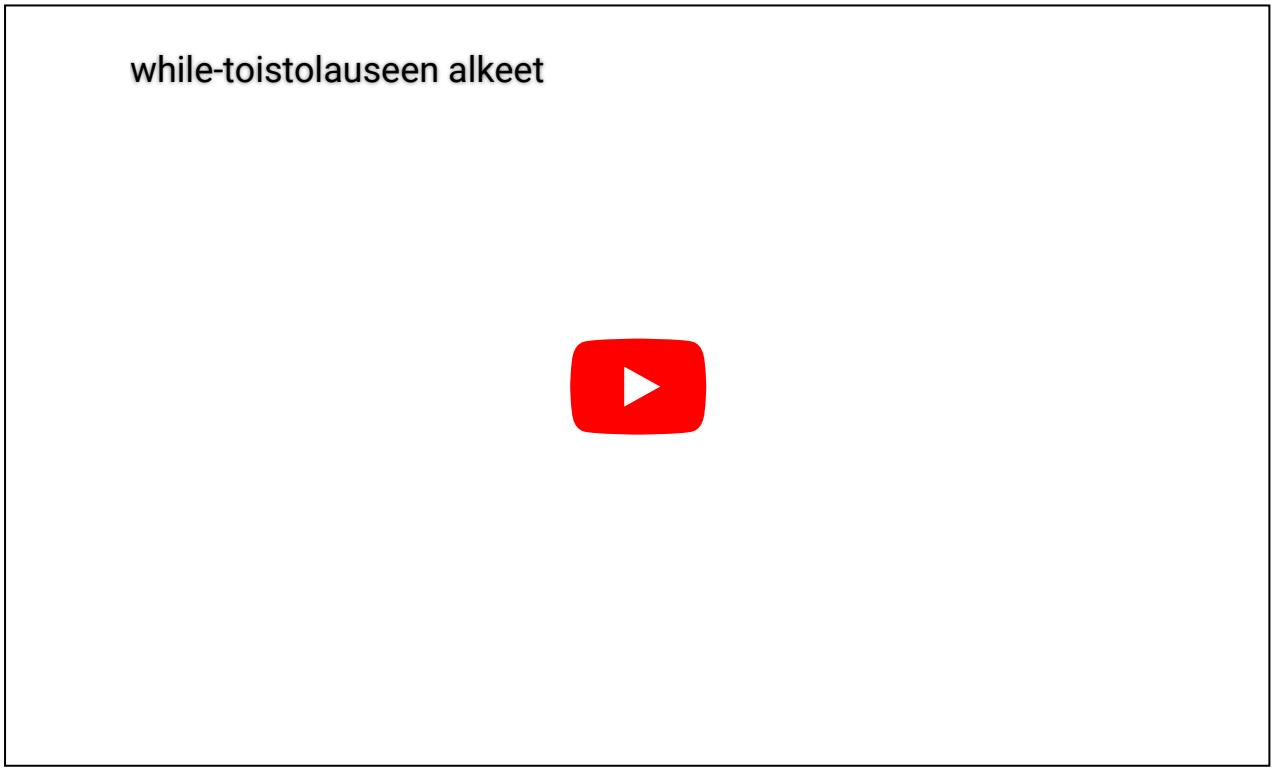
```
            System.out.println(number);
            number++;
        }
    }
}
```

Output

Prev 1 / 24 Next

Above, the value of the variable `number` is increased by one every time the loop body is executed.

Below is a video about using loops.



For Loop

Above, we learned how a `while` loop with a condition can be used to go through numbers in a certain interval.

The structure of this kind of loop is the following.

```
int i = 0;
while (i < 10) {
    System.out.println(i);
    i++;
}
```

The above loop can be split into three parts. First we introduce the variable `i`, used to count the number of times the loop has been executed so far, and set its value to 0: `int i = 0;`. This is followed by the definition of the loop — the loop's condition is `i < 10` so the loop is executed as long as the value of the variable `i` is less than 10. The loop body contains the functionality to be executed `System.out.println(i);`, which is followed by increasing the value of the variable `i++`. The command `i++` is shorthand for `i = i + 1`.

The same can be achieved with a `for` loop like so.

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

A `for` loop contains four parts: (1) introducing the variable for counting the number of executions; (2) the condition of the loop; (3) increasing (or decreasing or changing) the value of the counter variable; and (4) the functionality to be executed.

```
for (*introducing a variable*; *condition*; *increasing the counter*) {
    // Functionality to be executed
}
```

Loop execution is shown below step by step.

```
1 public class Example {
```

main:3

Name	Value

```
    public static void main(String[] args) {
        for (int i = 0; i < 5; i++) {
            System.out.println(i);
        }
    }
}
```

Output

The example above prints the numbers from zero to four. The interval can also be defined using variables — the example below uses variables `start` and `end` to define the interval of numbers the loop goes through.

```
int start = 3;
int end = 7;
for (int i = start; i < end; i++) {
    System.out.println(i);
}
```

```
public class Example {
    public static void main(String[] args) {
        int start = 3;
        int end = 7;
        for (int i = start; i < end; i++) {
            System.out.println(i);
        }
    }
}
```

main:3

Name Value

Output

We will continue practicing loops in the following exercises. You can use either a `while` loop with a condition, or a `for` loop.

Programming exercise:
Counting

Points
1/1

Write a program that reads an integer from the user. Next, the program prints numbers from 0 to the number given by the user. You can assume that the user always gives a positive number. Below are some examples of the expected functionality.

Sample output

```
4
0
1
2
3
4
```

Sample output

```
1
0
1
```

Exercise submission instructions

How to see the solution

Programming exercise:

Counting to hundred

Points 1/1

Write a program, which reads an integer from the user. Then the program prints numbers from that number to 100. You can assume that the user always gives a number less than 100. Below are some examples of the expected functionality.

Sample output

```
99
99
100
```

Sample output

```
-4
-4
-3
-2
-1
0
1
2
... (many numbers in between) ...
98
99
100
```

Exercise submission instructions

How to see the solution

Exercises with multiple parts

Note, that from now on exercises can have multiple parts. All of the parts are counted as separate exercises, so for example the following exercise counts as two separate exercises. Exercises with multiple parts can also typically be submitted even if all parts are not ready — points for the completed parts are added to your points count. Submitting a partial solution does not prevent you from submitting the full solution later on.

Programming exercise:

From where to where? (2 parts)

Points 2/2

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

This exercise is the first two-part exercise. When you complete both parts, you will get two exercise points. You can also submit the exercise after completing only the first part.

Part 1: Where to

Write a program which prints the integers from 1 to a number given by the user.

Sample output

```
Where to? 3
1
2
3
```

Sample output

```
Where to? 5
1
2
3
4
5
```

hint the number read from the user is now the upper limit of the condition. Remember that in Java `a <= b` means *a is smaller or equal to b*.

Part 2: Where from

Ask the user for the starting point as well.

Sample output

```
Where to? 8
Where from? 5
5
6
7
8
```

If the upper limit is smaller than the starting point, nothing is printed:

Sample output

```
Where to? 12
Where from? 16
```

NB remember that the lower and upper limits can be negative!

Exercise submission instructions

How to see the solution

On Stopping a Loop Execution

A loop does not stop executing immediately when its condition evaluates to true. A loop's condition is evaluated at the start of a loop, meaning when (1) the loop starts for the first time or (2) the execution of a previous iteration of the loop body has just finished.

Let's look at the following loop.

```
int number = 1;

while (number != 2) {
    System.out.println(number);
    number = 2;
    System.out.println(number);
    number = 1;
}
```

It prints the following:

Sample output

```
1
2
1
2
1
2
...
```

Even though `number` equals 2 at one point, the loop runs forever.

The condition of a loop is evaluated when the execution of a loop starts and when the execution of the loop body has reached the closing curly bracket. If the condition evaluates to `true`, execution continues from the top of the loop body. If the condition evaluates to `false`, execution continues from the first statement following the loop.

This also applies to `for` loops. In the example below, it would be incorrect to assume that the loop execution ends when `i` equals 100. However, it doesn't.

```
for (int i = 0; i != 100; i++) {
    System.out.println(i);
    i = 100;
    System.out.println(i);
    i = 0;
}
```

The loop above never stops executing.

Repeating Functionality

One common subproblem type is to "do something a certain amount of times". What's common to all these programs is repetition. Some functionality is done repeatedly, and a counter variable is used to keep track of the repetitions.

The following program calculates the product 4*3 somewhat clumsily, i.e., as the sum 3 + 3 + 3 + 3:

```
int result = 0;

int i = 0;
while (true) {
    result += 3; // shorthand for result = result + 3
    i++;  // shorthand for i = i + 1

    if (i == 4) {
        break;
    }
}

System.out.println(result);
```

The same functionality can be achieved with the following code.

```
int result = 0;

int i = 0;
while (i < 4) {
    result += 3; // shorthand for result = result + 3
    i++;  // shorthand for i = i + 1

}

System.out.println(result);
```

Or by using a for loop as seen in the following.

```
int result = 0;

for (int i = 0; i < 4; i++) {
    result += 3;
}

System.out.println(result);
```

The program execution using a while loop is visualized below.

```
1  public class Example {
2      public static void main(String[] args) {
3          int tulos = 0;
4
5          int i = 0;
6          while (i < 4) {
7              tulos += 3;
8              i++;
9          }
10
11         System.out.println(tulos);
12     }
13 }
```

main:3

Name	Value

Output

Simulating program execution

As the number of variables increases, understanding a program becomes harder. Simulating program execution can help in understanding it.

You can simulate program execution by drawing a table containing a column for each variable and condition of a program, and a separate space for program output. You then go through the source code line by line, and write down the changes to the state of the program (the values of each variable or condition), and the program output.

The values of variables `result` and `i` from the previous example have been written out onto the table below at each point the condition `i < 4` is evaluated.

result	i	i < 4
0	0	true
3	1	true
6	2	true
9	3	true
12	4	false

Programming exercise:

Sum of a sequence

Points
1/1

Implement a program, which calculates the sum 1+2+3+...+n where n is given as user input.

Sample output:

Sample output

```
Last number? 3
The sum is 6
```

The previous example calculated 1 + 2 + 3 = 6

Sample output

```
Last number? 7
The sum is 28
```

And this one calculated 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28

Exercise submission instructions

How to see the solution

Programming exercise:

Sum of a sequence - the sequel

Points
1/1

Implement a program which calculates the sum of a closed interval, and prints it. Expect the user to write the smaller number first and then the larger number.

You can base your solution to this exercise to the solution of last exercise — add the functionality for the user to enter the starting point as well.

Sample output:

Sample output

```
First number? 3
Last number? 5
The sum is 12
```

The above example internally calculated 3 + 4 + 5 = 12

Sample output

```
First number? 2
Last number? 8
The sum is: 35
```

And now the internal calculation was 2 + 3 + 4 + 5 + 6 + 7 + 8 = 35

Exercise submission instructions

How to see the solution

Programming exercise:

Factorial

Points

1/1

Implement a program which calculates the factorial of a number given by the user.

Factorial of n, denoted `n!`, is calculated as 1 * 2 * 3 * ... * n. For example, the factorial of 4 is 24 or 4! = 1 * 2 * 3 * 4 = 24. Additionally, it has been specified that the factorial of 0 is 1, so 0! = 1.

Sample output:

Sample output

```
Give a number: 3
Factorial: 6
```

The internal calculation here was 1 * 2 * 3 = 6

Sample output

```
Give a number: 10
Factorial: 3628800
```

And now the internal calculation was 1 * 2 * 3 * ... * 8 * 9 * 10 = 3628800

Additional info: Factorials are used especially in probability calculus when examining different possible orders of a set. For example, a group of five people can form 5! different lines, and a pack of 52 cards can be in 52! different orders. Factorial can also be used to calculate combinations; For example it is possible to deal 52! / (5! * (52-5)!) different hands from a 52 card pack, and you can form 40! / (7! * (40 - 7)!) different 7 number lottery lines from 40 numbers.

Exercise submission instructions

How to see the solution

On the Structure of Programs Using Loops

In the previous examples, we have concentrated on cases where the loop is executed a predetermined number of times. The number of repetitions can be based on user input — in these cases, the for loop is quite handy.

In programs where the loop body has to be executed until the user gives certain input, the for loop is not too great. In these cases, the while-true loop we practiced earlier works well.

Let's take a look at a somewhat more complex program that reads integers from the user. The program handles negative numbers as invalid, and zero stops the loop. When the user enters zero, the program prints the sum of valid numbers, the number of valid numbers and the number of invalid numbers.

A possible solution is detailed below. However, the style of the example is not ideal.

```
Scanner reader = new Scanner(System.in);

System.out.print("Write numbers, negative numbers are invalid: ");
int sum = 0;
int validNumbers = 0;
int invalidNumbers = 0;

while (true) {
    int input = Integer.valueOf(reader.nextLine());

    if (input == 0) {
        System.out.println("Sum of valid numbers: " + sum);
        System.out.println("Valid numbers: " + validNumbers);
        System.out.println("Invalid numbers: " + invalidNumbers);
        break;
    }

    if (input < 0) {
        invalidNumbers++;
        continue;
    }

    sum += input;
    validNumbers++;
}
```

In the code above, the computation executed after the loop has ended has been implemented inside of the loop. This approach is not recommended as it can easily lead to very complex program structure. If something else — for example, reading more input — is to be done when the loop ends, it could also easily end up being placed inside of the loop. As more and more functionality is needed, the program becomes increasingly harder to read.

Let's stick to the following loop structure:

```
Scanner reader = new Scanner(System.in);

// Create variables needed for the loop

while (true) {
    // read input

    // end the loop -- break

    // check for invalid input -- continue

    // handle valid input
}

// functionality to execute after the loop ends
```

In other words, the program structure is cleaner if the things to be done after the loop ends are placed outside of it.

```
Scanner reader = new Scanner(System.in);

System.out.print("Write numbers, negative numbers are invalid: ");
int sum = 0;
int validNumbers = 0;
int invalidNumbers = 0;

while (true) {
    int input = Integer.valueOf(reader.nextLine());

    if (input == 0) {
        break;
    }

    if (input < 0) {
        invalidNumbers++;
        continue;
    }

    sum += input;
    validNumbers++;
}

System.out.println("Sum of valid numbers: " + sum);
System.out.println("Valid numbers: " + validNumbers);
System.out.println("Invalid numbers: " + invalidNumbers);
```

Programming exercise:

Repeating, breaking and remembering (5 parts)

Points
5/5

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

Next, we'll implement a program one piece at a time. This is always strongly recommended when coding.

The series of exercises form a larger program whose functionality is implemented in small pieces. If you do not finish the whole series, you can still submit the parts you've completed to be checked. This can be done by clicking the "submit" button (the arrow pointing up) to the right of the "test" button. Although the submission system complains about the tests of unfinished parts, you get points for the parts you have finished.

NB: Remember that each sub-part of the series is equivalent to one individual exercise. As such, the series is equivalent to five individual exercises.

Note: the tests might fail a correct solution. This is a known bug that will be fixed in the future. In the meantime, you can avoid the error by printing "Give numbers:" without **any** spaces after ':'

Part 1: Reading

Implement a program that asks the user for numbers (the program first prints "Write numbers: ") until the user gives the number -1. When the user writes -1, the program prints "Thx! Bye!" and ends.

Sample output

```
Give numbers:
5
2
4
```

-1
Thx! Bye!

Part 2: Sum of numbers

Extend the program so that it prints the sum of the numbers (not including the -1) the user has written.

Sample output

Give numbers:
5
2
4
-1
Thx! Bye!
Sum: 11

Part 3: Sum and the number of numbers

Extend the program so that it also prints the number of numbers (not including the -1) the user has written.

Sample output

Give numbers:
5
2
4
-1
Thx! Bye!
Sum: 11
Numbers: 3

Part 4: Average of numbers

Extend the program so that it prints the mean of the numbers (not including the -1) the user has written.

Sample output

```
Give numbers:
5
2
4
-1
Thx! Bye!
Sum: 11
Numbers: 3
Average: 3.666666666666
```

Part 5: Even and odd numbers

Extend the program so that it prints the number of even and odd numbers (excluding the -1).

Sample output

```
Give numbers:
5
2
4
-1
Thx! Bye!
Sum: 11
Numbers: 3
Average: 3.666666666666
Even: 2
Odd: 1
```

Exercise submission instructions

How to see the solution

Implementing a program small part at a time

In the previous exercise, we used a series of exercises to practice implementing a program one piece at a time.

When you are writing a program, whether it's an exercise or a personal project, figure out the types of parts the program needs to function and proceed by implementing them one part at a time. Make sure to test the program right after implementing each part.

Never try solving the whole problem at once, because that makes running and testing the program in the middle of the problem-solving process difficult. Start with something easy that you know you can do. When one part works, you can move on to the next.

Some of the exercises are already split into parts. However, it's often the case in programming that these parts need to be split into even smaller parts. You should almost always run the program after every new line of code. This ensures that the solution is moving in the right direction.

You have reached the end of this section! Continue to the next section:

→ 4. Methods and dividing the program into smaller parts

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

5. End questionnaire

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI