Part 6

Separating the user interface from program logic

Learning Objectives

- Understand creating programs so that the user interface and the application logic are separated
- Can create a textual user interface, which takes program specific application logic and a Scanner object as parameters

Let's examine the process of implementing a program and separating different areas of responsibility from each other. The program asks the user to write words until they write the same word twice.

Sample output

```
Write a word: carrot
Write a word: turnip
Write a word: potato
Write a word: celery
Write a word: potato
You wrote the same word twice!
```

Let's build this program piece by piece. One of the challenges is that it is difficult to decide how to approach the problem, or how to split the problem into smaller subproblems, and from which subproblem to start. There is no one clear answer — sometimes it is good to start from the problem domain and its concepts and their connections, sometimes it better to start from the user interface.

We could start implementing the user interface by creating a class UserInterface. The user interface uses a Scanner object for reading user input. This object is given to the user interface.

```
public class UserInterface {
    private Scanner scanner;

    public UserInterface(Scanner scanner) {
        this.scanner = scanner;
    }

    public void start() {
        // do something
    }
}
```

Creating and starting up a user interface can be done as follows.

```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    UserInterface userInterface = new UserInterface(scanner);
    userInterface.start();
}
```

Looping and quitting

This program has (at least) two "sub-problems". The first problem is continuously reading words from the user until a certain condition is reached. We can outline this as follows.

```
public class UserInterface {
    private Scanner scanner;

    public UserInterface(Scanner scanner) {
        this.scanner = scanner;
    }

    public void start() {

        while (true) {
            System.out.print("Enter a word: ");
            String word = scanner.nextLine();
```

```
            if (*stop condition*) {
                break;
            }

        }

        System.out.println("You gave the same word twice!");
    }
}
```

The program continues to ask for words until the user enters a word that has already been entered before. Let us modify the program so that it checks whether the word has been entered or not. We don't know how to implement this functionality yet, so let us first build an outline for it.

```
public class UserInterface {
    private Scanner scanner;

    public UserInterface(Scanner scanner) {
        this.scanner = scanner;
    }

    public void start() {

        while (true) {
            System.out.print("Enter a word: ");
            String word = scanner.nextLine();

            if (alreadyEntered(word)) {
                break;
            }

        }

        System.out.println("You gave the same word twice!");
    }

    public boolean alreadyEntered(String word) {
        // do something here

        return false;
    }
}
```

It's a good idea to test the program continuously, so let's make a test version of the method:

```
public boolean alreadyEntered(String word) {
    if (word.equals("end")) {
        return true;
    }

    return false;
}
```

Now the loop continues until the input equals the word "end":

Sample output

```
Enter a word: carrot
Enter a word: celery
Enter a word: turnip
Enter a word: end
You gave the same word twice!
```

The program doesn't completely work yet, but the first sub-problem - quitting the loop when a certain condition has been reached - has now been implemented.

Storing relevant information

Another sub-problem is remembering the words that have already been entered. A list is a good structure for this purpose.

```
public class UserInterface {
    private Scanner scanner;
    private ArrayList<String> words;

    public UserInterface(Scanner scanner) {
        this.scanner = scanner;
        this.words = new ArrayList<String>();
    }

    //...
}
```

When a new word is entered, it has to be added to the list of words that have been entered before. This is done by adding a line that updates our list to the while-loop:

```
while (true) {
    System.out.print("Enter a word: ");
    String word = scanner.nextLine();

    if (alreadyEntered(word)) {
        break;
    }

    // adding the word to the list of previous words
    this.words.add(word);
}
```

The whole user interface looks as follows.

```
public class UserInterface {
    private Scanner scanner;
    private ArrayList<String> words;

    public UserInterface(Scanner scanner) {
        this.scanner = scanner;
        this.words = new ArrayList<String>();
    }

    public void start() {

        while (true) {
            System.out.print("Enter a word: ");
            String word = scanner.nextLine();

            if (alreadyEntered(word)) {
                break;
            }

            // adding the word to the list of previous words
            this.words.add(word);

        }

        System.out.println("You gave the same word twice!");
    }

    public boolean alreadyEntered(String word) {
```

```
        if (word.equals("end")) {
            return true;
        }

        return false;
    }
}
```

Again, it is a good idea to test that the program still works. For example, it might be useful to add a test print to the end of the start-method to make sure that the entered words have really been added to the list.

```
// test print to check that everything still works
for (String word: this.words) {
    System.out.println(word);
}
```

Combining the solutions to sub-problems

Let's change the method 'alreadyEntered' so that it checks whether the entered word is contained in our list of words that have been already entered.

```
public boolean alreadyEntered(String word) {
    return this.words.contains(word);
}
```

Now the application works as intended.

Objects as a natural part of problem solving

We just built a solution to a problem where the program reads words from a user until the user enters a word that has already been entered before. Our example input was as follows:

Sample output

```
Enter a word: carrot
Enter a word: celery
Enter a word: turnip
```

```
Enter a word: potato
Enter a word: celery
You gave the same word twice!
```

We came up with the following solution:

```
public class UserInterface {
    private Scanner scanner;
    private ArrayList<String> words;

    public UserInterface(Scanner scanner) {
        this.scanner = scanner;
        this.words = new ArrayList<String>();
    }

    public void start() {

        while (true) {
            System.out.print("Enter a word: ");
            String word = scanner.nextLine();

            if (alreadyEntered(word)) {
                break;
            }

            // adding the word to the list of previous words
            this.words.add(word);

        }

        System.out.println("You gave the same word twice!");
    }

    public boolean alreadyEntered(String word) {
       if (word.equals("end")) {
            return true;
        }

        return false;
    }
}
```

From the point of view of the user interface, the support variable 'words' is just a detail. The main thing is that the user interface remembers the *set*

of words that have been entered before. The set is a clear distinct "concept" or an abstraction. Distinct concepts like this are all potential objects: when we notice that we have an abstraction like this in our code, we can think about separating the concept into a class of its own.

Word set

Let's make a class called 'WordSet'. After implementing the class, the user interface's start method looks like this:

```
while (true) {
    String word = scanner.nextLine();

    if (words.contains(word)) {
        break;
    }

    wordSet.add(word);
}

System.out.println("You gave the same word twice!");
```

From the point of view of the user interface, the class WordSet should contain the method 'boolean contains(String word)', that checks whether the given word is contained in our set of words, and the method 'void add(String word)', that adds the given word into the set.

We notice that the readability of the user interface is greatly improved when it's written like this.

The outline for the class 'WordSet' looks like this:

```
public class WordSet {
    // object variable(s)

    public WordSet() {
        // constructor
    }

    public boolean contains(String word) {
        // implementation of the contains method
        return false;
    }
```

```
    public void add(String word) {
        // implementation of the add method
    }
}
```

Earlier solution as part of implementation

We can implement the set of words by making our earlier solution, the list, into an object variable:

```
import java.util.ArrayList;

public class WordSet {
    private ArrayList<String> words

    public WordSet() {
        this.words = new ArrayList<>();
    }

    public void add(String word) {
        this.words.add(word);
    }

    public boolean contains(String word) {
        return this.words.contains(word);
    }
}
```

Now our solution is quite elegant. We have separated a distinct concept into a class of its own, and our user interface looks clean. All the "dirty details" have been encapsulated neatly inside an object.

Let's now edit the user interface so that it uses the class WordSet. The class is given to the user interface as a parameter, just like Scanner.

```
public class UserInterface {
    private WordSet wordSet;
    private Scanner scanner;

    public userInterface(WordSet wordSet, Scanner scanner) {
        this.wordSet = wordSet;
        this.scanner = scanner;
    }
```

```
    public void start() {

        while (true) {
            System.out.print("Enter a word: ");
            String word = scanner.nextLine();

            if (this.wordSet.contains(word)) {
                break;
            }

            this.wordSet.add(word);
        }

        System.out.println("You gave the same word twice!");
    }
}
```

Starting the program is now done as follows:

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

    UserInterface userInterface = new UserInterface(set, scanner);
    userInterface.start();
}
```

Changing the implementation of a class

We have arrived at a situation where the class 'WordSet' "encapsulates" an ArrayList. Is this reasonable? Perhaps. This is because we can make other changes to the class if we so desire, and before long we might arrive at a situation where the word set has to be, for example, saved into a file. If we make all these changes inside the class WordSet without changing the names of the methods that the user interface uses, we don't have to modify the actual user interface at all.

The main point here is that changes made inside the class WordSet don't affect the class UserInterface. This is because the user interface uses WordSet through the methods that it provides — these are called its public interfaces.

Implementing new functionality: palindromes

In the future, we might want to augment the program so that the class 'WordSet' offers some new functionalities. If, for example, we wanted to know how many of the entered words were palindromes, we could add a method called 'palindromes' into the program.

```
public void start() {

    while (true) {
        System.out.print("Enter a word: ");
        String word = scanner.nextLine();

        if (this.wordSet.contains(word)) {
            break;
        }

        this.wordSet.add(word);
    }

    System.out.println("You gave the same word twice!");
    System.out.println(this.wordSet.palindromes() + " of the words were palindr
}
```

The user interface remains clean, because counting the palindromes is done inside the 'WordSet' object. The following is an example implementation of the method.

```
import java.util.ArrayList;

public class WordSet {
    private ArrayList<String> words;

    public WordSet() {
        this.words = new ArrayList<>();
    }

    public boolean contains(String word) {
        return this.words.contains(word);
    }
```

```
    public void add(String word) {
        this.words.add(word);
    }

    public int palindromes() {
        int count = 0;

        for (String word: this.words) {
            if (isPalindrome(word)) {
                count++;
            }
        }

        return count;
    }

    public boolean isPalindrome(String word) {
        int end = word.length() - 1;

        int i = 0;
        while (i < word.length() / 2) {
            // method charAt returns the character at given index
            // as a simple variable
            if(word.charAt(i) != word.charAt(end - i)) {
                return false;
            }

            i++;
        }

        return true;
    }
}
```

The method 'palindromes' uses the helper method 'isPalindrome' to check whether the word that's given to it as a parameter is, in fact, a palindrome.

Recycling

When concepts have been separated into different classes in the code, recycling them and reusing them in other projects becomes easy. For example, the class 'WordSet' could be used in a graphical user interface, and it could also be part of a mobile phone application. In addition, testing the program is much easier when it

has been divided into several concepts, each of which has its own separate logic and can function alone as a unit.

Programming tips

In the larger example above, we were following the advice given here.

- Proceed with small steps
    - Try to separate the program into several sub-problems and **work on only one sub-problem at a time**
    - Always test that the program code is advancing in the right direction, in other words: test that the solution to the sub-problem is correct
    - Recognize the conditions that require the program to work differently. In the example above, we needed a different functionality to test whether a word had been already entered before.
- Write as "clean" code as possible
    - Indent your code
    - Use descriptive method and variable names
    - Don't make your methods too long, not even the main method
    - Do only one thing inside one method
    - **Remove all copy-paste code**
    - Replace the "bad" and unclean parts of your code with clean code
- If needed, take a step back and assess the program as a whole. If it doesn't work, it might be a good idea to return into a previous state where the code still worked. As a corollary, we might say that a program that's broken is rarely fixed by adding more code to it.

Programmers follow these conventions so that programming can be made easier. Following them also makes it easier to read programs, to keep them up, and to edit them in teams.

Programming exercise:
Simple Dictionary (4 parts)

Points
4/4

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.

The exercise base contains a class `SimpleDictionary` that allows for storing words and their translations. The internal implementation of the class contains some techniques not (yet) covered on the course. Nevertheless, it's fairly simple to use it:

```
SimpleDictionary book = new SimpleDictionary();
book.add("one", "yksi");
book.add("two", "kaksi");

System.out.println(book.translate("one"));
System.out.println(book.translate("two"));
System.out.println(book.translate("three"));
```

Sample output

```
yksi
kaksi
null
```

In this exercise you will implement a text user interface that takes use of the `SimpleDictionary` class. And maybe pick up a few Finnish words while doing it!

Part 1: Starting and stopping the UI

Implement the class `TextUI` that receives as constructor parameters a `Scanner` and `SimpleDictionary` objects. Then give the class a method called `public void start()`. The method should work as follows:

1. The method asks the user for a command
2. If the command is `end`, the UI prints the string "Bye bye!" and the execution of the `start` method ends.
3. Otherwise the text UI prints the message `Unknown command` and asks for a new command, so it loops back to step 1.

```
Scanner scanner = new Scanner(System.in);
SimpleDictionary dictionary = new SimpleDictionary();
```

```
TextUI ui = new TextUI(scanner, dictionary);
ui.start();
```

Sample output

```
Command: something
Unknown command
Command: add
Unknown command
Command: end
Bye bye!
```

Part 2: Adding a translation

Modify the method `public void start()` so that it works in the following way:

1. The method asks the user for a command.
2. If the command is `end`, the UI prints the string "Bye bye!" and the execution of the `start` method ends.
3. If the command is `add`, the text UI asks the user for a word and a translation, each on its own line. After this the words are stored in the dictionary, and the method continues by asking for a new command (loops back to stage 1).
4. Otherwise the text UI prints the message `Unknown command` and asks for a new command, so it loops back to step 1.

Sample output

```
Command: something
Unknown command
Command: add
Word: pike
Translation: hauki
Command: change
Unknown command
Command: end
Bye bye!
```

In the example above, we added the word "pike" and its translation "hauki" to the SimpleDictionary object. After exiting the text user interface the dictionary could be used in the following manner:

```
Scanner scanner = new Scanner(System.in);
SimpleDictionary dictionary = new SimpleDictionary();

TextUI textUI = new TextUI(scanner, dictionary);
textUI.start();
System.out.println(dictionary.translate("pike")); // prints the string "hau
```

Part 3: Translating a word

Modify the method `public void start()` so that it works in the following:

1. The method asks the user for a command.
2. If the command is `end`, the UI prints the string "Bye bye!" and the execution of the `start` method ends.
3. If the command is `add`, the text UI asks the user for a word and a translation, each on its own line. After this the words are stored in the dictionary, and the method continues by asking for a new command (loops back to stage 1).
4. If the command is `search`, the text UI asks the user for the word to be translated. After this it prints the translation of the word, and the method continues by asking for a new command (loops back to stage 1).
5. Otherwise the text UI prints the message `Unknown command` and asks for a new command, so it loops back to step 1.

Sample output

```
Command: something
Unknown command
Command: add
Word: pike
Translation: hauki
Command: change
Unknown command
Command: search
To be translated: pike
Translation: hauki
```

Command: search
To be translated: carrot
Translation: null
Command: end
Bye bye!

Part 4: Cleaning up the translation

Modify the searching functionality of the UI so that if the word isn't found (i.e. the dictionary returns `null`), the UI prints the message "Word (searched word) was not found".

Sample output

Command: something
Unknown command
Command: add
Word: pike
Translation: hauki
Command: change
Unknown command
Command: search
To be translated: pike
Translation: hauki
Command: search
To be translated: carrot
Word carrot was not found
Command: end
Bye bye!

Exercise submission instructions

How to see the solution

Programming exercise:

To do list (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.

In this exercise we are going to create a program that can be used to create and modify a to-do list. The final product will work in the following manner.

Sample output

```
Command: add
Task: go to the store
Command: add
Task: vacuum clean
Command: list
1: go to the store
2: vacuum clean
Command: completed
Which task was completed? 2
Task go to the store tehty
Command: list
1: go to the store
Command: add
Task: program
Command: list
1: go to the store
2: program
Command: stop
```

We will build the program in parts.

Part 1: TodoList

Create a class called `TodoList`. It should have a constructor without parameters and the following methods:

- `public void add(String task)` - add the task passed as a parameter to the todo list.
- `public void print()` - prints the exercises. Each task has a number associated with it on the print statement — use the task's index here (+1).
- `public void remove(int number)` - removes the task associated with the given number; the number is the one seen associated with the task in the print.

```
TodoList list = new TodoList();
list.add("read the course material");
list.add("watch the latest fool us");
list.add("take it easy");

list.print();
list.remove(2);

System.out.println();
list.print();
```

Sample output

```
1: read the course material
2: watch the latest fool us
3: take it easy

1: read the course material
2: take it easy
```

NB! You may assume that the `remove` method is given a number that corresponds to a real task. The method only has to correctly work once after each print call.

Another example:

```
TodoList list = new TodoList();
list.add("read the course material");
list.add("watch the latest fool us");
list.add("take it easy");
list.print();
list.remove(2);
list.print();
list.add("buy raisins");
list.print();
```

```
list.remove(1);
list.remove(1);
list.print();
```

Sample output

```
1: read the course material
2: watch the latest fool us
3: take it easy
1: read the course material
2: take it easy
1: read the course material
2: take it easy
3: buy raisins
1: buy raisins
```

Part 2: User interface

Next, implement a class called `UserInterface`. It should have a constructor with two parameters. The first parameter is an instance of the class `TodoList`, and the second is an instance of the class `Scanner`. In addition to the constructor, the class should have the method `public void start()` that is used to start the text user interface. The text UI works with an eternal looping statement (`while-true`), and it must offer the following commands to the user:

- The command `stop` stops the execution of the loop, after which the execution of the program advances out of the `start` method.
- The command `add` asks the user for the next task to be added. Once the user enters this task, it should be added to the to-do list.
- The commmand `list` prints all the tasks on the to-do list.
- The command `remove` asks the user to enter the id of the task to be removed. When this has been entered, the specified task should be removed from the list of tasks.

Below is an example of how the program should work.

Sample output

```
Command: add
To add: write an essay
Command: add
To add: read a book
Command: list
1: write an essay
2: read a book
Command: remove
Which one is removed? 1
Command: list
1: read a book
Command: remove
Which one is removed? 1
Command: list
Command: add
To add: stop
Command: list
1: stop
Command: stop
```

NB! The user interface is to use the TodoList and Scanner that are passed as parameters to the constructor.

Exercise submission instructions

How to see the solution

From one entity to many parts

Let's examine a program that asks the user to enter exam points and turns them into grades. Finally, the program prints the distribution of the grades as stars. The program stops reading inputs when the user inputs an empty string. An example program looks as follows:

Sample output

```
Points: 91
Points: 98
Points: 103
Impossible number.
Points: 90
Points: 89
Points: 89
Points: 88
Points: 72
Points: 54
Points: 55
Points: 51
Points: 49
Points: 48
Points:

5: ***
4: ***
3: *
2:
1: ***
0: **
```

As almost all programs, this program can be written into main as one entity. Here is one possibility.

```
import java.util.ArrayList;
import java.util.Scanner;

public class Program {

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

        ArrayList<Integer> grades = new ArrayList<>();

        while (true) {
            System.out.print("Points: ");
            String input = scanner.nextLine();
            if (input.equals("")) {
```

```
                break;
            }

            int score = Integer.valueOf(input);

            if (score < 0 || score > 100) {
                System.out.println("Impossible number.");
                continue;
            }

            int grade = 0;
            if (score < 50) {
                grade = 0;
            } else if (score < 60) {
                grade = 1;
            } else if (score < 70) {
                grade = 2;
            } else if (score < 80) {
                grade = 3;
            } else if (score < 90) {
                grade = 4;
            } else {
                grade = 5;
            }

            grades.add(grade);
        }

        System.out.println("");
        int grade = 5;
        while (grade >= 0) {
            int stars = 0;
            for (int received: grades) {
                if (received == grade) {
                    stars++;
                }
            }

            System.out.print(grade + ": ");
            while (stars > 0) {
                System.out.print("*");
                stars--;
            }
            System.out.println("");

            grade = grade - 1;
        }
    }
}
```

Let's separate the program into smaller chunks. This can be done by identifying several discrete areas of responsibility within the program. Keeping track of grades, including converting scores into grades, could be done inside a different class. In addition, we could create a new class for the user interface.

Program logic

Program logic includes parts that are crucial for the execution of the program, like functionalities that store information. From the previous example, we can separate the parts that store grade information. From these we can make a class called 'GradeRegister', which is responsible for keeping track of the numbers of different grades students have received. In the register, we can add grades according to scores. In addition, we can use the register to ask how many people have received a certain grade.

An example class follows.

```
import java.util.ArrayList;

public class GradeRegister {

    private ArrayList<Integer> grades;

    public GradeRegister() {
        this.grades = new ArrayList<>();
    }

    public void addGradeBasedOnPoints(int points) {
        this.grades.add(pointsToGrades(points));
    }

    public int numberOfGrades(int grade) {
        int count = 0;
        for (int received: this.grades) {
            if (received == grade) {
                count++;
            }
        }

        return count;
    }

    public static int pointsToGrades(int points) {
```

```
        int grade = 0;
        if (points < 50) {
            grade = 0;
        } else if (points < 60) {
            grade = 1;
        } else if (points < 70) {
            grade = 2;
        } else if (points < 80) {
            grade = 3;
        } else if (points < 90) {
            grade = 4;
        } else {
            grade = 5;
        }

        return grade;
    }
}
```

When the grade register has been separated into a class, we can remove the functionality associated with it from our main program. The main program now looks like this.

```
import java.util.Scanner;

public class Program {

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

        GradeRegister register = new GradeRegister();

        while (true) {
            System.out.print("Points: ");
            String input = scanner.nextLine();
            if (input.equals("")) {
                break;
            }

            int score = Integer.valueOf(input);

            if (score < 0 || score > 100) {
                System.out.println("Impossible number.");
                continue;
            }

            register.addGradeBasedOnPoints(score);
```

```
        }

        System.out.println("");
        int grade = 5;
        while (grade >= 0) {
            int stars = register.numberOfGrades(grade);
            System.out.print(grade + ": ");
            while (stars > 0) {
                System.out.print("*");
                stars--;
            }
            System.out.println("");

            grade = grade - 1;
        }
    }
}
```

Separating the program logic is a major benefit for the maintenance of the program. Since the program logic — in this case the GradeRegister — is its own class, it can also be tested separately from the other parts of the program. If you wanted to, you could copy the class `GradeRegister` and use it in your other programs. Below is an example of simple manual testing — this experiment only concerns itself with a small part of the register's functionality.

```
GradeRegister register = new GradeRegister();
register.addGradeBasedOnPoints(51);
register.addGradeBasedOnPoints(50);
register.addGradeBasedOnPoints(49);

System.out.println("Number of students with grade 0 (should be 1): " + register
System.out.println("Number of students with grade 0 (should be 2): " + register
```

User interface

Typically each program has its own user interface. We will create the class `UserInterface` and separate it from the main program. The user interface receives two parameters in its constructor: a grade register for storing the grades, and a Scanner object used for reading input.

When we now have a separate user interface at our disposal, the main program that initializes the whole program becomes very clear.

```
import java.util.Scanner;

public class Program {

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

        GradeRegister register = new GradeRegister();

        UserInterface userInterface = new UserInterface(register, scanner);
        userInterface.start();
    }
}
```

Let's have a look at how the user interface is implemented. There are two essential parts to the UI: reading the points, and printing the grade distribution.

```
import java.util.Scanner;

public class UserInterface {

    private GradeRegister register;
    private Scanner scanner;

    public UserInterface(GradeRegister register, Scanner scanner) {
        this.register = register;
        this.scanner = scanner;
    }

    public void start() {
        readPoints();
        System.out.println("");
        printGradeDistribution();
    }

    public void readPoints() {
    }

    public void printGradeDistribution() {
    }
}
```

We can copy the code for reading exam points and printing grade distribution nearly as is from the previous main program. In the program below, parts of the code have indeed been copied from the earlier main program, and new method for printing stars has also been created — this clarifies the method that is used for printing the grade distribution.

```
import java.util.Scanner;

public class UserInterface {

    private GradeRegister register;
    private Scanner scanner;

    public UserInterface(GradeRegister register, Scanner scanner) {
        this.register = register;
        this.scanner = scanner;
    }

    public void start() {
        readPoints();
        System.out.println("");
        printGradeDistribution();
    }

    public void readPoints() {
        while (true) {
            System.out.print("Points: ");
            String input = scanner.nextLine();
            if (input.equals("")) {
                break;
            }

            int points = Integer.valueOf(input);

            if (points < 0 || points > 100) {
                System.out.println("Impossible number.");
                continue;
            }

            this.register.addGradeBasedOnPoints(points);
        }
    }

    public void printGradeDistribution() {
        int grade = 5;
        while (grade >= 0) {
            int stars = register.numberOfGrades(grade);
            System.out.print(grade + ": ");
```

```
            printStars(stars);
            System.out.println("");

            grade = grade - 1;
        }
    }

    public static void printStars(int stars) {
        while (stars > 0) {
            System.out.print("*");
            stars--;
        }
    }
}
```

Programming exercise:

Averages (3 parts)

Points 3/3

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.

The exercise base includes the previously constructed program to store grades. In this exercise you will further develop the class `GradeRegister` so that it can calculate the average of grades and exam results.

Part 1: Average grade

Create the method `public double averageOfGrades()` for the class `GradeRegister`. It should return the average of the grades. If the register contains no grades, the method should return `-1`. Use the `grades` list to calculate the average.

Example:

```
GradeRegister register = new GradeRegister();
register.addGradeBasedOnPoints(93);
register.addGradeBasedOnPoints(91);
```

```
register.addGradeBasedOnPoints(92);
register.addGradeBasedOnPoints(88);

System.out.println(register.averageOfGrades());
```

Sample output

```
4.75
```

Part 2: Average points

Give the class `GradeRegister` a new object variable: a list where you will store the exam points every time that the method `addGradeBasedOnPoints` is called. After this addition, create a method `public double averageOfPoints()` that calculates and returns the average of the exam points. If there are no points added to the register, the method should return the number `-1`.

Example:

```
GradeRegister register = new GradeRegister();
register.addGradeBasedOnPoints(93);
register.addGradeBasedOnPoints(91);
register.addGradeBasedOnPoints(92);

System.out.println(register.averageOfPoints());
```

Sample output

```
92.0
```

Part 3: Prints in the user interface

As a final step, add the methods implemented above as parts of the user interface. When the program prints the grade distribution, it should also print the averages of the points and the grades.

Sample output

```
Points: 82
Points: 83
```

```
Points: 96
Points: 51
Points: 48
Points: 56
Points: 61
Points:

5: *
4: **
3:
2: *
1: **
0: *
The average of points: 68.14285714285714
The average of grades: 2.4285714285714284
```

Exercise submission instructions

How to see the solution

Programming exercise:

Joke Manager (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.

The exercise base contains the following program that has been written "in the main".

```
Scanner scanner = new Scanner(System.in);
ArrayList<String> jokes = new ArrayList<>();
System.out.println("What a joke!");
```

```
while (true) {
    System.out.println("Commands:");
    System.out.println(" 1 - add a joke");
    System.out.println(" 2 - draw a joke");
    System.out.println(" 3 - list jokes");
    System.out.println(" X - stop");

    String command = scanner.nextLine();

    if (command.equals("X")) {
        break;
    }

    if (command.equals("1")) {
        System.out.println("Write the joke to be added:");
        String joke = scanner.nextLine();
        jokes.add(joke);
    } else if (command.equals("2")) {
        System.out.println("Drawing a joke.");

        if (jokes.isEmpty()) {
            System.out.println("Jokes are in short supply.");
        } else {
            Random draw = new Random();
            int index = draw.nextInt(jokes.size());
            System.out.println(jokes.get(index));
        }

    } else if (command.equals("3")) {
        System.out.println("Printing the jokes.");
        for (String joke : jokes) {
            System.out.println(joke);
        }
    }
}
```

The application is in practice a storage for jokes. You can add jokes, get a randomized joke, and the stored jokes can be printed. In this exercise the program is divided into parts in a guided manner.

Part 1: Joke manager

Create a class called `JokeManager` and move the functionality to manage jokes in it. The class must have a parameter-free constructor, and the following methods:

- `public void addJoke(String joke)` - adds a joke to the manager.
- `public String drawJoke()` - chooses one joke at random and returns it. It there are no jokes stored in the joke manager, the method should return the string "Jokes are in short supply.".
- `public void printJokes()` - prints all the jokes stored in the joke manager.

An example of how to use the class:

```
JokeManager manager = new JokeManager();
manager.addJoke("What is red and smells of blue paint? - Red paint.");
manager.addJoke("What is blue and smells of red paint? - Blue paint.");

System.out.println("Drawing jokes:");
for (int i = 0; i < 5; i++) {
    System.out.println(manager.drawJokes());
}

System.out.println("");
System.out.println("Printing jokes:");
manager.printJokes();
```

Below is a possible output of the program. Notice that the jokes will probably not be drawn as in this example.

Sample output

```
Drawing jokes:
What is blue and smells of red paint? - Blue paint.
What is red and smells of blue paint? - Red paint.
What is blue and smells of red paint? - Blue paint.
What is blue and smells of red paint? - Blue paint.
What is blue and smells of red paint? - Blue paint.

Printing jokes:
What is red and smells of blue paint? - Red paint.
What is blue and smells of red paint? - Blue paint.
```

Part 2: User interface

Create a class called `UserInterface` and move the UI functionality of the program there. The class must have a constructor with two parameters. The first parameter is an instance of the JokeManager class, and the second parameter is an instance of the Scanner class. In

addition, the class should have the method `public void start()` that can be used to start the user interface.

The user interface should provide the user with the following commands:

- `X` - ending: exits the method `start`.
- `1` - adding: asks the user for the joke to be added to the joke manager, and then adds it.
- `2` - drawing: chooses a random joke from the joke manager and prints it. If there are no jokes in the manager, thi string "Jokes are in short supply." will be printed.
- `3` - printing: prints all the jokes stored in the joke manager.

An example of how to use the UI:

```
JokeManager manager = new JokeManager();
Scanner scanner = new Scanner(System.in);

UserInterface ui = new UserInterface(manager, scanner);
ui.start();
```

Sample output

```
Commands:
1 - add a joke
2 - draw a joke
3 - list jokes
X - stop
1
Write the joke to be added:
Did you hear about the claustrophobic astronaut? — He just needed a little space.
Commands:
1 - add a joke
2 - draw a joke
3 - list jokes
X - stop
3
Printing the jokes.
Did you hear about the claustrophobic astronaut? — He just needed a little space.
Commands:
1 - add a joke
```

```
2 - draw a joke
3 - list jokes
X - stop
X
```

Exercise submission instructions

How to see the solution

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

→ 3. Introduction to testing

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

In this part:

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