Instruction Manual Programming

Seventh International Edition

Published September, 2021

VU UNIVERSITY AMSTERDAM

Contents

0 Syllabus

Course format

This course features a series of lectures and parallel lab sessions. During the lectures the theory of programming is taught, during the lab sessions programming is practiced by making assignments using Java. Assignments should be prepared in advance, at home. Students will be assigned to groups. Every group will have a teaching assistant, who will help with the assignments and grade the graded assignments.

Course documents and assignments

Book The book that will be used during this course is *Absolute Java*, International Edition, Fourth Edition, by Walter Savitch. Parts of this book will be handled during the lectures. During the lab sessions, you are supposed to take this book and/or your lecture notes. Depending on your study, the book will also be used again during other courses. It is therefore recommended to buy this book. It is available at Storm or the VU bookstore.

Modules The course material is devided in 7 modules. These modules contain additional theory and assignments and will be used as an instruction manual during the lab sessions. Depending on the course you take, a module will either take a week or two weeks. Theory handled during the lectures will be repeated as little as possible in the modules. The modules will however feature notes on programming style. This course also features a bonus assignment. The bonus assignment can be found in module 8. If you pass the bonus assingment, one tenth of your mark will be added to your lab grade.

Module	timetable for course in 1 period	timetable for course in 2 periods
1	week 1	week 1+2
2	week 2	week 3+4
3	week 3	week 5+6
4	week 4	week 7+9
5	week 5	week 10+11
6	week 6	week 12+13
7	week 7	week 14+15

Assignments Every module consists of theory and assignments. Only the last assignment of a module is a graded assignment. This does not mean that the other assignments are less important. All the assignments in a module will train essential skills needed to succesfully complete the graded assignment. *Every* assignment has to be approved by the teaching assistent to pass the course. You should not start on a assignment before completing all previous assignments. This way, you will not make the same mistakes twice.

Application Programming Interface (API) The program you hand in is not allowed to contain anything from the API except certain methods from the following:

- String
- Math
- Scanner
- PrintStream
- System

Deadlines and assessment

Graded assignments have to be submitted to Practool. The teaching assistent will only provide help with an assignment if all previous ungraded assignments have been approved. When the quality of an ungraded assignment is not sufficient, it has to be corrected according to the provided feedback. Graded assignments cannot be resubmitted once a grade has been given. Feedback on these assignments can therefore not be used to improve a program in order to receive a higher grade. Nevertheless, it is advised to process the feedback received on graded assignments as well.

Deadlines The last assignment of each module will be graded. **Deadlines will be posted on Canvas and will always be on friday 23:59.** Late submission will be accepted, but a point is deducted for every day the assignment is late. The grades are weighted in the following way:

Module	Graded Assignment	Weight
1	HelloWorld2	
2	All	slipday
3	Replay1	slipday
4	Statistics	2x
5	Pirate	2x
6	Snake	3x
7	LongestPath	3x
8	Life	see below*

* if you grade for the bonus assignment is a passing grade ($\geq 5,5$), one tenth of that grade will be added to your average lab grade

Slipdays The first three modules do not include graded assignments. There are regular assignments to be made, so there is also a deadline in the first two modules. You can earn a *slipday* if all the assignments have been approved before the deadline (you can earn at most 2 slipdays). The first module, in order to receive the slipday, attendance is mandatory. Slipdays can be used in the following modules. An assignment can be turned in a day late, without point deduction, using a slipday. Two slipdays can be used to submit a single assignment two days late, or two assignments one day late.

Final Grade You have passed the lab when:

1. all assignments have been approved
2. every graded assignment has been graded
3. the average grade of all the graded assignments is sufficient ($\geq 5,5$).

You passed the Introduction to Programming course if both the exam grade `E` and the lab grade `P` are a passing grade (both $\geq 5,5$). The final grade for this course is calculated using the following formula: `F = max(E, (2*E+P)/3)`, meaning that the lab can not influence your final grade negatively.

1 Editing, Compiling and Executing

Abstract

This module will introduce the IDE (Integrated Development Environment) Eclipse and explain how to organize Java files and execute programs.

Goals

- The use of Eclipse to open, edit, save and organize Java-files
- Compile and execute Java files
- Print source code

Introduction to Eclipse

Installing and starting Eclipse

VU Eclipse has already been installed on all the computers that will be used during the lab sessions. Using Windows, it can be started from the Start Menu. Using Linux, it can be started by typing *eclipse* in a terminal window.

At home, using Windows To use Eclipse at home, it is neccessary to install the Java Development Kit (JDK) and Eclipse itself. It is likely that Java has already been installed on your computer, in the form of a Java Runtime Environment (JRE). Whilst Java programs can be executed using this JRE, they cannot be compiled. To compile programs, a JDK is needed. The JDK can be downloaded from Oracles website: `http://www.oracle.com/technetwork/java/javase/downloads/`.

This site provides a number of different versions of the JDK: Java SE (Standard Edition), Java EE (Enterprise Edition), Java ME (Micro Edition) and more. For this course, Java SE will be used. Click "Download JDK" and download the Java SE Development Kit.

To download Eclipse, browse to `http://www.eclipse.org/downloads/` and download "Eclipse IDE for Java Developers".

At home, using Linux or Mac OS X Java is installed in different ways, depending on the distribution. In a Debian-based distribution, Java can be downloaded from the repository. For example, using Ubuntu: open a terminal window and type `sudo apt-get install sun-java6-jdk`. To install Java on a different distribution, visit their support site. The JDK is already installed on Mac OS X.

To download Eclipse, browse to `http://www.eclipse.org/downloads/` and download "Eclipse IDE for Java Developers".

Selecting a workspace The *workspace* is a folder in which all created files are saved. The workspace only has to be defined once. When Eclipse is started for the first time, it will automatically prompt for a location for the workspace. If, for whatever reason, this does not happen, it can be done manually: File → Switch Workspace. The home directory is usually a good place for the workspace `/home/`*vunet-id*`/eclipse/yearone`.

Arranging files

After selecting the workspace, a welcome screen will be shown. Click on Workbench. On the left of the current window, the Package Explorer is shown. This frame will hold all files, sorted on Project. Every Project consists of Source Folders, whilst Source Folders consist of Packages. This may sound complicated on first glance, but the next few steps will explain everything.

1. Create a new **Java Project**. To do this right-click in the Package Explorer and select New → Project. Select **Java Project** and click Next. The name of this project will be **Introduction to Programming**. Click Finish.

2. Create a new **Source Folder**. All *Source Files* used in one assignment will be logically grouped together in a **Source Folder**. In this course, every module will have its own **Source Folder**. Right-click on the project in the Package Explorer and select New → Source Folder. Name it **module1** and click Finish.

3. Every assignment will have its own **Package**. Right-click on the **Source Folder** that was just created and select New → Package. The name of the **Package** will be the same as the name of the assignment.

4. Create a new **Class**. Right-click on the current package and select New → Class. There are loads of options, but for now, only Name is relevant. Select "public static void main" everytime you create a new program. The first class in this course will be **HelloWorld1**.

Compiling and executing programs

The class HelloWorld1 in its current form is called a *skeleton*; an empty program, that does nothing. As a start, copy the following example:

```
package HelloWorld1;

import java.io.PrintStream;

class HelloWorld1 {
    // Name       : ...
    // Assignment : HelloWorld1
    // Date       : ...

    PrintStream out;

    HelloWorld1() {
        out = new PrintStream(System.out);
    }

    void start() {
        out.printf("Hello world!! ");
        out.printf("written by: %s\n", "...");
    }

    public static void main(String[] argv) {
        new HelloWorld1().start();
    }
}
```

Programs are compiled automatically in Eclipse. Make sure that the file is saved (Ctrl+s). Right-click on the file to be executed and select Run As → Java Application. To execute the same program again, use Ctrl+F11. The output of the program will be printed in the Console, at the bottom of the screen. If the program expects input, it can be typed into the Console as well.
Programs can only be executed if they are syntactically correct. If there are any errors, these are underlined in red. Hover the mouse over the underlined words to show an error message. Eclipse can automatically solve problems,

but be aware: Eclipse tends to do things you do not want it to do. *Only* use this feature, if you understand what the error is, and how it can be solved.

Setting the default Locale

A Locale object represents a specific geographical or political location. Some operations are *locale-sensitive*; this means that there behaviour is dependent on the current Locale. An example of such a *locale-sensitive* operation is the Scanner.nextDouble() function. The notation of decimal numbers is dependent on the location. The fractional part of a decimal is either separated from its integral part by a dot or by a comma. In this practical we will use the US decimal notation, i.e. use a dot as the decimal mark. To avoid compatibility problems encountered when a program is executed under different Locales, a default Locale has to be set. This can be done in two ways.

1. Add 'Locale.setDefault(Locale.US)' to the constructor of every program you write. This makes sure that the Locale will be set to US, no matter who executes it.
2. Set the default Locale of Eclipse. This will make sure that all programs executed by this Eclipse installation will be executed using the US Locale.
    - Navigate to Window → Preferences → Java → Installed JREs
    - Select the current JRE and click Edit
    - Add the following line to the Default VM Arguments: *-Duser.language=en -Duser.country=US*

Note the subtle difference between the two solutions. Using the first solution, the program will override the Locale of the environment in which it is executed. Using the second solution, Eclipse will override the Locale of the current environment. This means that only by using the first solution, it is guaranteed that the program will be executed using the US Locale. In this practical, both solutions will suffice, as it is only necessary to set the Locale to be able to read the provided input files. All programs will be evaluated using the US Locale and will thus not depend on the program to set the appropriate Locale.

Trial Submission

Create a new class HelloWorld2 with a new Package HelloWorld2 and copy HelloWorld1 to the Package HelloWorld2 and edit the code in the following way:

Hint: Copying files using Eclipse can be done easily using the refactor functionality provided by Eclipse. For more information about refactoring, visit: http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts%2Fconcept-refactoring.htm

- Add a second import-statement:

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

- Edit the start-method in such way that it will ask for your name:

```
void start() {
    Scanner in = new Scanner(System.in);

    out.printf("Enter your name: ");
    String name = in.nextLine();

    out.printf("Hello world!! ");
    out.printf("written by: %s\n", name);
}
```

Test the program. Does it work as expected? If so, you can hand it in.

Submitting Assignments

A graded assignment needs to be submitted to Practool. How to do this is explained in the following steps:

1. Export all the files of the assignment to a .jar-file. Right-click on the **Package** in the Package Explorer to be exported and select Export. Select Java → JAR file and click Next. Make sure the name of the file starts with the name of assignment and is followed by your VUnet-id separated by a hyphen. For example: `pirate-rhg600.jar`.

   Select **"Export Java source files and resources?"**, to ensure that the source code is included in the jar-file. Unselect **"Export generated class files and resources"**. Click Finish and the .jar-file has been created in the workspace.

2. Did you really select **"Export Java source files and resources?"** and unselect **"Export generated class files and resources"**? If not: redo step 1.

3. Surf to `http://phoenix.labs.vu.nl` . Choose the option 'register' and register. Now log in and choose the lab of your Teaching Assistent to enroll in that particular group. Now select your group, you'll see your current grades and an upload option.

☛ **Warning**
Assignments can only be processed if they are submitted in the format described above. Do not submit files in any other format!

This program is not graded like the other assignments that have to be submitted, however it is possible to earn a slipday if the program is submitted on time. The syllabus provides more details on slipdays. The goal of this program is to make sure that you can print and submit programs. These are essential skills required during the rest of this course.

2 If statements and loops

Abstract

The first few programs in this module will read from *standard input* and write output to *standard output*. These programs will be very simple. The focus in the first part of this module will be on writing programs with a clear layout using well chosen names. The second part of this module will introduce if statements and loops.

☛ **Warning**

This module contains ten assignments of variable size. Make sure to utilize the time given to you during the lab sessions. The lab sessions only provide sufficient time if you write your programs in advance. This way, any problems you encounter whilst writing your programs can be resolved during the lab sessions.

Goals

- The use of clear identifiers.
- Familiarize with if, else and else-if statements and recognize situations in which to apply these.
- Familiarize with for, while and do-while loops and recognize situations in which to apply these.

Instructions

- Read the theory about **Efficient programming** and **Constants**. With this information in mind, make the assignments **VAT**, **Plumber 1**, **Plumber 2** and **Othello 1**.
- Read the theory about **Identifiers** and **If-statements**. With this information in mind, make the assingments **Electronics** and **Othello 2**.

- Study your lecture note on **Loops**. With this information in mind, make the assignments **Manny**, **Alphabet**, **Collatz** and **SecondSmallest**.

Theory

Efficient programming

Once upon a time, running a computer was so expensive that any running time that could be saved was worthwile. Programs had to contain as few lines of code as possible and programs were designed to run fast; clear code was not a priority. Such a programming style is nowadays called machine-friendly. Luckily for us, that is no longer necessary.
Programs that have been written in the past often need altering in one way or another. If a program was written in a machine-friendly, but incomprehensible programming style, it is almost impossible to edit it. After half a year, one easily forgets how the program works. Imagine the problems that could occur, if the programmer that wrote the code no longer works for the company that wants to change it.
The direct result of this programming style is that programs are not changed at all. Everyone has to work with the, well-intentioned, 'features' that are no longer changeable.
Running programs is becoming increasingly less expensive. Programmers, on the other hand, are only getting more expensive. Efficient programming therefore does not mean:

> "writing programs that work as fast as possible."

but

> "writing programs that require as little effort and time possible to be
>
> - *comprehensible*
> - *reliable*
> - *easily maintained*."

This will be one of the major themes during this course. The Assignments that you'll make during this course will not be marked sufficient unless they do what the assignment requiers them to do *and* meet the requirements mentioned above.
Theory provided in this Instruction Manual is an addition to the lectures and the book. The book contains a thorough introduction into Java mechanics, this instruction manual will teach you how to write Java, taking the standards described above into account.

Constants

Imagine a program that reads a number of addresses from a file and prints them on labels - thirty characters wide, six lines high. All of the sudden, the wholesale company changes the size of the labels to thirty-six characters wide and five lines high.
Fortunately, the program looks like this:

```
class Labels {
    // Name       : ...
    // Assignment : Labels
    // Date       : ...

    /* This program reads adresses from input,
     * and prints them in a specific format.
     */

    static final int LABEL_WIDTH = 30;    // characters
                     LABEL_HEIGHT  =  6;  // lines

    // etc...
}
```

The only thing that needs to be done, is to change the two constants and recompile.

Errors that can occur when a program does not incorporate constants are:

- The code contains a `6 on 12 different places and is only replaced on 11 places by a 5`
- `Derived values like 5 (= LABEL_HEIGHT - 1) are not changed to 4 (= LABEL_HEIGHT - 1)!`

Constants cannot only ease the maintenance of a program, but can increase the comprehensibility of the code as well. When a constant, like `LABEL_HEIGHT`, is used, it is imediately clear what this number represents, instead of only knowing its numerical value. For this reason, the usage of constants is very valuable. Therefore, it is advisable to use constants in programs, even if the value of the constant will never change.

Rule of Thumb
All numbers used in a program are constants, except 0 and 1.

Example The following example program will read a number of miles from the standard input and prints the equivalent number of kilometers on the output. Take special notice to the use of identifiers, constants and layout.

```
import java.util.Scanner;
import java.io.PrintStream;

class MileInKilometers {
    static final double MILE_IN_KILOMETERS = 1.609344;

    PrintStream out;

    MileInKilometers() {
        out = new PrintStream(System.out);
    }

    void start() {
        Scanner in = new Scanner(System.in);

        out.printf("Enter the number of miles: ");
        double numberOfMiles = in.nextDouble();

        double numberOfKilometers = numberOfMiles
            * MILE_IN_KILOMETERS;

        out.printf("%f mile equals %f kilometer\n",
                   numberOfMiles, numberOfKilometers);
    }

    public static void main(String[] argv) {
        new MileInKilometers().start();
    }
}
```

☞ Make the assignments **VAT**, **Plumber 1**, **Plumber 2** and **Othello 1**.

Identifiers

All constants, types, variables, methods and classes have to be assigned a name. This name is called the identifier. This identifier has to be unique within the class it is defined in. This might seem easier than it is. In this practical you will learn to link the correct identifier to an object.

The importance of the right name The identifier that is assigned to an object should reflect the information it contains. When a variable is needed to maintain a record of the number of patients in a hospital, *n* would not suffice as the identifier for this variable. The identifier *n* does not give any information about this variable. When the identifier *number* is chosen, the problem seems to be resolved, but is not: it is still unclear to which number the identifier refers. Is it the number of doctors? Is it the number of beds? No, it is the number of patients. That is why this variable should be called *numberOfPatients*. It might take some time to find an appropriate identifier in some cases, but it is certainly worth the effort. This ensures that everyone will understand your program, including the teaching assistant.

Example A long time ago, the maximum length of identifiers was limited in some programming languages. All information about the contents of the variable had to be contained in six or seven characters. This meant that it was very difficult to find clear and understandable identifiers. As a result, programs were often hard to read. A program that had to find travel times in a timetable would contain identifiers like:

```
int ott,  // outward travel time, in minutes
    rtt;  // return travel time, in minutes
```

The introduction of programming languages like Pascal, significantly improved the readability of code by removing the restriction on identifier lengths. Like Pascal, Java does not limit the length of identifiers. Therefore the identifiers in the example can be rewritten:

```
int outwardTravelTime, // in minutes
    returnTravelTime;  // in minutes
```

Abbreviated identifiers Uncommon abbreviations should not be used as identifiers, as the example above illustrates. Identifiers do not necessarily have to be long to be understandable. In mathematics for example, characters are often used to denote variables in equations. Let's have a look at the quadratic equation:

$$ax^2 + bx + c = 0$$

A quadratic equation has at most two solutions if the discriminant is larger than zero:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

A program to solve a quadratic equation would contain the following code:

```
discriminant = (b * b) - (4.0 * a * c);
if (discriminant >= 0) {
    x1 = (-b + Math.sqrt(discriminant)) / (2.0 * a);
    x2 = (-b - Math.sqrt(discriminant)) / (2.0 * a);
}
```

Note that this implementation uses the identifiers *a*, *b* and *c* in the same way as the mathematical definition. Readability would not improve if these identifiers would be replaced by *quadraticCoefficient*, *linearCoefficient* and *constantTerm*. It is clear that using *a*, *b* and *c* is the better choice. The identifier *discriminant* is used as no specific mathematical character is defined for it. The class `Math` identifies the method to calculate a square root with `sqrt()`, it also identifies a number of other methods with equally well known abbreviations. For example: `cos()`, `ln()` and `max()`.

Exceptions There are some conventions for identifiers. An example for calculating the factorial of $n > 0$:

```
int factorial = n;
for(int i = n-1; i > 0; i--){
    factorial *= i;
}
```

The identifier for the variable *n* is not changed into *argument*. Numerical arguments are by convention often identified as *n*. Variables that are used for iterations are similarly not identified as *counter*, but as *i*. When more than one iterator is used, it is common practice to use *j* and *k* as identifiers for next iterators.
Let's look at another example. When programming a game of chess, the pieces on the board can be identified by *ki* (king), *qu* (queen), *ro* (rook), *bi* (bishop), *kn* (knight) and *pa* (pawn). Anyone a elemental knowledge of chess will surely understand these abbreviations. However, if someone else reads this program *kn* might be interpreted as king and *ki* as knight. This example shows the

need of *'psychological distance'* between two identifiers. The psychological distance between identifiers cannot be measured exactly. Psychological distance is roughly defined as 'large' when the chance of confusion between identifiers is nearly non-existing, and as 'small' when it is almost inevitable. Two identifiers with a very small psychological distance are the identifiers in the first timetable example.

✎ **Rule of Thumb**
Identifiers which are used a lot in the same context, need to have a large psychological distance.

Conventions

One important restriction for choosing identifiers is that they cannot contain whitespace. It is common practice to write identifiers consisting of multiple words by capitalising each word, except the first. In this practical the following guidelines are in place:

- Names of variables, methods and functions begin with a lower case letter. All following words are capitalised. No whitespace is used.

  Example: `int numberOfStudents;`
  Example: `void start() { ... }`
  Example: `X readX(Scanner xScanner);`

- Identifiers identifying constants are written in upper case. If an identifier for a constant consists of multiple words they are separated by underscores.

  Example: `static final int MAXIMUM_NUMBER_OF_STUDENTS = ...;`

- Identifiers identifying a class are written in the same way as variables except for the first letter, which is capitalised.

  Example: `class CollegeTimetable { ... }`

Self test

Expressions 1

The following questions are on expressions. These questions do not need to be turned in. Do make sure you are able to answer all the questions, as similar questions will be asked during the exam. For all questions give the generated output, or indicate an error. In addition write down every expression in a question and denote the type and value of the result of the expression.

Question 1

```
void start() {
    int result = 2 + 3;
}
```

Question 2

```
void start() {
    double result = 1.2 * 2 + 3;
}
```

Question 3

```
void start() {
    String result = "ab" + "cd";
}
```

Question 4

```
void start() {
    char result = ’c’ - ’a’ + ’A’;
}
```

Question 5

```
void start() {
    boolean result = true || false;
}
```

Question 6

```
void start() {
    int result = 17 / 4;
}
```

Question 7

```
void start() {
    int result = 17 % 4;
}
```

Question 8

```
void start() {
    if (true) {
        out.printf("not not true");
    }
}
```

Question 9

```
void start() {
    if (false) {
        out.printf("really not true");
    }
}
```

Question 10

```
void start() {
    if (2 < 3) {
        out.printf("2 is not larger or equal to 3");
    }
}
```

Question 11

```
void start() {
    if ((3 < 2 && 4 < 2 && (5 == 6 || 6 != 5)) || true) {
        out.printf("too much work");
    }
}
```

Question 12

```
void start() {
    char number = '7';

    out.printf("%c", number);
}
```

Question 13

```
void start() {
    if (false && (3 > 2 || 7 < 14 || (5 != 6))) {
        out.printf("finished quickly");
    }
}
```

Expressions 2

The following questions are about expressions. For all questions give the generated output, or indicate an error. In addition write down every expression in a question and denote the type and value of the result of the expression.

Question 1

```
double function() {
    int number = 2;
    return number / 3;
}

void start() {
    double result = function() * 3;
}
```

Question 2

```
boolean worldUpsideDown() {
    boolean numbersUpsideDown = 2 > 3;
    boolean booleansUpsideDown = true == false;

    return numbersUpsideDown && booleansUpsideDown;
}
```

```
void start() {
    if (worldUpsideDown()) {
        out.printf("The world is upside down!\n");
    } else {
        out.printf("The world is not upside down.\n");
    }
}
```

Question 3

```
int awkwardNumber() {
    char character = 'y';
    return 'z' - character;
}

void start() {
    out.printf("The result is awkward result: %s\n",
            awkwardNumber() );
}
```

Question 4

```
void start() {
    if ('a' < 'b') {
        out.printf("smaller\n");
    }
}
```

Question 5

```
void start() {
    if ('a' > 'B') {
        out.printf("hmmm\n");
    }
}
```

Question 6

```
void start() {
    char number = '7';
    out.printf("%d\n", number - 1);
}
```

If-statements

☞ Study your lecture notes on if-statements. Section **3.1** of the book will provide additional information on if-statements.

Example This example program will read an exam grade and prints whether this student has passed.

```java
import java.util.Scanner;
import java.io.PrintStream;

class Passed {
    static final double PASS_MINIMUM = 5.5;

    PrintStream out;

    Passed() {
        out = new PrintStream(System.out);
    }

    void start() {
        Scanner in = new Scanner(System.in);

        out.printf("Enter a grade: ");
        double grade = in.nextDouble();

        if (grade >= PASS_MINIMUM) {
            out.printf("The grade, %f, is a pass.\n", grade);
        } else {
            out.printf("The grade, %f, is not a pass.\n", grade);
        }
    }

    public static void main(String[] argv) {
        new Passed().start();
    }
}
```

This example can also be implemented using a ternary operator as described in the section Layout.

☞ Make the assignments **Electronics** and **Othello 2**.

Assignments

1. VAT

☞ Before starting with this assignment, read the theory about **Efficient programming** en **Constants**.

Write a program that takes the price of an article including VAT and prints the price of the article without VAT. The VAT is currently 21.00%.

Example Using an input of 121 the output will be:[1]

```
Enter the price of an article including VAT: 121
This article will cost 100.00 euro without 21.00% VAT.
```

2. Plumber 1

The employees at plumbery 'The Maverick Monkey' are notorious bad mathmaticians. Therefore the boss has decided to use a computer program to calculate the costs of a repair. The costs of a repair can be obtained with the following calculation: the hourly wages multiplied by the number of billable hours plus the call-out cost. The number of billable hours is always rounded. Plumbing laws fix the call-out cost at €16,00.
Write a program that calculates the costs of a repair. Take the hourly wages and number of billable hours as input for this program.

Example A plumber earning €31,50 an hour working for 5 hours should get the following output.

```
Enter the hourly wages: 31.50
Enter the number of billable hours: 5
The total cost of this repair is: €173.50
```

3. Plumber 2

After careful assessment of the new program it turns out that the employees of 'The Maverick Monkey' are as bad at rounding numbers as they are at making calculations. Therefore the boss decides to alter the program in such a way that the program only needs the number of hours an employee has actually worked. The program will determine the number of billable hours based on this input. Make a copy of the previous assignment and edit the code to implement this new feature. The number of billable hours is the number of hours worked rounded to an integer.

Example A plumber earning €31.50 an hour, working for 4.5 hours should get the following output.

```
Enter the hourly wages: 31.50
Enter the number of hours worked: 4.5
The total cost of this repair is: €173.50
```

Hint: How can a number be rounded to an integer?

[1]Examples will have input printed in italics.

4. Othello 1

During this course there will be multiple assignments concerning the game of Othello, also known as Reversi. More about Othello can be found at: http://en.wikipedia.org/wiki/Reversi.

The goal of this assignment is to give some information about the outcome once a game has finished. This information is obtained by two measurements:

- The percentage of black pieces of all the pieces on the board.
- The percentage of the board covered in black pieces.

The Othello board measures eight squares by eight squares, making the total number of squares sixty-four.
Write a program that takes the number of white pieces followed by the number of black pieces as input. Print the two percentages as output.

Example

```
Enter the number of white pieces on the board: 34
Enter the number of black pieces on the board: 23
The percentage of black pieces on the board is: 35.94%
The percentage of black pieces of all the pieces on the board is: 40.35%
```

5. Electronics

☞ Before starting this assignment, read the theory about **Identifiers** and **If-statements**.

The electrics company 'The Battered Battery' is nearly bankrupt. To avoid total disaster, the marketing branch has come up with a special sale to attract more customers. Whenever a customer buys three products, he or she receives a 15% reduction on the most expensive product. Write a program that takes the prices of three products as input and prints the discount and final price as output.

Example Determine the reduction and final price if the three products cost €200, €50 and €25 respectively.

```
Enter the price of the first article: 200
Enter the price of the second article: 50
Enter the price of the third article: 25
Discount: 30.00
Total: 245.00
```

6. Othello 2

During a game of Othello the time a player spends thinking about his moves is recorded. Write a program that takes the total time that two players have thought, one human, one computer, in milliseconds as input. The program determines which of the two players is human and prints the thinking time of the human in the following format: *hh:mm:ss*. It may be assumed that a computer does not require more than a thousand milliseconds to contemplate its moves.

Example

```
Enter the time the black player thought: 234432
Enter the time the white player thought: 36
The time the human player has spent thinking is: 00:03:54.
```

7. Manny

☞ The following four assignments use loops. Use the right loop for the right assignment, using all the following loops: for, while without hasNext(), while with hasNext() and do-while.

Mobster Manny thinks he has found the perfect way to part money from their rightful owners, using a computer program. Mobster Manny secretly installs the program on someone's computer and remains hidden in a corner, waiting for the program to finish. The program will ask the user how much he or she wants to donate to charity: the thirsty toads in the Sahara (Manny's Wallet). If the unsuspecting victim wants to donate less than €50, the program will ask again. The program will continue to ask for an amount until the user has agreed to donate €50 or more, after which Mobster Manny will show up to collect the money.
Write this malicious program, but make sure it does not fall in the wrong hands.

Example An example of a correct execution of the program is shown below:

```
Enter the amount you want to donate:
0
Enter the amount you want to donate:
10
Enter the amount you want to donate:
52.20
Thank you very much for your contribution of 52.20 euro.
```

8. Alphabet

Write a program that prints the alphabet on a single line.

9. Collatz

One of the most renowned unsolved problems is known as the Collatz conjecture. The problem is stated as follows:

> Start out with a random number n.
>
> - if n is even, the next number is $n/2$
> - is n is odd, the next number is $3n + 1$.
>
> This next number is treated exactly as the first. This process is repeated.
> An example starting with 11: 11 34 17 52 26 13 40 20 10 5 16 8 4 2 1 4 2 1 4 2 1 ...
> Once the sequence has reached 1, the values repeat indefinitely. The conjecture is that every sequence ends with 4 2 1 4 2 1 ...

This conjecture is probably correct. Using computers all numbers up to $10 * 2^{58}$ have been found to end with this sequence. This problem might seem very simple, but no one has proved the conjecture since Collatz stated it in 1937. There have even been mathmaticians that have spent years of continued study on the conjecture, without success. Fortunately, writing a program that generates the Collatz sequence is a lot less challenging.
Write a program that takes any positive integer and prints the corresponding Collatz sequence. End the sequence when it reaches one.

Hint Use the % (modulo) operator to test whether a number is even or odd.

10. SecondSmallest

Take an unknown number of positive integers as input. Assume that the first number is always smaller than the second, all numbers are unique and the input consists of at least two integers. Print the second smallest integer.

Example

```
10 12 2 5 15
The second smallest number is:5
```

To denote the end of the input, press enter followed by Ctrl-Z (Windows) or Ctrl-D (Linux and OSX).

3 Methods and functions

Abstract

The programs written in the previous module use if-statements and loops. Writing complicated programs with these statements will quickly result in confusing code. Introducing methods and functions to the code can solve this problem. This module will provide the neccesary knowledge on how to use these and more importantly, on how to structure the code.

Goals

- Familiarize with methods, functions and parameters.
- Use methods and functions to structure programs.

Instructions

- Read the theory about **Methods and functions**. With this knowledge in mind, make the assignments **NuclearPowerPlant**, **RepeatCharacter 1**, **RepeatCharacter 2**, **Pyramid** and **Pizza**.
- Make sure to import the library libUI.jar into Eclipse as described in **Importing the libUI.jar**. Now make the graded assignment **Replay 1**.

Theory

Methods and functions

The theory on how methods work, what they are used for and how to call them has been explained in the lectures. A method call is just another statement. The execution of this statement is slightly more complicated than the execution of a normal Java statement; instead of executing a single statement, a whole method, possibly calling other methods, has to be executed. The great thing about using methods is that at the moment that a method is called, it does not matter how the method is executed. The only thing that matters is *what* the method does, and not *how* the method does this.
An example. A program that translates Dutch text into flawless English will no doubt feature a piece of code like this:

```
class TranslateDutchToEnglish {
    void start() {
        while (in.hasNext()) {
            String dutchSentence = readSentence();
            String englishSentence = translateSentence(dutchSentence);
            writeSentence(englishSentence);
        }
    }

    // etc
}
```

It is very unlikely that someone will doubt the correct execution of this piece of code. Whilst writing a part of the program, it is assumed that the methods readSentence(), translateSentence() and writeSentence() exist. How these methods work, does not matter. Without knowing how these methods work, it cán be concluded that this piece of code is correct.
The method readSentence() is not that difficult to write. A sketch of this method:

```
String readSentence() {
    /* Returns a Dutch sentence. */

    String sentence = readWord();

    while (!endOfSentence()) {
        sentence += " " + readWord();
    }

    return sentence;
}
```

Methods are used to split the program in smaller parts, that have a clear and defined function. This can all be done without knowing how other methods do what they are supposed to do. When writing a part of the program, it is important not to be distracted by a detailed implementation somewhere else in the program. This also works the other way around. When writing a method, it is not important what it is used for in the part of the program that calls it. The only thing that matters, is that the method does exactly what it is supposed to do according to the method name.

Small pieces of code can easily be understood and can be checked easily whether they do what they are supposed to do.

✎ **Rule of Thumb**
A method consists of no more than 15 lines.

An elaborate example is provided below. Whilst studying this example, pay special attention to the constructor; this is the method sharing the name of the class. The constructor can be found in a lot of programs. All instances of a class are created by the operator new followed by a constructor. At last, study the use of the Scanner and PrintStream classes. These classes will be extensively used in the Replay assignment.

Example The world-renowned Swiss astrologer Professor Hatzelklatzer has discovered a new, very rare disease. This disease will be known to the world as the Hatzelklatzer-syndrom. The disease is charactarized by seizures lasting for approximately one hour.
Professor Hatzelklatzer suspects that these seizures happen more often in odd months. He has asked his assistent to write a program that will test this hypothesis. Professor Hatzelklatzer has observed a group of test subjects. A file contains all the reported seizures. Each line indicates the date on which one of the test subjects suffered from a seizure. The input is structured as follows:

```
12 01 2005
28 01 2005
etc...
```

The following example program will parse this input.

```java
import java.io.PrintStream;
import java.util.Scanner;

class Hatzelklatzer {
    // Name      : Heinz Humpelstrumpf
    // Assignment: Hatzelklatzer
    // Date      : 9-29-1997

    static final int STARTING_YEAR = 1950,
                     FINAL_YEAR = 2050;

    PrintStream out;

    Hatzelklatzer() {
        out = new PrintStream(System.out);
    }

    void printPercentageOfCases(double percentage) {
        out.printf("The percentage of illnesses that match " +
                   "the hypothesis is: %.2f\n", percentage);
    }

    /* Reads a number from the input. If the number is not
```

```java
     * in range the program will print an error message and
     * terminates. Otherwise, the number is returned.
     */
    int readInRange(Scanner input, int start, int end) {
        int result = input.nextInt();
        if (result < start || result > end) {
            out.printf("ERROR: %d is not in range (%d, %d)\n",
                       result, start, end);
            System.exit(1);
        }
        return result;
    }

    boolean oddMonth(Scanner input) {
        // the day is read, but not saved
        readInRange(input, 1, 31);

        // the month is read and saved in the variable "month"
        int month = readInRange(input, 1, 12);

        // the year is read, but not saved
        readInRange(input, STARTING_YEAR, FINAL_YEAR);

        return month % 2 != 0;
    }

    public void start() {
        Scanner in  = new Scanner(System.in);

        int totalNumberOfSeizures = 0,
            numberInOddMonths = 0;

        while (in.hasNext()) {
            // read the date and save it in the variable date
            String date = in.nextLine();

            // declare and initialize a scanner
            // to read from the string
            Scanner dateScanner = new Scanner(date);

            if (oddMonth(dateScanner)) {
                numberInOddMonths += 1;
            }
            totalNumberOfSeizures += 1;
        }

        double percentage = ((double) numberInOddMonths /
                            totalNumberOfSeizures) * 100.0;
        printPercentageOfCases(percentage);
    }

    public static void main(String[] argv) {
        new Hatzelklatzer().start();
    }
```

```
}
```

☞ You should now have sufficient knowledge to make the assignments **NuclearPowerPlant**, **RepeatCharacter 1**, **RepeatCharacter 2**, **Pyramid** and **Pizza**.

Importing the libUI.jar

☞ The **Replay 1** assignment is the first assignment that requires the libUI. Complete all previous assignments before starting on this assignment.

During the remaining part of this course, some of the assignments require the UserInterface. The UserInterface is a collection of classes that enables the use of a graphical interface instead of the console as input and output. Such a collection of classes, providing additional features, is called a library.

To use this library, the Java compiler and the Java interpreter have to know where to find it. First, download LibUI from Canvas (the file should be called `libUI.jar`, and save it somewhere on your computer where you can find it. Then, in Eclipse, right click your project folder. In the window that appears, click `Java Build Path`, then click the tab called `Libraries`, then click `Classpath`. On the right-hand side of the window there should be an option called `Add External JARs`. Click that, navigate to the `libUI.jar` file on your computer, and click OK. The LibUI should now be imported. **Warning:** don'move the location of the `libUI.jar` file to a different location on your computer, because your project might no longer work if you do so.

If you have trouble importing the LibUI, feel free to ask your teaching assistant for help.

Import-statements can now enable the use of specific elements of the UI-library. This works similarly to import-statements for the Scanner and PrintStream classes. For example, the program requires the UserInterfaceFactory from the libUI.jar. To enable this class to be used, add the following line to the top of the program:

```
import ui.UserInterfaceFactory;
```

Selecting input using the UserInterface Using the libUI.jar library, Eclipse can use files as input instead of the *standard input*, the keyboard. To select a file as input, the askUserForInput().getScanner() method is used. This method can be found in the UIAuxiliaryMethods class. Add the following line to the top of the program:

```
import ui.UIAuxiliaryMethods;
```

and add the following statement to the top of the start method:

```
Scanner fileScanner = UIAuxiliaryMethods.askUserForInput().getScanner();
```

When a program is executed that includes the above mentioned statement, the program will open a browser to select the input file. Browse to the location of the file and press Enter. The Scanner fileScanner can now be used to read the content of the selected file.

Using the UserInterface on laptops Using the UserInterface on a laptop with a small screen resolution may cause a part of the UserInterface to disappear off screen. To prevent this, add the following statement to the constructor:

```
UserInterfaceFactory.enableLowResolution(true);
```

Assignments

1. NuclearPowerPlant

☞ Before starting this assignment, read the theory on **Methods and functions**.

The nuclear powerplant at Threeyedfish will automatically run a program to print a warning message when the reactor core becomes unstable. The warning message reads:

```
NUCLEAR CORE UNSTABLE!!!
Quarantine is in effect.
Surrounding hamlets will be evacuated.
Anti-radiationsuits and iodine pills are mandatory.
```

Since the message contains crucial information, it should be printed three times. Write a program that prints this message three times using a single method. Do not use duplicate code. Seperate every warning message by a blank line.

2. RepeatCharacter 1

Write a program that will:

- read a number from standard input
- print that number of exclamation marks

Use methods with parameters for this assignment.

3. RepeatCharacter 2

This assignment takes of where RepeatCharacter1 has finished. Make a copy of RepeatCharacter1 and edit the code so that the program will:

- read a number from standard input
- print that number of exclamation marks
- read another number from standard input
- print that number of commas

Solve this problem with as little change to the previous program as possible. One method should suffice.

4. Pyramid

Write a program that prints a pyramid made of letters in the middle of the screen. Use methods with parameters for this assignment. The example shows the expected output, a pyramid of 15 levels. It can be assumed that the screen width is 80 characters.

Example

```
              a
             bbb
            ccccc
           ddddddd
          eeeeeeeee
         fffffffffff
        ggggggggggggg
       hhhhhhhhhhhhhhh
      iiiiiiiiiiiiiiiii
     jjjjjjjjjjjjjjjjjjj
    kkkkkkkkkkkkkkkkkkkkk
   lllllllllllllllllllllll
  mmmmmmmmmmmmmmmmmmmmmmmmm
 nnnnnnnnnnnnnnnnnnnnnnnnnnn
ooooooooooooooooooooooooooooo
```

5. Pizza

Mario owns a pizzaria. Mario makes all of his pizzas with 10 different ingredients, using 3 ingredients on each pizza. Mario's cousin Luigi owns a pizzaria as well. Luigi makes all his pizzas with 9 ingredients, using 4 ingredients on each pizza. Mario and Luigi made a bet: Mario believes that customers can order more pizzas in his pizzaria than they can order in Luigi's pizzaria.
Write a program that calculates the winner of this bet. Use functions for this assignment.

Hint When choosing k items from n possible items, the number of possibilities can be obtained using the following formula:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

Graded Assignment

6. Replay 1

☞ Starting whith this assignment, a lot of assignments will be using the library libUI.jar and read input from a file. Instruction on how to do this in Eclipse can be found in **Importing the libUI.jar**.

The goal of this assignment is to write a program that can replay a game of Othello. The input file contains all the changes that have been made to the board in every turn. All the changes that are made during a turn are to be visualized on screen.

Read and process one line of the input file at a time, letting the changes take place on the board. After each turn, the game halts as long as it has been instructed to by the input file. When the end of the file is reached, the game has finished.
The game starts with four stones in the center of the screen. Two white stones at `d 4` and `e 5` and two black stones at `d 5` and `e 4`. Make sure that these stones are present before processing any turns.

Input specification A number of games of Othello can be found on Canvas. A part of the input file could looks like this:

```
white   4198    move c 3
white   0       move d 4
black   0       move c 2
black   0       move c 3
white   1383    move b 2
white   0       move c 3
black   4       move a 2
black   0       move b 2
etc...
```

The line starts with indicating which player played this turn. The next number indicates the thinking time in milliseconds. After this there is either the word `"move"` or `"pass"`. Stones are only being placed when a turn is not passsed. The coordinate following a waiting time > 0 is the stone placed by the player currently playing. All other coordinates by the same player are captured stones.

Specification OthelloReplayUserInterface This assignment uses the OthelloReplayUserInterface to show the othello board. After declaring and initializing it, like any other variable, it is ready to use:

```java
OthelloReplayUserInterface ui;
ui = UserInterfaceFactory.getOthelloReplayUI();
```

If all is well, the OthelloReplayUserInterface should pop up.
All the methods available to the user are documented in the *Javadoc*. It can be found at: `https://phoenix.labs.vu.nl/doc/java/doc/`. All the methods required for this assignment are:

- `void wait(int milliseconds);`
- `void place(int x, int y, int type);`
- `void showChanges();`

The way these methods work is described in the Javadoc.

The class contains the following constants:

- `public static final int NUMBER_OF_COLUMNS = 8;`
- `public static final int NUMBER_OF_ROWS = 8;`
- `public static final int BLACK = 2;`
- `public static final int WHITE = 1;`
- `public static final int EMPTY = 0;`

Example The following example shows a piece of code placing a white stone in the top-left corner, a black stone in the bottom-left corner, waiting 5 seconds, changing the white stone in the top-left corner to black and finally deleting the black stone in the bottom-right corner.

```java
import ui.OthelloReplayUserInterface;
import ui.UserInterfaceFactory;

class UIExample {
    static final int WAITING_TIME = 5000;  // in milliseconds

    OthelloReplayUserInterface ui;

    UIExample() {
        ui = UserInterfaceFactory.getOthelloReplayUI();
    }

    void start() {
        ui.place(0, 0, ui.WHITE);
        ui.place(ui.NUMBER_OF_COLUMNS - 1, ui.NUMBER_OF_ROWS - 1,
                 ui.BLACK);
        ui.showChanges();

        ui.wait(WAITING_TIME);

        ui.place(0, 0, ui.BLACK);
        ui.place(ui.NUMBER_OF_COLUMNS - 1, ui.NUMBER_OF_ROWS - 1,
                 ui.EMPTY);
        ui.showChanges();
    }

    public static void main(String[] argv) {
        new UIExample().start();
    }
}
```

Submitting This assignment needs to be submitted on Practool. Instructions on how to do this can be found in module 1. Make sure that when exporting your files (to a .jar-file), "Export java source files and resources" is checked.

4 Parsing input

Abstract

A lot of programs depend on some sort of input. In previous modules the Scanner was used to read numbers. This module will introduce *structured reading* of *structured input* to parse complex input and write structured programs.

Goals

- Use next(), nextLine() and useDelimiter() from the Scanner class to read structured input.
- Write well structured code that reflects the way the input is parsed.

Theory

Layout

A good layout is essential to make comprehensible programs. There are many different layouts that will result in clear programs. There is no single 'best layout', it is important though to maintain the same layout throughout the whole program. Examples of a good layout can be found in all the examples in the book and in this instruction manual. A couple of rules of thumb:

Rule of Thumb

A } is always aligned vertically below the corresponding method, for, while, if or switch statements. The keyword itself is not indented. All code in the body of these control statements is indented by four or eight spaces, usually the width of one or two tabs.

Rule of Thumb

Methods are separated by at least one white line. The declaration of variables and assignments are also separated by a white line. White lines can be added anywhere, if this increases clarity.

Novice programmers often lack "white" in their programs. Indenting four spaces is not a lot, eight is better. The paper and screen are wide enough. If the code does reach the end of the page, by indenting twelve times, the code is probably too complicated. It will have to be simplified by introducing new methods and functions. The TAB key is useful for indenting pieces of code.
The layout of an if-statement is one of the most difficult statements to define. The layout is greatly influenced by the code following the statement. These examples show possible layouts:

```
if (boolean expression) {
    statement;
}
```

```
if (boolean expression) {
    statement;
} else {
    statement;
}
```

```
if (boolean expression) short statement;
```

```
if (boolean expression) short statement;
else short else-statement;
```

```
if (boolean expression) {
    a lot of statements
    ...
} else {  // opposite of the boolean expression
    statements
    ...
}
```

```
if (boolean expression 1) {
    statement 1;
}
else if (boolean expression 2) {
    statement 2;
}
else if (boolean expression 3) {
    statement 3;
}
else {   // explanation on the remaining cases
    statement 4;
}
```

Switch

The switch-statement can be used when a choice between different pieces of code is made, based on a value.

```
int a = ...;
switch (a) {
case 1:
    statement 1;
    break;
case 2:
    statement 2;
    break;
case 3:
    statement 3;
    break;
default:
    explain all remaining cases
    statement 4;
}
```

The switch-statement can only be used in combination with primitive types like integers and characters. Using the switch-statement in combination with strings will not work as expected. In addition to this restriction, the switch-statement can only be used when checking the value for equality `(==)`; this example uses `(a == 1)`, `(a == 2)` and `(a == 3)`. Other expressions, like < or >, can only be achieved by using if-statements.

The ternary operator

When a choice between two cases can be made based on a short expression, the following statement can be used:

```
expression ? value, if expression is true : value, if expression is false
```

This way, the following piece of code:

```
if (a < b) {
    minimum = a;
} else {
    minimum = b;
}
```

can be shortened to:

```
minimum = (a < b) ? a : b;
```

Comments

"Comments make sure a program is readable. Everyone knows this. In the past, when only very few could program, sometimes someone would write a program without comments. Now this is often considered as old-fashioned, maybe even offensive. Better safe than sorry and add some extra comments."

Wrong!

Comments are not meant to explain dodgy programs to readers. A program that can be understood without comments, is better than a program that cannot be understood without comments. Comments may never replace clear programming!
Comments should not be written wherever possible, but only on those occasions where they are neccesary. An example of unneccesary commenting:

```
// the sum of all values is assigned to sum.
sum = 0;
for (input.hasNext()){
    sum += input.nextInt();
}
```

It can be assumed the reader can understand Java. A clear piece of code does not need additional explanation.
However, there are some cases in which it is advisable to add comments, in the middle of a method (for example, the previous if-statements). Though usually comments are placed at the top of the method. These comments are usually placed to explain a complex method, describing:

- what the method does
- (if neccessary) how it does this
- (if neccessary) how the method changes external values. If, for example, a global variable is changed within the method, it might be useful to write this in a comment.

A well written program contains a lot of methods without any comments. Usually, the name of a method will indicate precisely what will happen and the code will be readable. For example:

```
void print(int[] row);
```

does not require explanation telling the reader that a row is printed. However, the method

```
/* Sorts the list using "rapidsort";
 * see Instruction Manual.
 */
void sort(int[] row) {
```

does require this kind of explanation. The execution of a sorting algorithm is not trivial. This can be solved in two ways: explaining the algorithm within the program, or reference another document describing the precise execution of this piece of code. In the latter case, the code has to exactly match the description ofcourse.

✎ **Rule of Thumb**

If the *name* of the method explains *what* it does and it is trivial *how* it does this, no comments are neccessary.
If one or both of the prerequisites are not met, a comment is needed.
There are very little or no comments within a method.

Advanced use of the Scanner

Using the Scanner class, structured input can be read. Until now, all input was quite simple; the nextInt() and nextDouble() functions were used to read numbers, while the hasNext() and while-loops allowed for an unknown amount of numbers to be read. The Scanner is capable of reading much more sophisticated input. This section will introduce three new Scanner features.

Reading strings All input so far consisted out of numbers. It is useful to be able to read strings as well; the next() and nextLine() functions will read strings.

Reading from a string Scanners made with the `new Scanner(System.in)` statement read from standard input, the keyboard. Scanners made with a `new Scanner(`*string*`)` statement will read from the defined string. It is possible to create Scanners that read from a string, read by another Scanner. For example:

```
Scanner in = new Scanner("a,b,c,d 2#4#6#8");
String letters = in.next();
String numbers = in.next();

Scanner letterScanner = new Scanner(letters);
Scanner numberScanner = new Scanner(numbers);
```

Reading using delimiters All next-functions, except nextLine() read up to the next delimiter. The delimiter is defined as whitespace by default. Whitespace includes spaces, tabs and newlines. For example: reading the text `"2 4    6 8"` with four calls of the nextInt(), next() or nextDouble will result in four seperate values. Calling the nextLine() function will ignore the delimiter and read up to the end-of-line. In other words: nextLine() will always read with the `"\n"` delimiter.

The delimiter of a scanner can be changed with the useDelimiter() method. Reading the string `"2#4###6#8"` with nextInt() would try to read the whole piece of text and throw an exception stating that there is no integer. This problem can be solved by changing the delimiter of this Scanner to `"#"`. The following example will continue where the previous example left off:

```
numberScanner.useDelimiter("#");

// read all the numbers, and print the sum of all the numbers.
int sum = 0;
while (numberScanner.hasNext()) {
    sum += numberScanner.nextInt();
}
out.printf("%d\n", sum);
```

Example Input is often structured, this means that the input is made up of different parts, often again divided in seperate parts, and so on and so forth. Such input can be read in a structured way by first reading the large parts and forwarding these parts to a different method that will read the sub-parts.

The example uses the following structured input:

```
Melissa White-Admiral Nelsonway;12;2345 AP;Seaty
Richard of Hughes-Green Lawn;1;2342 SS;Seaty
Godwyn Large-Calferstreet;101;2341 NG;Seaty
Petronella Diesel-The Mall;1102;2342 MW;Seaty
enz...
```

The input is made up of an unknown number of students and is read with the Scanner in. Every line states the name and address of a single student. The name is separated from the address by a '-'. The address consists of a street, house number, postal code and city. The components are separated by a ';'.

One of the most important skills is to recognize such structures. Study the example and the explanation below. The program will read the input defined above and print the addresses in format suitable for letters.

```
class ExampleProgram {
    PrintStream out;

    ExampleProgram() {
        out = new PrintStream(System.out);
    }

    void printAddress(Scanner addressScanner) {
        addressScanner.useDelimiter(";");

        String street = addressScanner.next();
        int houseNumber = addressScanner.nextInt();
        String postalCode = addressScanner.next();
        String city = addressScanner.next();

        out.printf("%s %d\n%s %s\n", street, houseNumber,
                   postalCode, city);
    }

    void printStudent(Scanner studentScanner) {
        studentScanner.useDelimiter("-");

        String name = studentScanner.next();
        String address = studentScanner.next();
        Scanner addressScanner = new Scanner(address);

        printAddress(addressScanner);
    }

    void start() {
        Scanner in = new Scanner(System.in);

        while (in.hasNext()) {
            String student = in.nextLine();
            Scanner studentScanner = new Scanner(student);

            printStudent(studentScanner);
        }
    }

    public static void main(String[] argv) {
        new ExampleProgram().start();
    }
}
```

The program is very comprehensible, even without comments. The program has three methods, each reading a different aspect of the input:

- start() makes a studentScanner for each student:

  ```
  Melissa White-Admiral Nelsonway;12;2345 AP;Seaty
  Richard of Hughes-Green Lawn;1;2342 SS;Seaty
  Godwyn Large-Calferstreet;101;2341 NG;Seaty
  Petronella Diesel-The Mall;1102;2342 MW;Seaty
  ```
  etc...

- printStudent(Scanner studentScanner) reads the name and address seperately and creates a new Scanner for the address to forward to the printAddress method.

  ```
  Melissa White-Admiral Nelsonway;12;2345 AP;Seaty
  ```

- printAddress(Scanner addressScanner) reads every component, but does not create any more Scanners, as the components do not need to be parsed further.

  ```
  Admiral Nelsonway;12;2345AP;Seaty
  ```

Assignments

1. Replay 2

☞ Before starting this assignment, read the theory about **Advanced use of the Scanner**.

The input of Replay1 was structured like this:

```
black   1035    move b 3
black   0       move c 3
black   0       move d 3
etc...
```

This causes a lot of unneccesary information. Every turn is being devided amongs multiple lines. There is currently no distinction between a turn and a move. The input has been re-aranged, and now looks like this:

```
black   1035    move b 3 c 3 d 3
white   4058    move h 1 g 2 f 3 e 4
black   24423   pass
white   8494    move a 6
etc...
```

In this arrangement, every line consists of a turn, and every turn consists of multiple coordinates. Make a copy of Replay1 and edit the code such that it can handle this new arrangement. In addition, print the number of pieces that have been conquered during each turn. The Javadoc describes how to print in the status bar of the OthelloReplayUserInterface.

Example

```
black: conquered 2 pieces
white: conquered 3 pieces
black: passed
white: conquered 0 pieces
```

2. Replay 3

For this assignment, multiple games of Othello have been combined into a single file, separated by a '='. This file can be found on Canvas. Make a copy of Replay2 and edit the code in such a way that it can handle this input file.
Games should be played one after another, waiting five secconds between the end of one game and the start of another. Before starting a new game, make sure to erase the board and status bar. Information on how to achieve this can be found in the Javadoc.

Graded Assignment

3. Administration

For the end of year administration of Programming for History of Arts students you are to write a program that has 2 functions:

1. calculate a final grade
2. print a small graph of similarity scores and, if applicable, list the students under investigation

The input is structured as follows:

```
Piet van Gogh_5 6 7 4 5 6
5=20=22=10=2=0=0=1=0=1;Vincent Appel,Johannes Mondriaan
Karel van Rijn_7 8 6 6
2=30=15=8=4=3=2=0=0=0;
```

The first line should be interpreted as follows:

```
<name of the student><underscore><one or more grades separated by spaces>
```

You have to calculate the final grade of the student. All grades have the same weight. The final grade is rounded as follows:

- a grade that is >= 5.5 AND < 6 should be noted as a "6-"
- otherwise a grade will be rounded to the nearest half

The second line should be interpreted as follows:

```
<10 numbers separated by '='>;<zero or more names separated by ','>
```

The first 10 numbers are the similarity scores. These scores represent the number of programs matching a certain percentage of the current program in steps of 10%. This means the first numbers indicates the matches from 1%-10% and the last number indicates the matches from 91%-100%.

Since this is not very readable, the professor would like a simple graph according to these rules:

- if there are zero matches, display an underscore: _
- if there are less than 20 matches, display a minus sign: −
- if there are 20 or more matches, display a caret: ∧

The names of the students after the semicolon are the names of the students with matches in the final 3 categories. The names of these students should be printed under the graph. If there are no matches, the program should print "No matches found".

The output for the aforementioned input should be:

```
Piet van Gogh has an average of 6-
        -^^--__-_-
        Vincent Appel
        Johannes Mondriaan
Karel van Rijn has an average of 7.0
        -^-----___
        No matches found
```

5 Arrays and classes

Abstract

In the previous modules methods and functions were used to structure a program. For the more advanced programms another way to structure a program is by using classes. Classes are used to join a number of relating methods and variables. With classes it is possible to represent real-world objects. Because classes can be used for this, Java is called an object-oriented programming language. This module shows how to use classes.

Goals

- Using arrays to save a(n) (un)known number of values.
- Using more classes and recognising when to use extra classes.
- Recognising the right class for the right method, variable or constant.

Theory

Classes

Complex programs almost always require the use of two or more classes. Each class contains a number of logically connected data and/or methods. Two types of classes can be distinguished:

1. Firstly, classes that contain several logically connected methods. An example of such a class is the Scanner class. This class contains all methods used to read input from various sources. All resulting programs from the previous assignments belong to this set of classes as well.
2. Secondly, classes that define an object from the real world. An example of such a class is the class Person. This class could contain data like: name, address, date of birth, etc. In addition, this class could contain a function int age(), which returns the age of this person by using the date of birth and the current date. Another example of such a class is Circle. This class contains a center and a radius. Methods that are logically connected to this class are for example double surfaceArea() and double circumference().

In the first four modules, only classes of the first category were required. Assignments from this module onwards will also use classes from the second category. An example will show how to create and use such a second class. It is good practice to use separate files for different classes.

Example This example will feature a program used by airline 'FlyLo' to calculate the profit of a flight to London. The airline uses four different fares:

1. Toddlers, aged 0 to 4 years old are charged 10% of the regular fare.
2. Children, aged 5 to 12 years old are charged half the regular fare.
3. Adults are charged the regular fare.
4. Elderly peoply, aged 65 years or more pay an extra 10% on top of the regular fare, as they have more money anyway.

The regular fare for a single ticket to London is €99. A Boeing 747, the largest plane in the airline's fleet, will accomodate 400 passengers.
The airline wants to know what consequences a change in the maximum age of a toddler may have. The airline wants to know what the new profit of a flight will be, and if there is an increase or decrease in the profit.

The program uses a two-line input. The first line contains all the passenger's ages. The second line contains the new maximum toddler age. The program will print the two profits and the difference between these two. It is useful to save all the ages in a row. This way, the row can simply be 'asked' how many passengers will fit each category. This data will allow the program to calculate the total profit. A row, such as the one proposed above is a good example of a second class. The class holds data, the ages of all the passengers and the number of passengers. Apart from the constructor, the class has a method to add an age to the row and a method to calculate how many passengers fit into a certain category.

This row can be implemented using an array, which does not necessarily have to be completely filled. This means that the number of ages currently saved in the array has to be maintained. The method add() will add a new age to the back of the row. The function int numberInRange(int startingAge, int endingAge) calculates the number of passengers that fit this range. The methods AgeRow readAgeRow() and double calculateTotalProfit() are self-explanatory.

```java
class AgeRow {
    static final int MAX_NUMBER_OF_PASSENGERS = 400;

    int[] ageArray;
    int numberOfPassengers;

    AgeRow() {
        ageArray = new int[MAX_NUMBER_OF_PASSENGERS];
        numberOfPassengers = 0;
    }

    void add(int age) {
        ageArray[numberOfPassengers] = age;
        numberOfPassengers += 1;
    }

    int numberInRange(int startingAge, int finalAge) {
        // Calculates how many passengers are in the range
        // bounded by startingAge and finalAge

        int result = 0;

        for (int i = 0;i < numberOfPassengers; i++) {
            if (startingAge <= ageArray[i] &&
                ageArray[i] <= finalAge) {
                result += 1;
            }
        }
        return result;
    }
}
```

```
import java.io.PrintStream;
import java.util.Scanner;

class Airplane {
    // Name      : Martijn Bot
    // Assignment: Airplane
    // Date      : August 6th 1997

    static final int MAX_TODDLER_AGE =   4,  // year
                     MAX_CHILD_AGE   =  12,  // year
                     MAX_ADULT_AGE   =  64,  // year
                     MAX_AGE         = 135;  // year

    static final double ADULT_FARE   = 99.0,             // euro
                        TODDLER_FARE = ADULT_FARE * 0.1, // euro
                        CHILD_FARE   = ADULT_FARE * 0.5, // euro
                        ELDERLY_FARE = ADULT_FARE * 1.1; // euro

    PrintStream out;

    Airplane() {
        out = new PrintStream(System.out);
    }

    int readInRange(Scanner input, int start, int end) {
        int result = input.nextInt();
        if (result < start || result > end) {
            out.printf("ERROR: %d is not in range (%d, %d)\n",
                       result, start, end);
            System.exit(1);
        }
        return result;
    }

    int readAge(Scanner input) {
        return readInRange(input, 0, MAX_AGE);
    }

    AgeRow readAgeRow(Scanner input) {
        AgeRow result = new AgeRow();
        while (input.hasNext()) {
            result.add(readAge(input));
        }
        return result;
    }

    double calculateProfit(int start, int end,
                           double fare, AgeRow row) {
        return row.numberInRange(start, end) * fare;
    }

    double calculateTotalProfit(int maxToddlerAge, AgeRow row) {
        double toddlerProfit = calculateProfit(
                0, maxToddlerAge, TODDLER_FARE, row);
        double childrenProfit = calculateProfit(
                maxToddlerAge + 1, MAX_CHILD_AGE, CHILD_FARE, row);

        double adultProfit = calculateProfit(
                MAX_CHILD_AGE + 1, MAX_ADULT_AGE, ADULT_FARE, row);

        double elderlyProfit = calculateProfit(
                MAX_ADULT_AGE + 1, MAX_AGE, ELDERLY_FARE, row);

        return toddlerProfit +  childrenProfit +
            adultProfit + elderlyProfit;
    }

    void printChangeInProfit(double oldProfit, double newProfit) {
        out.printf("When using the new age limits for " +
                   "toddlers and children \n the profit" +
                   "changes from EUR %8.2f to EUR %.2f.\n",
                   oldProfit, newProfit);
        out.printf("The difference is EUR %8.2f.\n",
                    newProfit - oldProfit);
    }

    void start() {
        Scanner in = new Scanner(System.in);

        String passengers = in.nextLine();
        Scanner passengerScanner = new Scanner(passengers);
        String age = in.nextLine();
        Scanner maxToddlerAgeScanner = new Scanner(age);

        AgeRow ageRow = readAgeRow(passengerScanner);
        int newMaxToddlerAge = readAge(maxToddlerAgeScanner);

        double normalProfit = calculateTotalProfit(
                              MAX_TODDLER_AGE, ageRow);
        double newProfit = calculateTotalProfit(
                              newMaxToddlerAge, ageRow);

        printChangeInProfit(normalProfit, newProfit);
    }

    public static void main(String[] argv) {
        new Airplane().start();
    }
}
```

Assignments

1. Array

Read exactly twenty numbers from standard input and print them in reversed order. Do not use a second class yet.

2. BodyMassIndex

Professor Hatzelklatzer has researched the extremely rare Hatzelklatzer-sydrome. There seem to be fewer cases of the sydrome in odd months than in even months. Further research should reveal if the syndrome affects people more often if they are too heavy.
A way of determining whether someone is too heavy is the *body-mass index* (BMI). This is a measure of a person's weight taking into account their height. The BMI is defined as $weight/length^2$. The World Health Organization (WHO) considers a BMI between 18,5 and 25 as ideal and considers people with such a BMI healthy.
The program receives input consisting of two persons with their name, sex, length and weight.

```
Dean Johnson      M    1.78    83
Sophia Miller     V    1.69    60
```

Process this input into structured data. To achieve this, use an useful extra class with useful methods to enhance the structure of the program. Use this structured data to print for each person: an appropriate style of address, surname, the BMI and a statement whether this is considered healthy or not.

Example

```
Mr. Johnson's BMI is 26.2 and is unhealthy.
Mrs. Miller's BMI is 21.0 and is healthy.
```

3. Reverse

The input for this assignment consists of two rows of an unknown number of numbers, at most 20, at least 1. A row is printed on a single line, with the numbers separated by spaces.
Reverse the rows and print them on standard output. Next, print the largest number of each row and print which row has the largest number.

Example

```
5 8 2 1
-100 100 200

1 2 8 5
200 100 -100
Largest number of row 1: 8
Largest number of row 2: 200
The largest number is in row 2.
```

Graded Assignment

4. Travel Distance

For the annual scavenger hunt and puzzle race, each contestant is given a file that contains some number of coordinates. The contestants have to visit these coordinates in a certain order, but that order is different from the order in which the coordinates appear in the file.

An example of such a file looks like this:

```
2,6
b 5,8
b 3,9
f 4,6
b 6,11
f 5,3
f 2,3
```

The first line of the file contains a coordinate (in the form `x,y`). Each other line starts with the letter `b` or `f`, followed by a coordinate. In order to find the correct route, start off with the coordinate on the first line. Then go over the rest of the lines. If a line starts off with the letter `f`, put the following coordinate in front of the route. If the line starts with the letter `b`, put the following coordinate to the back of the route.

Write a program that can read such an input file, and find the correct route. You should write a class called `Coordinate`, and a class called `CoordinateRow`. The `CoordinateRow` class should contain methods to add a coordinate to the front or the back of the row, which you can use to put the coordinates in the correct order. The program should print the correct route, and the total distance of that route. You can assume that you can always travel in a straight line from one coordinate to the other, meaning that you can calculate the distance using the Pythagorean theorem.

The output for the aforementioned input should be:

```
The correct route is:
2,3
5,3
4,6
2,6
5,8
3,9
6,11

The total distance is 17.61
```

Input and output files can be found on Canvas.

6 Events and animations

Abstract

This module introduces events and animations. These notions are essential for programming interactive programs. The graded assignment of this module will involve programming the game *Snake*. A game such as *Snake* is quite complex and therefore it requires a careful approach. Using *stepwise refinement* one begins with a rough sketch of a program, which is developed with increasing detail. Before starting with programming the graded assignment it is compulsary to make such a sketch of the program, which has to be approved before being allowed to continue.

Goals

- Familiarize with events.
- Use events to program an animated program.
- Use events to program an interactive program.
- Use *stepwise refinement* to program complex programs.

Theory

Events

The Replay assignment introduced the Graphical User Interface (GUI). This program was not interactive, it did not react to input provided by the user. To make a program interact with the user, this course uses *events*. An event is for example a mouse-click, a keystroke or even the fact that it is 2 o'clock.
Using the following functions from the GUI will allow the program to work with events.

```
Event getEvent();
```

Calling this function will make the program halt and wait for an event to arise. After an event has risen, the function returns an Event-object containing information on the event. The Event class looks like this:

```
class Event {
    String name;
    String data;
}
```

Pressing the letter 'a' will generate an Event-object containing the name `"letter"` and the data `"a"`. Clicking field 4,3 will return an Event-object containing the name `"click"` and the data `"4 3"`.

An interactive program, a program that reacts to input generated by the user, works as follows:

1. wait for an event to arise
2. process event
3. repeat.

In Java this can be implemented like this:

```
while (true) {  // infinite loop
    Event event = ui.getEvent();
    processEvent(event);
}
```

The method processEvent needs to determine what has happened, and what should happen as a result. Such a method, that calls different methods according to a specific condition is called a *dispatch*-method.

```
void processEvent(Event event) {
    if (event.name.equals("click")) {
        processClick(event.data);
    } else if (event1) {
        processEvent1(event.data);
    } else if (event2) {
        processEvent2(event.data);
    } else {
        ....
    } else {
        ....
    }
}
```

More information on the UserInterface can be found at https://phoenix.labs.vu.nl/doc/java/doc/.

☞ Make the assignment **Events**.

Animations

The way events have been used so far, only allows the program to react to input given by the user. It is not possible for the program to change anything on the screen on it's own account. In order to program an animated program, a program has to be able to do something without requiring input from the user. A program is animated when it does not require events produced by the user to make changes to the screen. This means a program is not animated if the screen changes very rapidly because of a lot of events produced by the user. The first animated program during this course was the Replay assignment. A game of Othello was replayed without any input from the user.
Computer games like snake are called interactive animations. In these programs, the user can influence an animation. In the Replay assignment, the wait() method was used to regulate the speed of the animation. This approach is not useable for interactive animation, as the whole program halts when the wait() method is executed. The program cannot react to events, not even those generated by the user. Waiting for ten seconds, will cause a mouse-click for example to be processed with a ten seccond delay. To solve this issue, the SnakeUserInterface contains the following method:

```
void setFramesPerSecond(double framesPerSecond);
```

When this method is called with, for example 24.0 frames per second, the program will generate 24 events per second. These events all have the name `"alarm"` and data `"refresh"`. The program can now be made to react to these events by refreshing the screen. This way, the wait() method does not have to be used, and events generated by the user can be processed instantly.

☞ Make the assignment **Animation**.

Stepwise Refinement

An important part of writing structured programs is *stepwise refinement*. This can be roughly defined in the following way:

- Write down exactly what the program should do, in English or in another natural language.
- Next, step by step elaborate on the description of the program. Again, write in a natural language, or, when it is trivial, directly in Java.
- This process is repeated until the whole program is written in Java.

When the algorithm is correctly executed, the result will be a flawless structured program.

Example Write a program that reads a date in the format: day month year, separated by spaces. The program prints whether the date is correct. This program will be made using stepwise refinement. Important to note is that all programs execute precisely the same assignment. The only difference is the amount of English replaced by Java.

Program 1

```
class CheckDate {

    CheckDate() {
    }

    void start() {
        // read the date
        // check the date
        // print output
    }

    public static void main(String[] argv) {
        new CheckDate().start();
    }
}
```

Program 2

```
import java.util.Scanner;
import java.io.PrintStream;

class CheckDate {

    PrintStream out;

    CheckDate() {
        out = new PrintStream(System.out);
    }

    boolean isCorrect(int day, int month, int year) {
        // return true if date is correct,
        // return false otherwise
    }

    int readInRange(Scanner input, int start, int end) {
        // read an int and return this int if
        // it is between start and end. Print
        // an error message otherwise and
        // terminate.
    }

    void start() {
        Scanner in = new Scanner(System.in);

        int day   = readInRange(in, 1, 31);
        int month = readInRange(in, 1, 12);
        int year  = readInRange(in, 0, 2500);

        if (isCorrect(day, month, year)) {
            out.printf("The date is correct.\n");
        } else {
            out.printf("The date is not correct.\n");
        }
    }

    public static void main(String[] argv) {
        new CheckDate().start();
    }
}
```

Program 3

```
import java.io.PrintStream;
import java.util.Scanner;

class CheckDate {

    PrintStream out;

    static final int[] NUMBER_OF_DAYS_IN_A_MONTH = {
        // list of thirteen values, for each month
        // the maximum number of days, and a random
        // value at index 0
    };

    CheckDate() {
        out = new PrintStream(System.out);
    }

    boolean isLeapYear(int year) {
        // return true if the year is a leap year
        // return false, otherwise
    }

    boolean isCorrect(int day, int month, int year) {
        if (day == 29 && month == 2) {
            return isLeapYear(year);
        } else {
            return day <= NUMBER_OF_DAYS_IN_A_MONTH[month];
        }
    }

    int readInRange(Scanner input, int start, int end) {
        int result = input.nextInt();
        if (result < start || result > end) {
            out.printf("ERROR: %d is not in range (%d, %d)\n",
                       result, start, end);
            System.exit(1);
        }
        return result;
    }

    void start() {
        Scanner in = new Scanner(System.in);

        int day   = readInRange(in, 1, 31);
        int month = readInRange(in, 1, 12);
        int year  = readInRange(in, 0, 2500);

        if (isCorrect(day, month, year)) {
            out.printf("The date is correct.\n");
        } else {
            out.printf("The date is not correct.\n");
        }
    }
}
```

Program 4

```
import java.io.PrintStream;
import java.util.Scanner;

class CheckDate {
    PrintStream out;

    static final int[] NUMBER_OF_DAYS_IN_A_MONTH = {
        0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };

    CheckDate() {
        out = new PrintStream(System.out);
    }

    boolean isLeapYear(int year) {
        return year % 400 == 0 ||
            (year % 4 == 0 && year % 100 != 0);
    }

    boolean isCorrect(int day, int month, int year) {
        if (day == 29 && month == 2) {
            return isLeapYear(year);
        } else {
            return day <= NUMBER_OF_DAYS_IN_A_MONTH[month];
        }
    }

    int readInRange(Scanner input, int start, int end) {
        int result = input.nextInt();
        if (result < start || result > end) {
            out.printf("ERROR: %d is not in range (%d, %d)\n",
                       result, start, end);
            System.exit(1);
        }
        return result;
    }

    void start() {
        Scanner in = new Scanner(System.in);

        int day   = readInRange(in, 1, 31);
        int month = readInRange(in, 1, 12);
        int year  = readInRange(in, 0, 2500);

        if (isCorrect(day, month, year)) {
            out.printf("The date is correct.\n");
        } else {
            out.printf("The date is not correct.\n");
        }
    }
}
```

Assignments

1. Events

Before starting this assignment, read the theory about **Events**.

Write a program using a SnakeUserInterface of 40×30 which has the following features:

- Clicking on a square results in a piece of wall to be placed on that square.
- Pressing the space bar erases all the walls.
- The program prints the name and data of all events that occur.

2. Animation

Before starting this assignment, read the theory about **Animations**

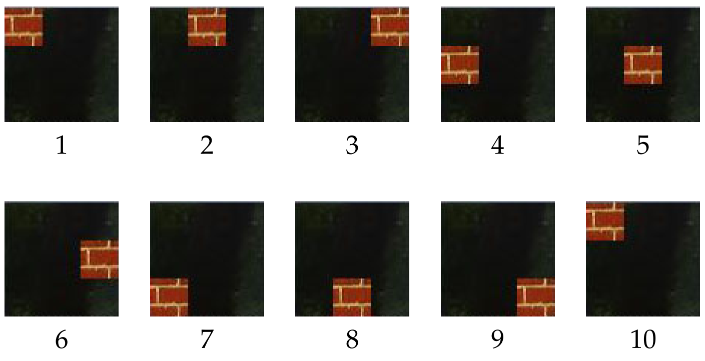
The goal of this assignment is to make an animated program in which a piece of wall moves across the screen. The piece of wall starts out on (0,0) and moves right a square at a time. Upon reaching the end of a row, the piece of wall will move to the first square of the next row. When the piece off wall reaches the end of the last row, it is transferred back to the initial (0,0) position.

On top of this make sure the program implements the following features:

- The animation should slow down 0.5 frames per second when $\leftarrow$ (left arrow) is pressed.
- The animation should speed up 0.5 frames per second when $\rightarrow$ (right arrow) is pressed.
- The piece of wall should change into a green sphere (a part of a snake) when g is pressed. Pressing g again will revert the change.

Use the SnakeUserInterface for this assignment.

Example



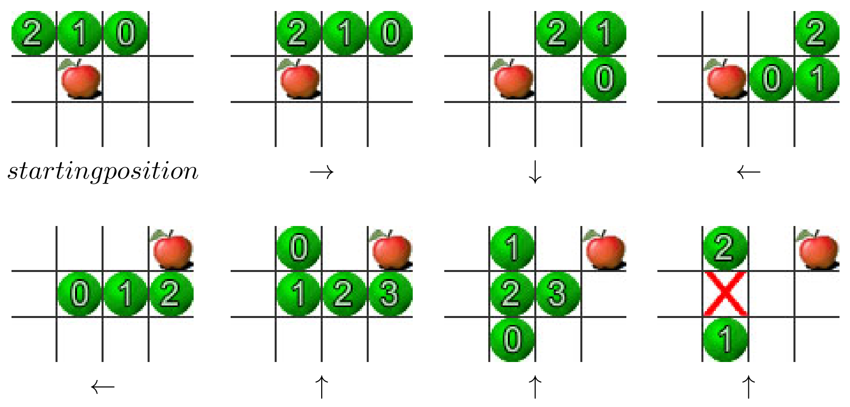
Graded Assignment

3. Snake

☞ Before starting this assignment, read the theory about **Stepwise Refinement**. In addition, before starting with programming, a draft of the program **has** to be approved.

A logical step forward from interactive animated programs is games. The goal of this assignment is to program the classic computer game, *Snake*.
The goal of Snake is to create a snake as long as possible. This is achieved by guiding the snake to apples, lying about on the field. The snake cannot stop moving, and dies whenever it hits something. Because the snake is growing longer and longer as the game progresses, it is increasingly difficult to avoid collisions with the snake itself.
At the start of the game, the snake consists of two pieces at the coordinates (0,0) and (0,1). As said before, the snake is always moving. At the start of the game, it moves to the right. When the user presses one of the arrow keys, the snake changes direction.
At every moment in the game, there is always an apple somewhere in the field. If the snake hits an apple, the snake becomes one piece longer at the next screen refresh. A new apple is placed on a random location, excluding all places covered by the snake.
When the snake reaches the end of the screen, it will re-emerge at the other end.
Note that if the snake goes into a certain direction, it can't move in the opposite direction within one refresh (so for instance, if the snake goes right, and you click left, nothing should happen). Also within one refresh the snake can't change direction more than once.

Example The example below shows a short game of snake, played on a 4x3 field. The game to be designed in this assignment will have a field measuring 32x24. The arrow indicates in which direction the snake is travelling. The numbers on the snake indicate its position in the row.



This assignment uses the SnakeUserInterface. Information about the SnakeUserInterface can be found on `https://phoenix.labs.vu.nl/doc/java/doc/`. To generate a random location for the apple use methods provided in the UIAuxiliaryMethods class.

Bonus Edit the program in such a way that it accepts a level as input. A level defines a number of walls, which the player has to avoid. Levels can be found on Canvas. The structure of these files is as follows: <the coordinates at which the snake starts, starting with the head of the snake>=<the initial direction of the snake>=<the coordinates of the walls>.
Coordinates are formatted in the following way: one coordinate per line, in the format: <x><space><y>. The initial direction is one of four strings: `"L"` (Left), `"R"` (Right), `"U"` (Up) of `"D"` (Down).
An example of a piece of such a file:

```
1 0
0 0=R=3 3
4 3
5 3
6 3
7 3
8 3
etc...
```

7 Recursion

Abstract

When a method or function makes a call to the same method or function, it is called recursion. In some cases a complex problem can be solved by recursively solving similar subproblems. The graded assignment of this module uses recursion to solve the problem of finding the longest track in a labyrinth in an elegant way.

Goals

- Understand the notion of recursion
- Recognize situations in which to apply recursion

Assignments

1. Numbers

Print the numbers 10 to 1 in a recursive way. Next, adapt the recursive function to print the numbers 1 to 10. Using this knowledge, write a combined recursive function that prints the numbers 10 to 1 followed by the numbers 1 to 10. One method should suffice to solve this problem.

What happens when the base case is removed?

Example The output of the final program should look like this:

```
10 9 8 7 6 5 4 3 2 1 1 2 3 4 5 6 7 8 9 10
```

2. Reversed Number

Take a large number as input and print it in reverse on the output. Zeros at the end of a number should also be visible at the start of the reversed number.

Example

```
123456780
087654321
```

Graded Assignment

3. LongestPath

One of the uses of recursive programming is searching the best solution by trying all possible solutions. Examples include searching for a solution of a Rubik's Cube or the best possible next move in a game of Othello. The goal of this assignment is to find the longest path that can be traced through a labyrinth without visiting the same place twice.
A number of labyrinths can be found on Canvas. This is an example of such a file:

```
1 1=30 22=31 23
31 4
31 3
31 2
31 1
31 0
30 0
29 0
```

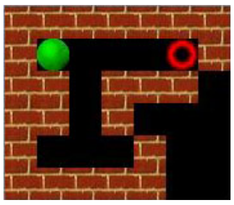
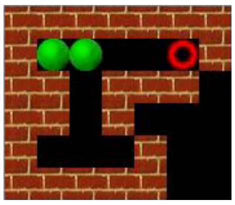
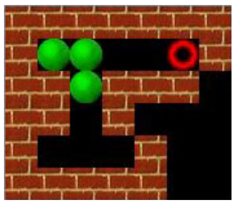
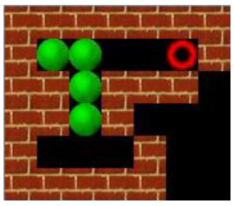
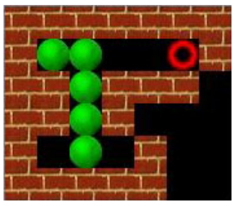
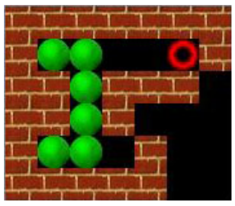
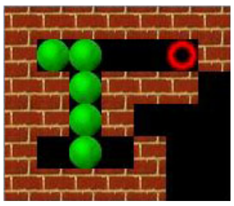
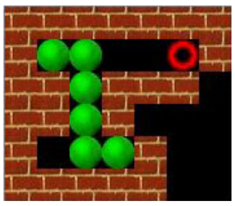
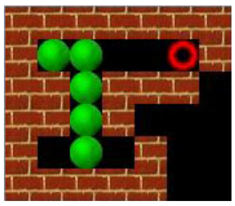
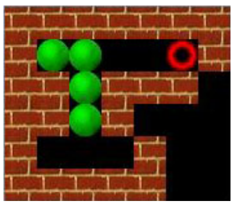
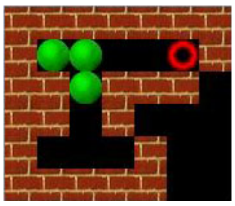
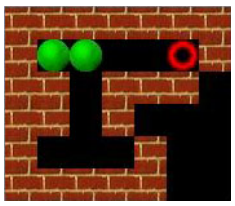
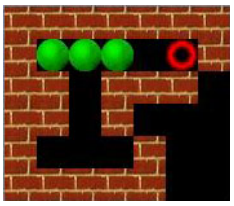
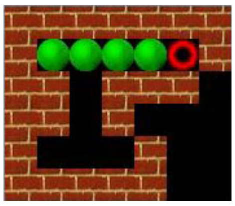
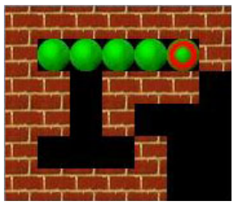
These files contain three elements, separated by `'='`:

- the starting coordinate of the path,
- the final coordinate of the path, and
- the coordinates of the pieces of wall, separated by an end-of-line (`'\n'`).

The process of searching for the longest path should be visible on screen. This means that every change in the current path should be shown on screen. To find the longest path, use the following steps:

- show the path that has been found so far,
- try walking in every direction (recursion),
- retrace a step (this should be visible).

The following example shows the tracing of all possible paths in a small labyrinth. It uses the steps described above. It tries all directions in the following order: west, south, east, north.


1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

To create a labyrinth, use the LabyrinthUserInterface, which can be obtained in the following way:

```
LabyrinthUserInterface ui;
ui = UserInterfaceFactory.getLabyrinthUI(width, height);
```

The default width and height are 32 and 24 respectively.

The LabyrinthUserInterface has the same methods as the OthelloReplayUserInterface and those used in the Replay assignment.

8 Bonus

Abstract

This module contains the bonus assignment. Students who complete the bonus assignment suffuciently will be given one tenth of the bonus grade on top of their lab grade.

☛ **Warning**

As this is a bonus assignment, there is no time reserved for this assignment during the lab sessions. This assignment is solely meant for those students who have finished all other assignments. Students are only allowed to start on this assignment if all other assignments have been submitted.

Graded Assignment

1. Life

The Game of Life was invented by J.H. Conway. Two publications in the "Scientific American" by Martin Gardner saw the game introduced to the public. Life is played on a board of n x n squares, representing a population of dead and living cells. A living cell can either die or continue to live, based on a set of rules. A dead cell can either become alive again, or remain dead. Every cell has eight neighbours, except the cells on the edge of the board:

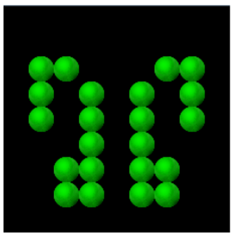
1	2	3
4	*	5
6	7	8

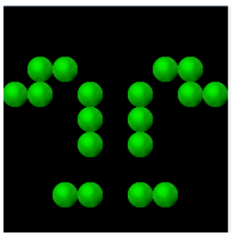
The set of rules determining the fate of a cell:

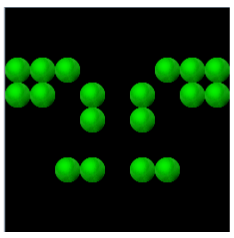
1. X is currently dead: If X has exactly three living neighbours, X becomes alive again. In all other cases, X remains dead.

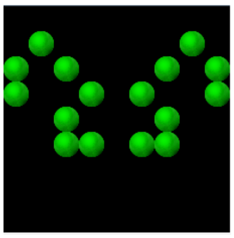
2. X is currenly alive: If X has zero or one neighbour(s), X dies of loneliness. If X has two or three living neighbours, X remains alive. In all other cases X dies of a shortage of living space.

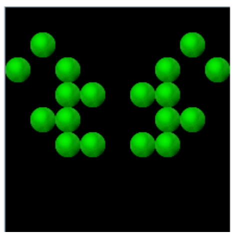
An example using a 9 x 9 board:

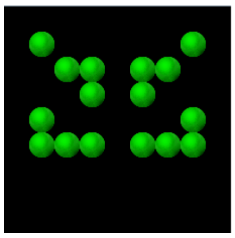

1

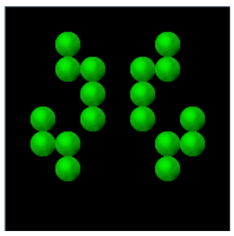

2

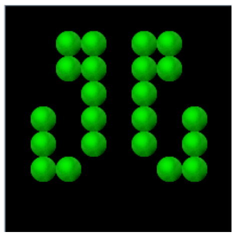

3

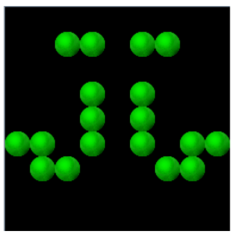

4

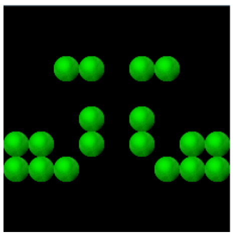

5

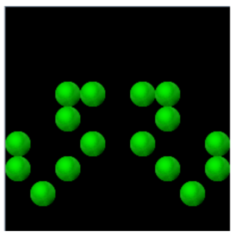

6

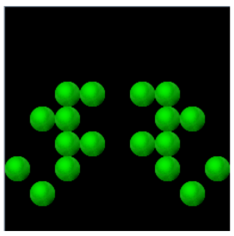

7

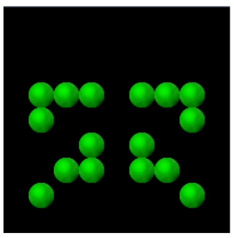

8

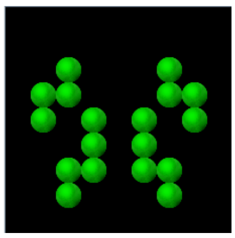

9


10


11


12


13


14

It is possible for a figure to die (an empty board) or become an oscillator, i.e. generation n = generation n + p, for any n above a certain value. If the period equals 1 (p=1), it is called a still figure.
Write a program that takes a starting configuration from a file and generates generations as long as the figure has not died, become an oscillator with a certain p, or exceeds the maximum number of generations. When the program terminates, print a message stating why the program has terminated and if the figure has become an oscillator, its period. If the period of the oscillator is 1, the message should read “Still figure” instead of “Oscillator”. The input files on Canvas have the following structure:

- On the first line, the maximum number of generations, ranging from 1 to 100.
- On the second line, the largest period for which the figure should be tested to oscillate, ranging from 2 to 15.
- After this, a starting configuration for a 9 x 9 board, made up from 9 line of 9 characters. A living cell is represented by an ‘x’, a dead cell is represented by ‘ ’

This assignment uses the LifeUserInterface. This works exactly like the ReplayUserInterface. The only difference are the constants declaring the images: LifeUserInterface.DEAD and LifeUserInterface.ALIVE. The LifeUserInterface can be intialized with UserInterfaceFactory.getLifeUI(width,height).