# Object-Oriented Programming with Java, part II »

Authors: Arto Vihavainen, Matti Luukkainen
Translators: Simone Romeo, Kenny Heinonen

# 54. SOME USEFUL TECHNIQUES

Before the course comes to its end, we can still have a look at some useful particular features of Java.

## 54.1 REGULAR EXPRESSIONS

A regular expression is a compact form to define a string. Regular expressions are often used to check the validity of strings. Let's have a look at an exercise where we have to check whether the student number given by the user is written in the valid form or not. Finnish student numbers start with the string "01" which is followed by seven numerical digits from 0 to 9.

We can check the validity of a student number parsing its each character with the help of the method `charAt`. Another way would be checking whether the first character is "0", and using the method `Integer.parseInt` to translate the string into a number. Then, we could check whether that number is smaller than 20000000.

Validity check with the help of regular expressions requires we define a suitable regular expression. Then we can use the `matches` method of the class `String`, which checks whether the string matches with the regular expression in parameter. In the case of a student number, a suitable regular expression is `"01[0-9]{7}"`, and you can check the validity of what the user has input in the following way:

```
System.out.print("Give student number: ");
String num = reader.nextLine();

if (num.matches("01[0-9]{7}")) {
    System.out.println("The form is valid.");
} else {
    System.out.println("The form is not valid.");
}
```

Next, we can go through the most commonly used regular expressions.

### 54.1.1 VERTICAL BAR: LOGICAL OR

The vertical bar means that the parts of the regular expression are optional. For instance, the expression `00|111|0000` defines the strings `00`, `111` and `0000`. The method `matches` returns `true` if the string matches one of the alternatives defined.

```
    String string = "00";

    if(string.matches("00|111|0000")) {
        System.out.println("The string was matched by some of the alternatives");
    } else {
        System.out.println("The string not was matched by any of the alternatives");
    }
```

```
The string was matched by some of the alternatives
```

The regular expression `00|111|0000` requires the exactly same form of the string: its functionality is not like "*contains*".

```
    String string = "1111";

    if(string.matches("00|111|0000")) {
        System.out.println("The string was matched by some of the alternatives");
    } else {
        System.out.println("The string not was matched by any of the alternatives");
    }
```

```
The string not was matched by any of the alternatives
```

### 54.1.2 ROUND BRACKETS: A DELIMITED PART OF THE STRING

With the help of round brackets it is possible to define what part of the regular expression is affected by the symbols. If we want to allow for the alternatives `00000` and `00001`, we can define it with the help of a vertical bar: `00000|00001`. Thanks to round brakers we can delimit the choice to only a part of the string. The expression `0000(0|1)` defines the strings `00000` and `00001`.

Accordingly, the regular expression `look(|s|ed)` defines the basic form of the verb to look (look), the third person (looks), and the past (looked).

```
System.out.print("Write a form of the verb to look: ");
String word = reader.nextLine();

if (word.matches("look(|s|ed|ing|er)")) {
    System.out.println("Well done!");
} else {
    System.out.println("Check again the form.");
}
```

## 54.1.3 REPETITIONS

We often want to know whether a substring repeats within another string. In regular expressions, we can use repetition symbols:

- The symbol `*` stands for a repetition from 0 to n times, for instance

```
String string = "trolololololo";

if(string.matches("trolo(lo)*")) {
    System.out.println("The form is right.");
} else {
    System.out.println("The form is wrong.");
}
```

```
The form is right.
```

- The symbol `+` stands for a repetition from 1 to n times, for instance

```
String string = "trolololololo";

if(characterString.matches("tro(lo)+")) {
    System.out.println("The form is right.");
} else {
    System.out.println("The form is wrong.");
}
```

```
The form is right.
```

```
String characterString = "nänänänänänänänä Bätmään!";

if(characterString.matches("(nä)+ Bätmään!")) {
    System.out.println("The form is right.");
} else {
    System.out.println("The form is wrong.");
}
```

```
The form is right.
```

- The symbol `?` stands for a repetition of 0 or 1 time, for instance

```
String string = "You have accidentally the whole name";

if(characterString.matches("You have accidentally (deleted )?the whole name")) {
    System.out.println("The form is right.");
} else {
    System.out.println("The form is wrong.");
}
```

```
The form is right.
```

- The symbol `{a}` stands for a repetition of `a` times, for instance

```
String string = "1010";

if(string.matches("(10){2}")) {
    System.out.println("The form is right.");
} else {
    System.out.println("The form is wrong.");
}
```

```
The form is right.
```

- The symbol `{a,b}` stands for a repetition from `a` to `b` times, for instance

```
String string = "1";

if(string.matches("1{2,4}")) {
    System.out.println("The form is right.");
} else {
    System.out.println("The form is wrong.");
}
```

```
The form is wrong.
```

- The symbol `{a,}` stands for a repetition from `a` to n times, for instance

  ```
  String string = "11111";

  if(string.matches("1{2,}")) {
      System.out.println("The form is right.");
  } else {
      System.out.println("The form is wrong.");
  }
  ```

  ```
  The form is right.
  ```

You can also use various different repetition symbols within one regular expression. For instance, the regular expression `5{3}(1|0)*5{3}` defines strings which start and end with three fives. In between there can be an indefinite number of 1 and 0.

### 54.1.4 SQUARE BRACKETS: CHARACTER GROUPS

With the help of square brackets we can quickly define groups of characters. The characters are written inside the brackets, and we can define an interval with the help of a hyphen (-). For instance, `[145]` means the same as `(1|4|5)`, whereas `[2-36-9]` means the same as `(2|3|6|7|8|9)`. Accordingly, `[a-c]*` defines a regular expression with a string made only of characters `a`, `b` and `c`.

#### Exercise 44: Regular Expressions

Let's train to use regular expressions. The exercises are done in the `Main` class of the default package .

#### EXERCISE 44.1: WEEK DAYS

Create the method `public static boolean isAWeekDay(String string)` in the class `Main`, using regular expressions. The method returns `true` if its parameter string is the abbreviation of a week day (mon, tue, wed, thu, fri, sat or sun).

The following is a sample print output of the method:

```
Give a string: tue
The form is fine.
```

```
Give a string: abc
The form is wrong.
```

#### EXERCISE 44.2: VOWEL INSPECTION

Create the method `public static boolean allVowels(String string)` in the class `Main`, which makes use of regular expressions and checks whether the String argument contains only vowel characters.

The following is a sample print output of the method:

```
Give a string: aie
The form is fine.
```

```
Give a string: ane
The form is wrong.
```

### EXERCISE 44.3: CLOCK TIME

Regual expressions suit in particular situations. In some cases, the expressions become too complicate and it may be useful to check the "regularity" of a string in a different way, or it may be appropriate to use regular expressions to manage only a part of the inspection.

Create the method `public static boolean clockTime(String string)` in the class `Main`, which makes use of regular expressions and checks whether the String argument conforms with the clock time `hh:mm:ss` (two-digit hours, minutes, and seconds). In this exercise you can use whatever tecnique, in addition to regular expressions.

The following is a sample print output of the method:

```
Give a string: 17:23:05
The form is fine.
```

```
Give a string: abc
The form is wrong.
```

```
Give a string: 33:33:33
The form is wrong.
```

## 54.2 ENUM: ENUMERATED TYPE

Previously, we implemented the class `Card` which represented a playing card:

```
public class Card {

    public static final int DIAMONDS = 0;
    public static final int SPADES = 1;
    public static final int CLUBS = 2;
    public static final int HEARTS = 3;

    private int value;
    private int suit;

    public Card(int value, int suit) {
        this.value = value;
        this.suit = suit;
    }

    @Override
    public String toString() {
        return suitName() + " "+value;
    }

    private String suitName() {
        if (suit == 0) {
            return "DIAMONDS";
        } else if (suit == 1) {
            return  "SPADES";
        } else if (suit == 2) {
            return "CLUBS";
        }
        return "HEARTS";
    }

    public int getSuit() {
        return suit;
    }
}
```

The card suit is stored as object variable integer. Indicating the suit is made easier by constants which help the legibility. The constants which represent cards and suits are used in the following way:

```
public static void main(String[] args) {
        Card card = new Card(10, Card.HEARTS);

        System.out.println(card);

        if (card.getSuit() == Card.CLUBS) {
            System.out.println("It's clubs");
        } else {
            System.out.println("It's not clubs");
        }

}
```

Representing the suit as a number is a bad solution, because the following absurd ways to use cards are possible:

```
        Card absurdCard = new Card(10, 55);

        System.out.println(absurdCard);

        if (absurdCard.getSuit() == 34) {
            System.out.println("The card's suit is 34");
        } else {
            System.out.println("The card's suit is not 34");
        }

        int suitPower2 = absurdCard.getSuit() * absurdCard.getSuit();

        System.out.println("The card's suit raised to the power of two is " + suitPower2);
```

If we already know the possible values of our variables, we can use a `enum` class to represent them: an enumerated type. In addition to being classes and interfaces, enumerated types are also a class type of their own. Enumerated types are defined with the keyword `enum`. For instance the following `Suit` enum class defines four values: `DIAMONDS`, `SPADES`, `CLUBS` and `HEARTS`.

```
public enum Suit {
    DIAMONDS, SPADES, CLUBS, HEARTS
}
```

In its most basic from, `enum` lists its constant values divided by a comma. Enum constants are usually written in capital letters.

Enums are usually created in their own file, in the same way as classes and interfaces. In Netbeans, you can create a enum by clicking to *new/other/java/java enum* on your project name.

The following `Card` class is represented with the help of enum:

```
public class Card {

    private int value;
    private Suit suit;

    public Card(int value, Suit suit) {
        this.value = value;
        this.suit = suit;
    }

    @Override
    public String toString() {
        return suit + " "+value;
    }

    public Suit getSuit() {
        return suit;
    }

    public int getValue() {
        return value;
    }
}
```

The new version of the card is used in the following way:

```
public class Main {

    public static void main(String[] args) {
        Card first = new Card(10, Suit.HEARTS);

        System.out.println(first);

        if (first.getSuit() == Suit.CLUBS) {
            System.out.println("It's clubs");
        } else {
            System.out.println("It's not clubs");
        }


    }
}
```

Prints:

```
HEARTS 10
It's not clubs
```

We notice that enum names are printed smoothly! Because card suits' type is `Suit`, absurd practices like the one above -- raising a suit to the power of two -- do not work. Oracle has a tutorial for `enum` type at http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html.

## 54.3 ITERATOR

Let's have a look at the following class `Hand` which represents the cards a player has in his hand in a card game:

```
public class Hand {
    private ArrayList<Card> cards;

    public Hand() {
        cards = new ArrayList<Hand>();
    }

    public void add(Card card){
        cards.add(card);
    }

    public void print(){
        for (Card card : cards) {
            System.out.println( card );
        }
    }
}
```

The `print` method prints each card in the hand by using a "for each" statement. ArrayList and other "object containers" which implement the *Collection* interface indirectly implement the interface *Iterable*. Objects which implement *Iterable* can be parses, or better "iterated", with statements such as for *each*.

Object containers can also be iterated using a so called *iterator*, that is an object, which was thought to parse a particular object collection. Below, there is a version of an iterator used to print cards:

```
public void print() {
    Iterator<Card> iterator = cards.iterator();

    while ( iterator.hasNext() ){
        System.out.println( iterator.next() );
    }
}
```

The iterator is taken from the ArrayList `cards`. The iterator is like a finger, which always points out a specific object of the list, from the first to the second, to the third, and so on, till the finger has gone through each object.

The iterator provides a couple of methods. The method `hasNext()` asks whether there are still objects to be iterated. If there are, we can retrieve the following object using the method `next()`. The method returns the following object in the collection, and makes the iterator -- the "finger" -- point out the following object.

The object reference returned by the Iterator's next() method can be stored into a variable, of course; in fact, we could modify the method `print` in the following way:

```
public void print(){
    Iterator<Card> iterator = cards.iterator();

    while ( iterator.hasNext() ){
        Card nextCard = iterator.next();
        System.out.println( nextCard );
    }
}
```

We can create a method to delete the cards which are smaller than a specific value:

```
public class Hand {
    // ...

    public void deleteWorst(int value) {
        for (Card card : cards) {
            if ( card.getValue() < value ) {
                cards.remove(card);
            }
        }
    }
}
```

We notice that running the method causes a strange error:

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.AbstractList$Itr.checkForComodification(AbstractList.java:372)
        at java.util.AbstractList$Itr.next(AbstractList.java:343)
        at Hand.deleteWorst(Hand.java:26)
        at Main.main(Main.java:20)
Java Result: 1
```

The reason is that you can't delete objects from a list while you are parsing it: the for each statement "gets all worked up".

If we want to delete a part of the objects while we parse our list, we have to use an iterator. If we call the `remove` method of our iterator object, we can neatly delete the value which was returned by the iterator with its previous `next` method call. The following method works fine:

```
public class Hand {
    // ...

    public void deleteWorst(int value) {
        Iterator<Card> iterator = cards.iterator();

        while (iterator.hasNext()) {
            if (iterator.next().getValue() < value) {
                // delete the object returned by the iterator with its previous met
                iterator.remove();
            }
        }
    }
}
```

## Exercise 45: Enum and Iterator

Let's create a program to manage the staff personnel of a small business.

### EXERCISE 45.1: EDUCATION

Create the enumerated type, or enum, `Education` in the package `personnel`. The enum has the titles `D` (doctor), `M` (master), `B` (bachelor), `GRAD` (graduate).

### EXERCISE 45.2: PERSON

Create the class `Person` in `personnel`. Person is assigned a name and an education title as constructor parameters. Person has also a method to communicate their education, `public Education getEducation()`, as well as a `toString` method which returns the person information following the example below.

```
    Person arto = new Person("Arto", Education.D);
    System.out.println(arto);
```

```
Arto, D
```

## EXERCISE 45.3: EMPLOYEES

Create the class `Employees` in `personnel`. An Employees object contains a list of Person objects. The class has a parameterless constructor plus the following methods:

- `public void add(Person person)` adds the parameter person to the employees
- `public void add(List<Person> persons)` adds the parameter list of people to the employees
- `public void print()` prints all the employees
- `public void print(Education education)` prints all the employees, who have the same education as the one specified as parameter

**ATTENTION:** The `Print` method of the class `Employees` have to be implemented using an iterator!

## EXERCISE 45.4: FIRING

Create the method `public void fire(Education education)` in the class `Employees`. The method deletes all the employees whose education is the same as the method argument.

**ATTENTION:** implement the method using an iterator!

Below, you find an example of the class usage:

```
public class Main {

    public static void main(String[] args) {
        Employees university = new Employees();
        university.add(new Person("Matti", Education.D));
        university.add(new Person("Pekka", Education.GRAD));
        university.add(new Person("Arto", Education.D));

        university.print();

        university.fire(Education.GRAD);

        System.out.println("==");
        university.print();
}
```

Prints:

```
Matti, D
Pekka, GRAD
Arto, D
==
Matti, D
Arto, D
```

## 54.4 LOOPS AND CONTINUE

In addition to the `break` statement, loops have also got the `continue` statement, which allows you to skip to the following loop stage.

```
List<String> names = Arrays.asList("Matti", "Pekka", "Arto");

for(String name: names) {
    if (name.equals("Arto")) {
        continue;
    }

    System.out.println(name);
}
```

```
Matti
Pekka
```

The `continue` statement is used especially when we know the iterable variables have got values with which we do not want to handle at all. The classic manner of approach would be using an if statement, but the `continue` statement allows for another approach to handle with the values, which avoids indentations and possibly helps readability. Below, you find two examples, where we go through the numbers of a list. If the number is smaller than 5 and contains 100, or if it contains 40, it is not printed; otherwise it is.

```
List<Integer> values = Arrays.asList(1, 3, 11, 6, 120);

for(int num: values) {
    if (num > 4 && num % 100 != 0 && num % 40 != 0) {
        System.out.println(num);
    }
}
```

```
    for(int num: values) {
        if (num < 5) {
            continue;
        }

        if (num % 100 == 0) {
            continue;
        }

        if (num % 40 == 0) {
            continue;
        }

        System.out.println(num);
    }
```

```
11
6
11
6
```

## 54.5 MORE ABOUT ENUMS

Next, we create enums which contain object variables and implement an interface.

### 54.5.1 ENUMERATED TYPE CONSTRUCTOR PARAMETERS

Enumerated types can contain object variables. Object variable values have to be set up in the constructor of the class defined by enumerated type. Enum-type classes cannot have `public` constructors.

```
public enum Colour {
    RED("red"), // the constructor parameters are defined as constant values when th
    GREEN("green"),
    BLUE("blue");

    private String name; // object variable

    private Colour(String name) { // constructor
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

The enumerated value `Colour` can be used in the following way:

```
    System.out.println(Colour.GREEN.getName());
```

```
green
```

## Exercise 46: Film Reference

Recently, in October 2006 after arriving to Finland, Netflix promised one million dollars to the person or group of people who developed a program, which would be 10% better than their own program. The challenge was met September 2009 (http://www.netflixprize.com/).

With this exercise, we create a program to recommend films. Below, you see how it should work:

```
    EvaluationRegister ratings = new EvaluationRegister();

    Film goneWithTheWind = new Film("Gone with the Wind");
    Film theBridgesOfMadisonCounty = new Film("The Bridges of Madison County");
    Film eraserhead = new Film("Eraserhead");

    Person matti = new Person("Matti");
    Person pekka = new Person("Pekka");
    Person mikke = new Person("Mikke");
    Person thomas = new Person("Thomas");

    ratings.addRating(matti, goneWithTheWind, Rating.BAD);
    ratings.addRating(matti, theBridgesOfMadisonCounty, Rating.GOOD);
    ratings.addRating(matti, eraserhead, Rating.FINE);

    ratings.addRating(pekka, goneWithTheWind, Rating.FINE);
    ratings.addRating(pekka, theBridgesOfMadisonCounty, Rating.BAD);
    ratings.addRating(pekka, eraserhead, Rating.MEDIOCRE);

    ratings.addRating(mikke, eraserhead, Rating.GOOD);


    Reference reference = new Reference(votes);
    System.out.println(thomas + "'s recommendation: " +
            reference.recommendFilm(thomas));
    System.out.println(mikke + "'s recommendation: " +
            reference.recommendFilm(mikke));
```

```
Thomas's recommendation: The Bridges of Madison County
Mikke's recommendation: Gone with the Wind
```

The program is able to recommend films both according to their common appraisal and according to the ratings given by a specific person. Let's start to build our program.

## EXERCISE 46.1: PERSON AND FILM

Create the package `reference.domain`, and there you add the classes `Person` and `Film`. Both classes have a public constructor `public Class(String name)`, as well as the method `public String getName()`, which returns the name received with the argument.

```
Person p = new Person("Pekka");
Film f = new Film("Eraserhead");

System.out.println(p.getName() + " and " + f.getName());
```

```
Pekka and Eraserhead
```

Also add the method `public String toString()` which returns the name received with the argument, as well as the method `equals` and `hashCode`.

Override `equals` so that the equivalence is checked according to the object variable `name`. Look at the example in section 45.1. In Section 45.2 there are guidelines to override `hashCode` methods. At least, you'd better generate the HashCode automatically, following the instructions at the end of the section:

*NetBeans allows for the automatic creation of the `equals` and `hashCode` methods. From the menu Source -> Insert Code, you can choose equals() and hashCode(). After this, NetBeans asks which object variables the methods shall use.*

**Attention:** to help finding mistakes, you may want to implement toString methods to Person and Film, but the tests do not require them.

## EXERCISE 46.2: RATING

Create the enumerated type `Rating` in `reference.domain`. The enum class `Rating` has a public method `public int getValue()`, which returns the value of the rating. The value names and their grades have to be the following:

| Rating | Value |
|---|---|
| BAD | -5 |
| MEDIOCRE | -3 |
| NOT_WATCHED | 0 |
| NEUTRAL | 1 |
| FINE | 3 |
| GOOD | 5 |

The class could be used in the following way:

```
Rating given = Rating.GOOD;
System.out.println("Rating " + given + ", value " + given.getValue());
```

```
    given = Rating.NEUTRAL;
    System.out.println("Rating " + given + ", value " + given.getValue());
```

```
Rating GOOD, value 5
Rating NEUTRAL, value 1
```

### EXERCISE 46.3: RATINGREGISTER, PART 1

Let's get started with the implementation necessary to store the ratings.

Create the class `RatingRegister` in the package `reference`; the class has the constructor `public RatingRegister()`, as well as the following methods:

- `public void addRating(Film film, Rating rating)` adds a new rating to the parameter film. The same film can have various same ratings.
- `public List<Rating> getRatings(Film film)` returns a list of the ratings which were added in connection to a film.
- `public Map<Film, List<Rating>> filmRatings()` returns a map whose keys are the evaluated films. Each film is associated to a list containing the ratings for that film.

Test the methods with the following source code:

```
    Film theBridgesOfMadisonCounty = new Film("The Bridges of Madison County");
    Film eraserhead = new Film("Eraserhead");

    RatingRegister reg = new RatingRegister();
    reg.addRating(eraserhead, Rating.BAD);
    reg.addRating(eraserhead, Rating.BAD);
    reg.addRating(eraserhead, Rating.GOOD);

    reg.addRating(theBridgesOfMadisonCounty, Rating.GOOD);
    reg.addRating(theBridgesOfMadisonCounty, Rating.FINE);

    System.out.println("All ratings: " + reg.filmRatings());
    System.out.println("Ratings for Eraserhead: " + reg.getRatings(eraserhead));
```

```
All ratings: {The Bridges of Madison County=[GOOD, FINE], Eraserhead=[BAD, BAD, 
Ratings for Eraserhead: [BAD, BAD, GOOD]
```

### EXERCISE 46.4: RATINGREGISTER, PART 2

Let's make possible to add personal ratings.

Add the following methods to the class `RatingRegister`:

- `public void addRating(Person person, Film film, Rating rating)` adds the rating of a specific film to the parameter person. The same person can recommend a specific film only once. The person rating has also to be added to the ratings connected to all the films.

- `public Rating getRating(Person person, Film film)` returns the rating the paramater person has assigned to the parameter film. If the person hasn't evaluated such film, the method returns `Rating.NOT_WATCHED`.
- `public Map<Film, Rating> getPersonalRatings(Person person)` returns a HashMap which contains the person's ratings. The HashMap keys are the evaluated films, and their values are the ratings of these films.
- `public List<Person> reviewers()` returns a list of the people who have evaluate the films.

People's ratings should be stored into a HashMap, and the people should act as keys. The values of the HashMap is another HashMap, whose keys are films and whose values are ratings.

Test your improved `RatingRegister` with the following source code:

```
RatingRegister ratings = new RatingRegister();

Film goneWithTheWind = new Film("Gone with the Wind");
Film eraserhead = new Film("Eraserhead");

Person matti = new Person("Matti");
Person pekka = new Person("Pekka");

ratings.addRating(matti, goneWithTheWind, Rating.BAD);
ratings.addRating(matti, eraserhead, Rating.FINE);

ratings.addRating(pekka, goneWithTheWind, Rating.GOOD);
ratings.addRating(pekka, eraserhead, Rating.GOOD);

System.out.println("Ratings for Eraserhead: " + ratings.getRatings(eraserhead));
System.out.println("Matti's ratings: " + ratings.getPersonalRatings(matti));
System.out.println("Reviewers: " + ratings.reviewers());
```

```
Ratings for Eraserhead: [FINE, GOOD]
Matti's ratings: {Gone with the Wind=BAD, Eraserhead=FINE}
Reviewers: [Pekka, Matti]
```

Next, we create a couple of helping classes to help evaluation.

## EXERCISE 46.5: PERSONCOMPARATOR

Create the class `PersonComparator` in the package `reference.comparator`. The class `PersonComparator` has to implement the interface `Comparator<Person>`, and it has to have the constructor `public PersonComparator(Map<Person, Integer> peopleIdentities)`. The class `PersonComparator` is used later on to sort people according to their number.

Test the class with the following source code:

```
Person matti = new Person("Matti");
Person pekka = new Person("Pekka");
```

```
    Person mikke = new Person("Mikke");
    Person thomas = new Person("Thomas");

    Map<Person, Integer> peopleIdentities = new HashMap<Person, Integer>();
    peopleIdentities.put(matti, 42);
    peopleIdentities.put(pekka, 134);
    peopleIdentities.put(mikke, 8);
    peopleIdentities.put(thomas, 82);

    List<Person> ppl = Arrays.asList(matti, pekka, mikke, thomas);
    System.out.println("People before sorting: " + ppl);

    Collections.sort(ppl, new PersonComparator(peopleIdentities));
    System.out.println("People after sorting: " + ppl);
```

```
People before sorting: [Matti, Pekka, Mikke, Thomas]
People after sorting: [Pekka, Thomas, Matti, Mikke]
```

## EXERCISE 46.6: FILMCOMPARATOR

Create the class `FilmComparator` in the package `reference.comparator`. The class `FilmComparator` has to implement the interface `Comparator<Film>`, and it has to have the constructor `public FilmComparator(Map<Film, List<Rating>> ratings)`. The class `FilmComparator` will be used later on to sort films according to their ratings.

The class FilmComparator has to allow for film sorting according to the average of the rating values they have received. The films with the greatest average should be placed first, and the ones with the smallest average should be the last.

Test the class with the following source code:

```
    RatingRegister ratings = new RatingRegister();

    Film goneWithTheWind = new Film("Gone with the Wind");
    Film theBridgesOfMadisonCounty = new Film("The Bridges of Madison County");
    Film eraserhead = new Film("Eraserhead");

    Person matti = new Person("Matti");
    Person pekka = new Person("Pekka");
    Person mikke = new Person("Mikke");

    ratings.addRating(matti, goneWithTheWind, Rating.BAD);
    ratings.addRating(matti, theBridgesOfMadisonCounty, Rating.GOOD);
    ratings.addRating(matti, eraserhead, Rating.FINE);

    ratings.addRating(pekka, goneWithTheWind, Rating.FINE);
    ratings.addRating(pekka, theBridgesOfMadisonCounty, Rating.BAD);
    ratings.addRating(pekka, eraserhead, Rating.MEDIOCRE);

    ratings.addRating(mikke, eraserhead, Rating.BAD);
    Map<Film, List<Rating>> filmRatings = ratings.filmRatings();

    List<Film> films = Arrays.asList(goneWithTheWind, theBridgesOfMadisonCounty,
    System.out.println("The films before sorting: " + films);

    Collections.sort(films, new FilmComparator(filmRatings));
    System.out.println("The films after sorting: " + films);
```

```
The films before sorting: [Gone with the Wind, The Bridges of Madison County, Era
The films after sorting: [The Bridges of Madison County, Gone with the Wind, Eras
```

## EXERCISE 46.7: REFERENCE, PART 1

Implement the class `Reference` in the package `reference`. The class `Reference` receives a `RatingRegister` object as constructor parameter. Reference uses the ratings in the rating register to elaborate a recommendation.

Implement the method `public Film recommendFilm(Person person)`, which implements films to people. Hint: you need three things to find out the most suitable film. These are at least the class `FilmComparator` which you created earlier on; the method `public Map<Film, List<Rating>> filmRatings()` of the class `RatingRegister`; and a list of the existing films.

Test your program with the following source code:

```
    RatingRegister ratings = new RatingRegister();

    Film goneWithTheWind = new Film("Gone with the Wind");
    Film theBridgesOfMadisonCounty = new Film("The Bridges of Madison County");
    Film eraserhead = new Film("Eraserhead");

    Person matti = new Person("Matti");
    Person pekka = new Person("Pekka");
    Person mikke = new Person("Mikke");

    ratings.addRating(matti, goneWithTheWind, Rating.BAD);
    ratings.addRating(matti, theBridgesOfMadisonCounty, Rating.GOOD);
    ratings.addRating(matti, eraserhead, Rating.FINE);

    ratings.addRating(pekka, goneWithTheWind, Rating.FINE);
    ratings.addRating(pekka, theBridgesOfMadisonCounty, Rating.BAD);
    ratings.addRating(pekka, eraserhead, Rating.MEDIOCRE);

    Reference ref = new Reference(ratings);
    Film recommended = ref.recommendFilm(mikke);
    System.out.println("The film recommended to Michael is: " + recommended);
```

```
The film recommended to Michael is: The Bridges of Madison County
```

Now, our first part works fine exclusively for people who have never evaluated any movie. In such cases, we can't say anything about their film tastes, and the best choice is recommending them the film which has received the hightest average among the ratings.

## EXERCISE 46.8: REFERENCE, PART 2

*Attention! The exercise is challenging. First you should do the previous exercises and coming back to this one later. You can return the sequence of exercises in TMC; even though you don't get the point for this part, you'd get the points for the perious ones, as it is with all the exercises.*

Unfortunately, the error diagnostics of this part is not similar to the previous parts.

If people have added their own preferences to the reference service, we know something about their film tastes. Let's extend the functionality of our reference to create a personal recommendation if the person has evaluated films. The functionality implemented in the previous part has to be preserved: if a person hasn't evaluated any film, we recommend them a film according to film ratings.

Personal recommendations are based on the similarity between the person ratings and other people's ratings. Let's reason about it with the help of the following table; in the first line on the top there are films, and the people who have rated are on the left. The brackets describe the ratings given.

| **Person \ Film** | Gone with the Wind | The Bridges of Madison County | Eraserhead | Blues Brothers |
|---|---|---|---|---|
| Matti | BAD (-5) | GOOD (5) | FINE (3) | - |
| Pekka | FINE (3) | - | BAD (-5) | MEDIOCRE (-3) |
| Mikael | - | - | BAD (-5) | - |
| Thomas | - | GOOD (5) | - | GOOD (5) |

If we want to find the suitable film for Mikael, we can explore the similarity between Mikael's and other people's preferences. The similarity is calculated based on the ratings: as the sum of the products of the ratings for the films watched by both. For instance, Mikael and Thomas's similarity is 0, because they haven't watched the same films.

If we calculate Mikael and Pekka's similarity, we find out that the sum of the products of the films they have in common is 25. Mikael and Pekka have both watched only one film, and they have both given it the grade bad (-5).

```
-5 * -5 = 25
```

Mikael and Matti's similarity is -15. Mikael and Matti have also watched only one same film. Mikael gave the grade bad (-5) to the film, whereas Matti gave it the grade fine (3).

```
-5 * 3 = -15
```

Based on that Mikael can be recommended films according to Pekka's taste: the recommendation is Gone with the Wind.

On the other hand, if we want to find a suitable film for Matti, we have to find the similarity between Matti and everyone else. Matti and Pekka have watched two same films. Matti gave Gone with the Wind the grade bad (-5), Pekka the grade fine (3). Matti gave fine (3) to Eraserhead, and Pekka gave bad (-5). Matti and Pekka's similarity is -30.

```
-5 * 3 + 3 * -5 = -30
```

Matti and Mikael's similarity is -15, which we know according to out previous calculations. Similarities are symmetrical.

Matti and Thomas have watched Gone with the Wind, and they both gave it the grade good (5). Matti and Thomas's similarity is 25, then.

```
5 * 5 = 25
```

Matti has to be recommended a film according to Thomas' taste: the recommendation will be the Blues Brothers.

Implement the recommendation mechanism described above. The method `recommendFilm` should return `null` in two cases: if you cannot find any film to recommend; if you find a, say, person1 whose film taste is appropriate to recommend films to, say, person2, but person1 has rated bad, mediocre, or neutral, all the films person2 hasn't watched, yet. The approach described above has to work also if the person hasn't added any rating.

Do not suggest films which have already been watched.

You can test your program with the following source code:

```
    RatingRegister ratings = new RatingRegister();

    Film goneWithTheWind = new Film("Gone with the Wind");
    Film theBridgesOfMadisonCounty = new Film("The Bridges of Madison County");
    Film eraserhead = new Film("Eraserhead");
    Film bluesBrothers = new Film("Blues Brothers");

    Person matti = new Person("Matti");
    Person pekka = new Person("Pekka");
    Person mikke = new Person("Mikael");
    Person thomas = new Person("Thomas");
    Person arto = new Person("Arto");

    ratings.addRating(matti, goneWithTheWind, Rating.BAD);
    ratings.addRating(matti, theBridgesOfMadisonCounty, Rating.GOOD);
    ratings.addRating(matti, eraserhead, Rating.FINE);

    ratings.addRating(pekka, goneWithTheWind, Rating.FINE);
    ratings.addRating(pekka, eraserhead, Rating.BAD);
    ratings.addRating(pekka, bluesBrothers, Rating.MEDIOCRE);

    ratings.addRating(mikke, eraserhead, Rating.BAD);

    ratings.addRating(thomas, bluesBrothers, Rating.GOOD);
    ratings.addRating(thomas, theBridgesOfMadisonCounty, Rating.GOOD);
    Reference ref = new Reference(ratings);
    System.out.println(thomas + " recommendation: " + ref.recommendFilm(thomas));
    System.out.println(mikke + " recommendation: " + ref.recommendFilm(mikke));
    System.out.println(matti + " recommendation: " + ref.recommendFilm(matti));
    System.out.println(arto + " recommendation: " + ref.recommendFilm(arto));
```

```
Thomas recommendation: Eraserhead
Mikael recommendation: Gone with the Wind
Matti recommendation: Blues Brothers
Arto recommendation: The Bridges of Madison County
```

Have we made one million bucks? Not yet, maybe. In the course Introduction to Artificial Intelligence and Machine Learning we learn more techniques to build learning programs.

## 54.6 VARIABLE NUMBER OF METHOD PARAMETERS

So far, we have been creating methods which had a clearly defined number of parameters. Java makes it possible to give an indefinite number of a specific type of parameters by placing an ellipsis after the parameter type. For instance the method `public int sum(int... values)` can be given as parameter as many integers (`int`) as the user wants. The parameter values can be handled as a table.

```
public int sum(int... values) {
    int sum = 0;
    for (int i = 0; i < values.length; i++) {
        sum += values[i];
    }
    return sum;
}
```

```
System.out.println(sum(3, 5, 7, 9));  // values = {3, 5, 7, 9}
System.out.println(sum(1, 2));        // values = {1, 2}
```

```
24
3
```

Note that the parameter definition above `int... values` depends on the fact that the method has a table-like variable called `values`.

A method can be assigned only one parameter which receives an indefinite number of values, and this must be the first parameter in the method definition. For instance:

```java
public void print(String... characterStrings, int times) // right!
public void print(int times, String... strings) // wrong!
```

An indefinite number of parameter values is used for instance when we want to create an interface which would not force the user to use a precise number of parameters. An alternative approach would be defining a list of that precise type as parameter. In this case, the objects can be assigned to the list before the method call, and the method can be called and given the list as parameter.

## Exercise 47: Flexible Filtering Criteria

In some exercises (see Library in Introduction to Programming, and Word Inspection in Advanced Programming), we have run into such situations where we had to filter out list objects according to particular criteria; for instance, in Word Inspection the methods `wordsContainingZ, wordsEndingInL, palindromes, wordsWhichContainAllVowels` all to the same same thing: they go through the file content one word after the other, and they make sure that the specific filtering criteria are satisfied, in which case they store the word. Because the method criteria are different, we haven't been able to get rid of this repetition, and the code of all those methods made wide use of copypaste.

With this exercise, we create a program to filter the lines of the books found on the Project Guttenberg pages. In the following example we analyze Dostojevski's Crime and Punishment. We want to have various different filtering criteria, and that it would be possible to filter according to different criteria combinations. The program structure should also allow for adding new criteria later on.

A suitable solution to the problem, is defining the filtering criteria as objects of their own which implement the interface `Criterion`. The interface definition is below:

```java
public interface Criterion {
    boolean complies(String line);
}
```

A filtering class which implements the interface:

```java
public class ContainsWord implements Criterion {

    String word;

    public ContainsWord(String word) {
        this.word = word;
    }

    @Override
    public boolean complies(String line) {
        return line.contains(word);
    }
}
```

The class objects are quite simple, in fact, and they only remember the word given as constructor parameter. The only method of the object can be asked whether the criterion complies to the parameter String; if so, this means that the object contains the word stored into the String object.

Together with the excercise body, you find the pre-made class `GutenbergReader`, which helps you to analyze book lines according to the filtering criteria given as parameter:

```
public class GutenbergReader {

    private List<String> lines;

    public GutenbergReader(String address) throws IllegalArgumentException {
        // the code which retrieves the book from the Internet
    }

    public List<String> linesWhichComplyWith(Criterion c){
        List<String> complyingLines = new ArrayList<String>();

        for (String line : lines) {
            if (c.complies(line)) {
                complyingLines.add(line);
            }
        }

        return complyingLines;
    }
}
```

With the following code, we print all the lines in Crime and Punishment which contain the word "beer":

```
public static void main(String[] args) {
    String address = "http://www.gutenberg.myebook.bg/2/5/5/2554/2554-8.txt";
    GutenbergReader book = new GutenbergReader(address);

    Criterion criterion = new ContainsWord("beer");

    for (String line : book.linesWhichComplyWith(criterion)) {
        System.out.println(line);
    }
}
```

## EXERCISE 47.1: ALL LINES

Create the class `AllLines` which implements `Criterion`, which accepts all the lines. This and the other classes of the exercise have to be implemented in the package `reader.criteria`.

```
public static void main(String[] args) {
    String address = "http://www.gutenberg.myebook.bg/2/5/5/2554/2554-8.txt";
    GutenbergReader book = new GutenbergReader(address);

    Criterion criterion = new AllLines();

    for (String line : book.linesWhichComplyWith(criterion)) {
        System.out.println(line);
    }
}
```

## EXERCISE 47.2: ENDS WITH QUESTION OR EXCLAMATION MARK

Implement class `EndsWithQuestionOrExclamationMark`, which implements the interface `Criterion` and accepts the lines whose last character is a question or an exclamation mark.

```
public static void main(String[] args) {
    String address = "http://www.gutenberg.myebook.bg/2/5/5/2554/2554-8.txt";
    GutenbergReader book = new GutenbergReader(address);

     Criterion criterion = new EndsWithQuestionOrExclamationMark();

    for (String line : book.linesWhichComplyWith(criterion)) {
        System.out.println(line);
    }
}
```

**Reminder:** you compare character in Java with the == operator:

```
String name = "pekka";

// ATTENTION: 'p' is a character, that is to say char p; differently, "p" is a S
if ( name.charAt(0) == 'p' ) {
    System.out.println("beginning with p");
} else {
    System.out.println("beginning with something else than p");
}
```

## EXERCISE 47.3: LENGTH AT LEAST

Implement the class `LengthAtLeast`, which implements the interface `Criterion` and accepts the lines whose length is equal or greater than the number received as constructor parameter.

```
public static void main(String[] args) {
    String address = "http://www.gutenberg.myebook.bg/2/5/5/2554/2554-8.txt";
    GutenbergReader book = new GutenbergReader(address);

     Criterion criterion = new LengthAtLeast(40);
    for (String line : book.linesWhichComplyWith(criterion)) {
        System.out.println(line);
    }
}
```

## EXERCISE 47.4: BOTH

Create the class `Both` which implements the interface `Criterion`. The objects of this class receive two objects as constructor parameter, both implementing the interface `Criterion`. `Both` objects accept the lines which comply with both the criteria received as constructor parameters. We print below all the lines which end with a question or an exclamation mark and that contain the word "beer".

```
public static void main(String[] args) {
    String address = "http://www.gutenberg.myebook.bg/2/5/5/2554/2554-8.txt";
    GutenbergReader book = new GutenbergReader(address);

    Criterion criterion = new Both(
                    new EndsWithQuestionOrExclamationMark(),
                    new ContainsWord("beer")
                );

    for (String line : book.linesWhichComplyWith(criterion)) {
        System.out.println(line);
    }
}
```

## EXERCISE 47.5: NEGATION

Create the class `Not` which implement the interface `Criterion` and accepts the lines, which don't comply with the criterion received as parameter. We print below the lines whose length is less than 10.

```
public static void main(String[] args) {
    String address = "http://www.gutenberg.myebook.bg/2/5/5/2554/2554-8.txt";
    GutenbergReader book = new GutenbergReader(address);

    Criterion criterion = new Not( new LengthAtLeast(10) );

    for (String line : book.linesWhichComplyWith(criterion)) {
        System.out.println(line);
    }
}
```

## EXERCISE 47.6: AT LEAST ONE

Create the class `AtLeastOne` which implements the interface `Criterion`. The objects of this class receive as parameter an optional amount of objects which implement the interface

`Criterion`; this means that the constructor receives a list of variable length as parameter. `AtLeastOne` objects accept the lines which comply with at least one of the criteria received as constructor parameter. We print below the lines which contain one ot the words "beer", "milk" or "oil".

```
public static void main(String[] args) {
    String address = "http://www.gutenberg.myebook.bg/2/5/5/2554/2554-8.txt";
    GutenbergReader book = new GutenbergReader(address);

    Criterion criterion =new AtLeastOne(
                    new ContainsWord("beer"),
                    new ContainsWord("milk"),
                    new ContainsWord("oil")
                );

    for (String line : book.linesWhichComplyWith(criterion)) {
        System.out.println(line);
    }
}
```

Note that the criteria can be combined as preferred. You find below a criterion which accepts the lines which have at least one of the words "beer", "milk" and "oil" and whose length is between 20-30 characters.

```
    Criterion words = new AtLeastOne(
                    new ContainsWord("beer"),
                    new ContainsWord("milk"),
                    new ContainsWord("oil")
                );

    Criterion rightLength = new Both(
                         new LengthAtLeast(20),
                         new Not( new LengthAtLeast(31))
                       );

    Criterion wanted = new Both(words, rightLength);
```

## 54.7 STRINGBUILDER

We have got accustomed to building strings in the following way:

```
public static void main(String[] args) {
    int[] t = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    System.out.println(build(t));
}
```

```
    public static String build(int[] t) {
        String str = "{";

        for (int i = 0; i < t.length; i++) {
            str += t[i];
            if (i != t.length - 1) {
                str += ", ";

            }
        }

        return str + "}";
    }
```

Result:

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

This works, but it could be more effective. As we remember, strings are *immutable* objects. The result of a String operation is always a new String object. This means that in the interphases of previous example 10 String objects were created. If the dimension of the input was bigger, creating new objects in the various interphases would start to have an unpleasant impact on the program execution time.

## Exercise 48: String builder

In situations like the previous one, it is better to use `StringBuilder` objects when it comes to building strings. Differently from Strings, StringBuilders are not immutable, meaning StringBuilder-objects can be modified. Get acquainted with the description of the StringBuilder API (you'll find it by googling "stringbuilder java api 6") and modify the method in the exercise body, `public static String build(int[] t)`, so that it would use StringBuilder and work in the following way:

```
{
 1, 2, 3, 4,
 5, 6, 7, 8,
 9, 10
}
```

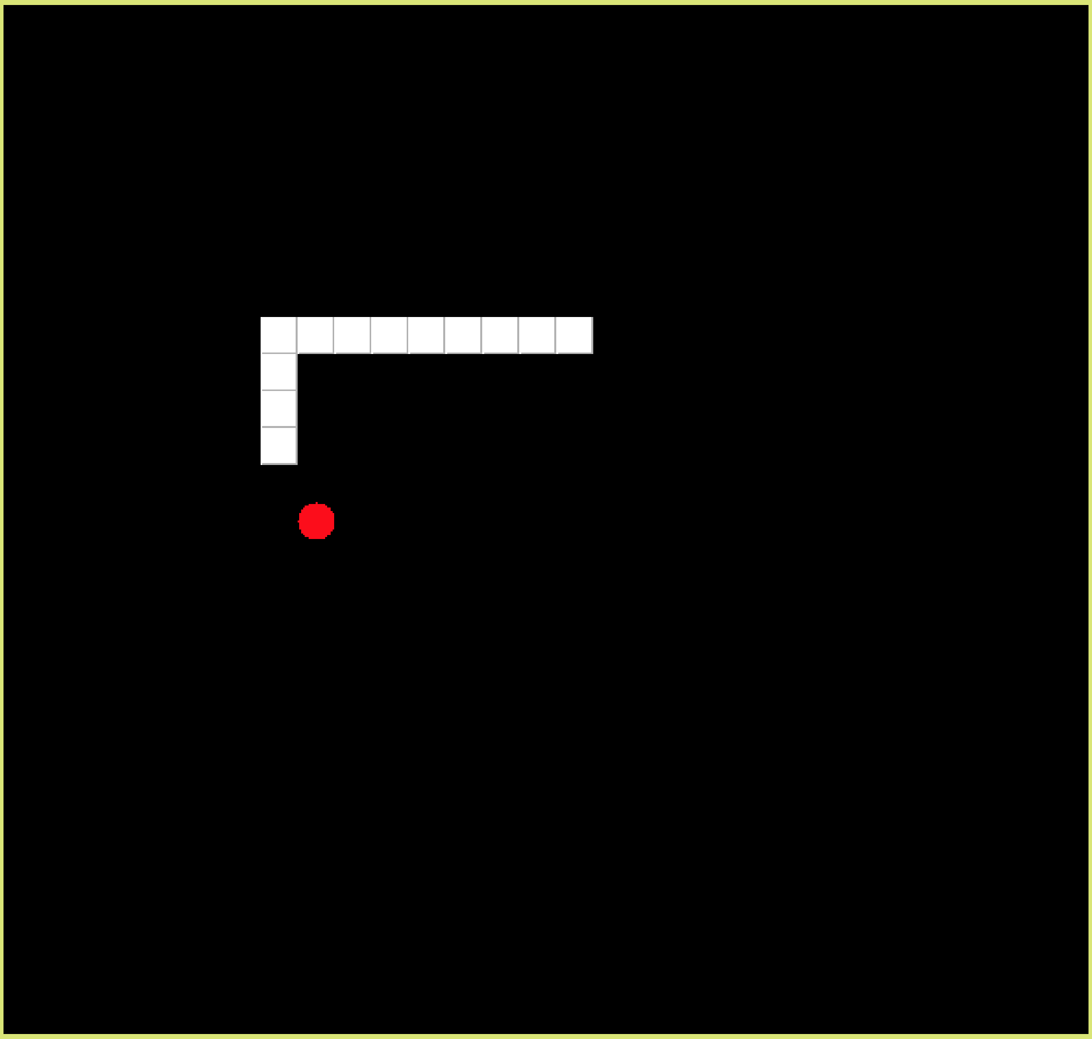
The curly brackets are on their own lines. There are up to four values in each line of the table, and the first value requires an initial space. Before each number, and after the comma there must be exactly one space.

# 55. GRANDE FINALE

The course is nearing the end, and it's time for the grande finale!

**Exercise 49: Worm Game**

In this exercise, we create the structure and functionality for the following worm game. Unlike in the picture below, our worm game will be coloured differently: worm is black, apple is red and background is gray.



### EXERCISE 49.1: PIECE AND APPLE

Create the class `Piece` in `wormgame.domain`. The class `Piece` has the constructor `public Piece(int x, int y)`, which receives the position of the piece as parameter. Moreover, the class `Piece` has the following methods.

- `public int getX()` returns the x coordinate of the piece, which was assigned in the constructor.
- `public int getY()` returns the y coordinate of the piece, which was assigned in the constructor.

- `public boolean runsInto(Piece piece)` returns true if the object has the same coordinates as the Piece instance received as paramater.
- `public String toString()` returns the piece position following the pattern `(x,y)`. For instance. `(5,2)` if the value of x is 5 and the value of y is 2.

Also implement the class `Apple` in `wormgame.domain`, and let Apple inherit `Piece`.

## EXERCISE 49.2: WORM

Implement the class `Worm` in the package `wormgame.domain`. The class `Worm` has the constructor `public Worm(int originalX, int originalY, Direction originalDirection)`, which creates a new worm whose direction is the parameter `originalDirection`. Worm is composed of a list of instances of the class `Piece`. Attention: the pre-made enum `Direction` can be found in the package `wormgame`.

When it's created, Worm's length is one, but its "mature" length is three. Worm has to grow by one piece while it moves. When its length is three, Worm grows only by eating.

Implement the following methods

- `public Direction getDirection()` return's Worm's direction.
- `public void setDirection(Direction dir)` sets a new direction for the worm. The worm starts to move in the new direction when the method `move` is called again.
- `public int getLength()` returns the Worm's length. The Worm's length has to be equal to the length of the list returned by the method `getPieces()`.
- `public List<Piece> getPieces()` returns a list of Piece objects which the worm is composed of. The pieces in the list are in order, with the worm head at the end of the list.
- `public void move()` moves the worm one piece forward.
- `public void grow()` grows the worm by one piece. The worm grows together with the following `move` method call; after the first move method call the worm does not grow any more.
- `public boolean runsInto(Piece piece)` checks whether the worm runs into the parameter piece. If so -- that is, if a piece of the worm runs into the parameter piece -- the method returns the value `true`; otherwise it returns `false`.
- `public boolean runsIntoItself()` check whether the worm runs into itself. If so -- that is, if one of the worm's pieces runs into another piece -- the method returns the value `true`. Otherwise it returns `false`.

The functionality of `public void grow()` and `public void move()` has to be implemented so that the worm grows only with the following move.

Motion should be implemented in a way that the worm is always given a new piece. The position of the new piece depends on the moving direction: if moving left, the new piece location should on the left of the head piece, i.e. the x coordinate of the head should be smaller by one. If the location of the new piece is under the old head -- if the worm's direction is down, the y coordinate of the new piece should be by one bigger than the y coordinate of the head (when we draw, we will have to get accustometd to a coordinate system where the y axe is reverse).

When the worm moves, a new piece is added to the list, and the first piece is deleted from the beginning of the list. In this way, you don't need to update the coordinates of each single piece. Implement the growth so that the first piece is deleted if the `grow` method has just been called.

Attention! The worm has to grow constantly if its length is less than 3.

```
Worm worm = new Worm(5, 5, Direction.RIGHT);
System.out.println(worm.getPieces());
worm.move();
System.out.println(worm.getPieces());
worm.move();
System.out.println(worm.getPieces());
worm.move();
System.out.println(worm.getPieces());

worm.grow();
System.out.println(worm.getPieces());
worm.move();
System.out.println(worm.getPieces());

worm.setDirection(Direction.LEFT);
System.out.println(worm.runsIntoItself());
worm.move();
System.out.println(worm.runsIntoItself());
```

```
[(5,5)]
[(5,5), (6,5)]
[(5,5), (6,5), (7,5)]
[(6,5), (7,5), (8,5)]
[(6,5), (7,5), (8,5)]
[(6,5), (7,5), (8,5), (9,5)]
false
true
```

## EXERCISE 49.3: WORM GAME, PART 1

Let's modify the class `WormGame` which is contained in `wormgame.game`, and encapsulates the functionality of the game. The class WormGame inherits the class `Timer`, which provides the time functionality to update the game. In order to work, the class `Timer` requires a class which implements the interface `ActionListener`, and we have implemented it in `WormGame`.

Modify the functionality of WormGame's constructor so that the game's `Worm` is created in the constructor. Create the worm so that the position of the worm depends on the parameters received in the constructor of the class WormGame. The worm's x coordinate has to be `width / 2`, the y coordinate `height / 2` , and the direction `Direction.DOWN`.

Also create an apple inside the constructor. The apple coordinates have to be random, but the apple x coordinate has to be contained between `[0, width[`, and the y coordinate between `[0, height[`.

Also add the following methods to the WormGame:

- `public Worm getWorm()` returns the WormGame worm.
- `public void setWorm(Worm worm)` sets on the game the method parameter worm. If the method `getWorm` is called after the worm has been set up, it has to return a reference to the *same* worm.
- `public Apple getApple` returns the apple of the WormGame.
- `public void setApple(Apple apple)` sets the method parameter apple on the worm game. If the method `getApple` is called after the apple has been set up, it has to return a reference to the *same* apple.

## EXERCISE 49.4: WORM GAME, PART 2

Modify the functionality of the method `actionPerformed` so that it would implement the following tasks in the given order.

- Move the worm
- If the worm runs into the apple, it eats the apple and calls the grow method. A new apple is randomly created.
- If the worm runs into itself, the variable `continue` is assigned the value `false`
- Call `update`, which is a method of the variable `updatable` which implements the interface `Updatable`.
- Call the `setDelay` method which is inherited from the Timer class. The game velocity should grow with respect to the worm length. The call `setDelay(1000 / worm.getLength());` will work for it: in the call we expect that you have defined the object variable `worm`.

Let's start next to build our user interface components.

## EXERCISE 49.5: KEYBOARD LISTENER

Implement the class `KeyboardListener` in `wormgame.gui`. The class has the constructor `public KeyboardListener(Worm worm)`, and it implements the interface `KeyListener`. Replace the method `keyPressed` so that the worm is assigned direction up when the up arrow key. Respectively, the worm is assigned the directions down, left, or right, when the user presses the down, left, or right arrow key.

## EXERCISE 49.6: DRAWINGBOARD

Create the class `DrawingBoard` in `wormgame.gui`. The `DrawingBoard` inherits the class `JPanel`, and its constructor receives an instance of the class `WormGame` and the int variable `pieceLength` as parameters. The variable `pieceLength` tells the dimension of the pieces; length and height of the pieces are equal.

Replace the `paintComponent` method which was inherited from the class `JPanel` so that the method draws the worm and the apple. Use the Graphics object's `fill3DRect` method to draw the worm. The worm colour has to be black (`Color.BLACK`). The apple has to be drawn with the Graphics object's `fillOval` method, and its colour has to be red.

Also implement the interface `Updatable` in `DrawingBoard`. The method `update` of Updatable has to call the `repaint` method of the class JPanel.

### EXERCISE 49.7: USER INTERFACE

Modify the class `UserInterface` to contain DrawingBoard. In the method `createComponents`, you have to create an instance of DrawingBoard and add it into the Container object. Create an instance of `KeyboardListener` at the end of `createComponents`, and add it to the Frame object.

Also add the method `public Updatable getUpdatable()` to the class `UserInterface`, returning the drawing board which was created in `createComponents`.

You can start the user interface from the `Main` class in the following way. Before the game starts, we wait that the user interface is created. When the user interface is created, it gets connected to the worm game and the game gets started.

```
        WormGame game = new WormGame(20, 20);

        UserInterface ui = new UserInterface(game, 20);
        SwingUtilities.invokeLater(ui);

        while (ui.getUpdatable() == null) {
            try {
                Thread.sleep(100);
            } catch (InterruptedException ex) {
                System.out.println("The drawing board hasn't been created yet.");
            }
        }

        game.setUpdatable(ui.getUpdatable());
        game.start();
```

# 56. COURSE FEEDBACK

**Exercise 50: Course Feedback**

We have received a lot of valuable feedback through TMC. Therefore, as your last task in the course we would like to have feedback on the course in general. Give feedback by filling out the comment section which appears when you have submitted the exercise. If comment section doesn't pop up, please write your feedback in the `Main`-class of the downloaded exercise and submit it again.

In order to receive the points of this exercise submit the exercise to our server.