Object-Oriented Programming with Java, part II ››

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

43. SINGLE RESPONSIBILITY PRINCIPLE

When we design bigger programs, we often reason about what class has to deal with what task. If we delegate the implementation of the whole program to one class, the result is inevitably chaos. A sector of software design, *object-oriented design*, includes the *Single Responsibility Principle*, which we should follow.

The Single Responsibility Principle states that each class should have only one clear role. If the class has one clear role, modifying that role is easy, and only one class will have to be modified. *Each class should have only one reason to be modified.*.

Let us focus on the following class `Worker`, which has methods to calculate his salary and to report his working hours.

```
public class Worker {
    // object variables

    // worker's constructor and methods

    public double calculateSalary() {
        // the logic concerning salary count
    }

    public String reportHours() {
        // the logic concerning working hours bookkeeping
    }
}
```

Even if the examples above do not show the concrete implementations, an alarm should go off. Our `Worker` class has at least three different responsibilities. It represents a worker, it performes the role of a salary calculator, and the role of a working hour bookkeeping system by reporting working hours. The class above should be split into three: one should represent the worker, another should represent the salary calculator, and the third should deal with time bookkeeping.

```
public class Worker {
    // object variables

    // worker's constructor and methods
}
```

```
public class SalaryCalculator {
    // object variables

    // methods for salary count

    public double calculateSalary(Person person) {
        // salary calculation logic
    }
}
```

```
public class TimeBookkeeping {
    // object variables

    // methods concerning time bookkeeping

    public String createHourReport(Person person) {
        // working hours bookeeping logic
    }
}
```

Each variable, each code raw, each method, each class, and each program should have only one responsibility. Often a "better" program stucture is clear to the programmer only once the program is implemented. This is completely acceptable: even more important it is that we always try to change a program to make it clearer. ***Always refactor -- i.e. always improve your program when it is needed!***

44. ORGANISING CLASSES INTO PACKAGES

When we design and implement bigger programs, the number of classes rapidly grows. When the number of classes grows, remembering their functionality and methods becomes more difficult. Giving sensible names to classes helps to remember their funcitonality. In addition to giving sensible names, it is good to split the source code files into packages according to their functionality, use, and other logical reasons. In fact, the *packages* are but folders we use to organise our source code files. Directories are often called folders, both in windows and colloqually. We will use the term directory, anyway.

Programming environments provide made-up tools for package management. So far, we have been creating classes and interfaces only in the `default package` of the `Source Packages` partition. In NetBeans, we can create a new package by clicking on `Source Packages`, and choosing `New -> Java Package...`. In the created package, we can create classes in the same way as we do in the `default package`.

You can read the name of the package that contains a certain class at the beginning of the source code files in the sentence `package` *`packageName`* before the other statements. For instance, the below class `Implementation` is contained in the package `library`.

```
package library;

public class Implementation {

    public static void main(String[] args) {
        System.out.println("Hello packageworld!");
    }
}
```

Packages can contain other packages. For instance, the package definition `package library.domain` means that the package `domain` is contained in the package `library`. By placing packages into other packages, we design the hierachy of classes and interfaces. For instance, all Java's classes are located in packages that are contained in the package `java`. The package name `domain` is often used to represent the storage location of the classes which deal with concepts specific for the domain. For instance, the class `Book` could be stored in the package `library.domain` because it represents a concept specific of the library.

```
package library.domain;

public class Book {
```

```
    private String name;

    public Book(String name) {
        this.name = name;
    }

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

We can uses the classes stored in our packages through the `import` statement. For instance, the class `Implementation`, which is contained in the package `library` could make use of a class stored in `library.domain` through the assignment `import library.domain.Book`.

```
package library;

import library.domain.Book;

public class Implementation {

    public static void main(String[] args) {
        Book book = new Book("The ABC of Packages!");
        System.out.println("Hello packageworld: " + book.getName());
    }
}
```

```
Hello packageworld: The ABC of Packages!
```

The import statements are defined in our source code file after the package statement but before the class statement. There can be many of them -- for instance, when we want to use different classes. Java's made-up classes are usually stored in `java` package child packages. Hopefully, the statements which appear at the beginning of our classes -- such as `import java.util.ArrayList` and `import java.util.Scanner;` -- are starting to look more meaningful now.

From now on, in *all* our exercises we will use packages. Next, we will create our first packages ourselves.

Exercise 17: First Packages

EXERCISE 17.1: UI INTERFACE

Create the package `mooc` in your project. We create the functionality of our application inside this package. Add the package `ui` to your application; at this point, you should

have the package `mooc.ui`. Create a new interface in it, and call it `UserInterface`.

The interface `UserInterface` has to determine the method `void update()`.

EXERCISE 17.2: TEXT USER INTERFACE

Create the class `TextUserInterface` in the same package; make it implement the interface `UserInterface`. Implement the method `public void update()` which is required by the interface `UserInterface` which `TextUserInterface` implements: its only duty should be printing the string "`Updating the user interface`" with a `System.out.println` method call.

EXERCISE 17.3: APPLICATION LOGIC

Create now the package `mooc.logic`, and add the class `ApplicationLogic` in it. The application logic API should be the following:

- the constructor `public ApplicationLogic(UserInterface ui)`

. It receives as parameter a class which implements the interface UserInterface. Note: your application logic has to see the interface and therefore to import it; in other words, the line `import mooc.ui.UserInterface` must appear at the beginning of he file

- the method `public void execute(int howManyTimes)`

prints the string "The application logic works" as many times as it is defined by its parameter variable. After each "The application logic works" printout, the code has to call the `update()` method of the object which implements the interface `UserInterface` and which was assigned to the constructor as its parameter.

You can test your application with the following main class.

```
import mooc.logic.ApplicationLogic;
import mooc.ui.UserInterface;
import mooc.ui.TextUserInterface;

public class Main {

    public static void main(String[] args) {
        UserInterface ui = new TextUserInterface();
        new ApplicationLogic(ui).execute(3);
    }
}
```

The program output should be the following:

```
The application logic works
Updating the user interface
The application logic works
Updating the user interface
The application logic works
Updating the user interface
```

44.1 A CONCRETE DIRECTORY CONSTRUCTION

All the projects which can be seen are stored in your computer *file system*. Each project has its own directory (folder) which contains the project directories and files.

The project directory `src` contains the program source code. If a class package is a library, it is located in the directory `library` of the project source code directory `src`. If you are interested in it, it is possible to have a look at the concrete project structure in NetBeans, by going to the *Files* tab which is next to the *Projects* tab. If you can't see the *Files* tab, you can display it by choosing *Files* from the *Window* menu.

Application development is usually done through the *Projects* tab, where NetBeans has hidden the project files which the programmer doesn't have to care about.

44.2 VISIBILITY DEFINITIONS AND PACKAGES

We have already managed to know two visibility definitions. The method and variables with the visibility definition `private` are visible only inside the class that defines them. They cannot be used outside the class. Differently, the method and variables with visibility definition `public` are visible for any class.

```
package library.ui;

public class UserInterface {
    private Scanner reader;

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

    public void start() {
        printTitle();

        // more functionality
    }

    private void printTitle() {
        System.out.println("***********");
        System.out.println("* LIBRARY *");
        System.out.println("***********");
    }
}
```

The object constructor and `start` method of the above class `UserInterface` can be called from whatever program. The method `printTitle` and the variable `reader` can be used only inside their class.

When we want to assign package visibility to a variable or a method, we do not need to use any prefix. We can modify the example above assigning package visibility to the method `printTitle`.

```
package library.ui;

public class UserInterface {
    private Scanner reader;

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

    public void start() {
        printTitle();

        // more functionality
    }

    void printTitle() {
        System.out.println("***********");
        System.out.println("* Library *");
        System.out.println("***********");
    }
}
```

Now, the classes *inside the same package* can use the method `printTitle`.

```
package library.ui;

import java.util.Scanner;

public class Main {

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

        userInterface.printTitle(); // it works!
    }
}
```

If the class is in a different package, the method `printTitle` can't be used.

```
package library;
```

```
import java.util.Scanner;
import library.ui.UserInterface;

public class Main {

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

        userInterface.printTitle(); // it doesn't work  !
    }
}
```

45. MANY INTERFACES, AND INTERFACE FLEXIBILITY

Last week we were introduced to interfaces. An interface defines one or more methods which have to be implemented in the class which implements the interface. The interfaces can be stored into packages like any other class. For instance, the interface `Identifiable` below is located in the package `application.domain`, and it defines that the classes which implement it have to implement the method `public String getID()`.

```
package application.domain;

public interface Identifiable {
    String getID();
}
```

The class makes use of the interface through the keyword `implements`. The class `Person`, which implements the `Idenfifiable` interface. The `getID`of Person class always returns the person ID.

```
package application.domain;

public class Person implements Identifiable {
    private String name;
    private String id;

    public Person(String name, String id) {
        this.name = name;
        this.id = id;
    }

    public String getName() {
```

```
        return this.name;
    }

    public String getPersonID() {
        return this.id;
    }

    @Override
    public String getID() {
        return getPersonID();
    }

    @Override
    public String toString(){
        return this.name + " ID: " +this.id;
    }
}
```

An interface strength is that interfaces are also *types*. All the objects which are created from classes that implement an interface also have that interface's type. This effictively helps us to build our applications.

We create the class `Register`, which we can use to search for people against their names. In addition to retrieve single people, `Register` provides a method to retrieve a list with all the people.

```
public class Register {
    private HashMap<String, Identifiable> registered;

    public Register() {
        this.registered = new HashMap<String, Identifiable>();
    }

    public void add(Identifiable toBeAdded) {
        this.registered.put(toBeAdded.getID(), toBeAdded);
    }

    public Identifiable get(String id) {
        return this.registered.get(id);
    }

    public List<Identifiable> getAll() {
        return new ArrayList<Identifiable>(registered.values());
    }
}
```

Using the register is easy.

```
Register personnel = new Register();
personnel.add( new Person("Pekka", "221078-123X") );
personnel.add( new Person("Jukka", "110956-326B") );
```

```
System.out.println( personnel.get("280283-111A") );

Person found = (Person) personnel.get("110956-326B");
System.out.println( found.getName() );
```

Because the people are recorded in the register as `Identifiable`, we have to change back their type if we want to deal with people through those methods which are not defined in the interface. This is what happens in the last two lines.

What about if we wanted an operation which returns the people recorded in our register sorted according to their ID?

One class can implement various different interfaces, and our `Person` class can implement `Comparable` in addition to `Identifiable`. When we implement various different interfaces, we separate them with a comma (`public class ... implements` *`FirstInterface`*`,` *`SecondInterface`* `...`). When we implement many interfaces, we have to implement all the methods required by all the interfaces. Below, we implement the interface `Comparable` in the class `Person`.

```
package application.domain;

public class Person implements Identifiable, Comparable<Person> {
    private String name;
    private String id;

    public Person(String name, String id) {
        this.name = name;
        this.id = id;
    }

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

    public String getPersonID() {
        return this.id;
    }

    @Override
    public String getID() {
        return getPersonID();
    }

    @Override
    public int compareTo(Person another) {
        return this.getID().compareTo(another.getID());
    }
}
```

Now, we can add to the register method sortAndGetEverything:

```
    public List<Identifiable> sortAndGetEverything() {
        ArrayList<Identifiable> all = new ArrayList<Identifiable>(registered.values
        Collections.sort(all);
        return all;
    }
```

However, we notice that our solution does not work. Because the people are recorded into the register as if their type was `Identifiable`, Person has to implement the interface `Comparable<Identifiable>` so that our register could sort people with its method `Collections.sort()`. This means we have to modify Person's interface:

```
public class Person implements Identifiable, Comparable<Identifiable> {
    // ...

    @Override
    public int compareTo(Identifiable another) {
        return this.getID().compareTo(another.getID());
    }
}
```

Now our solution works!

Our Register is unaware of the real type of the objects we record. We can use the class Register to record objects of different types than Person, as long as the object class implements the interface `Identifiable`. For instance, below we use the register to manage shop sales:

```
public class Sale implements Identifiable {
    private String name;
    private String barcode;
    private int stockBalance;
    private int price;

    public Sale(String name, String barcode) {
        this.name = name;
        this.barcode = barcode;
    }

    public String getID() {
        return barcode;
    }

    // ...
}

Register products = new Register();
products.add( new Product("milk", "11111111") );
products.add( new Product("yogurt", "11111112") );
products.add( new Product("cheese", "11111113") );
```

```
System.out.println( products.get("99999999") );

Product product = (Product)products.get("11111112");
product.increaseStock(100);
product.changePrice(23);
```

The class `Register` is quite universal now that it is not dependent on concrete classes. Whatever class which implements `Identifiable` is compatible with Register. However, the method `sortAndGetEverything` can only work if we implement the interface Comparable<Identifiable>.

NetBeans Tips

- All NetBeans tips can be found here
- **Implement all abstract methods**

  Let us suppose that your program contains the interface `Interface`, and you are building the class `Class` which implements the interface. It will be annoying to write the declaration raws of all the interface methods.

  However it is possible to ask NetBeans to fill in the method bodies automatically. When you have defined the interface a class should implement, i.e. when you have written

  ```
  public class Class implements Interface {
  }
  ```

  NetBeans paints the class name red. If you go to lamp icon on the left corner of the raw, click, and choose *Implement all abstract methods*, the method bodies will appear in your code!

- **Clean and Build**

  Sometimes, NetBeans may get confused and try to run a code version without noticing all the corrected changes made to it. Usually you notice it because something "strange" happens. Usually, you can fix the problem by using *Clean and build* operation. The operation is found in the Run menu, and you can execute it also by clicking on the *brush and hammer* symbol. Clean and build deletes the translated versions of the code and generates a new translation.

Exercise 18: Moving

Before moving, you pack your things and put them into boxes trying to keep the number of boxes needed as small as possible. In this exercise we simulate packing things into boxes. Each thing has a volume, and boxes have got a maximum capacity.

EXERCISE 18.1: THINGS AND ITEMS

The removers will later on move your things to a track (which is not implemented here); therefore, we first implement the interface `Thing`, which represents all things and boxes.

The Thing interface has to determine the method `int getVolume()`, which is needed to understand the size of a thing. Implement the interface `Thing` in the package `moving.domain`.

Next, implement the class `Item` in the package `moving.domain`. The class receives the item name (String) and volume (int) as parameter. The class has to implement the interface `Thing`.

Add the method `public String getName()` to `Item`, and replace the method `public String toString()` so that it returns strings which follow the pattern "`name (volume dm^3)`". Item should now work like the following

```
Thing item = new Item("toothbrash", 2);
System.out.println(item);
```

```
toothbrash (2 dm^3)
```

EXERCISE 18.2: COMPARABLE ITEM

When we pack our items into boxes, we want to start in order from the first items. Implement the interface `Comparable` in the class `Item`; the item *natural order* must be ascending against volume. When you have implemented the interface `Comparable`, the `sort` method of class `Collection` has to work in the following way:.

```
List<Item> items = new ArrayList<Item>();
items.add(new Item("passport", 2));
items.add(new Item("toothbrash", 1));
items.add(new Item("circular saw", 100));

Collections.sort(items);
System.out.println(items);
```

```
[toothbrash (1 dm^3), passport (2 dm^3), circular saw (100 dm^3)]
```

EXERCISE 18.3: MOVING BOX

Implement now the class `Box` in the package `moving.domain`. At first, implement the following method for your Box:

- the constructor `public Box(int maximumCapacity)`

receives the box maximum capacity as parameter;

- the method `public boolean addThing(Thing thing)`

adds an item which implements the interface `Thing` to the box. If it does not fit in the box, the method returns `false`, otherwise `true`. The box must store the things into a list.

Also, make your `Box` implement the `Thing` interface. The method `getVolume` has to return the current volume of the things inside the box.

EXERCISE 18.4: PACKING ITEMS

Implement the class `Packer` in the package `moving.logic`. The constructor of the class `Packer` is given the parameter `int boxesVolume`, which determines how big boxes the packer should use.

Afterwards, implement the method `public List<Box> packThings(List<Thing> things)`, which packs things into boxes.

The method should move all the things in the parameter list into boxes, and these boxes should be contained by the list the method returns. You don't need to pay attention to such situations where the things are bigger than the boxes used by the packer. The tests do not check the way the packer makes use of the moving boxes.

The example below shows how our packer should work:

```
// the things we want to pack
List<Thing> things = new ArrayList<Thing>();
things.add(new Item("passport", 2));
things.add(new Item("toothbrash", 1));
things.add(new Item("book", 4));
things.add(new Item("circular saw", 8));

// we create a packer which uses boxes whose valume is 10
Packer packer = new Packer(10);

// we ask our packer to pack things into boxes
List<Box> boxes = packer.packThings( things );

System.out.println("number of boxes: "+boxes.size());

for (Box box : boxes) {
    System.out.println("  things in the box: "+box.getVolume()+" dm^3");
}
```

Prints:

```
number of boxes: 2
  things in the box: 7 dm^3
  things in the box: 8 dm^3
```

The packer has packed the things into two boxes, the first box has the firts three things, whose total volume was 7, and the last thing in the list -- the circular saw, whose volume was 8 -- has gone to the third box. The tests do not set a limit to the number of boxes

used by the packer; each thing could have been packed into a different box, and the output would have been:

```
number of boxes: 4
  things in the box: 2 dm^3
  things in the box: 1 dm^3
  things in the box: 4 dm^3
  things in the box: 8 dm^3
```

Note: to help testing, it would be convinient to create a toString method for the class `Box`, for instance; this would help printing the content of the box.

46. EXCEPTIONS

Exceptions are such situations where the program executions is different from our expectations. For instance, the program may have called a method of a *null* reference, in which case the user is *thrown* a `NullPointerException`. If we try to retrieve a index outside a table, the user is thrown a `IndexOutOfBoundsException`. All of them are a type of `Exception`.

We deal with exception using the block `try { } catch (Exception e) { }`. The code contained within the brackets which follows the keyword `try` can *possibly* go through an exception. The keyword the code within the brackets which follow the keyword `catch` defines what should happen when the try-code throws an exception. We also define the type of the exception we want to catch (`catch (Exception e)`).

```
try {
    // code which can throw an exception
} catch (Exception e) {
    // code which is executed in case of exception
}
```

The `parseInt` method of class `Integer` which turns a string into a number can throw a `NumberFormatException` if its string parameter cannot be turned into a number. Now we implement a program which tries to turn into a number a user input string.

```
Scanner reader = new Scanner(System.in);
System.out.print("Write a number: ");

int num = Integer.parseInt(reader.nextLine());
```

```
Write a number: tatti
Exception in thread "..." java.lang.NumberFormatException: For input string: "tatti
```

The program above throws an exception because the user digits an erroneous number. The program execution ends up with a malfunction, and it cannot continue. We add an exception management statement to our program. The call, which may throw an exception is written into the `try` block, and the action which takes place in case of exception is written into the `catch` block.

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

    System.out.print("Write a number: ");

    try {
        int num = Integer.parseInt(reader.nextLine());
    } catch (Exception e) {
        System.out.println("You haven't written a proper number.");
    }
```

```
Write number: 5
```

```
Write number: oh no!
You haven't written a proper number.
```

In case of exception, we move from the chunk of code defined by the `try` keyword to the `catch` chunk. Let's see this by adding a print statement after the `Integer.parseInt` line in the `try` chunk.

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

    System.out.print("Write a number: ");

    try {
        int num = Integer.parseInt(reader.nextLine());
        System.out.println("Looks good!");
    } catch (Exception e) {
        System.out.println("You haven't written a proper number.");
    }
```

```
Write a number: 5
Looks good!
```

```
Write a number: I won't!
you haven't written a proper number.
```

String `I won't!` is given as parameter to the method `Integer.parseInt`, which throws an exception if the String parameter can't be changed into a number. Note that the code in the `catch` chunk is executed *only* in case of exception -- otherwise the program do not arrive till there.

Let's make something more useful out of our number translator: let's do a method which keeps on asking to type a number till the user does it. The user can return *only* if they have typed the right number.

```
public int readNumber(Scanner reader) {
    while (true) {
        System.out.print("Write a number: ");

        try {
            int num = Integer.parseInt(reader.nextLine());
            return num;
        } catch (Exception e) {
            System.out.println("You haven't written a proper number.");
        }
    }
}
```

The method `readNumber` could work in the following way:

```
Write a number: I won't!
You haven't written a proper number.
Write a number: Matti has a mushroom on his door.
You haven't written a proper number.
Write a number: 43
```

46.1 THROWING EXCEPTIONS

Methods and constructors can *throw* exceptions. So far, there are two kinds of exceptions which can be thrown. There are the ones which have to be handled, and the ones which don't have to be dealt with. When we have to handle the exceptions, we do it either in a `try-catch` chunk, or *throwing them from a* `method`.

In the clock exercise of Introduction to Programming, we explained that we can stop our program of one second, by calling the method `Thread.sleep(1000)`. The method may throw an exception, which we *must* deal with. In fact, we handle the exception using the `try-catch` sentence; in the following example we skip the exception, and we leave empty the `catch` chunk.

```
    try {
        // we sleep for 1000 milliseconds
        Thread.sleep(1000);
    } catch (Exception e) {
        // In case of exception, we do not do anything.
    }
```

It is also possible to avoid handling the exceptions in a method, and *delegate the responsibility* to the method caller. We delegate the responsibility of a method by using the statement `throws Exception`.

```
    public void sleep(int sec) throws Exception {
        Thread.sleep(sec * 1000);   // now we don't need the try-catch block
    }
```

The `sleep` method is called in another method. Now, this other method can either handle the exception in a `try-catch` block or delegate the responsibility forward. Sometimes, we delegate the responsibility of handling an exception, till the very end, and even the `main` method delegates it:

```
public class Main {
    public static void main(String[] args) throws Exception {
        // ...
    }
}
```

In such cases, the exception ends up in Java's virtual machine, which interrupts the program in case there is an error which causes the problem.

There are some exceptions which the programmer does not always have to address, such as the `NumberFormatException` which is thrown by `Integer.parseInt`. Also the `RuntimeException`s do not always require to be addressed; next week we will go back to *why* variables can have more than one type.

We can throw an exception ourself from the source code using the `throw` statement. For instance, if we want to throw an exception which was created in the class `NumberFormatException`, we could use the statement `throw new NumberFormatException()`.

Another exception which hasn't got to be addressed is `IllegalArgumentException`. With `IllegalArgumentException` we know that a method or a constructor has received an *illegal* value as parameter. For instance, we use the IllegalArgumentException when we want to make sure that a parameter has received particular values. We create the class `Grade` whose constructor has a integer parameter: the grade.

```
public class Grade {
    private int grade;

    public Grade(int grade) {
        this.grade = grade;
```

```
    }

    public int getGrade() {
        return this.grade;
    }
}
```

Next, we want to validate the value of the constructor parameter of our Grade class. The grades in Finland are from 0 to 5. If the grade is something else, we want to *throw an exception*. We can add an if statement to our `Grade` class constructor, which checks whether the grade is outside range 0-5. If so, we throw an `IllegalArgumentException` telling `throw new IllegalArgumentException("The grade has to be between 0-5");`.

```
public class Grade {
    private int grade;

    public Grade(int grade) {
        if (grade < 0 || grade > 5) {
            throw new IllegalArgumentException("The grade has to be between 0-5");
        }
        this.grade = grade;
    }

    public int getGrade() {
        return this.grade;
    }
}
```

```
    Grade grade = new Grade(3);
    System.out.println(grade.getGrade());

    Grade wrongGrade = new Grade(22);
    // it causes an exception, we don't continue
```

```
3
Exception in thread "..." java.lang.IllegalArgumentException: The grade has to be b
```

Exercise 19: Method Argument Validation

Let's train method argument validation with the help of the `IllegalArgumentException`. The excercise layout shows two classes `Person` and `Calculator`. Change the class in the following way:

EXERCISE 19.1: PERSON VALIDATION

The constructor of `Person` has to make sure its parameter's Name variable is not null, empty, or longer than 40 characters. The age has also to be between 0-120. If one of the conditions above are not satisfied, the constructor has to throw an `IllegalArgumentException`.

EXERCISE 19.2: CALCULATOR VALIDATION

The `Calculator` methods have to be changed in the following way: the method `multiplication` has to work only if its parameter is not negative (greater than or equal to 0). The method `binomialCoefficient` has to work only if the parameters are not negative and the size of a subset is smaller than the set's size. If one of the methods receives invalid arguments when they are called, they have to throw a `IllegalArgumentException`.

Exercise 20: Sensors and Temperature Measurement

All the code in our application has to be placed into the package `application`.

We have got the following interface available for our use:

```
public interface Sensor {
    boolean isOn();   // returns true if the sensor is on
    void on();        // switches the sensor on
    void off();       // switches the sensor off
    int measure();    // returns the sensor reading if the sensor is on
                      // if the sensor is off, it throws an IllegalStateException
}
```

EXERCISE 20.1: CONSTANT SENSOR

Create the class `Constant Sensor` which implements the interface `Sensor`.

The constant sensor is online all the time. The methods on() and off() do not do anything. The constant sensor has a constructor with an int parameter. The `measure` method call returns the number received as constructor parameter.

For instance:

```
public static void main(String[] args) {
  ConstantSensor ten = new ConstantSensor(10);
  ConstantSensor minusFive = new ConstantSensor(-5);

  System.out.println( ten.measure() );
  System.out.println( minusFive.measure() );
```

```
    System.out.println( ten.isOn() );
    ten.off();
    System.out.println( ten.isOn() );
}
```

Prints:

```
10
-5
true
true
```

EXERCISE 20.2: THERMOMETER

Create the class `Thermometer` which implements the interface `Sensor`.

At first, the thermometer is off. When the `measure` method is called, if the thermometer is on it returns a random number between -30 and 30. If the thermometer is off, it throws an `IllegalStateException`.

EXERCISE 20.3: AVERAGESENSOR

Create the class `AverageSensor` which implements the interface `Sensor`.

An average sensor contains many sensors. In addition to the methods defined by the interface `Sensor`, the class has the method `public void addSensor(Sensor additional)` which adds a new sensor to the AverageSensor.

The average sensor is on when *all* its sensors are on. When the average sensor is switched on, all its sensors have to be switched on if they were not on already. When the average sensor is closed, at least one of its sensors has to be switched off. It's also possible that all its sensors are switched off.

The `measure` method of our AverageSensor returns the average of the readings of all its sensors (because the return value is `int`, the readings are rounded down as it is for integer division). If the `measure` method is called when the average sensor is off, or if the average sensor was not added any sensor, the method throws an `IllegalStateException`.

Below, you find an example of a sensor program (note that both the Thermometer and the AverageSensor constructors are without parameter):

```
public static void main(String[] args) {
    Sensor kumpula = new Thermometer();
    kumpula.on();
    System.out.println("the temperature in Kumpula is "+kumpula.measure() + " degrees");

    Sensor kaisaniemi = new Thermometer();
    Sensor helsinkiVantaa = new Thermometer();
```

```
    AverageSensor helsinkiArea = new AverageSensor();
    helsinkiArea.addSensor(kumpula);
    helsinkiArea.addSensor(kaisaniemi);
    helsinkiArea.addSensor(helsinkiVantaa);

    helsinkiArea.on();
    System.out.println("the temperature in Helsinki area is "+helsinkiArea.measure() + "
}
```

Prints (the printed readings depend on the random temperature readings):

```
the temperature in Kumpula is -7 degrees
the temperature in Helsinki area is -10 degrees
```

Note: you'd better use a ConstantSensor object to test your average sensor!

EXERCISE 20.4: ALL READINGS

Add the method `public List<Integer> readings()` to your AverageSensor; it returns a list of the reading results of all the measurements executed through your AverageSensor. Below is an example of how the method works:

```
public static void main(String[] args) {
    Sensor kumpula = new Thermometer();
    Sensor kaisaniemi = new Thermometer();
    Sensor helsinkiVantaa = new Thermometer();

    AverageSensor helsinkiArea = new AverageSensor();
    helsinkiArea.addSensor(kumpula);
    helsinkiArea.addSensor(kaisaniemi);
    helsinkiArea.addSensor(helsinkiVantaa);

    helsinkiArea.on();
    System.out.println("the temperature in Helsinki area is "+helsinkiArea.measure() + "
    System.out.println("the temperature in Helsinki area is "+helsinkiArea.measure() + "
    System.out.println("the temperature in Helsinki area is "+helsinkiArea.measure() + "

    System.out.println("readings: "+helsinkiArea.readings());
}
```

Prints (again, the printed readings depend on the random temperature readings):

```
the temperature in Helsinki area is -10 degrees
the temperature in Helsinki area is -4 degrees
the temperature in Helsinki area is -5 degrees

readings: [-10, -4, 5]
```

46.2 EXCEPTIONS AND INTERFACES

Interfaces do not have a method body, but the method definition can be freely chosen when the developer implements the interface. Interfaces can also define the exceptions throw. For instance, the classes which implement the following `FileServer` can *possibly* throw an exception in their methods `download` and `save`.

```
public interface FileServer {
    String download(String file) throws Exception;
    void save(String file, String string) throws Exception;
}
```

If an interface defines the `throws Exception` attributes for the methods -- i.e. the methods may throw an exception -- the classes which implement the interface must be defined in the same way. However, they do not have to throw an exception, as it becomes clear in the following example.

```
public class TextServer implements FileServer {

    private Map<String, String> data;

    public TextServer() {
        this.data = new HashMap<String, String>();
    }

    @Override
    public String download(String file) throws Exception {
        return this.data.get(file);
    }

    @Override
    public void save(String file, String string) throws Exception {
        this.data.put(file, stirng);
    }
}
```

46.3 THE EXCEPTION INFORMATION

The `catch` block tells how we handle an exception, and it tells us what exception we should be prepared for: `catch (Exception e)`. The exception information is saved into the `e` variable.

```
    try {
        // the code, which may throw an exception
    } catch (Exception e) {
        // the exception information is saved into the variable e
    }
```

The class `Exception` can provide useful methods. For instance, the method `printStackTrace()` prints a *path* which tells us where the exception came from. Let's check the following error printed by the method `printStackTrace()`.

```
Exception in thread "main" java.lang.NullPointerException
  at package.Class.print(Class.java:43)
  at package.Class.main(Class.java:29)
```

Reading the stack trace happens button up. The lowest is the first call, i.e. the program execution has started from the `main()` method of class `Class`. At line 29 of the main method of `Class`, we called the method `print()`. Line 43 of the method `print` caused a `NullPointerException`. Exception information are extremely important to find out the origin of a problem.

47. READING A FILE

A relevant part of programming is related to stored files, in one way or in another. Let's take the first steps in Java file handling. Java's API provides the class File, whose contents can be read using the already known Scanner class.

If we read the desciption of the `File` API we notice the `File` class has the constructor `File(String pathname)`, which *creates a new File instance by converting the given pathname string into an abstract pathname*. This means the `File` class constructor can be given the pathname of the file we want to open.

In the NetBeans programming environment, files have got their own tab called Files, which contains all our project files. If we add a file to a project root -- that is to say outside all folders -- we can refer to it by writing only the its name. We create a file object by fiving the file pathname to it as parameter:

```
    File file = new File("file-name.txt");
```

`System.in` input stream is not the only reading source we can give to the constructor of a Scanner class. For instance, the reading source can be a file, in addition to the user keyboard. Scanner provides the same methods to read a keyboard input and a file. In the following example, we open a file and we print all the text contained in the file using the `System.out.println` statement. At the end, we close the file using the statement `close`.

```
        // The file we read
        File file = new File("filename.txt");

        Scanner reader = new Scanner(file);
        while (reader.hasNextLine()) {
            String line = reader.nextLine();
            System.out.println(line);
        }
```

```
        reader.close();
```

The Scanner class constructor `public Scanner(File source)` (*Constructs a new Scanner that produces values scanned from the specified file.*) throws a `FileNotFoundException` when the specified file is not found. The `FileNotFoundException` is different than `RuntimeException`, and we have either to handle it or throw it forward. At this point, you only have to know that the programming environment tells you whether you have to handle the exception or not. Let's first create a try-catch block where we handle our file as soon as we open it.

```
public void readFile(File f) {
    // the file we read
    Scanner reader = null;

    try {
        reader = new Scanner(f);
    } catch (Exception e) {
        System.out.println("We couldn't read the file. Error: " + e.getMessage());
        return; // we exit the method
    }

    while (reader.hasNextLine()) {
        String line = reader.nextLine();
        System.out.println(line);
    }

    reader.close();
}
```

Another option is to delegate the exception handling responsibility to the method caller. We delegate the exception handling responsibility by adding the definition `throws ExceptionType` to the method. For instance, we can add `throws Exception` because the type of all exceptions is `Exception`. When a method has the attribute `throws Exception`, whatever chunk of code which calls that method knows that it may throw an exception, and it should be prepared for it.

```
public void readFile(File f) throws Exception {
    // the file we read
    Scanner reader = new Scanner(f);

    while (reader.hasNextLine()) {
        String line = reader.nextLine();
        System.out.println(line);
    }

    reader.close();
}
```

In the example, the method `readFile` receives a file as parameter, and prints all the file lines. At the end, the reader is closed, and the file is closed with it, too. The attribute `throws Exception` tells us that the method may throw an exception. Same kind of attributes can be added to all the methods that handle files.

Note that the `Scanner` object's method `nextLine` returns a string, but it does not return a new line at the end of it. If you want to read a file and still maintain the new lines, you can add a new line at the end of each line:

```
public String readFileString(File f) throws Exception {
    // the file we read
    Scanner reader = new Scanner(f);

    String string = "";

    while (reader.hasNextLine()) {
        String line = reader.nextLine();
        string += line;
        string += "\n";
    }

    reader.close();
    return string;
}
```

Because we use the `Scanner` class to read files, we have all Scanner methods available for use. For instance the method `hasNext()` returns the boolean value `true` if the file contains something more to read, and the method `next()` reads the following word and returns a `String` object.

The following program creates a `Scanner` object which opens the file `file.txt`. Then, it prints every fifth word of the file.

```
File f = new File("file.txt");
Scanner reader = new Scanner(f);

int whichNumber = 0;
while (reader.hasNext()) {
    whichNumber++;
    String word = reader.next();

    if (whichNumber % 5 == 0) {
        System.out.println(word);
    }
}
```

Below, you find the text contained in the file, followed by the program output.

```
Exception handling is the process of responding to the occurrence, during computati
```

```
process
occurrence,
-
requiring
changing
program
```

47.1 CHARACTER SET ISSUES

When we read a text file (or when we save something into a file), Java has to find out the character set used by the operating system. Knowledge of the character set is required both to save text on the computer harddisk in binary format, and to translate binary data into text.

There have been developed standard character sets, and "UTF-8" is the most common nowadays. UTF-8 character set contains both the alphabet letters of everyday use and more particular characters such as the Japanese kanji characters or the information need to read and save the chess pawns. From a simplified programming angle, we could think a character set both as a character-number hashmap and a number-character hashmap. The character-number hashmap shows what binary number is used to save each character into a file. The number-character hashmap shows how we can translate into characters the values we obtain reading a file.

Almost each operating system producer has also got their own standards. Some support and want to contribute to the use of open source standards, some do not. If you have got problems with the use of Scandinavian characters such as ä and ö (expecially Mac and Windows users), you can tell which character set you want to use when you create a `Scanner` object. In this course, we always use the the "UTF-8" character set.

You can create a Scanner object which to read a file which uses the UTF-8 character set in the following way:

```
File f = new File("examplefile.txt");
Scanner reader = new Scanner(f, "UTF-8");
```

Anther thing you can do to set up a character set is using an environment variable. Macintosh and Windows users can set up an the value of the environment variable `JAVA_TOOL_OPTIONS` to the string `-Dfile.encoding=UTF8`. In such case, Java always uses UTF-8 characters as a default.

Exercise 21: Printer

Create the class `Printer`, its constructor `public Printer(String fileName)` which receives a String standing for the file name, and the method `public void printLinesWhichContain(String word)` which prints the file lines which contain the parameter word (*lower and upper case make difference in this excercise; for instance, "test" and "Test" are not the considered the same*); the lines are printed in the same order as they are inside the file.

If the argument is an empty String, all of the file is printed.

If the file is not found, the constructor delegates the exception with no need for a try-catch statement; the constructor simply has to be defined in the following way:

```
public Printer {

    public Printer(String fileName) throws Exception {
```

```
        // ...
    }

    // ...
}
```

The file *textFile* has been place into the default package of your project to help the tests. When you define the file name of the constructor of Printer, you have to write `src/textfile.txt`. The file contains an extract of Kalevala, a Finnish epic poem:

```
Siinä vanha Väinämöinen
katseleikse käänteleikse
Niin tuli kevätkäkönen
näki koivun kasvavaksi
Miksipä on tuo jätetty
koivahainen kaatamatta
Sanoi vanha Väinämöinen
```

The following example shows what the program should do:

```
Printer printer = new Printer("src/textfile.txt");

printer.printLinesWhichContain("Väinämöinen");
System.out.println("-----");
printer.printLinesWhichContain("Frank Zappa");
System.out.println("-----");
printer.printLinesWhichContain("");
System.out.println("-----");
```

Prints:

```
Siinä vanha Väinämöinen
Sanoi vanha Väinämöinen
-----
-----
Siinä vanha Väinämöinen
katseleikse käänteleikse
Niin tuli kevätkäkönen
näki koivun kasvavaksi
Miksipä on tuo jätetty
koivahainen kaatamatta
Sanoi vanha Väinämöinen
```

In the project, you also find the whole Kalevala; the file name is `src/kalevala.txt`

Exercise 22: File Analysis

In this exercise, we create an application to calculate the number of lines and characters.

EXERCISE 22.1: NUMBER OF LINES

Create the class `Analysis` in the package `file`; the class has the constructor `public Analysis(File file)`. Create the method `public int lines()`, which returns the number of lines of the file the constructor received as parameter.

The method cannot be "disposable", that is to say it has to return the right value even though it is called different times in a raw. Note that after you create a Scanner object for a file and read its whole contents using `nextLine` method calls, you can't use the *same* scanner to read the file again!

Attention: if the tests report a *timeout*, it probably means that you haven't been reading the file at all, meaning that the `nextLine` method calls miss!

EXERCISE 22.2: NUMBER OF CHARACTERS

Create the method `public int characters()` in the class `Analysis`; the method returns the number of characters of the file the constructor received as parameter.

The method cannot be "disposable", that is to say it has to return the right value even though it is called different times in a raw.

You can decide yourself what to do if the constructor parameter file does not exist.

The file *testFile* has been place into the test package of your project to help the tests. When you define the file name of the constructor of Analysis, you have to write `test/testfile.txt`. The file contains the following text:

```
there are 3 lines, and characters
because line breaks are also
characters
```

The following example shows what the program should do:

```
File file = new File("test/testfile.txt");
Analysis analysis = new Analysis(file);
System.out.println("Lines: " + analysis.lines());
System.out.println("Characters: " + analysis.characters());
```

```
Lines: 3
Characters: 74
```

Exercise 23: Word Inspection

Create the class WordInspection, which allows for different kinds of analyses on words. Implement the class in the package `wordinspection`.

The Institute for the Languages of Finland (Kotimaisten kielten tutkimuskeskus, Kotus) has published online a list of Finnish words. In this exercise we use a modified version of that list, which can be found in the exercise source folder `src` with the name `wordList.txt`; the relative path is `"src/wordList.txt"`. Because the word list if quite long, in fact, a `shortList.txt` was created in the project for the tests; the file can be found following the path `"src/shortList.txt"`.

If you have problems with Scandinavian letters (Mac and Windows users) create your `Scanner` object assigning it the "UTF-8" character set, in the following way: `Scanner reader = new Scanner(file, "UTF-8");` Problems come expecially when the tests are executed.

EXERCISE 23.1: WORD COUNT

Create the constructor `public WordInspection(File file)` to your WordInspection class. The constructor creates a new WordInspection object which inspects the given file.

Create the method `public int wordCount()`, which counts the file words and prints their number. In this part, you don't have to do anything with the words, you should only count how many there are. For this exercise, you can expect there is only one word in each row.

EXERCISE 23.2: Z

Create the method `public List<String> wordsContainingZ()`, which returns all the file words which contain a z; for instance, jazz and zombie.

EXERCISE 23.3: ENDING L

Create the method `public List<String> wordsEndingInL()`, which returns all the Finnish words of the file which end in l; such words are, for instance, kannel and sammal.

Attention! If you read the file various different times in your program, you notice that your code contains a lot of copy-paste, so far. It would be useful to think whether it would be possible to read the file in an different place, maybe inside the constructor or as a method, which the constructor calls. In such case, the methods could use a list which was read before and then create a new list which suits their search criteria. In week 12, we will come back again with an ortodox way to eliminate copy-paste.

EXERCISE 23.4: PALINDROMES

Create the method `public List<String> palindromes()`, which returns all the palindrome words of the file. Such words are, for instance, ala and enne.

EXERCISE 23.5: ALL VOWELS

Create the method `public List<String> wordsWhichContainAllVowels()`, which returns all the words of the file which contain all Finnish vowels (aeiouyäö). Such words are, for instance, myöhäiselokuva and ympäristönsuojelija.

48. HASHMAPS AND SETS

48.1 MANY VALUES AND ONE KEY

As we remember, we can save only one value per key using HashMap. In the following examples we save people's mobile phone numbers in a HashMap.

```
Map<String, String> phoneNumbers = new HashMap<String, String>();

phoneNumbers.put("Pekka", "040-12348765");

System.out.println( "Pekka's number: "+ phoneNumbers.get("Pekka") );

phoneNumbers.put("Pekka", "09-111333");

System.out.println( "Pekka's number: "+ phoneNumbers.get("Pekka") );
```

as expected, the output tells us:

```
Pekka's number: 040-12348765
Pekka's number: 09-111333
```

What about if we wanted to save various different values per one key, what about if a person had many phone numbers? Can we manage with an HashMap? Of course! For instance, instead of saving Strings as HashMap values we could save ArrayLists, mapping more than one object to one key. Let's change the way we save phone numbers as follows:

```
Map<String, ArrayList<String>> phoneNumbers = new HashMap<String, ArrayList<Strin
```

Now, a list is mapped to each HashMap key. Even though the new command creates a HashMap, the list which will be saved inside has to be created separately. In the following example, we add two numbers to the HashMap for Pekka, and we print them:

```
Map<String, ArrayList<String>> phoneNumbers = new HashMap<String, ArrayList<Strin

// We map an empty ArrayList to Pekka
phoneNumbers.put( "Pekka", new  ArrayList<String>() );

// we add Pekka's number to the list
phoneNumbers.get("Pekka").add("040-12348765");
```

```
// and we add a second phone number
phoneNumbers.get("Pekka").add("09-111333");

System.out.println( "Pekka's numbers: "+ phoneNumbers.get("Pekka") );
```

Prints

```
Pekka's numbers: [040-12348765, 09-111333]
```

We define the phone number type as `Map<String, ArrayList<String>>`, that is a Map whose key is a String and whose value is a list containing strings. The concrete implementation -- that is to say the created object -- was a HashMap. We could have defined a variable also in the following way:

```
Map<String, List<String>> phoneNumbers = new HashMap<String, List<String>>();
```

Now, the variable type is a Map, whose key is a String and value is a `List` containing strings. In fact, a List is an interface which defines the List functionality, and ArrayLists implement this interface, for instance. The concrete object is a HashMap.

The values we save into the HashMap are concrete object which implement the interface `List<String>`, ArrayLists, for instance. Again, we can add values to the HashMap in the following way:

```
// first, we map an empty ArrayList to Pekka
phoneNumbers.put( "Pekka", new  ArrayList<String>() );

// ...
```

In the future, instead of using concrete classes (such as `HashMap` and `ArrayList`, for instance), we will always try to use their respective interfaces `Map` and `List`.

48.2 SETS

Differently from lists, in a `Set` there can be up to one same entry, that is to say the same object can not be contained twice in a set. The similarity between two objects is inspected using the methods `equals` and `hashCode`.

One of the classes which implement the `Set` interface is `HashSet`. Let's use it to implement the class `ExerciseAccounting`, which allows us to keep an account of the exercise we do and to print them. Let's suppose the the exercises are always integers.

```
public class ExerciseAccounting {
    private Set<Integer> doneExercises;

    public ExerciseAccounting() {
        this.doneExercises = new HashSet<Integer>();
    }
```

```
    public void add(int exercise) {
        this.doneExercises.add(exercise);
    }

    public void print() {
        for (int exercise: this.doneExercises) {
            System.out.println(exercise);
        }
    }
}
```

```
        ExerciseAccounting account = new ExerciseAccounting();
        account.add(1);
        account.add(1);
        account.add(2);
        account.add(3);

        account.print();
```

```
1
2
3
```

The solution above is useful if we don't need information about the exercises done by each different user. We can change the saving logic of the exercises in a way to have them save in relation to each user, using a HashMap. The users are recognized through a unique string (for instance, their student number), and each user has their own set of finished exercises.

```
public class ExerciseAccounting {
    private Map<String, Set<Integer>> doneExercises;

    public ExerciseAccounting() {
        this.doneExercises = new HashMap<String, Set<Integer>>();
    }

    public void add(String user, int exercise) {
        // note that when we create a new user we have first to map an empty exerci
        if (!this.doneExercises.containsKey(user)) {
            this.doneExercises.put(user, new HashSet<Integer>());
        }

        // first, we retrieve the set containing the user's exercises and then we a
        Set<Integer> finished = this.doneExercises.get(user);
        finished.add(exercise);

        // the previous would have worked out without helping variable in the follo
        //  this.doneExercises.get(user).add(exercise);
    }
```

```
    public void print() {
        for (String user: this.doneExercises.keySet()) {
            System.out.println(user + ": " + this.doneExercises.get(user));
        }
    }
}
```

```
        ExerciseAccounting accounting = new ExerciseAccounting();
        accounting.add("Mikael", 3);
        accounting.add("Mikael", 4);
        accounting.add("Mikael", 3);
        accounting.add("Mikael", 3);

        accounting.add("Pekka", 4);
        accounting.add("Pekka", 4);

        accounting.add("Matti", 1);
        accounting.add("Matti", 2);

        accounting.print();
```

```
Matti: [1, 2]
Pekka: [4]
Mikael: [3, 4]
```

Note that the user names are not printed in order, in our example. This depends on the saving process of the `HashMap` entries, which happens through the value returned by the `hashCode` method, and does not involve the entry order in any way.

Exercise 24: Multiple Entry Dictionary

Let's make an extended version of the dictionary of week 1. Your task is to implement the class `PersonalMultipleEntryDictionary`, which can save one or more entry for each word translated. The class has to implement the interface in the exercise source, `MultipleEntryDictionary`, with the following methods:

- `public void add(String word, String entry)`

, which adds a new entry to a word, maintaining the old ones

- `public Set<String> translate(String word)`

, which returns a `Set` object, with all the entries of the word, or a `null` reference, if the word is not in the dictionary

- `public void remove(String word)`

, which removes a word and all its entries from the dictionary

As for the ExampleAccounting above, it's good to store the translations into a `Map<String, Set<String>>` object variable.

The interface code:

```
package dictionary;

import java.util.Set;

public interface MultipleEntryDictionary {
    void add(String word, String translation);
    Set<String> translate(String word);
    void remove(String word);
}
```

An example program:

```
    MultipleEntryDictionary dict = new PersonalMultipleEntryDictionary();
    dict.add("kuusi", "six");
    dict.add("kuusi", "spruce");

    dict.add("pii", "silicon");
    dict.add("pii", "pi");

    System.out.println(dict.translate("kuusi"));
    dict.remove("pii");
    System.out.println(dict.translate("pii"));
```

Prints:

```
[six, spruce]
null
```

Exercise 25: Duplicate Remover

Your task is to implement inside the package `tools` a class `PersonalDuplicateRemover`, which stores the given characterStrings so that equal characterStrings are removed (a.k.a duplicates). Class also holds a record of the amount of duplicates. Class should implement the given interface `DuplicateRemover`, which has the following methods:

- `public void add(String characterString)`

stores a characterString if it's not a duplicate.

- `public int getNumberOfDetectedDuplicates()`

returns the number of detected duplicates.

- `public Set<String> getUniqueCharacterStrings()`

returns an object which implements the interface `Set<String>`. Object should have all unique characterStrings (no duplicates!). If there are no unique characterStrings, method returns an empty set.

- `public void empty()`

removes stored characterStrings and resets the amount of detected duplicates.

Code of the interface:

```java
package tools;

import java.util.Set;

public interface DuplicateRemover {
    void add(String characterString);
    int getNumberOfDetectedDuplicates();
    Set<String> getUniqueCharacterStrings();
    void empty();
}
```

Interface can be used like this for example:

```java
    public static void main(String[] args) {
        DuplicateRemover remover = new PersonalDuplicateRemover();
        remover.add("first");
        remover.add("second");
        remover.add("first");

        System.out.println("Current number of duplicates: " +
            remover.getNumberOfDetectedDuplicates());

        remover.add("last");
        remover.add("last");
        remover.add("new");

        System.out.println("Current number of duplicates: " +
            remover.getNumberOfDetectedDuplicates());

        System.out.println("Unique characterStrings: " +
            remover.getUniqueCharacterStrings());

        remover.empty();

        System.out.println("Current number of duplicates: " +
            remover.getNumberOfDetectedDuplicates());

        System.out.println("Unique characterStrings: " +
            remover.getUniqueCharacterStrings());
    }
```

Code above would print: (order of characterStrings can change, it doesn't matter)

```
Current number of duplicates: 1
Current number of duplicates: 2
Unique characterStrings: [first, second, last, new]
Current number of duplicates: 0
Unique characterStrings: []
```

48.3 ONE OBJECT IN MANY LISTS, A MAP CONSTRUCTION OR A SET

As we remember, object variable are reference-type, which means that the variable does not memorize the object itself, but the reference to the object. Respectively, if we put an object into an ArrayList, for instance, the List does not memorize the object itself but the `reference` to the object. There is no reason why we should not be able to save an object in various different lists or HashMaps, for instance.

Let's have a look at our library example, which saves books into HashMaps, both based on their writer and ISB number. In addition to this, the library. Moreover, the library has two lists for the books on loan and for the ones that are on the shelves.

```
public class Book {
    private String ISBN;
    private String writer;
    private String name;
    private int date;
    // ...
}

public class Library {
    private Map<String, Book> ISBNBooks;
    private Map<String, List<String>> writerBooks;
    private List<Book> loanBooks;
    private List<Book> shelfBooks;

    public void addBook(Book newBook){
        ISBNBooks.put(newBook.getIsbn(), newBook);
        writerBooks.get(newBook.getWriter()).add(newBook);
        shelfBooks.add(newBook);
    }

    public Book getBookBasedOnISBN(String isbn){
        return ISBNBooks.get(isbn);
    }

    // ...
}
```

If an object is listed in different places at the same time (in a list, a set, or a map construction), you have to pay particular attention so to make sure the state of the different collections is consistent. For instance, if we decide to delete a book, it must be deleted from both maps as well as from the two lists which contain the books on loan and on the shelves.

Exercise 26: Phone Search

Attention: you can create only one Scanner object so that your tests would work well. Also, do not use static variables, the tests execute your program many different times, and the static variable values left from the previous execution would possibly disturb them!

Let's create an application to manage people phone numbers and addresses.

The exercise can be worth 1-5 points. To receive one point, you should implement the following functionality:

- 1 adding a phone number to the relative person
- 2 phone number search by person

to receive two points we also require

- 3 name search by phone number

to receive three points also

- 4 adding an address to the relative person
- 5 personal information search (search for a person's address and phone number)

if you want to receive four points, also implement

- 6 removing a person's information

and to receive all the points:

- 7 filtered search by keyword (retrieving a list which must be sorted by name in alphabetic order), the keyword can appear in the name or address

An example of how the program works:

```
phone search
available operations:
 1 add a number
 2 search for a number
 3 search for a person by phone number
 4 add an address
 5 search for personal information
 6 delete personal information
 7 filtered listing
 x quit

command: 1
whose number: pekka
number: 040-123456

command: 2
whose number: jukka
  not found

command: 2
whose number: pekka
 040-123456
```

```
command: 1
whose number: pekka
number: 09-222333

command: 2
whose number: pekka
 040-123456
 09-222333

command: 3
number: 02-444123
 not found

command: 3
number: 09-222333
 pekka

command: 5
whose information: pekka
  address unknown
  phone numbers:
   040-123456
   09-222333

command: 4
whose address: pekka
street: ida ekmanintie
city: helsinki

command: 5
whose information: pekka
  address: ida ekmanintie helsinki
  phone numbers:
   040-123456
   09-222333

command: 4
whose address: jukka
street: korsontie
city: vantaa

command: 5
whose information: jukka
  address: korsontie vantaa
  phone number not found

command: 7
keyword (if empty, all listed): kk

 jukka
  address: korsontie vantaa
  phone number not found

 pekka
```

```
  address: ida ekmanintie helsinki
  phone numbers:
   040-123456
   09-222333

command: 7
keyword (if empty, all listed): vantaa

 jukka
  address: korsontie vantaa
  phone number not found

command: 7
keyword (if empty, all listed): seppo
 keyword not found

command: 6
whose information: jukka

command: 5
whose information: jukka
  not found

command: x
```

Some remarks:

- Because of the tests, it is essential that the *user interface* works exactly as in the example above. The application can optionally decide in which way invalid inputs are handled. The tests contain only valid inputs.
- ***The program has to start when the main method is executed; you can only create one Scanner object.***
- Do not use static variables, the tests execute your program many different times, and the static variable values left from the previous execution would possibly disturb them!
- In order to make things easier, we assume the name is a single string; if we want to print our lists sorted by surname in alphabetic order, the name has to be given in the form *surname name*.
- A person can have more than one phone number and address. However, these are not necessarily stored.
- If a person is deleted, no search should retrieve them.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

TIETOJENKÄSITTELYTIETEEN LAITOS
INSTITUTIONEN FÖR DATAVETENSKAP
DEPARTMENT OF COMPUTER SCIENCE