Object-Oriented Programming with Java, part II ››

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

39. OBJECT

In our course, we have been using frequently the method `public String toString()` when we wanted to print an object in the shape of a string. Calling the method without setting it up properly does usually cause an error. We can have a look at the class `Book`, which does not contain the method `public String toString()` yet, and see what happens when the program uses the method `System.out.println()` and tries to print an object of `Book` class.

```
public class Book {
    private String name;
    private int publishingYear;

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

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

```
    }

    public int getPublishingYear() {
        return this.publishingYear;
    }
}
```

```
Book objectBook = new Book("Object book", 2000);
System.out.println(objectBook);
```

if we take an object of `Book` class and use it as the parameter of the method `System.out.println()`, our program does not print an error message. Our program does not crash, and instead of reading an error message, we notice an interesting print output. The print output contains the name of the class, `Book`, plus an indefinite String which follows a @ character. Notice that when we call `System.out.println(objectBook)` Java calls `System.out.println(objectBook.toString())`, in fact, but this does not cause an error.

The explanation is related to the way Java classes are built. Each Java class automatically *inherits* the `Object` class, which contains a set of methods that are useful to each Java class. Heritage means that our class has access to the features and functions defined in the inherited class. Among the others, the class `Object` contains the method `toString`, which is inherited by the classes we create.

The `toString` method inherited from the object class is not usually the one we'd want. That's why we will want to *replace* it with one we make personally. Let us add the method `public String toString()` to our `Book` class. This method will replace the `toString` method inherited from the `Object` class.

```
public class Book {
    private String name;
    private int publishingYear;

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

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

    public int getPublishingYear() {
        return this.publishingYear;
    }

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

```
}
```

If now we create an object instance, and we set it into the print method, we notice that the `toString` method of the `Book` class produces a string.

```
Book objectBook = new Book("Object book", 2000);
System.out.println(objectBook);
```

```
Object book (2000)
```

Above the `toString` method of class `Book` we see the `@Override` *annotation*. We use annotations to give guidelines to both the translator and the reader about how to relate to the methods. The `@Override` annotation tells that the following method replaces the one defined inside the inherited class. If we don't add an annotation to the method we replace, the translator gives us a *warning*, however avoiding writing annotations is not a mistake.

There are also other useful methods we inherit from the `Object` class. Let us now get acquainted with the methods `equals` and `hashCode`.

39.1 EQUALS METHOD

The `equals` method is used to compare two objects. The method is particularly used when we compare two `String` objects.

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

System.out.print("Write password: ");
String password = reader.nextLine();

if (password.equals("password")) {
    System.out.println("Right!");
} else {
    System.out.println("Wrong!");
}
```

```
Write password: mightycarrot
Wrong!
```

The `equals` method is defined in the `Object` class, and it makes sure that both the parameter object and the compared object have the same reference. In other words, by default the method makes sure that we are dealing with *one* unique object. If the reference is the same, the method returns

`true`, otherwise `false`. The following example should clarify the concept. The class `Book` doesn't implement its own `equals` method, and therefore it uses the one created by the `Object` class.

```
Book objectBook = new Book("Objectbook", 2000);
Book anotherObjectBook = objectBook;

if (objectBook.equals(objectBook)) {
    System.out.println("The books were the same");
} else {
    System.out.println("The books were not the same");
}

// Now we create an object with the same contents, which is however a different, in
anotherObjectBook = new Book("Objectbook", 2000);

if (objectBook.equals(anotherObjectBook)) {
    System.out.println("The books were the same");
} else {
    System.out.println("The books were not the same");
}
```

Print output:

```
The books were the same
The books were not the same
```

Even if the internal structure of both `Book` objects (i.e. the object variable values) is exactly the same, only the first comparison prints "`The books were the same`". This depends on the fact that only in the first case also the references were the same, i.e. we were comparing an object with itself. In the second example, we had two different objects even though they both had the same values.

When we use the `equals` method to compare strings, it works as we want it to: it identifies two strings as equal if the have the same *contents* even though they are two different objects. In fact, the default `equals` method is replaced with a new implementation in the String class.

We want that book comparison happened against name and year. We replace the `equals` method in the `Object` class with an implementation in the `Book` class. The `equals` method has to make sure whether the object is the same as the one received as parameter. First, we define a method according to which all the objects are the same.

```
    public boolean equals(Object object) {
        return true;
    }
```

Our method is a little too optimistic, so let us change its functionality slightly. Let us define that the objects are not the same if the parameter object is *null* or if the the two object types are different. We can find out the type of an object with the method `getClass()` (which is denifed in the `oject` class). Otherwise, we expect that the objects are the same.

```
public boolean equals(Object object) {
    if (object == null) {
        return false;
    }

    if (this.getClass() != object.getClass()) {
        return false;
    }

    return true;
}
```

The `equals`method finds out the class difference of two objects, but it is not able to distinguish two similar objects from each other. In order to compare our object with the object we received as parameter, and whose reference is `Object` type, we have to change the type of the Object reference. The reference type can be changed if and only if the object type is really such as we are converting it into. Type casting happens by specifying the desired class within brackets on the right side of the assignment sentence:

```
WantedType variable = (WantedType) oldVariable;
```

Type casting is possible because we know two objects are the same type. If they are different type, the above `getClass` method returns false. Let us change the `Object` parameter received with the `equals` method into `Book` type, and let us identify two different books against their publishing year. The books are otherwise the same.

```
public boolean equals(Object object) {
    if (object == null) {
        return false;
    }

    if (getClass() != object.getClass()) {
        return false;
    }

    Book compared = (Book) object;

    if(this.publishingYear != compared.getPublishingYear()) {
        return false;
    }

    return true;
}
```

Now, our comparison method is able to distinguish books against their publishing year. Wa want to check still that our book names are the same, and our own book name is not *null*.

```
    public boolean equals(Object object) {
        if (object == null) {
            return false;
        }

        if (getClass() != object.getClass()) {
            return false;
        }

        Book compared = (Book) object;

        if (this.publishingYear != compared.getPublishingYear()) {
            return false;
        }

        if (this.name == null || !this.name.equals(compared.getName())) {
            return false;
        }

        return true;
    }
```

Excellent, we have got a method for comparison which works, finally! Below is our `Book` class as it looks like at the moment.

```
public class Book {
    private String name;
    private int publishingYear;

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

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

    public int getPublishingYear() {
        return this.publishingYear;
    }

    @Override
    public String toString() {
        return this.name + " (" + this.publishingYear + ")";
    }

    @Override
    public boolean equals(Object object) {
        if (object == null) {
            return false;
```

```
        }

        if (getClass() != object.getClass()) {
            return false;
        }

        Book compared = (Book) object;

        if (this.publishingYear != compared.getPublishingYear()) {
            return false;
        }

        if (this.name == null || !this.name.equals(compared.getName())) {
            return false;
        }

        return true;
    }
}
```

Now, our book comparison returns `true`, if the book contents are the same.

```
Book objectBook = new Book("Objectbook", 2000);
Book anotherObjectBook = new Book("Objectbook", 2000);

if (objectBook.equals(anotherObjectBook)) {
    System.out.println("The books are the same");
} else {
    System.out.println("The books are not the same");
}
```

```
The books are the same
```

39.1.1 EQUALS AND ARRAYLIST

Various different Java made-up methods make use of the `equals` method to implement their search functionality. For instance, the `contains` mehod of class `ArrayList` compares objects through the `equals` method. Let us continue to use the `Book` class we defied for our examples. If our objects do not implement the `equals` method, we can't use the `contains` method, for instance. Try out the code below in two different `book` classes. The first class implements the `equals` method, the other does not.

```
ArrayList<Book> books = new ArrayList<Book>();
Book objectBook = new Book("Objectbook", 2000);
books.add(objectBook);
```

```
if (books.contains(objectBook)) {
    System.out.println("The object book was found.");
}

objectBook = new Book("Objectbook", 2000);

if (!books.contains(objectBook)) {
    System.out.println("The object book was not found.");
}
```

39.2 HASHCODE METHOD

The `hashCode` method takes an object and returns a numeric value, i.e. a hash value. We need numeric values for instance when we use and object as HashMap keys. So far, we have been using only String and Integer objects as HashMap keys, and their `hashCode` method is implemented by default. Let us make an example where it is not so: let us continue with our book examples and let us start to take note of our books on loan. We want to implement our bookkeeping through Hashmap. The key is the book, and the book's value is a string, which tells the loaner's name:

```
HashMap<Book, String> loaners = new HashMap<Book, String>();

Book objectbook = new Book("Objectbook", 2000);
loaners.put( objectbook, "Pekka" );
loaners.put( new Book("Test Driven Development",1999), "Arto" );

System.out.println( loaners.get( objectbook ) );
System.out.println( loaners.get( new Book("Objectbook", 2000) ) );
System.out.println( loaners.get( new Book("Test Driven Development", 1999) ) );
```

Print output:

```
Pekka
null
null
```

We can find the loaner by searching against the same object which was given as HashMap key with the `put` method. However, if our search item is the same book but a different object, we are not able to find its loaner and we are retured with a *null* reference. This is again due to the default implementation of the `hashCode` method of `Object` class. The default implementation creates an index based on the reference; this means that different objects with the same content receive different hashCode method outputs, and therefore it is not possible to find the right place of the object in the HashMap.

To be sure the HashMap worked in the way we want - i.e. it returned the loaner when the key is an object with the right *content* (not necessarily the same object as the original value) - the class

which works as key must overwrite both the `equals` method and the `hashCode` method. The method must be overwritten in such a way, so that it would assign the same numeric value to all objects which have the same content. Some objects with different content may eventually be assigned the same hashCode; however, different content objects should be assigned the same hashCode as rarely as possible, if we want our HashMap to be efficient.

Previously, we have successfully used `String` objects as HashMap keys, and we can therefore say that the `String` class has a `hashCode` implementation which works as expected. Let us *delegate* the calculation to the `String` object.

```
public int hashCode() {
    return this.name.hashCode();
}
```

The solution above is quite good; but if `name` is *null*, we are thrown a `NullPointerException`. We can fix this by setting the condition: if the value of the `name` variable is is *null*, return value 7. Seven is a value chosen casually, thirteen could have done as well.

```
public int hashCode() {
    if (this.name == null) {
        return 7;
    }

    return this.name.hashCode();
}
```

We can still improve the `hashCode` method by taking into consideration the book publishing year, in our calculations:

```
public int hashCode() {
    if (this.name == null) {
        return 7;
    }

    return this.publishingYear + this.name.hashCode();
}
```

An additional remark: the output of the hashCode method of HashMap key objects tells us their value slot in the hash construction, i.e. their index in the HashMap. You may now be wondering: "doesn't this lead to a situation where more than one object ends up with the same index in the HashMap?". The answer is yes and no. Even if the `hashCode` method gave the same value to two different objects, HashMaps are built in such way that various different obejcts may have the same index. In order to distinguish objects with the same index, the key objects of the HashMap must have implemented the `equals` method. You will find more information about Hashmap implementation in the course *Data Structures and Algorithms*.

The final `Book` class now.

```java
public class Book {

    private String name;
    private int publishingYear;

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

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

    public int getPublishingYear() {
        return this.publishingYear;
    }

    @Override
    public String toString() {
        return this.name + " (" + this.publishingYear + ")";
    }

    @Override
    public boolean equals(Object object) {
        if (object == null) {
            return false;
        }

        if (getClass() != object.getClass()) {
            return false;
        }

        Book compared = (Book) object;

        if (this.publishingYear != compared.getPublishingYear()) {
            return false;
        }

        if (this.name == null || !this.name.equals(compared.getName())) {
            return false;
        }

        return true;
    }

    public int hashCode() {
        if (this.name == null) {
            return 7;
        }

        return this.publishingYear + this.name.hashCode();
    }
```

```
}
```

Let us sum up everything again: in order to use a class as HashMap key, we have to define

- The `equals` method in a way that objects with the same content will return true when compared, whereas different-content objects shall return false
- The `hashCode` method in a way that it assigns the same value to all the objects whose content is regarded as similar

The equals and hashCode methods of our Book class fulfill these two conditions. Now, the problem we faced before is solved, and we can find out the book loaners:

```
        HashMap<Book, String> loaners = new HashMap<Book, String>();

        Book objectbook = new Book("Objectbook", 2000);
        loaners.put( objectbook, "Pekka" );
        loaners.put( new Book("Test Driven Development",1999), "Arto" );

        System.out.println( loaners.get( objectbook ) );
        System.out.println( loaners.get( new Book("Objectbook", 2000) ) );
        System.out.println( loaners.get( new Book("Test Driven Development", 1999) ) );
```

Print output:

```
Pekka
Pekka
Arto
```

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.

Exercise 9: Car Registration Centre

EXERCISE 9.1: REGISTRATION PLATE EQUALS AND HASHCODE

European registration plates are composed of two parts: the country ID -- one or two letters long -- and possibly a regitration code specific for the country, which in turn is composed of numbers and letters. Registaration plates are defined using the following class:

```
public class RegistrationPlate {
    // ATTENTION: the object variable types are final, meaning that their value c
    private final String regCode;
    private final String country;
```

```
    public RegistrationPlate(String regCode, String country) {
        this.regCode = regCode;
        this.country = country;
    }

    public String toString(){
         return country+ " "+regCode;
    }
}
```

We want to store the registration plates into say ArrayLists, using a HashMap as key. As mentioned before, it means we have to implement the methods `equals` and `hashCode` in their class, otherwise they can't work as we want.

Suggestion: take the equals and hashCode models from the Book example above. The registration plate hashCode can be created say combining the hashCodes of the country ID and of the registration code.

Example program:

```
    public static void main(String[] args) {
        RegistrationPlate reg1 = new RegistrationPlate("FI", "ABC-123");
        RegistrationPlate reg2 = new RegistrationPlate("FI", "UXE-465");
        RegistrationPlate reg3 = new RegistrationPlate("D", "B WQ-431");

        ArrayList<RegistrationPlate> finnish = new ArrayList<RegistrationPlate>()
        finnish.add(reg1);
        finnish.add(reg2);

        RegistrationPlate new = new RegistrationPlate("FI", "ABC-123");
        if (!finnish.contains(new)) {
            finnish.add(new);
        }
        System.out.println("Finnish: " + finnish);
        // if the equals method hasn't been overwritten, the same registration p

        HashMap<RegistrationPlate, String> owners = new HashMap<RegistrationPlate
        owners.put(reg1, "Arto");
        owners.put(reg3, "Jürgen");

        System.out.println("owners:");
        System.out.println(owners.get(new RegistrationPlate("FI", "ABC-123")));
        System.out.println(owners.get(new RegistrationPlate("D", "B WQ-431")));
        // if the hashCode hasn't been overwritten, the owners are not found
    }
```

If equals hashCode have been implemented well, the output should look like this:

```
Finnish: [FI ABC-123, FI UXE-465]
owners:
Arto
Jürgen
```

EXERCISE 9.2: THE OWNER, BASED OF THE REGISTRATION PLATE

Implement the class `VehicleRegister` which has the following methods:

- `public boolean add(RegistrationPlate plate, String owner)`, which adds the parameter owner of the car which corresponds to the parameter registration plate. The method returns true if the car had no owner; if the car had an owner already, the method returns false and it doesn't do anything
- `public String get(RegistrationPlate plate)`, which returns the car owner which corresponds to the parameter register number. If the car was not registered, it returns `null`
- `public boolean delete(RegistrationPlate plate)`, which delete the information connected to the parameter registration plate. The method returns true if the information was deleted, and false if there was no information connetted to the parameter in the register.

Attention: the vehicle register has to store the owner information into a `HashMap<RegistrationPlate, String> owners` object variable!

EXERCISE 9.3: MORE FOR THE VEHICLE REGISTER

Add still the following methods to your VehicleRegister:

- `public void printRegistrationPlates()`, which prints out all the registration plates stored
- `public void printOwners()`, which prints all the car owners stored. Each owner's name has to be printed only once, even though they had more than one car

40. INTERFACE

Interface is an instrument we have to define the functionality our classes should have. Interfaces are defined as normal Java classes, but instead of the definition "`public class ...`", we write "`public interface ...`". The interfaces influence class behaviour by defining the method names and return values, but they *do not contain method implementation*. The access modifier is not specified,

because it is always `public`. Let us have a look at the interface *Readable*, which defines whether an object can be read.

```
public interface Readable {
    String read();
}
```

The interface `Readable` defines the method `read()`, which returns a string object. The classes which implement an interface decide *in which way* the methods defined in the interface have to be implemented, in the end. A class implements an interface by adding the keyword *implements* between the class and the interface name. Below, we create the class `SMS` which implements `Readable` interface.

```
public class SMS implements Readable {
    private String sender;
    private String content;

    public SMS(String sender, String content) {
        this.sender = sender;
        this.content = content;
    }

    public String getSender() {
        return this.sender;
    }

    public String read() {
        return this.content;
    }
}
```

Because the class `SMS` implements the interface `Readable` (`public class SMS implements Readable`), the class `SMS` *must* implement the method `public String read()`. The implementations of methods defined in the interface must always have public access.

An interface is a behavioural agreement. In order to implement the behaviour, the class must implement the methods defined by the interface. The programmer of a class which implements an interface has to define what the behaviour will be like. Implementing an interface means to agree that the class will offer all the actions defined by the interface, i.e. the behaviour defined by the interface. A class which implements an interface but does not implement some of the interface methods can not exist.

Let us implement another class which implements the `Readable` interface, in addition to our `SMS` class. The class `EBook` is the electronic implementation of a book, and it contains the book name and page number. The EBook reads one page at time, and the `public String read()` method always returns the string of the following page.

```
public class EBook implements Readable {
    private String name;
```

```
    private ArrayList<String> pages;
    private int pageNumber;

    public EBook(String name, ArrayList<String> pages) {
        this.name = name;
        this.pages = pages;
        this.pageNumber = 0;
    }

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

    public int howManyPages() {
        return this.pages.size();
    }

    public String read() {
        String page = this.pages.get(this.pageNumber);
        nextPage();
        return page;
    }

    private void nextPage() {
        this.pageNumber = this.pageNumber + 1;
        if(this.pageNumber % this.pages.size() == 0) {
            this.pageNumber = 0;
        }
    }
}
```

Classes which implement interfaces generate objects as well as normal classes, and they can be used as ArrayList types too.

```
    SMS message = new SMS("ope", "Awesome stuff!");
    System.out.println(message.read());

    ArrayList<SMS> messages = new ArrayList<SMS>();
    messages.add(new SMS("unknown number", "I hid the body.");
```

```
Awesome stuff!
```

```
    ArrayList<String> pages = new ArrayList<String>();
    pages.add("Split your method into short clear chunks.");
    pages.add("Devide the user interface logic from the application logic.");
    pages.add("At first, always code only a small program which solves only a part of the problem."
    pages.add("Practice makes perfect. Make up your own fun project.");
```

```
EBook book = new EBook("Programming Hints.", pages);
for(int page = 0; page < book.howManyPages(); page++) {
    System.out.println(book.read());
}
```

```
Split your method into short clear chunks.
Divide the user interface logic from the application logic.
At first, always code only a small program which solves only a part of the problem.
Practice makes perfect. Make up your own fun project.
```

Exercise 10: NationalService

In the exercise layout, you find the premade interface `NationalService`, which contains the following operations:

- the method `int getDaysLeft()` which returns the number of days left on service
- the method `void work()`, which reduces the working days by one. The working days number can not become negative.

```
public interface NationalService {
    int getDaysLeft();
    void work();
}
```

EXERCISE 10.1: CIVILSERVICE

Create the class `CivilService` which implements your `NationalService` interface. The class constructor is without parameter. The class has the object variable daysLeft which is initialised in the constructor receiving the value 362.

EXERCISE 10.2: MILITARYSERVICE

Create the class `MilitaryService` which implements your `NationalService` interface. The class constructor has one parameter, defining the days of service (`int daysLeft`).

40.1 AN INTERFACE AS VARIABLE TYPE

When we create a new variable we always specify its type. There are two types of variable types: primitive-type variables (int, double, ...) and reference-type (all objects). As far as reference-type variables are concerned, their class has also been their type, so far.

```
String string = "string-object";
SMS message = new SMS("teacher", "Something crazy is going to happen");
```

The type of an object can be different from its class. For instance, if a class implements the interface `Readable`, its type is `Readable`, too. For instance, since the class `SMS` implements the interface `Readable`, it has two types: `SMS` and `Readable`.

```
SMS message = new SMS("teacher", "Awesome stuff!");
Readable readable = new SMS("teacher", "The SMS is Readable!");
```

```
ArrayList<String> pages = new ArrayList<String>();
pages.add("A method can call itself.");

Readable book = new EBook("Recursion Principles", pages);
for(int page = 0; page < ((EBook)book).howManyPages(); page++) {
    System.out.println(book.read());
}
```

Because an interface can be used as type, it is possible to create a list containing interface-type objects.

```
ArrayList<Readable> numberList = new ArrayList<Readable>();

numberList.add(new SMS("teacher", "never been programming before..."));
numberList.add(new SMS("teacher", "gonna love it i think!"));
numberList.add(new SMS("teacher", "give me something more challenging! :)"));
numberList.add(new SMS("teacher", "you think i can do it?"));
numberList.add(new SMS("teacher", "up here we send several messages each day"));

for (Readable readable: numberList) {
    System.out.println(readable.read());
}
```

The `EBook` class implements the interface `Readable`. However, notice that even though the type of the class `EBook` is an interface, `EBook` is not the type of all the classes which implement the `Readable` interface. It is possible to assign an `EBook` object to a `Readable` variable, but the assignment does not work in the opposite way without a particular type change.

```
Readable readable = new SMS("teacher", "The SMS is Readable!"); // works
SMS message = readable; // not possible
```

```
    SMS transformedMessage = (SMS) readable; // works
```

Type casting works if and only if the variable's type is really what we try to change it into. Type casting is not usually a best practice; one of the only cases where that is legitimate is in connection with the `equals` method.

40.2 AN INTERFACE AS METHOD PARAMETER

The real use of interfaces becomes clear when we use them for the type of a method parameter. Because interfaces can be used as variable type, they can be used in method calls as parameter type. For instance, the below method `print` of class `Printer` receives a `Readable` variable.

```
public class Printer {
    public void print(Readable readable) {
        System.out.println(readable.read());
    }
}
```

The real value of the `print` method of class `Printer` is that its parameter can be *whatever* class instance which implements our `Readable` interface. When we call the method of an object, the method will work regardless of the class of this object, as long as the object implements Readable.

```
    SMS message = new SMS("teacher", "Wow, this printer is able to print them, actually!");
    ArrayList<String> pages = new ArrayList<String>();
    pages.add("{3, 5} are the numbers in common between {1, 3, 5} and {2, 3, 4, 5}.");

    EBook book = new EBook("Introduction to University Mathematics.", pages);

    Printer printer = new Printer();
    printer.print(message);
    printer.print(book);
```

```
Wow, this printer is able to print them, actually!
{3, 5} are the numbers in common between {1, 3, 5} and {2, 3, 4, 5}.
```

Let us implement another `numberList` class, where we can add interesting readable stuff. The class has an `ArrayList` instance as object variable where we save things to read. We add items to our number list through the `add` method which receives a `Readable` variable as parameter.

```
public class NumberList {
    private ArrayList<Readable> readables;
```

```
    public NumberList() {
        this.readables = new ArrayList<Readable>();
    }

    public void add(Readable readable) {
        this.readables.add(readable);
    }

    public int howManyReadables() {
        return this.readables.size();
    }
}
```

Number lists are usually readable, so we can implement the `Readable` interface to the `NumberList` class. The number list `read` method reads all the objects of the `readables` list, and it adds them one by one to a string which is returned by the `read()` method.

```
public class NumberList implements Readable {
    private ArrayList<Readable> readables;

    public NumberList() {
        this.readables = new ArrayList<Readable>();
    }

    public void add(Readable readable) {
        this.readables.add(readable);
    }

    public int howManyReadables() {
        return this.readables.size();
    }

    public String read() {
        String read = "";
        for(Readable readable: this.readables) {
            read += readable.read() + "\n";
        }

        this.readables.clear();
        return read;
    }
}
```

```
    NumberList joelList = new NumberList();
    joelList.add(new SMS("matti", "have you already written the tests?"));
    joelList.add(new SMS("matti", "did you have a look at the submissions?"));

    System.out.println("Joel has " + joelList.howManyReadables() + " messages to read");
```

```
Joel has got 2 messages to read
```

Because the type of `NumberList` is `Readable`, we can add `NumberList` objects to our number list, too. In the example below, Joel has a lot of messages to read, luckily Mikael deals with it and reads the messages on behalf of Joel.

```
NumberList joelList = new NumberList();
for (int i = 0; i < 1000; i++) {
    joelList.add(new SMS("matti", "have you already written the tests?"));
}

System.out.println("Joel has " + joelList.howManyReadables() + " messages to read");
System.out.println("Let's delegate some reading to Mikael");

NumberList mikaelList = new NumberList();
mikaelList.add(joelList);
mikaelList.read();

System.out.println();
System.out.println("Joel has " + joelList.howManyReadables() + " messages to read");
```

```
Joel has 1000 messages to read
Let's delegate some reading to Mikael

Joel has 0 messages to read
```

The `read` method which is called in connection to Mikael's list parses all the `Readable` objects contained in the list, and calls their `read` method. At the end of each `read` method call the list is cleared. In other words, Joel's number list is cleared as soon as Mikael reads it.

At this point, there are a lot of references; it would be good to draw down the objects and try to grasp how the `read` method call connected to `mikaelList` works!

Exercise 11: Boxes and Things

EXERCISE 11.1: TOBESTORED

We need storage boxes when we move to a new apartment. The boxes are used to store different things. All the things which are stored in the boxes have to implement the following interface:

```
public interface ToBeStored {
    double weight();
```

```
}
```

Add the interface to your program. New interfaces are added almost in the same way as classes: you choose *new Java interface* instead of *new Java class*.

Create two classes which implement the interface `Book` and `CD`. Book receives its writer (String), name (String), and weight (double), all as parameter. CD's parameter contains its artist (String), title (String), and publishing year (int). All CDs weigh 0.1 kg.

Remember that the classes also have to implement the interface `ToBeStored`. The classes have to work in the following way:

```
public static void main(String[] args) {
    Book book1 = new Book("Fedor Dostojevski", "Crime and Punishment", 2);
    Book book2 = new Book("Robert Martin", "Clean Code", 1);
    Book book3 = new Book("Kent Beck", "Test Driven Development", 0.5);

    CD cd1 = new CD("Pink Floyd", "Dark Side of the Moon", 1973);
    CD cd2 = new CD("Wigwam", "Nuclear Nightclub", 1975);
    CD cd3 = new CD("Rendezvous Park", "Closer to Being Here", 2012);

    System.out.println(book1);
    System.out.println(book2);
    System.out.println(book3);
    System.out.println(cd1);
    System.out.println(cd2);
    System.out.println(cd3);
}
```

Print output:

```
Fedor Dostojevski: Crime and Punishment
Robert Martin: Clean Code
Kent Beck: Test Driven Development
Pink Floyd: Dark Side of the Moon (1973)
Wigwam: Nuclear Nightclub (1975)
Rendezvous Park: Closer to Being Here (2012)
```

Attention! The weight is not reported here.

EXERCISE 11.2: BOX

Create the class box, which has to store Things that implement the interface `ToBeStored`. The box receives as constructor the maximum weight, expressed in kilograms. The box can't be added more things than its maximum capacity allows for. The weight of the things contained in the box can never exceed the box maximum capacity.

The following example clarifies the box use:

```
    public static void main(String[] args) {
        Box box = new Box(10);

        box.add( new Book("Fedor Dostojevski", "Crime and Punishment", 2) ) ;
        box.add( new Book("Robert Martin", "Clean Code", 1) );
        box.add( new Book("Kent Beck", "Test Driven Development", 0.7) );

        box.add( new CD("Pink Floyd", "Dark Side of the Moon", 1973) );
        box.add( new CD("Wigwam", "Nuclear Nightclub", 1975) );
        box.add( new CD("Rendezvous Park", "Closer to Being Here", 2012) );

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

Printing:

```
Box: 6 things, total weight 4.0 kg
```

Note: because the weights are represented as double, the rounding can cause small mistakes. You don't need to care about it when you do the exercise.

EXERCISE 11.3: BOX WEIGHT

If you created the object variable `double weght` in your box which records the weight of your things, replace it now with a method which calculates the weight:

```
public class Box {
    //...

    public double weight() {
        double weight = 0;
        // it calculates the total weight of the things which had been stored
        return weight;
    }
}
```

When you need the box weight -- if you have to add a new thing, for instance -- you can simply call the method which calculates it.

In fact, the method could work well even though it retured the value of an object variable. However, we train a situation where you don't need to maintain the object variable explicitly, but you can calculate it when you need it. With the following exercise, the information stored in a box object variable would not necessarily work properly, however. Why?

EXERCISE 11.4: BOXES ARE STORED TOO!

Implementing the interface `ToBeStored` requires that the class has the method `double weight()`. In fact, we just added this method to Box. Boxes can be stored!

Boxes are objects where we can store object which implement the interface `ToBeStored`. Boxes also implement this interface. This means that **you can also put boxes inside your boxes!**

Try this out: create a couple of boxes in your program, put things into the boxes and put smaller boxes into the bigger ones. Try also what happens when you put a box into itself. Why does it happen?

40.3 AN INTERFACE AS METHOD RETURN VALUE

As well as any other variable type, an interface can also be used as method return value. Below you find `Factory`, which can be used to produce different objects that implement the interface `Item`. In the beginning, Factory produces books and disks at random.

```
public class Factory {
   public Factory(){
       // Attention: it is not necessary to write an empty constructor if there
   // In such cases, Java creates a default constructor, i.e a constructor witho
   }

    public Item produceNew(){
        Random random = new Random();
        int num = random.nextInt(4);
        if ( num==0 ) {
            return new CD("Pink Floyd", "Dark Side of the Moon", 1973);
        } else if ( num==1 ) {
            return new CD("Wigwam", "Nuclear Nightclub", 1975);
        } else if ( num==2 ) {
            return new Book("Robert Martin", "Clean Code", 1 );
        } else {
            return new Book("Kent Beck", "Test Driven Development", 0.7);
        }
    }
}
```

It is possible to use our Factory without knowing precisely what kind of classes are present in it, as long as they all implement Item. Below you find the class `Packer` which can be used to get a boxful of items. The Packer knows the factory which produces its Items:

```
public class Packer {
    private Factory factory;

    public Packer(){
```

```
            factory = new Factory();
        }

        public Box giveABoxful() {
            Box box = new Box(100);

            for ( int i=0; i < 10; i++ ) {
                Item newItem = factory.produceNew();
                box.add(newItem);
            }

            return box;
        }
    }
```

Because the packer doesn't know the classes which implement the Item interface, it is possble to add new classes which implement the interface without having to modify the packer. Below, we create a new class which implements our Item interface - `ChocolateBar`. Our Factory was modified to produce chocolate bars in addition to books and CDs. The class `Packer` works fine with the extended factory version, without having to change it.

```
public class ChocolateBar implements Item {
    // we don't need a constructor because Java is able to generate a default one

    public double weight(){
        return 0.2;
    }
}

public class Factory {
    // we don't need a constructor because Java is able to generate a default one

    public Item produceNew(){
        Random random = new Random();
        int num = random.nextInt(5);
        if ( num==0 ) {
            return new CD("Pink Floyd", "Dark Side of the Moon", 1973);
        } else if ( num==1 ) {
            return new CD("Wigwam", "Nuclear Nightclub", 1975);
        } else if ( num==2 ) {
            return new Book("Robert Martin", "Clean Code", 1 );
        } else if ( num==3 ) {
            return new Book("Kent Beck", "Test Driven Development", 0.7);
        } else {
            return new ChocolateBar();
        }
    }
}
```

Using interfaces while programming permits us to reduce the number of dependences among our classes. In our example, Packer is not dependent on the classes which implement Item interface, it is only dependent on the interface itself. This allows us to add classes wihout having to change the class Packer, as long as they implement our interface. We can even add classes that implement the interface to the methods which make use of our packer without compromising the process. In fact, less dependences make it easy to extend a program.

40.4 MADE-UP INTERFACES

Java API offers a sensible number of made-up interfaces. Below, we get to know some of Java's most used interfaces: `List`, `Map`, `Set` and `Collection`.

40.4.1 LIST

The List interface defines lists basic functionality. Because the class ArrayList implements the `List` interface, it can also be initialized through the `List` interface.

```
List<String> strings = new ArrayList<String>();
strings.add("A String object within an ArrayList object!");
```

As we notice from the List interface Java API, there are a lot of classes which implement the interface `List`. A list construction which is familiar to hakers like us is the linked list. A linked list can be used through the List interface in the same way as the objects created from ArrayList.

```
List<String> strings = new LinkedList<String>();
strings.add("A string object within a LinkedList object!");
```

Both implementations of the `List` interface work in the same way, in the user point of view. In fact, the interface *abstracts* their internal functionality. ArrayList and linkedList internal construction is evidently different, anyway. ArrayList saves the objects into a table, and the search is quick with a specific index. Differently, LinkedList builds up a list where each item has a reference to the following item. When we search for an item in a linked list, we have to go through all the list items till we reach the index.

When it comes to bigger lists, we can point out more than evident performance differences. LinkedList's strength is that adding new items is always fast. Differently, behind ArrayList there is a table which grows as it fills up. Increasing the size of the table means creating a new one and copying there the information of the old. However, searching against an index is extremely fast with an ArrayList, whereas we have to go thourgh all the list elements one by one before reaching the one we want, with a LinkedList. More information about data structures such as ArrayList and LinkedList internal implementation comes with the course *Data structures and algorithms*.

In our programming course you will rather want to choose ArrayList, in fact. Programming to interface is worth of it, anyway: implement your program so that you'll use data structures via interfaces.

40.4.2 MAP

The Map Interface defines HashMap basic fuctionality. Because HashMaps implement the `Map` interface, it is possible to initialize them trough the `Map` interface.

```
Map<String, String> translations = new HashMap<String, String>();
translations.put("gambatte", "tsemppiä");
translations.put("hai", "kyllä");
```

You get HashMap keys thourgh the method `keySet`.

```
Map<String, String> translations = new HashMap<String, String>();
translations.put("gambatte", "good luck");
translations.put("hai", "yes");

for(String key: translations.keySet()) {
    System.out.println(key + ": " + translations.get(key));
}
```

```
gambatte: good luck
hai: yes
```

The `keySet` method returns a set made of keys which implement `Set` interface. The set which implement the `Set` interface can be parsed with a for-each loop. HashMap values are retrieved through the `values` method, which returns a set of values which implement the `Collection` interface. We should now focus on Set and Collection interfaces.

40.4.3 SET

The Set interface defines the functionality of Java's sets. Java's sets always contain 0 or 1 element of a certain type. Among the others, `HashSet` is one of the classes which implement the Set interface. We can parse a key set through a for-each loop, in the following way

```
Set<String> set = new HashSet<String>();
set.add("one");
set.add("one");
set.add("two");

for (String key: set) {
    System.out.println(key);
}
```

```
one
two
```

Notice that HashSet is not concerned on the order of its keys.

40.4.4 COLLECTION

The Collection interface defines the functionality of collections. Among the others, Java's lists and sets are collections -- that is, List and Set interfaces implement the Collection interface. Collection interface provides methods to check object existence (the `contains` method) and to check the collection size (`size` method). We can parse any class which implements the Collection interface with a `for-each` loop.

We now create a HashMap and parse first its keys, and then its values.

```
Map<String, String> translations = new HashMap<String, String>();
translations.put("gambatte", "good luck");
translations.put("hai", "yes");

Set<String> keys = translations.keySet();
Collection<String> keySet = keys;

System.out.println("Keys:");
for(String key: keySet) {
    System.out.println(key);
}

System.out.println();
System.out.println("Values:");
Collection<String> values = translations.values();
for(String value: values) {
    System.out.println(value);
}
```

```
Keys:
gambatte
hai

Values:
yes
good luck
```

The following example would have produced the same output, too.

```
Map<String, String> translations = new HashMap<String, String>();
translations.put("gambatte", "good luck");
```

```
translations.put("hai", "yes");

System.out.println("Keys:");
for(String key: translations.keySet()) {
    System.out.println(key);
}

System.out.println();
System.out.println("Values:");
for(String value: translations.values()) {
    System.out.println(value);
}
```

In the following exercise we build an online shop, and we train to use classes through their interfaces.

Exercise 12: Online Shop

Next, we create some programming components which are useful to manage an online shop.

EXERCISE 12.1: STOREHOUSE

Create the class Storehouse with the following methods:

- `public void addProduct(String product, int price, int stock)`, adding to the storehouse a product whose price and number of stocks are the parameter values
- `public int price(String product)` returns the price of the parameter product; if the product is not available in the storehouse, the method returns -99

Inside the storehouse, the prices (and soon also the stocks) of the products have to be stored into a `Map<String, Integer>` variable! The type of the object so created can be `HashMap`, but you should use the interface `Map` for the variable type (see 40.4.2)

The next example clarifies storehouse use:

```
Storehouse store = new Storehouse();
store.addProduct("milk", 3, 10);
store.addProduct("coffee", 5, 7);

System.out.println("prices:");
System.out.println("milk:   " + store.price("milk"));
System.out.println("coffee:  " + store.price("coffee"));
System.out.println("sugar: " + store.price("sugar"));
```

Prints:

```
prices:
milk:  3
coffee:  5
sugar: -99
```

EXERCISE 12.2: PRODUCT STOCK

Store product stocks in a similar `Map<String, Integer>` variable as the one you used for their prices. Fill the Storehouse with the following methods:

- `public int stock(String product)` returns the stock of the parameter product.
- `public boolean take(String product)` decreases the stock of the parameter product by one, and it returns *true* if the object was available in the storehouse. If the product was not in the storehouse, the method returns *false*, the product stock cannot go below zero.

An example of how to use the storehouse now:

```
Storehouse store = new Storehouse();
store.addProduct("coffee", 5, 1);

System.out.println("stocks:");
System.out.println("coffee:  " + store.stock("coffee"));
System.out.println("sugar: " + store.stock("sugar"));

System.out.println("we take a coffee " + store.take("coffee"));
System.out.println("we take a coffee " + store.take("coffee"));
System.out.println("we take sugar " + store.take("sugar"));

System.out.println("stocks:");
System.out.println("coffee:  " + store.stock("coffee"));
System.out.println("sugar: " + store.stock("sugar"));
```

Prints:

```
stocks:
coffee:  1
sugar: 0
we take coffee true
we take coffee false
we take sugar false
stocks:
coffee:  0
sugar: 0
```

EXERCISE 12.3: LISTING THE PRODUCTS

Let's add another method to our storehouse:

- `public Set<String> products()` returns a name *set* of the products contained in the storehouse

The method can be implemented easily. Using the Map's method `keySet`, you can get the storehouse products by asking for their prices or stocks.

An example of how to use the storehose now:

```
Storehouse store = new Storehouse();
store.addProduct("milk", 3, 10);
store.addProduct("coffee", 5, 6);
store.addProduct("buttermilk", 2, 20);
store.addProduct("jogurt", 2, 20);

System.out.println("products:");
for (String product : store.products()) {
    System.out.println(product);
}
```

Prints:

```
products:
buttermilk
jogurt
coffee
milk
```

EXERCISE 12.4: PURCHASE

We add *purchases* to our shopping basket. With purchase we mean a specific number of a specific product. For instance, we can put into our shopping basket either a purchase corresponding to one bread, or a purchase corresponding to 24 coffees.

Create the class `Purchase` with the following functionality:

- a constructor `public Purchase(String product, int amount, int unitPrice)`, which creates a purchase corresponding the parameter product. The product unit amount of purchase is clarified by the parameter *amount*, and the third parameter is the *unit price*
- `public int price()`, which returns the purchase price. This is obtained by raising the unit amount by the unit price
- `public void increaseAmount()` increases by one the purchase unit amount
- `public String toString()` returns the purchase in a string form like the following

An example of how to use a purchase

```
Purchase purchase = new Purchase("milk", 4, 2);
System.out.println( "the total price of a purchase containing four milks is " + pur
System.out.println( purchase );
purchase.increaseAmount();
System.out.println( purchase );
```

Prints:

```
the total price of a purchase containing four milks is 8
milk: 4
milk: 5
```

Note: *toString* follows the pattern *product: amount*, but the price is not reported!

EXERCISE 12.5: SHOPPING BASKET

Finally, we can implement our shopping basket class!

The shopping basket stores its products as *Purchase objects*. The shopping basket has to have an object variable whose type is either `Map<String, Purchase>` or `List<Purchase>`. Do not create any other object variable for your shopping basket in addition to the Map or List needed to store purchases.

Attention: if you store a Purchase object in a Map helping variable, it will be useful to use the Map method values() in this and in the following exercise; with it, it's easy to go through all the stored Purchase objects.

Let's create a constructor without parameter for our shopping basket, as well as the following methods:

- `public void add(String product, int price)` adds a purchase to the shopping basket; the purchase is defined by the parameter product, and its price is the second parameter.
- `public int price()` returns the shopping basket total price

Example code of using basket

```
ShoppingBasket basket = new ShoppingBasket();
basket.add("milk", 3);
basket.add("buttermilk", 2);
basket.add("cheese", 5);
System.out.println("basket price: " + basket.price());
basket.add("computer", 899);
System.out.println("basket price: " + basket.price());
```

Prints:

```
basket price: 10
basket price: 909
```

EXERCISE 12.6: PRINTING OUT THE SHOPPING BASKET

Let's create the method `public void print()` for our shopping basket. This prints out the *Purchase* objects which are contained by the basket. The printing order is not important. The output of the shopping basket in the previous example would be:

```
butter: 1
cheese: 1
computer: 1
milk: 1
```

Note that the number stands for the unit amount of the products, not for their price!

EXERCISE 12.7: ONLY ONE PURCHASE OBJECT FOR ONE PRODUCT

Let's update our Shopping Basket; if the basket already contains the product which we add, we don't create a new Purchase object, but we update the Purchase object corresponding to the existing product by calling its method *increaseAmount()*.

Example:

```
ShoppingBasket basket = new ShoppingBasket();
basket.add("milk", 3);
basket.print();
System.out.println("basket price: " + basket.price() +"\n");

basket.add("buttermilk", 2);
basket.print();
System.out.println("basket price: " + basket.price() +"\n");

basket.add("milk", 3);
basket.print();
System.out.println("basket price: " + basket.price() +"\n");

basket.add("milk", 3);
basket.print();
System.out.println("basket price: " + basket.price() +"\n");
```

Prints:

```
milk: 1
basket price: 3
```

```
buttermilk: 1
milk: 1
basket price: 5

buttermilk: 1
milk: 2
basket price: 8

buttermilk: 1
milk: 3
basket price: 11
```

This means that first, we add milk and buttermilk, creating new Purchase objects for them. When we add more milk to the basket, we don't create a new purchase object for the milk, but we update the unit amount of the purchase object representing the milk we already have in the basket.

EXERCISE 12.8: SHOP

Now, all the parts of our online shop are ready. Our online shop has a storage house, which contains all products. We have got a shopping basket to manage all our customers. Whenever a customer chooses a purchase, we add it to the shopping basket if the product is available in our storage house. Meanwhile, the storage stocks are reduced by one.

Below, you find a ready-made code body for your online shop. Create the class `Shop` to your project, and copy the code below into it.

```
import java.util.Scanner;

public class Shop {

    private Storehouse store;
    private Scanner reader;

    public Shop(Storehouse store, Scanner reader) {
        this.store = store;
        this.reader = reader;
    }

    // the method to deal with a customer in the shop
    public void manage(String customer) {
        ShoppingBasket basket = new ShoppingBasket();
        System.out.println("Welcome to our shop " + customer);
        System.out.println("below is our sale offer:");

        for (String product : store.products()) {
            System.out.println( product );
        }

        while (true) {
```

```
            System.out.print("what do you want to buy (press enter to pay):");
            String product = reader.nextLine();
            if (product.isEmpty()) {
                break;
            }

            // here, you write the code to add a product to the shopping basket,
            // and decreases the storehouse stocks
            // do not touch the rest of the code!

        }

        System.out.println("your purchases are:");
        basket.print();
        System.out.println("basket price: " + basket.price());
    }
}
```

The following main program fills the shop storehouse and manages the customer Pekka:

```
Storehouse store = new Storehouse();
store.addProduct("coffee", 5, 10);
store.addProduct("milk", 3, 20);
store.addProduct("milkbutter", 2, 55);
store.addProduct("bread", 7, 8);

Shop shop = new Shop(store, new Scanner(System.in));
shop.manage("Pekka");
```

The shop is almost ready. There are comments in the method `public void manage(String customer)`, showing the part that you should implement. In that point, implement the code to check whether the object the customer wants is available in the storehouse. If so, reduce the storehouse stocks by one unit, and add the product to the shopping basket.

Now you have done something! verkkokauppa.com!

41. GENERICS

We speak about *Generics* in connection to the way classes can conserve objects of genric type. Generics is based on the generic type parameter which is used when we define a class, and which helps us to define the types that have to be chosen when an *object is created*. A class generics can be defined by setting up the number of type parameters we want. This number is written after the

class name and between the greater-than and less-than signs. We now implement our own generic class `Slot` which be assigned whatever object.

```
public class Slot<T> {
    private T key;

    public void setValue(T key) {
        this.key = key;
    }

    public T getValue() {
        return key;
    }
}
```

The definition `public class Slot<T>` tells us that we have to give a type parameter to the constructor of the class `Slot`. After the constructor call the object variables have to be the same type as what established with the call. We now create a slot which memorizes strings.

```
    Slot<String> string = new Slot<String>();
    string.setValue(":)");

    System.out.println(string.getValue());
```

```
:)
```

If we change the type parameter we can create different kinds of `Slot` ojects, whose purpose is to memorize objects. For instance, we can memorize an integer in the following way:

```
    Slot<Integer> num = new Slot<Integer>();
    num.setValue(5);

    System.out.println(slot.getValue());
```

```
5
```

An important part of Java data structures are programmed to be generic. For instance, ArrayList receives one parameter, HashMap two.

```
    List<String> string = new ArrayList<String>();
    Map<String, String> keyCouples = new HashMap<String, String>();
```

In the future, when you see the type `ArrayList<String>`, for instance, you know that its internal structure makes use of a generic type parameter.

41.1 THE INTERFACE WHICH MAKES USE OF GENERICS: COMPARABLE

In addition to normal interfaces, Java has interfaces which make use of generics. The internal value types of generic interfaces are defined in the same way as for generic classes. Let us have a look at Java made-up `Comparable` interface. The `Comparable` interface defines the `compareTo` method, which returns the place of `this` object, in relation to the parameter object (a negative number, 0, or a positive number). If `this` object is placed before the parameter object in the comparison order, the method returns a negative value, whereas it returns a positive value if it is placed after the parameter object. If the objects are placed at the same place in the comparison order, the method returns 0. With comparison order we mean the object order of magnitude defined by the programmer, i.e. the object order, when they are sorted with the sort method.

One of the advantages of the `Comparable` interface is that it allows us to sort a list of Comparable type keys by using the standard library method `Collections.sort`, for instance. `Collections.sort` uses the `compareTo` method of a key list to define in which order these keys should be. We call *Natural Ordering* this ordering technique which makes use of the `compareTo` method.

We create the class `ClubMember`, which depicts the young people and children who belong to the club. The members have to eat in order of height, so the club members will implement the interface `Comparable`. The interface `Comparable` also takes as type parameter the class which it is compared to. As type parameter, we use the `ClubMember` class.

```
public class ClubMember implements Comparable<ClubMember> {
    private String name;
    private int height;

    public ClubMember(String name, int height) {
        this.name = name;
        this.height = height;
    }

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

    public int getHeight() {
        return this.height;
    }

    @Override
    public String toString() {
        return this.getName() + " (" + this.getHeight() + ")";
    }

    @Override
```

```
    public int compareTo(ClubMember clubMember) {
        if(this.height == clubMember.getHeight()) {
            return 0;
        } else if (this.height > clubMember.getHeight()) {
            return 1;
        } else {
            return -1;
        }
    }
}
```

The interface requires the method `compareTo`, which returns an integer that tells us the comparison order. Our `compareTo()` method has to return a negative number if `this` object is smaller than its parameter object, or zero, if the two members are equally tall. Therefore, we can implement the above `compareTo` method, in the following way:

```
    @Override
    public int compareTo(ClubMember clubMember) {
        return this.height - clubMember.getHeight();
    }
```

Sorting club members is easy, now.

```
    List<ClubMember> clubMembers = new ArrayList<ClubMember>();
    clubMembers.add(new ClubMember("mikael", 182));
    clubMembers.add(new ClubMember("matti", 187));
    clubMembers.add(new ClubMember("joel", 184));

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

```
[mikael (182), matti (187), joel (184)]
[mikael (182), joel (184), matti (187)]
```

If we want to sort the members in descending order, we only have to switch the variable order in our `compareTo` method.

Exercise 13: Rich First, Poor Last

You find the pre-made class Person. People have got name and salary information. Make Person implement the `Comparable` interface, so that the `compareTo` method would sort the people according to their salary -- rich first, poor last.

Exercise 14: Students Sorted by Name

You find the pre-made class Student. Students have got a name. Make Student implement the `Comparable` interface, so that the `compareTo` method would sort the students in alphabetic order.

Tip: student names are Strings, the class String is `Comparable` itself. You can use the String's `compareTo` method when you implement your Student class. `String.compareTo` gives a different value to characters according to their case; because of this, String has also got the method `compareToIgnoreCase` which, in fact, ignores the case while comparing. You can use either of them, when you sort your students.

Exercise 15: Sorting Cards

Together with the exercise layout, you find a class whose objects represent playing cards. Cards have got a value and a suit. Card values are *2, 3, ..., 10, J, Q, K* and *A*, and the suits are *Spades, Hearts, Diamonds* and *Clubs*. Value and suit are however shown as integers in the objects. Cards have also got a toString method, which is used to print the card value and suit in a "friendly way".

Four constants -- that is `public static final` variables -- are defined in the class, so that the user didn't need to handle card's suits as numbers:

```
public class Card {
    public static final int SPADES  = 0;
    public static final int DIAMONDS  = 1;
    public static final int HEARTS = 2;
    public static final int CLUBS   = 3;

    // ...
}
```

Now, instead of writing the number 1, we can use the constant `Card.DIAMONDS` in our program. In the following example, we create three cards and print them:

```
Card first = new Card(2, Card.DIAMONDS);
Card second = new Card(14, Card.CLUBS);
Card third = new Card(12, Card.HEARTS);

System.out.println(first);
System.out.println(second);
System.out.println(third);
```

Prints:

```
2 of Diamonds
A of Clubs
Q of Hearts
```

Note: using constants as shown above is not the best way deal with things. Later on in the course we learn a better way to show suits!

EXERCISE 15.1: COMPARABLE CARDS

Make your Cards class Comparable. Implement the `compareTo` method so that cards would be sorted in ascending order according to their value. If the value of two classes have got the same values, we compare them against their suit in ascending order: *spades first, diamonds second, hearts third, and clubs last.*

The smallest card would then be the two spades and the greatest would be the clubs ace.

EXERCISE 15.2: HAND

Next, let's create the class `Hand` which represents the player hand set of cards. Create the following method to the hand:

- `public void add(Card card)` adds a card to the hand
- `public void print()` prints the cards in the hand following the below example pattern

```
Hand hand = new Hand();

hand.add( new Card(2, Card.SPADES) );
hand.add( new Card(14, Card.CLUBS) );
hand.add( new Card(12, Card.HEARTS) );
hand.add( new Card(2, Card.CLUBS) );

hand.print();
```

Prints:

```
2 of Spades
A of Clubs
Q of Hearts
2 of Clubs
```

Store the hand cards into an ArrayList.

EXERCISE 15.3: SORTING THE HAND

Create the method `public void sort()` for your hand, which sorts the cards in the hand. After being sorted, the cards are printed in order:

```
Hand hand = new Hand();

hand.add( new Card(2, Card.SPADES) );
hand.add( new Card(14, Card.CLUBS) );
hand.add( new Card(12, Card.HEARTS) );
hand.add( new Card(2, Card.CLUBS) );

hand.sort();

hand.print();
```

Prints:

```
2 of Spades
2 of Clubs
Q of Hearts
A of Clubs
```

EXERCISE 15.4: COMPARING HANDS

In one card game, the most valuable hand, where the sum of the cards value is the biggest. Modify the class `Hand` so that it could be compared according to this criterion: make it implement the interface `Comparable<Hand>`.

Below, you find an example of a program where we compare hands:

```
Hand hand1 = new Hand();

hand1.add( new Card(2, Card.SPADES) );
hand1.add( new Card(14, Card.CLUBS) );
hand1.add( new Card(12, Card.HEARTS) );
hand1.add( new Card(2, Card.CLUBS) );

Hand hand2 = new Hand();

hand2.add( new Card(11, Card.DIAMONDS) );
hand2.add( new Card(11, Card.CLUBS) );
hand2.add( new Card(11, Card.HEARTS) );

int comparison = hand1.compareTo(hand2);

if ( comparison < 0 ) {
  System.out.println("the most valuable hand contains the cards");
  hand2.print();
} else if ( comparison > 0 ){
  System.out.println("the most valuable hand contains the cards");
  hand1.print();
} else {
```

```
    System.out.println("the hands are equally valuable");
  }
```

Prints:

```
the most valuable hand contains the cards
J of Diamonds
J of Clubs
J of Hearts
```

EXERCISE 15.5: SORTING THE CARDS AGAINST DIFFERENT CRITERIA

What about if we wanted to sort cards in a slightly different way, sometimes; for instance, what about if we wanted to have all same-suit cards in a raw? The class can have only one compareTo method, which means that we have to find out other ways to sort cards against different orders.

If you want to sort your cards in optional orders, you can make use of different classes which execute the comparison. These classes have to implement the interface `Comparator<Card>`. The object which determines the sorting order compares two cards it receives as parameter. There is only one method, a compare(Card card1, Card card2) method which has to return a negative value if card1 is before card2, a positive value if card2 is before card1, and 0 otherwise.

The idea is creating a specific comparison class for each sorting order; for instance, a class which places same suit cards together in a row:

```
import java.util.Comparator;

public class SortAgainstSuit implements Comparator<Card> {
    public int compare(Card card1, Card card2) {
        return card1.getSuit()-card2.getSuit();
    }
}
```

Sorting against suit works in the same way as the card method `compareTo` thought for suits, that is *spades first, diamonds second, hearts third, clubs last.*

Sorting is still possible through the Collections' sort method. The method now receives as second parameter an object of the class that determines the sorting order:

```
  ArrayList<Card> cards = new ArrayList<Card>();

  cards.add( new Card(3, Card.CLUBS) );
  cards.add( new Card(2, Card.DIAMONDS) );
  cards.add( new Card(14, Card.CLUBS) );
  cards.add( new Card(12, Card.HEARTS) );
  cards.add( new Card(2, Card.CLUBS) );
```

```
SortAgainstSuit suitSorter = new SortAgainstSuit();
Collections.sort(cards, suitSorter );

for (Card c : cards) {
  System.out.println( c );
}
```

Prints:

```
2 of Diamonds
Q of Hearts
3 of Clubs
A of Clubs
2 of Clubs
```

The sorting object can also be created directly together with the sort call:

```
Collections.sort(cards, new SortAgainstSuit() );
```

Further information about comparator classes in here.

Create now the class `SortAgainstSuitAndValue` which implements the Comparator interface and sorts cards as it is done in the example above, plus same suit cards are also sorted according to their value.

EXERCISE 15.6: SORT AGAINST SUIT

Add the method `public void sortAgainstSuit()` to the class `Hand`; when the method is called the hand's cards are sorted according to the comparator `SortAgainstSuitAndValue`. After sorting them, the cards are printed in order:

```
Hand hand = new Hand();

hand.add( new Card(12, Card.HEARTS) );
hand.add( new Card(4, Card.CLUBS) );
hand.add( new Card(2, Card.DIAMONDS) );
hand.add( new Card(14, Card.CLUBS) );
hand.add( new Card(7, Card.HEARTS) );
hand.add( new Card(2, Card.CLUBS) );

hand.sortAgainstSuit();

hand.print();
```

Prints:

```
2 of Diamonds
7 of Hearts
Q of Hearts
2 of Clubs
4 of Clubs
A of Clubs
```

42. COLLECTIONS

The class library `Collections` is Java's general-purpose library for collection classes. As we can see, `Collections` provides methods to sort objects either through the interface `Comparable` or `Comparator`. In addition to sorting, we can use this class library to retrieve the minimum and maximum values (through the methods `min` and `max`, respectively), retrieve a specific value (`binarySearch` method), or reverse the list (`reverse` method).

42.1 SEARCH

The Collections class library provides a pre-made binary search functionality. The method `binarySearch()` returns the index of our searched key, if this is found. If the key is not found, the search algorithm returns a negative value. The method `binarySearch()` makes use of the Comparable interface to retieve objects. If the object's `compareTo()` method returns the value 0, i.e. if it is the same object, the key is considered found.

Our ClubMember class compares people's heights in its `compareTo()` method, i.e. we look for club members whose height is the same while we parse our list.

```
List<ClubMember> clubMembers = new ArrayList<ClubMember>();
clubMembers.add(new ClubMember("mikael", 182));
clubMembers.add(new ClubMember("matti", 187));
clubMembers.add(new ClubMember("joel", 184));

Collections.sort(clubMembers);

ClubMember wanted = new ClubMember("Name", 180);
int index = Collections.binarySearch(clubMembers, wanted);
if (index >= 0) {
    System.out.println("A person who is 180 centimeters tall was found at index " + index)
    System.out.println("name: " + clubMembers.get(index).getName());
}
```

```
wanted = new ClubMember("Name", 187);
index = Collections.binarySearch(clubMembers, wanted);
if (index >= 0) {
    System.out.println("A person who is 187 centimeters tall was found at index " + index)
    System.out.println("name: " + clubMembers.get(index).getName());
}
```

The print output is the following:

```
A person who is 187 centimeters tall was found at index 2
name: matti
```

Notice that we also called the method `Collections.sort()`, in our example. This is because binary search cannot be done if our table or list are not already sorted up.

Exercise 16: Ski Jumping

Once again, you can train to build the program structure yourself; the appearance of the user interface and its functionality are pre-defined.

Note: you can create only one Scanner object lest the tests fail. Also, no 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!

Ski jumping is a beloved sport for Finns; they attempt to land as far as possible down the hill below, in the most stylish way. In this exercise, you create a simulator for a ski jumping tournament.

First, the simulator asks the user for the jumper names. If the user inputs an empty string (i.e. presses enter), we move to the jumping phase. In the jumping phase, the jumpers jump one by one in reverse order according to their points. The jumper with the less points always jumps first, then the ones with more points, till the person with the most points.

The total points of a jumper are calculated by adding up the points from their jumps. Jump points are decided in relation to the jump length (use a random integer between 60-120) and judge decision. Five judges vote for each jump (a vote is a random number between 10-20). The judge decision takes into consideration only three judge votes: the smallest and the greatest votes are not taken into account. For instance, if Mikael jumps 61 meters and the judge votes are 11, 12, 13, 14, and 15, the total points for the jump are 100.

There are as many rounds as the user wants. When the user wants to quit, we print the tournament results. The tournament results include the jumpers, the jumper total points, and the lengths of the jumps. The final results are sorted against the jumper total points, and the jumper who received the most points is the first.

Among the other things, you will need the method `Collections.sort` and `Collections.reverse`. First, you should start to wonder what kind of classes and objects there could be in your program. Also, it would be good to arrive to a situation where your user interface is the only class with printing statements.

```
Kumpula ski jumping week

Write the names of the participants one at a time; an empty string brings you to
  Participant name: Mikael
  Participant name: Mika
  Participant name:

The tournament begins!

Write "jump" to jump; otherwise you quit: jump

Round 1

Jumping order:
  1. Mikael (0 points)
  2. Mika (0 points)

Results of round 1
  Mikael
    length: 95
    judge votes: [15, 11, 10, 14, 14]
  Mika
    length: 112
    judge votes: [14, 12, 18, 18, 17]

Write "jump" to jump; otherwise you quit: jump

Round 2

Jumping order:
  1. Mikael (134 points)
  2. Mika (161 points)

Results of round 2
  Mikael
    length: 96
    judge votes: [20, 19, 15, 13, 18]
  Mika
    length: 61
    judge votes: [12, 11, 15, 17, 11]

Write "jump" to jump; otherwise you quit: jump

Round 3

Jumping order:
  1. Mika (260 points)
  2. Mikael (282 points)
```

```
Results of round 3
  Mika
    length: 88
    judge votes: [11, 19, 13, 10, 15]
  Mikael
    length: 63
    judge votes: [12, 19, 19, 12, 12]

Write "jump" to jump; otherwise you quit: quit

Thanks!

Tournament results:
Position    Name
1           Mikael (388 points)
            jump lengths: 95 m, 96 m, 63 m
2           Mika (387 points)
            jump lengths: 112 m, 61 m, 88 m
```

Note1: it is essential that the *user interface* works exactly as displayed above; for instance, the number of spaces at the beginning of the lines must be right. **The space at the beginning of the lines must be made of spaces**, the tests do not work if you use tabulatation. Also, it is good that you copy the text which has to be printed by your program and you paste it into your code; you can copy it either from the exercise layout or from the test error messages. *The exercise is worth of four separate exercise points.*

The program has to start when we execute the main method in the example layout. Also, remember again that you can create only one Scanner object in your exercise.