Object-Oriented Programming with Java, part I ››

Authors: Arto Vihavainen, Matti Luukkainen
Translators to English: Emilia Hjelm, Alex H. Virtanen, Matti Luukkainen, Virpi Sumu, Birunthan Mohanathas

23. MORE ABOUT OBJECTS AND CLASSES

23.1 MULTIPLE CONSTRUCTORS

Let us return to the class that handles Persons again. The class `Person` currently looks like this:

```
public class Person {

    private String name;
    private int age;
    private int height;
    private int weight;

     public Person(String name) {
```

```
        this.name = name;
        this.age = 0;
        this.weight = 0;
        this.height = 0;
    }

    public void printPerson() {
        System.out.println(this.name + " I am " + this.age + " years old");
    }

    public void becomeOlder() {
        this.age++;
    }

    public boolean adult(){
        if ( this.age < 18 ) {
            return false;
        }

        return true;
    }

    public double weightIndex(){
        double heightInMeters = this.height/100.0;

        return this.weight / (heightInMeters*heightInMeters);
    }

    public String toString(){
        return this.name + " I am " + this.age + " years old, my weight index is " + this.w
    }

    public void setHeight(int height){
        this.height = height;
    }

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

    public int getWeight() {
        return this.weight;
    }

    public void setWeight(int weight) {
        this.weight = weight;
    }

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

All person objects are 0 years old at creation, since the constructor sets it to 0:

```
public Person(String name) {
    this.name = name;
    this.age = 0;
    this.weight = 0;
    this.height = 0;
}
```

We also want to create a person so that in addition to name, can be given an age as a parameter. This can be achieved easily, since multiple constructors can exist. Let us make an alternative constructor. You do not need to remove the old one.

```
public Person(String name) {
    this.name = name;
    this.age = 0;
    this.weight = 0;
    this.height = 0;
}

public Person(String name, int age) {
    this.name = name;
    this.age = age;
    this.weight = 0;
    this.height = 0;
}
```

Now, creating objects can be done in two different ways:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka", 24);
    Person esko = new Person("Esko");

    System.out.println( pekka );
    System.out.println( esko );
}
```

```
Pekka, age 24 years
Esko, age 0 years
```

The technique in which a class has two constructors is called *constructor overloading*. A class can have multiple constructors, which are different from one another according to parameter quanitities and/or types. However, it is not possible to create two different constructors that have exactly the same type of parameters. We cannot add a constructor `public Person(String name, int weight)` on top of the old ones, since it is impossible for Java to tell the difference between this one and the one in which the integer stands for the age.

23.2 CALLING YOUR OWN CONSTRUCTOR

But wait, in chapter 21 we noted that "copy-paste" code is not too great of an idea! When we inspect the overloaded constructors above, we notice that they have the same code repeated in them. We are not ok with this.

The old constructor actually is a special case of the new constructor. What if the old constructor could 'call' the new constructor? This can be done, since you can call another constructor from within a constructor with `this`!

Let us change the old constructor that does nothing, but only calls the new constructor below it and asks it to set the age to 0:

```
public Person(String name) {
    this(name, 0);  // run here the other constructor's code and set the age parame
}

public Person(String name, int age) {
    this.name = name;
    this.age = age;
    this.weight = 0;
    this.height = 0;
}
```

Calling the own constructor of a class `this(name, 0);` might seem a little peculiar. But we can imagine that during the call it will automatically copy-paste the code from the constructor below and that 0 is entered to the age parameter.

23.3 OVERLOADING A METHOD

Just like constructors, methods can also be overloaded and multiple versions of a method can exist. Again, the parameter types of different versions have to be different. Let us create another version of the `becomeOlder`, which enables aging the person the amount of years that is entered as a parameter:

```
public void becomeOlder() {
    this.age = this.age + 1;
}

public void becomeOlder(int years) {
    this.age = this.age + years;
}
```

In the following, "Pekka" is born as a 24-year old, ages one year, and then 10:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka", 24);
```

```
        System.out.println(pekka);
        pekka.becomeOlder();
        System.out.println(pekka);
        pekka.becomeOlder(10);
        System.out.println(pekka);
    }
```

Prints:

```
Pekka, age 24 years
Pekka, age 25 years
Pekka, age 35 years
```

Now, a person has two `becomeOlder` methods. The method that is chosen to be run depends on the amount of parameters entered in to the method call. The method `becomeOlder` can also be run through the method `becomeOlder(int years)`:

```
public void becomeOlder() {
    this.becomeOlder(1);
}

public void becomeOlder(int years) {
    this.age = this.age + years;
}
```

Exercise 84: Overloaded counter

EXERCISE 84.1: MULTIPLE CONSTRUCTORS

Make a class `Counter` that holds a number that can be decreased and increased. The counter also has an optional *check* that prevents the counter from going below 0. The class has to have the following constructors:

- `public Counter(int startingValue, boolean check)` creates a new counter with the given value. The check is on if the parameter given to `check` was `true`.
- `public Counter(int startingValue)` creates a new counter with the given value. The check on the new counter should be off.
- `public Counter(boolean check)` creates a new counter with the starting value 0. The check is on if the parameter given to `check` was `true`.
- `public Counter()` creates a new counter with the starting value of 0 and with checking off.

and the following methods:

- `public int value()` returns the current value of the counter

- `public void increase()` increases the value of the counter by one
- `public void decrease()` decreases the value of the counter by one, but not below 0 if the check is on

EXERCISE 84.2: ALTERNATIVE METHODS

Create also a one parametered versions of the methods `increase` and `decrease`:

- `public void increase(int increaseAmount)` increases the value by the amount of the parameter. If the value of the parameter is negative, the value will not change.
- `public void decrease(int decreaseAmount)` decreases the value of the counter by the amount given by the parameter, but not below 0 if the check is on. If the value of the parameter is negative, the value of the counter will not change.

23.4 OBJECT IS AT THE END OF A WIRE

In chapter 20, we noted that `ArrayList` is at the end of a wire. Also objects are 'at the end of a wire'. What does this mean? Let us inspect the following example:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka", 24);

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

When we run the sentence `Person pekka = new Person("Pekka", 24);` an object is born. The object can be accessed through the variable `pekka`. Technically speaking, the object is not within the variable `pekka` (in the box 'pekka'), but `pekka` refers to the object that was born. In other words, the object is 'at the end of a wire' that is attached to a variable named `pekka`. The concept could be visualized like this:

Let us add to the program a variable `person` of the type `Person` and set its starting value to `pekka`. What happens now?

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka", 24);

    System.out.println( pekka );
```

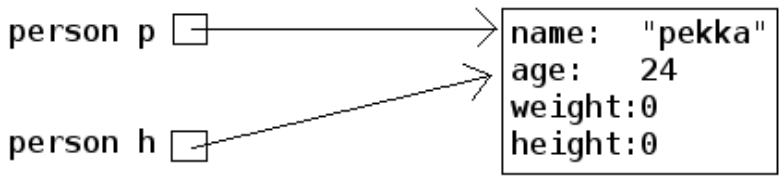
```
    Person person = pekka;
    person.becomeOlder(25);

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

Prints:

```
Pekka, age 24 years
Pekka, age 49 years
```

In the beginning, Pekka was 24 years old. Then a Person object at the end of a wire attached to a `Person` variable is aged by 25 years and as a consequence of that Pekka becomes older! What is going on here?

The command `Person person = pekka;` makes `person` refer to the same object that `pekka` refers to. So, a copy of the object is not born, but instead both of the variables refer to the same object. With the command `Person person = pekka;` a *copy of the wire* is born. The same thing as a picture (Note: in the picture p refers to the variable pekka, and h to the variable person in the main program. The variable names have also been abbreviated in some of the following pictures.):



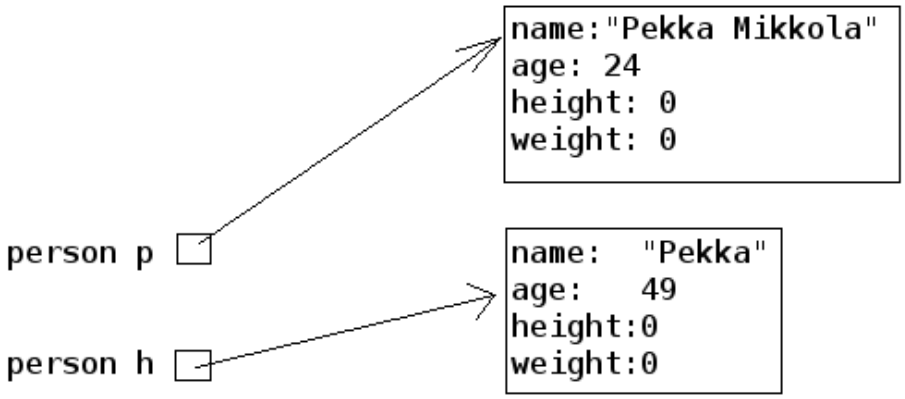
In the example, "an unknown `person` steals Pekka's identity". In the following, we have expanded the example so that a new object is created and `pekka` begins to refer to a new object:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka", 24);

    System.out.println( pekka );

    Person person = pekka;
    person.becomeOlder(25);

    System.out.println( pekka );

    pekka = new Person("Pekka Mikkola", 24);
    System.out.println( pekka );
}
```

Prints:

```
Pekka, age 24 years
Pekka, age 49 years
Pekka Mikkola, age 24 years
```

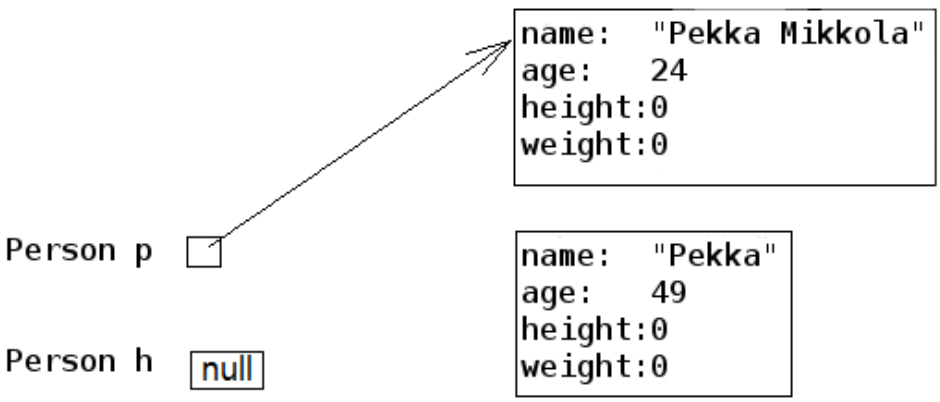
The variable `pekka` refers to one object, but then begins to refer to another. Here is the situation after running the previous line of code:



Let's develop the example further by making `person` to refer to 'nothing', to `null`:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka", 24);

    System.out.println( pekka );

    Person person = pekka;
    person.becomeOlder(25);

    System.out.println( pekka );

    pekka = new Person("Pekka Mikkola", 24);
    System.out.println( pekka );

    person = null;
    System.out.println( person );
}
```

After running that, the situation looks like this:



Nothing refers to the second object. The object has become 'garbage'. Java's garbace collector cleans up the garbage every now and then by itself. If this did not happen, the garbage would pile up in the computer's memory until the execution of the program is done.

We notice this on the last line whine we try to print 'nothing' (`null`) on the last line:

```
Pekka, age 24 years
Pekka, age 49 years
Pekka Mikkola, age 24 years
null
```

What happens if we try to call a "nothing's" method, for example the method `weightIndex`:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka", 24);

    System.out.println( pekka );

    Person person = null;
    System.out.println( person.weightIndex() );
}
```

Result:

```
Pekka, age 24 years
Exception in thread "main" java.lang.NullPointerException
        at Main.main(Main.java:20)
Java Result: 1
```

Not good. This might be the first time in your life that you see the text **NullPointerException**. But we can assure you that it will not be the last. NullPointerException is an exception state, when we try to call methods of an object with the value `null`.

23.5 AN OBJECT AS A METHOD PARAMETER

We have seen that a method can have, for example `int`, `double`, `String` or ArrayList as its parameter. ArrayLists and character strings are objects, so as one might guess a method can take any type of object as a parameter. Let us demonstrate this with an example.

People whose weight index exceeds a certain limit are accepted into the Weight Watchers. The limit is not the same in all Weight Watchers associations. Let us make a class corresponding to the Weight Watchers association. As the object is being created, the lowest acceptance limit is passed to the constructor as a parameter.

```
public class WeightWatchersAssociation {
    private double lowestWeightIndex;

    public WeightWatchersAssociation(double indexLimit) {
        this.lowestWeightIndex = indexLimit;
    }

}
```

Next we will create a method, with which we can check if a person is eligible to the association, in other words we check if a person's weight index is large enough. The method returns `true` if the person that is passed in as a parameter is eligible and `false` if not.

```
public class WeightWatchersAssociation {
    // ...

    public boolean isAcceptedAsMember(Person person) {
        if ( person.weightIndex() < this.lowestWeightIndex ) {
            return false;
        }

        return true;
    }
}
```

The method `isAcceptedAsMember` of the WeightWatchersAssociation object gets a `Person` object as its parameter (or more accurately the wire to the person), and then calls the method `weightIndex` of the person that it received as a parameter.

In the following, is a test main program in which a person object `matti` and a person object `juhana` is passed to the weight watchers association's method:

```
public static void main(String[] args) {
    Person matti = new Person("Matti");
    matti.setWeight(86);
    matti.setHeight(180);

    Person juhana = new Person("Juhana");
    juhana.setWeight(64);
    juhana.setHeight(172);

    WeightWatchersAssociation kumpulasWeight = new WeightWatchersAssociation(25);

    if ( kumpulasWeight.isAcceptedAsMember(matti) ) {
        System.out.println( matti.getName() + " is accepted as a member");
    } else {
        System.out.println( matti.getName() + " is not accepted as a member");
    }

    if ( kumpulasWeight.isAcceptedAsMember(juhana) ) {
        System.out.println( juhana.getName() + " is accepted as a memberksi");
    } else {
        System.out.println( juhana.getName() + " is not accepted as a member");
    }
}
```

The program prints:

```
Matti is accepted as a member
Juhana is not accepted as a member
```

A few NetBeans-tips

- All NetBeans-tips are found here
- **The automatic generating of constructors, getters and setters.**

  Go inside of the code block of the class, but outside of all methods and simultaneously press Ctrl+Space. If your class, for example, has an object variable `balance`, NetBeans will offer you the opportunity to generate the getter and setter methods, and a constructor that sets a starting value for the object variable.

Exercise 85: Reformatory

In this assignment, we use the already given class `Person` and are supposed to build a new class `Reformatory`. Reformatory objects do certain things to persons, e.g. measure their weight and feed them.

Note: you should not alter the code in the class Person!

EXERCISE 85.1: WEIGHT OF A PERSON

The reformatory class already has a method skeleton `public int weight(Person person)`:

```
public class Reformatory {

    public int weight(Person person) {
        // returns the weight of the parameter
        return -1;
    }
}
```

The method gets a person object as a parameter. The method is supposed to return the weight of the parameter, so the method should call a suitable method of `person`, get the return value and then return it to the caller.

In the following a reformatory weight's two persons:

```
public static void main(String[] args) {
    Reformatory eastHelsinkiReformatory = new Reformatory();

    Person brian = new Person("Brian", 1, 110, 7);
    Person pekka = new Person("Pekka", 33, 176, 85);

    System.out.println(brian.getName() + " weight: " + eastHelsinkiReformatory.weig
    System.out.println(pekka.getName() + " weight: " + eastHelsinkiReformatory.weig
```

```
}
```

The output should be:

```
Brian weight: 7 kilos
Pekka weight: 85 kilos
```

EXERCISE 85.2: FEEDING A PERSON

In the previous part of the assignment, the method `weight` queried some information from the parameter object by calling its method. It is also possible to change the state of the parameter. Add to class Reformatory the method `public void feed(Person person)` that increases the weight of its parameter by one.

Next, an example where first the weight of Pekka and Brian is measured and printed. Then Reformatory feeds Brian three times and after that the weights are measured and printed again.

```
public static void main(String[] args) {
    Reformatory eastHelsinkiReformatory = new Reformatory();

    Person brian = new Person("Brian", 1, 110, 7);
    Person pekka = new Person("Pekka", 33, 176, 85);

    System.out.println(brian.getName() + " weight: " + eastHelsinkiReformatory.weig
    System.out.println(pekka.getName() + " weight: " + eastHelsinkiReformatory.weig

    eastHelsinkiReformatory.feed(brian);
    eastHelsinkiReformatory.feed(brian);
    eastHelsinkiReformatory.feed(brian);

    System.out.println("");

    System.out.println(brian.getName() + " weight: " + eastHelsinkiReformatory.weig
    System.out.println(pekka.getName() + " weight: " + eastHelsinkiReformatory.weig
}
```

The output should reveal that Brian has gained 3 kilos:

```
Brian weight: 7 kilos
Pekka weight: 85 kilos

Brian weight: 10 kilos
Pekka weight: 85 kilos
```

EXERCISE 85.3: NUMBER OF TIMES A WEIGHT HAS BEEN MEASURED

Add to class Reformatory the method `public int totalWeightsMeasured()` that returns the total number of times a weight has been measured.

With the following main program:

```
public static void main(String[] args) {
    Reformatory eastHelsinkiReformatory = new Reformatory();

    Person brian = new Person("Brian", 1, 110, 7);
    Person pekka = new Person("Pekka", 33, 176, 85);

    System.out.println("total weights measured "+eastHelsinkiReformatory.totalWeights

    eastHelsinkiReformatory.weight(brian);
    eastHelsinkiReformatory.weight(pekka);

    System.out.println("total weights measured "+eastHelsinkiReformatory.totalWeights

    eastHelsinkiReformatory.weight(brian);
    eastHelsinkiReformatory.weight(brian);
    eastHelsinkiReformatory.weight(brian);
    eastHelsinkiReformatory.weight(brian);

    System.out.println("total weights measured "+eastHelsinkiReformatory.totalWeights
}
```

the output should be:

```
total weights measured 0
total weights measured 2
total weights measured 6
```

Exercise 86: Lyyra card and Cash Register

EXERCISE 86.1: THE "STUPID" LYYRA CARD

In the last set of exercises, we implemented the class LyyraCard. The card had methods for paying economical and gourmet lunches and a method for loading money.

Last week's version of the card is however somehow problematic. The card knew the lunch prices so that it could take the right price from the balance if a lunch was paid. What if the lunch prices change? Or what if it is decided that LyyraCards could also be used to purchase coffee? A change like these would mean that all the existing LyyraCards

should be replaced with the new ones with the right prices and/or new methods. This does not sound good at all!

A better solution is to store only the balance on the card and have all the inteligence in a *cash register*.

We will soon program the cash register but let us start by completing the "stupid" version of the Lyyra card. The card holds the balance and has only two methods, `public void loadMoney(double amount)` that is already implemented and `public boolean pay(double amount)` that you should complete according to the instructions below:

```
public class LyyraCard {
    private double balance;

    public LyyraCard(double balance) {
        this.balance = balance;
    }

    public double balance() {
        return this.balance;
    }

    public void loadMoney(double amount) {
        this.balance += amount;
    }

    public boolean pay(double amount){
       // the method checks if the balance of the card is at least the amount gi
       // if not, the method returns false meaning that the card could not be us
       // if the balance is enough, the given amount is taken from the balance a
    }
}
```

With the following main:

```
public class Main {
    public static void main(String[] args) {
        LyyraCard cardOfPekka = new LyyraCard(10);

        System.out.println("money on the card " + cardOfPekka.balance() );
        boolean succeeded = cardOfPekka.pay(8);
        System.out.println("money taken: " + succeeded );
        System.out.println("money on the card " + cardOfPekka.balance() );

        succeeded = cardOfPekka.pay(4);
        System.out.println("money taken: " + succeeded );
        System.out.println("money on the card " + cardOfPekka.balance() );
    }
}
```

the output should be

```
money on the card 10.0
money taken: true
money on the card 2.0
money taken: false
money on the card 2.0
```

EXERCISE 86.2: CASH REGISTER AND PAYING WITH CASH

In Unicafe, a client pays either with cash or with a Lyyra Card. The personnel uses a cash register to charge the client. Let us start by implementig the part of CashRegister that takes care of cash payments.

Below is the skeleton of CashRegister that also has the information on how the methods should be implemented:

```
public class CashRegister {
    private double cashInRegister;    // the amount of cash in the register
    private int economicalSold;       // the amount of economical lunches sold
    private int gourmetSold;          // the amount of gourmet lunches sold

    public CashRegister() {
        // at start the register has 1000 euros
    }

    public double payEconomical(double cashGiven) {
        // the price of the economical lunch is 2.50 euros
        // if the given cash is at least the price of the lunch:
        //    the price of lunch is added to register
        //    the amount of the sold lunches is incremented by one
        //    the method returns cashGiven - lunch price
        // if not enough money is given, all is returned and nothing else happens
    }

    public double payGourmet(double cashGiven) {
        // the price of the gourmet lunch is 4.00 euros
        // if the given cash is at least the price of the lunch:
        //    the price of lunch is added to the register
        //    the amount of the sold lunches is incremented by one
        //    the method returns cashGiven - lunch price
        // if not enough money is given, all is returned and nothing else happens
    }

    public String toString() {
        return "money in register "+cashInRegister+" economical lunches sold: "+economical
    }
}
```

When correctly implemented, the following main:

```
public class Main {
    public static void main(String[] args) {
        CashRegister unicafeExactum = new CashRegister();

        double theChange = unicafeExactum.payEconomical(10);
        System.out.println("the change was " + theChange );

        theChange = unicafeExactum.payEconomical(5);
        System.out.println("the change was "  + theChange );

        theChange = unicafeExactum.payGourmet(4);
        System.out.println("the change was "  + theChange );

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

should output:

```
the change was 7.5
the change was 2.5
the change was 0.0
money in register 1009.0 economical lunches sold: 2 gourmet lunches sold: 1
```

EXERCISE 86.3: PAYING WITH CARD

Extend the cash register with methods to charge a lunch price from a Lyyra Card. See below how the methods should appear and behave:

```
public class CashRegister {
    // ...

    public boolean payEconomical(LyyraCard card) {
        // the price of the economical lunch is 2.50 euros
        // if the balance of the card is at least the price of the lunch:
        //    the amount of sold lunches is incremented by one
        //    the method returns true
        // if not, the method returns false
    }

    public boolean payGourmet(LyyraCard card) {
        // the price of the gourmet lunch is 4.00 euros
        // if the balance of the card is at least the price of the lunch:
        //    the amount of sold lunches is incremented by one
        //    the method returns true
        // if not, the method returns false
    }
```

```
    // ...
}
```

Note: card payments do not affect the amount of money in the register!

Example main and output:

```
public class Main {
    public static void main(String[] args) {
        CashRegister unicafeExactum = new CashRegister();

        double theChange = unicafeExactum.payEconomical(10);
        System.out.println("the change was " + theChange );

        LyyraCard cardOfJim = new LyyraCard(7);

        boolean succeeded = unicafeExactum.payGourmet(cardOfJim);
        System.out.println("payment success: " + succeeded);
        succeeded = unicafeExactum.payGourmet(cardOfJim);
        System.out.println("payment success: " + succeeded);
        succeeded = unicafeExactum.payEconomical(cardOfJim);
        System.out.println("payment success: " + succeeded);

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

```
the change was 7.5
payment success: true
payment success: false
payment success: true
money in register 1002.5 economical lunches sold: 2 gourmet lunches sold: 1
```

EXERCISE 86.4: LOADING MONEY

To complete the assignment, extend the cash register with a method that can be used to load cash to Lyyra Cards. When a certain amount is loaded to the card, the amount stored in the register increases correspondingly. Remember that the amount to be loaded needs to be positive! The method skeleton:

```
public void loadMoneyToCard(LyyraCard card, double sum) {
    // ...
}
```

Example main and its output:

```
public class Main {
    public static void main(String[] args) {
        CashRegister unicafeExactum = new CashRegister();
        System.out.println( unicafeExactum );

        LyyraCard cardOfJim = new LyyraCard(2);

        System.out.println("the card balance " + cardOfJim.balance() + " euros");

        boolean succeeded = unicafeExactum.payGourmet(cardOfJim);
        System.out.println("payment success: " + succeeded);

        unicafeExactum.loadMoneyToCard(cardOfJim, 100);

        succeeded = unicafeExactum.payGourmet(cardOfJim);
        System.out.println("payment success: " + succeeded);

        System.out.println("the card balance " + cardOfJim.balance() + " euros");

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

```
money in register 1000.0 economical lunches sold: 0 gourmet lunches sold: 0
money on the card 2.0 euros
payment success: false
payment success: true
the card balance 98.0 euros
money in register 1100.0 economical lunches sold: 0 gourmet lunches sold: 1
```

23.6 ANOTHER OBJECT OF THE SAME TYPE AS A PARAMETER TO A METHOD

We will keep on working with the `Person` class. As we recall, persons know their age:

```
public class Person {

    private String name;
    private int age;
    private int height;
    private int weight;
```

```
    // ...
}
```

We want to compare ages of two persons. The comparison can be done in a number of ways. We could define a getter method `getAge` for a person. Comparing two persons in that case would be done like this:

```
Person pekka = new Person("Pekka");
Person juhana = new Person("Juhana")

if ( pekka.getAge() > juhana.getAge() ) {
    System.out.println(pekka.getName() + " is older than " + juhana.getName());
}
```

We will learn a slightly more object-oriented way to compare the ages of two people.

We will create a method `boolean olderThan(Person compared)` for the Person class, with which we can compare a certain person with a person that is given as a parameter.

The method is meant to be used in the following way:

```
public static void main(String[] args) {
    Person pekka = new Person("Pekka", 24);
    Person antti = new Person("Antti", 22);

    if (pekka.olderThan(antti)) {  //  same as pekka.olderThan(antti)==true
        System.out.println(pekka.getName() + " is older than " + antti.getName());
    } else {
        System.out.println(pekka.getName() + " isn't older than " + antti.getName());
    }
}
```

Here, we ask Pekka if he is older than Antti, Pekka replies true if he is, and false if he is not. In practice, we call the method `olderThan` of the object that `pekka` refers to. For this method, we give as a parameter the object that `antti` refers to.

The program prints:

```
Pekka is older than Antti
```

The program gets a person object as its parameter (or more accurately a reference to a person object, which is at 'the end of a wire') and then compares its own age `this.age` to the age of the compared `compared.age`. The implementation looks like this:

```
public class Person {
    // ...

    public boolean olderThan(Person compared) {
        if ( this.age > compared.age ) {
            return true;
        }
```

```
        return false;
    }
}
```

Even though `age` is a `private` object variable, we can read the value of the variable by writing `compared.age`. This is because `private` variables can be read in all methods that the class in question contains. Note that the syntax resembles the call of a method of an object. Unlike calling a method, we refer to a field of an object, in which case we do not write the parentheses.

23.7 THE DATE AS AN OBJECT

Another example of the same theme. Let us create a class, which can represent dates.

Within an object, the date is represented with three object variables. Let us also make a method, which can compare whether the date is earlier than a date that is given as a parameter:

```
public class MyDate {
    private int day;
    private int month;
    private int year;

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public String toString() {
        return this.day + "." + this.month + "." + this.year;
    }

    public boolean earlier(MyDate compared) {
        // first we'll compare years
        if ( this.year < compared.year ) {
            return true;
        }

        // if the years are the same, we'll compare the months
        if ( this.year == compared.year && this.month < compared.month ) {
            return true;
        }

        // years and months the same, we'll compare the days
        if ( this.year == compared.year && this.month == compared.month &&
                this.day < compared.day ) {
            return true;
        }

        return false;
```

```
    }
}
```

Example of usage:

```
public static void main(String[] args) {
    MyDate p1 = new MyDate(14, 2, 2011);
    MyDate p2 = new MyDate(21, 2, 2011);
    MyDate p3 = new MyDate(1, 3, 2011);
    MyDate p4 = new MyDate(31, 12, 2010);

    System.out.println( p1 + " earlier than " + p2 + ": " + p1.earlier(p2));
    System.out.println( p2 + " earlier than " + p1 + ": " + p2.earlier(p1));

    System.out.println( p2 + " earlier than " + p3 + ": " + p2.earlier(p3));
    System.out.println( p3 + " earlier than " + p2 + ": " + p3.earlier(p2));

    System.out.println( p4 + " earlier than " + p1 + ": " + p4.earlier(p1));
    System.out.println( p1 + " earlier than " + p4 + ": " + p1.earlier(p4));
}
```

```
14.2.2011 earlier than 21.2.2011: true
21.2.2011 earlier than 14.2.2011: false
21.2.2011 earlier than 1.3.2011: true
1.3.2011 earlier than 21.2.2011: false
31.12.2010 earlier than 14.2.2011: true
14.2.2011 earlier than 31.12.2010: false
```

Exercise 87: Apartment comparison

The information system of a Housing service represents the apartments it has for sale using objects of the following class:

```
public class Apartment {
    private int rooms;
    private int squareMeters;
    private int pricePerSquareMeter;

    public Apartment(int rooms, int squareMeters, int pricePerSquareMeter){
        this.rooms = rooms;
        this.squareMeters = squareMeters;
        this.pricePerSquareMeter = pricePerSquareMeter;
    }
}
```

Next you should implement a couple of methods that help in apartment comparisons.

EXERCISE 87.1: LARGER

Implement the method `public boolean larger(Apartment otherApartment)` that returns true if the Apartment object for which the method is called (`this`) is larger than the apartment object given as parameter (`otherApartment`).

Example of the usage:

```
Apartment studioManhattan = new Apartment(1, 16, 5500);
Apartment twoRoomsBrooklyn = new Apartment(2, 38, 4200);
Apartment fourAndKitchenBronx = new Apartment(3, 78, 2500);

System.out.println( studioManhattan.larger(twoRoomsBrooklyn) );        // false
System.out.println( fourAndKitchenBronx.larger(twoRoomsBrooklyn) );    // true
```

EXERCISE 87.2: PRICE DIFFERENCE

Implement the method `public int priceDifference(Apartment otherApartment)` that returns the absolute value of the price difference of the Apartment object for which the method is called (`this`) and the apartment object given as parameter (`otherApartment`). The price of an apartment is `squareMeters * pricePerSquareMeter`.

Example of the usage:

```
Apartment studioManhattan = new Apartment(1, 16, 5500);
Apartment twoRoomsBrooklyn = new Apartment(2, 38, 4200);
Apartment fourAndKitchenBronx = new Apartment(3, 78, 2500);

System.out.println( studioManhattan.priceDifference(twoRoomsBrooklyn) );
System.out.println( fourAndKitchenBronx.priceDifference(twoRoomsBrooklyn) );
```

EXERCISE 87.3: MORE EXPENSIVE THAN

Implement the method `public boolean moreExpensiveThan(Apartment otherApartment)` that returns true if the Apartment-object for which the method is called (`this`) has a higher price than the apartment object given as parameter (`otherApartment`).

Example of the usage:

```
Apartment studioManhattan = new Apartment(1, 16, 5500);
Apartment twoRoomsBrooklyn = new Apartment(2, 38, 4200);
Apartment fourAndKitchenBronx = new Apartment(3, 78, 2500);

System.out.println( studioManhattan.moreExpensiveThan(twoRoomsBrooklyn) );
System.out.println( fourAndKitchenBronx.moreExpensiveThan(twoRoomsBrooklyn) );
```

23.8 OBJECTS ON A LIST

We've used `ArrayList`s in a lot of examples and assignments already. You can add character strings, for example, to an ArrayList object and going through the strings, searching, removing and sorting them and so forth, are painless actions.

You can put any type of objects in ArrayLists. Let's create a person list, an `ArrayList<Person>` and put a few person objects in it:

```
public static void main(String[] args) {
    ArrayList<Person> teachers = new ArrayList<Person>();

    // first we can take a person into a variable
    Person teacher = new Person("Juhana");
    // and then add it to the list
    teachers.add(teacher);

    // or we can create the object as we add it:
    teachers.add( new Person("Matti") );
    teachers.add( new Person("Martin") );

    System.out.println("teachers as newborns: ");
    for ( Person prs : teachers ) {
        System.out.println( prs );
    }

    for ( Person prs : teachers ) {
        prs.becomeOlder( 30 );
    }

    System.out.println("in 30 years: ");
    for ( Person prs : teachers ) {
        System.out.println( prs );
    }
}
```

The program prints:

```
teachers as newborns:
Juhana, age 0 years
Matti, age 0 years
Martin, age 0 years
in 30 years:
Juhana, age 30 years
Matti, age 30 years
Martin, age 30 years
```

Exercise 88: Students

EXERCISE 88.1: CLASS STUDENT

Implement class `Student` that holds the following information about a student:

- name (`String`)
- studentNumber (`String`)

The class should have the following methods:

- A constructor that initializes the name and the student number with the given parameters.
- `getName`, that returns the student name
- `getStudentNumber`, that returns the student number
- `toString`, that returns a String representation of the form: Pekka Mikkola (013141590)

With the following code:

```
public class Main {
    public static void main(String[] args) {
        Student pekka = new Student("Pekka Mikkola", "013141590");
        System.out.println("name: " + pekka.getName());
        System.out.println("studentnumber: " + pekka.getStudentNumber());
        System.out.println(pekka);
    }
}
```

The output should be:

```
name: Pekka Mikkola
studentnumber: 013141590
Pekka Mikkola (013141590)
```

EXERCISE 88.2: LIST OF STUDENTS

Implement a main program that works as follws:

```
name: Alan Turing
studentnumber: 017635727
name: Linus Torvalds
studentnumber: 011288989
name: Steve Jobs
```

```
studentnumber: 013672548
name:

Alan Turing (017635727)
Linus Torvalds (011288989)
Steve Jobs (013672548)
```

So the program asks for student information from the user until the user gives a student an empty name. After the student info has been enteres, all the students are printed. From each inputted name-studentnumber-pair, the program should create a Student object. The program should store the students in an ArrayList which is defined as follows:

```
ArrayList<Student> list = new ArrayList<Student>();
```

EXERCISE 88.3: SEARCH

Extend the program of the previous part so that after the student info has been entered and students printed, the user can search the student list based on a given search term. The extended program should work in the following manner:

```
name: Carl Gustaf Mannerheim
studentnumber: 015696234
name: Steve Jobs
studentnumber: 013672548
name: Edsger Dijkstra
studentnumber: 014662803
name:

Carl Gustaf Mannerheim (015696234)
Steve Jobs (013672548)
Edsger Dijkstra (014662803)

Give search term: st
Result:
Carl Gustaf Mannerheim (015696234)
Edsger Dijkstra (014662803)
```

TIP: in the search you should iterate (using for or while) through the student list and by using the method `contains` of String check if a student's name (obtained with method `getName`) matches the search term.

23.9 AN OBJECT WITHIN AN OBJECT

Objects can have objects within them, not only character strings but also self-defined objects. Let's get back to the `Person`-class again and add a birthday for the person. We can use the `MyDate`-object we created earlier here:

```
public class Person {
    private String name;
    private int age;
    private int weight;
    private int height;
    private MyDate birthMyDate;

    // ...
```

Let's create a new constructor for persons, which enables setting a birthday:

```
    public Person(String name, int day, int month, int year) {
        this.name = name;
        this.weight = 0;
        this.height = 0;
        this.birthMyDate = new MyDate(day, month, year);
    }
```

So because the parts of the date are given as constructor parameters (day, month, year), the date object is created out of them and then *inserted* to the object variable `birthMyDate`.

Let's edit `toString` so that instead of age, it displays the birthdate:

```
public String toString() {
    return this.name + ", born " + this.birthMyDate;
}
```

And then let's test how the renewed Person class works:

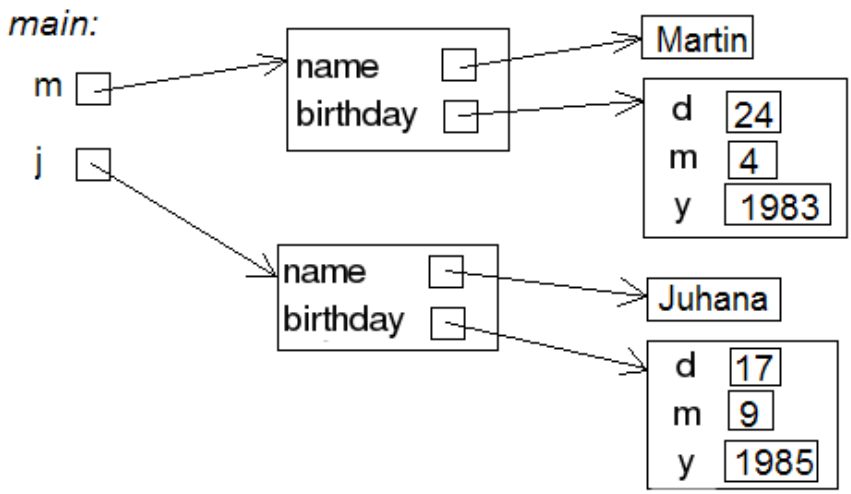
```
public static void main(String[] args) {
    Person martin = new Person("Martin", 24, 4, 1983);

    Person juhana = new Person("Juhana", 17, 9, 1985);

    System.out.println( martin );
    System.out.println( juhana );
}
```

Prints:

```
Martin, born 24.4.1983
Juhana, born 17.9.1985
```

In chapter 24.4, we noted that objects are 'at the end of a wire'. Take a look at that chapter again for good measure.

Person objects have the object variables `name`, which is a String-object and `birthMyDate`, which is a MyDate object. The variables of person are consequently both objects, so technically speaking they don't actually exist within a person object, but are 'at the end of a wire'. In other words a person has a reference to the objects stored in its object variables. The concept as a picture:



The main program now has two person programs at the ends of wires. The persons have a name and a birthdate. Because both are objects, both are at the ends of wires the person holds.

Birthday seems like a good expansion to the Person class. We notice, however, that the object variable `age` is becoming obsolete and should probably be removed since the age can be determined easily with the help of the current date and birthday. In Java, the current day can be figured out, for example, like this:

```
int day = Calendar.getInstance().get(Calendar.DATE);
int month = Calendar.getInstance().get(Calendar.MONTH) + 1; // January is 0 so we a
int year = Calendar.getInstance().get(Calendar.YEAR);
System.out.println("Today is " + day + "." + month + "." + year );
```

When age is removed, the `olderThan` method has to be changed so that it compares birthdates. We'll do this as an excersise assignment.

Exercise 89: Clock object

In assignment 78 we used objects of the class `BoundedCounter` to implement a clock in the main method. In this assignment we will tranform the clock to an object. The skeleton of the class clock looks like the following:

```
public class Clock {
    private BoundedCounter hours;
    private BoundedCounter minutes;
    private BoundedCounter seconds;

    public Clock(int hoursAtBeginning, int minutesAtBeginning, int secondsAtBegi
```

```
        // the counters that represent hours, minutes and seconds are created and
        // set to have the correct initial values
    }

    public void tick(){
        // Clock advances by one second
    }

    public String toString() {
        // returns the string representation
    }
}
```

Copy the class `BoundedCounter` from assignment 78 to the project of this assignment!

Implement constructor and method `tick` for the class `Clock`. Use the following main to test your clock:

```
public class Main {
    public static void main(String[] args) {
        Clock clock = new Clock(23, 59, 50);

        int i = 0;
        while( i < 20) {
            System.out.println( clock );
            clock.tick();
            i++;
        }
    }
}
```

The output should be:

```
23:59:50
23:59:51
23:59:52
23:59:53
23:59:54
23:59:55
23:59:56
23:59:57
23:59:58
23:59:59
00:00:00
00:00:01
...
```

23.10 A LIST OF OBJECTS WITHIN AN OBJECT

Let's expand the `WeightWatchersAssociation` object so that the association records all its members into an `ArrayList` object. So in this case the list will be filled with `Person` objects. In the extended version the association is given a name as a constructor parameter:

```
public class WeightWatchersAssociation {
    private double lowestWeightIndex;
    private String name;
    private ArrayList<Person> members;

    public WeightWatchersAssociation(String name, double lowestWeightIndex) {
        this.lowestWeightIndex = lowestWeightIndex;
        this.name = name;
        this.members = new ArrayList<Person>();
    }

    //..
}
```

Let's create a method with which a person is added to the association. The method won't add anyone to the association but people with a high enough weight index. Let's also make a toString with which the members' names are printed:

```
public class WeightWatchersAssociation {
    // ...

    public boolean isAccepted(Person person) {
        if ( person.weightIndex() < this.lowestWeightIndex ) {
            return false;
        }

        return true;
    }

    public void addAsMember(Person person) {
        if ( !isAccepted(person) ) { // same as isAccepted(person) == false
            return;
        }

        this.members.add(person);
    }

    public String toString() {
        String membersAsString = "";

        for ( Person member : this.members ) {
            membersAsString += "  " + member.getName() + "\n";
        }
```

```
        return "Weightwatchers association " + this.name + " members: \n" + membersAsString
    }
}
```

The method `addAsMember` uses the method `isAccepted` that was creater earlier.

Let's try out the expanded weightwatchers association:

```
public static void main(String[] args) {
    WeightWatchersAssociation weightWatcher = new WeightWatchersAssociation("Kumpulan

    Person matti = new Person("Matti");
    matti.setWeight(86);
    matti.setHeight(180);
    weightWatcher.addAsMember(matti);

    Person juhana = new Person("Juhana");
    juhana.setWeight(64);
    juhana.setHeight(172);
    weightWatcher.addAsMember(juhana);

    Person harri = new Person("Harri");
    harri.setWeight(104);
    harri.setHeight(182);
    weightWatcher.addAsMember(harri);

    Person petri = new Person("Petri");
    petri.setWeight(112);
    petri.setHeight(173);
    weightWatcher.addAsMember(petri);

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

In the output we can see that Juhana wasn't accepted as a member:

```
The members of weight watchers association 'kumpulan paino':
  Matti
  Harri
  Petri
```

Exercise 90: Team and Players

EXERCISE 90.1: CLASS TEAM

Implement a class `Team`. At this stage team has only a name (`String`) and the following functionality:

- a constructor that sets the team name
- `getName`, that returns the name

With the code:

```
public class Main {
    public static void main(String[] args) {
    Team barcelona = new Team("FC Barcelona");
    System.out.println("Team: " + barcelona.getName());
    }
}
```

the output should be::

```
Team: FC Barcelona
```

EXERCISE 90.2: PLAYER

Create a class `Player` with the instance variables for the player name and the amount of goals. A player should have two constructors: one that initializes the name and an another that initializes the name and the amount of goals. Implement also the following methods:

- `getName`, returns the player name
- `goals`, returns the amount of goals
- `toString`, returns a string representation that is formed as in the example below

Example usage:

```
public class Main {
    public static void main(String[] args) {
    Team barcelona = new Team("FC Barcelona");
    System.out.println("Team: " + barcelona.getName());

        Player brian = new Player("Brian");
        System.out.println("Player: " + brian);

        Player pekka = new Player("Pekka", 39);
        System.out.println("Player: " + pekka);
    }
}
```

and the expected output:

```
Team: FC Barcelona
Player: Brian, goals 0
```

```
Player: Pekka, goals 39
```

EXERCISE 90.3: ADDING PLAYERS TO A TEAM

Add to the class `Team` the following methods:

- `addPlayer`, adds a player to the team
- `printPlayers`, prints the players in the team

You should store the players to an instance variable of the type `ArrayList<Player>` within the class `Team`.

With the code:

```
public class Main {
    public static void main(String[] args) {
    Team barcelona = new Team("FC Barcelona");

        Player brian = new Player("Brian");
        Player pekka = new Player("Pekka", 39);

        barcelona.addPlayer(brian);
        barcelona.addPlayer(pekka);
        barcelona.addPlayer(new Player("Mikael", 1)); // works similarly as the ab

        barcelona.printPlayers();
    }
}
```

the output should be:

```
Brian, goals 0
Pekka, goals 39
Mikael, goals 1
```

EXERCISE 90.4: THE TEAM MAXIMUM SIZE AND CURRENT SIZE

Add to the class `Team` the methods

- `setMaxSize(int maxSize)`, sets the maximum number of players that the team can have
- `size`, returns the number of players in the team

By default the maximum number of players should be set to 16, and that can be changed with the method `setMaxSize`. Change the method `addPlayer` so that it does not add players to the team if the team already has the maximum number of players.

With the code:

```
public class Main {
    public static void main(String[] args) {
    Team barcelona = new Team("FC Barcelona");
        barcelona.setMaxSize(1);

        Player brian = new Player("Brian");
        Player pekka = new Player("Pekka", 39);
        barcelona.addPlayer(brian);
        barcelona.addPlayer(pekka);
        barcelona.addPlayer(new Player("Mikael", 1)); // works similarly as the ab

        System.out.println("Number of players: " + barcelona.size());
    }
}
```

the output should be

```
Number of players: 1
```

EXERCISE 90.5: GOALS OF A TEAM

Add to the class `Team` the method

- `goals`, returns the total number of goals for all the players in the team

With the code:

```
public class Main {
    public static void main(String[] args) {
        Team barcelona = new Team("FC Barcelona");

        Player brian = new Player("Brian");
        Player pekka = new Player("Pekka", 39);
        barcelona.addPlayer(brian);
        barcelona.addPlayer(pekka);
        barcelona.addPlayer(new Player("Mikael", 1)); // works similarly as the ab

        System.out.println("Total goals: " + barcelona.goals());
    }
}
```

the output should be

```
Total goals: 40
```

23.11 METHOD RETURNS AN OBJECT

We've seen methods that return booleans, numbers, lists and strings. It's easy to guess that a method can return any type of an object. Let's make a method for the weight watchers association that returns the person with the highest weight index.

```
public class WeightWatchersAssociation {
    // ...

    public Person personWithHighestWeightIndex() {
        // if members list is empty, we'll return null-reference
        if ( this.members.isEmpty() ) {
            return null;
        }

        Person heaviestSoFar = this.members.get(0);

        for ( Person person : this.members) {
            if ( person.weightIndex() > heaviestSoFar.weightIndex() ) {
                heaviestSoFar = person;
            }
        }

        return heaviestSoFar;
    }
}
```

The logic in this method works in the same way as when finding the largest number in a list. We use a dummy variable `heaviestSoFar` which is initially made to refer to the first person on the list. After that the list is read through and we see if there's anyone with a greater weight index in it, if so, we make `heaviestSoFar` refer to that one instead. At the end we return the value of the dummy variable, or in other words the *reference to a person object*.

Let's make an expansion to the previous main program. The main program receives the reference returned by the method to its variable `heaviest`.

```
public static void main(String[] args) {
    WeightWatchersAssociation weightWatcher = new WeightWatchersAssociation("Kumpluan

    // ..

    Person heaviest = weightWatcher.personWithHighestWeightIndex();
    System.out.print("member with the greatest weight index: " + heaviest.getName() );
    System.out.println(" weight index " + String.format( "%.2f", heaviest.weightIndex()
}
```

Prints:

```
member with the greatest weight index: Petri
weight index 37,42
```

23.12 METHOD RETURNS AN OBJECT IT CREATES

In the last example a method returned one Person object that the WeightWatcers object had in it. It's also possible that a method returns an entirely new object. In the following is a simple counter that has a method `clone` with which a clone - an entirely new counter object - can be made from the counter, which at creation has the same value as the counter that is being cloned:

```
public Counter {
    private int value;

    public Counter(){
        this(0);
    }

    public Counter(int initialValue){
        this.value = initialValue;
    }

    public void grow(){
        this.value++;
    }

    public String toString(){
        return "value: "+value;
    }

    public Counter clone(){
        // lets create a new counter object, that gets as its initial value
        // the value of the counter that is being cloned
        Counter clone = new Counter(this.value);

        // return the clone to the caller
        return clone;
    }
}
```

Here's a usage example:

```
Counter counter = new Counter();
counter.grow();
counter.grow();

System.out.println(counter);          // prints 2
```

```
Counter clone = counter.clone();

System.out.println(counter);          // prints 2
System.out.println(clone);            // prints 2

counter.grow();
counter.grow();
counter.grow();
counter.grow();

System.out.println(counter);          // prints 6
System.out.println(clone);            // prints 2

clone.grow();

System.out.println(counter);          // prints 6
System.out.println(clone);            // prints 3
```

The value of the object being cloned and the value of the clone - after the cloning has happened - are the same. However they are two different objects, so in the future as one of the counters grows the value of the other isn't affected in any way.

Exercise 91: Extending MyDate

In this assignment we will extend the class `MyDate`, that was developed in chapter 24.7. The code of the class:

```
public class MyDate {
    private int day;
    private int month;
    private int year;

    public MyDate(int day, int month, int year) {
        this.day = day;
        this.month = month;
        this.year = year;
    }

    public String toString() {
        return this.day + "." + this.month + "." + this.year;
    }

    public boolean earlier(MyDate compared) {
        // first we'll compare years
        if ( this.year < compared.year ) {
            return true;
        }

        // if the years are the same, we'll compare the months
```

```
        if ( this.year == compared.year && this.month < compared.month ) {
            return true;
        }

        // years and months the same, we'll compare the days
        if ( this.year == compared.year && this.month == compared.month &&
                this.day < compared.day ) {
            return true;
        }

        return false;
    }
}
```

EXERCISE 91.1: NEXT DAY

Add to the class MyDate the method `public void advance()` that advances the date by one. **Note:** In this assignment we assume that all the months have 30 days!

EXERCISE 91.2: ADVANCING MANY DAYS

Add also overloaded version `public void advance(int numberOfDays)`. This method should advance the day by the number given as parameter. Implement this method so that it calls the method `advance()` that was defined in the previous part of the assignment, e.g. the call `advance(5)` should call `advance()` 5 times. Again assume that all the months have 30 days!

EXERCISE 91.3: CREATION OF A NEW DATE

Add to the class `MyDate` the method `MyDate afterNumberOfDays(int days)`, that returns a **new** `MyDate`-object that has the date which equals the date of the object for which the method was called advance by the parameter of the method `days`. Again assume that all the months have 30 days!

Note that the object for which this method is called should not change!

Since the method creates a **new object**, the skeleton is of the form:

```
public MyDate afterNumberOfDays(int days){
    MyDate newMyDate = new MyDate( ... );

    // some code here

    return newMyDate;
}
```

The following code

```
public static void main(String[] args) {
    MyDate day = new MyDate(25, 2, 2011);
    MyDate newDate = day.afterNumberOfDays(7);
    for (int i = 1; i <= 7; ++i) {
        System.out.println("Friday after  " + i + " weeks is " + newDate);
        newDate = newDate.afterNumberOfDays(7);
    }
    System.out.println("This week's Friday is " + day);
    System.out.println("The date 790 days from this week's Friday is  " + day.afterNumberO
}
```

should print:

```
Friday after  1 weeks is 2.3.2011
Friday after  2 weeks is 9.3.2011
Friday after  3 weeks is 16.3.2011
Friday after  4 weeks is 23.3.2011
Friday after  5 weeks is 30.3.2011
Friday after  6 weeks is 7.4.2011
Friday after  7 weeks is 14.4.2011
This week's Friday is 25.2.2011
The date 790 days from this week's Friday is  5.5.2013
```

23.13 MORE ASSIGNMENTS

All the new theory for this week has already been covered. However, since this week's topics are quite challenging, we will practise our routine with a couple of more exercises.

Exercise 92: Difference of two dates

In this assignment we'll further extend the class MyDate. This assignment does not depend on the previous one, so the project contains the MyDate class that does not have the extensions of the previous assignment.

EXERCISE 92.1: DIFFERENCE IN YEARS, FIRST VERSION

Add to the class MyDate the method `public int differenceInYears(MyDate comparedDate)`, that calculates the difference in years of the object for which the method is called and the object given as parameters.

Note the following

- the first vesion of the method is not very precise, it only calculates the difference of the years and does not take into account the day and month of the dates
- The method needs to work only in the case where the date given as parameter is before the date for which the method is called

With the code

```
public class Main {
    public static void main(String[] args) {
        MyDate first = new MyDate(24, 12, 2009);
        MyDate second = new MyDate(1, 1, 2011);
        MyDate third = new MyDate(25, 12, 2010);

        System.out.println( second + " and " + first + " difference in years: " + seco

        System.out.println( third + " and " + first + " difference in years: " + third

        System.out.println( second + " and " + third + " difference in years: " + seco
    }
}
```

the output should be:

```
1.1.2011 and 24.12.2009 difference in years: 2     // since 2011-2009 = 2
25.12.2010 and 24.12.2009 difference in years: 1   // since 2010-2009 = 1
1.1.2011 and 25.12.2010 difference in years: 1     // since 2011-2010 = 1
```

EXERCISE 92.2: MORE ACCURACY

Calculation of the previous version was not very exact, e.g. the difference of dates 1.1.2011 and 25.12.2010 was claimed to be one year. **Modify the method so that it can calculate the difference properly.** Only the full years in difference count. So if the difference of two dates would be 1 year and 364 days, only the full years are counted and the result is thus one.

The method still needs to work only in the case where the date given as parameter is before the date for which the method is called

The output for the previous example is now:

```
1.1.2011 and 24.12.2009 difference in years: 1
25.12.2010 and 24.12.2009 difference in years: 1
1.1.2011 and 25.12.2010 difference in years: 0
```

EXERCISE 92.3: AND THE FINAL VERSION

Modify the method so that it works no matter which date is later, the one for which the method is called or the parameter. Example code:

```
public class Main {
    public static void main(String[] args) {
        MyDate first = new MyDate(24, 12, 2009);
        MyDate second = new MyDate(1, 1, 2011);
        MyDate third = new MyDate(25, 12, 2010);

        System.out.println( first + " and " + second + " difference in years: " + seco
        System.out.println( second + " and " + first + " difference in years: " + firs
        System.out.println( first + " and " + third + " difference in years: " + third
        System.out.println( third + " and " + first + " difference in years: " + first
        System.out.println( third + " and " + second + " difference in years: " + seco
        System.out.println( second + " and " + third + " difference in years: " + thir
    }
}
```

and the output

```
24.12.2009 and 1.1.2011 difference in years: 1
1.1.2011 and 24.12.2009 difference in years: 1
24.12.2009 and 25.12.2010 difference in years: 1
25.12.2010 and 24.12.2009 difference in years: 1
1.1.2011 and 25.12.2010 difference in years: 0
25.12.2010 and 1.1.2011 difference in years: 0
```

Exercise 93: Person extended

EXERCISE 93.1: CALCULATING THE AGE BASED ON THE BIRTHDAY

In chapter 24.9. Person was extended by adding to it a birthday represented as a MyDate object. It was noticed that after the addition the instance variable `age` has no role since the age could easily be calculated based on the current date and the birthday.

Now implement the method `age` that calucates and returns the age of the person.

Note: in the previous assignment we added the class `MyDate` method `public int differenceInYears(MyDate compared)`. Copy the method here since it eases this assignment considerably.

```
import java.util.Calendar;

public class Person {
    private String name;
    private MyDate birthday;
```

```
    public Person(String name, int pp, int kk, int vv) {
        this.name = name;
        this.birthday = new MyDate(pp, kk, vv);
    }

    public int age() {
        // calculate the age based on the birthday and the current day
        // you get the current day as follows:
        // Calendar.getInstance().get(Calendar.DATE);
        // Calendar.getInstance().get(Calendar.MONTH) + 1; // January is 0 so we
        // Calendar.getInstance().get(Calendar.YEAR);
    }

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

    public String toString() {
        return this.name +", born "+ this.birthday;
    }
}
```

You can use the following program to test your method. Add also yourself to the program and ensure that your age is calculated correctly.

```
public class Main {
    public static void main(String[] args) {
        Person pekka = new Person("Pekka", 15, 2, 1993);
        Person steve = new Person("Thomas", 1, 3, 1955);

        System.out.println( steve.getName() + " age " + steve.age() + " years");
        System.out.println( pekka.getName() + " age " + pekka.age() + " years");
    }
}
```

Output:

```
Thomas age 59 years
Pekka age 21 years
```

EXERCISE 93.2: COMPARING AGES BASED ON BIRTHDATE

Add to the class Person the method `boolean olderThan(Person compared)` which compares the ages of the object for which the method is called and the object given as parameter. The method returns true if the object itself is older than the parameter.

```
public class Person {
    // ...

    public boolean olderThan(Person compared) {
       // compare the ages based on birthdate
    }
}
```

Test the method with the code:

```
public class Main {
    public static void main(String[] args) {
        Person pekka = new Person("Pekka", 15, 2, 1983);
        Person martin = new Person("Martin", 1, 3, 1983);

        System.out.println( martin.getName() + " is older than " +  pekka.getName()
        System.out.println( pekka.getName() + " is older than " +  martin.getName()
    }
}
```

The output should be:

```
Martin is older than Pekka: false
Pekka is older than Martin: true
```

EXERCISE 93.3: NEW CONSTRUCTORS

Add to the class Person two new constructors:

- `public Person(String name, MyDate birthday)` constructor sets the given MyDate-object to be the birthday of the person
- `public Person(String name)` constructor sets the current date (i.e., the date when the program is run) to be the birthday of the person

Example program:

```
public class Main {
    public static void main(String[] args) {
        Person pekka = new Person("Pekka", new MyDate(15, 2, 1983));
        Person steve = new Person("Steve");

        System.out.println( pekka );
        System.out.println( steve );
    }
}
```

Output:

```
Pekka, born 15.2.1983
Steve, born 9.2.2012
```

Note: The last line depends on the day when the code is executed!

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

TIETOJENKÄSITTELYTIETEEN LAITOS
INSTITUTIONEN FÖR DATAVETENSKAP
DEPARTMENT OF COMPUTER SCIENCE