Part 5

Objects and references

Learning Objectives

- You will brush up on using classes and objects.
- You know what a `null` reference is, and what causes the NullPointerException error.
- You can use an object as an object variable and a method parameter.
- You can create a method that returns an object.
- You can create the method equals, which can be used to check if two objects of the same type have the same contents or state.

Let's continue working with objects and references. Assume we can use the class that represents a person, shown below. Person has object variables name, age, weight and height. Additionally, it contains methods to calculate the body mass index, among other things.

```
public class Person {

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

    public Person(String name) {
        this(name, 0, 0, 0);
    }

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

```
    // other constructors and methods

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

    public int getAge() {
        return this.age;
    }

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

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

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

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

    public double bodyMassIndex() {
        double heightPerHundred = this.height / 100.0;
        return this.weight / (heightPerHundred * heightPerHundred);
    }

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

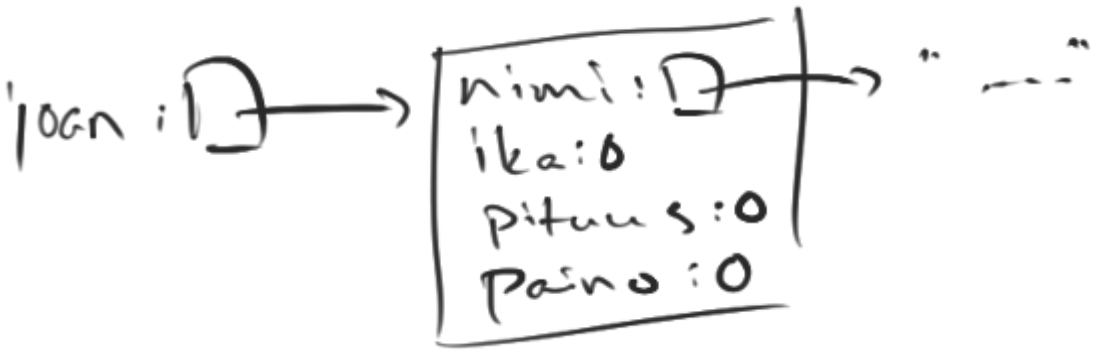
Precisely what happens when a new object is created?

```
Person joan = new Person("Joan Ball");
```

Calling a constructor with the command `new` causes several things to happen. First, space is reserved in the computer memory for storing object variables. Then default or initial values are set to object variables

(e.g. an `int` type variable receives an initial value of 0). Lastly, the source code in the constructor is executed.

A constructor call returns a reference to an object. A **reference** is information about the location of object data.



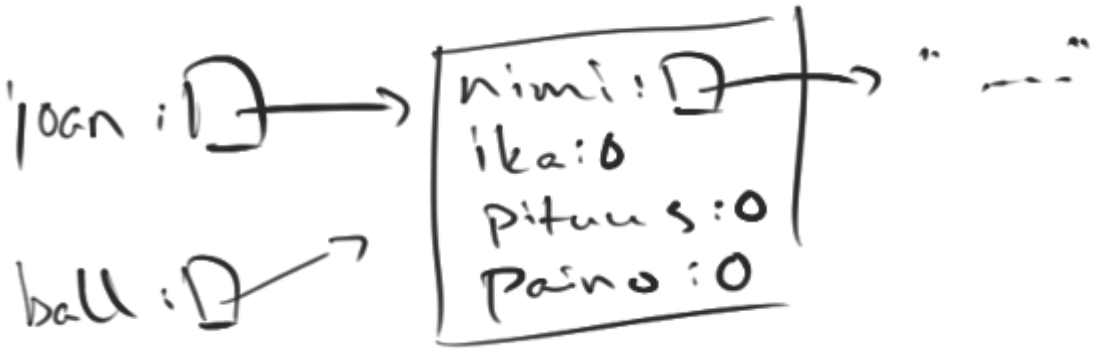
So the value of the variable is set to be a reference, i.e. knowledge about the location of related object data. The image above also reveals that strings — the name of our example person, for instance — are objects, too.

Assigning a reference type variable copies the reference

Let's add a `Person` type variable called `ball` into the program, and assign `joan` as its initial value. What happens then?

```
Person joan = new Person("Joan Ball");
System.out.println(joan);

Person ball = joan;
```

The statement `Person ball = joan;` creates a new Person variable `ball`, and copies the value of the variable `joan` as its value. As a result, `ball` refers to the same object as `joan`.



Let's inspect the same example a little more thoroughly.

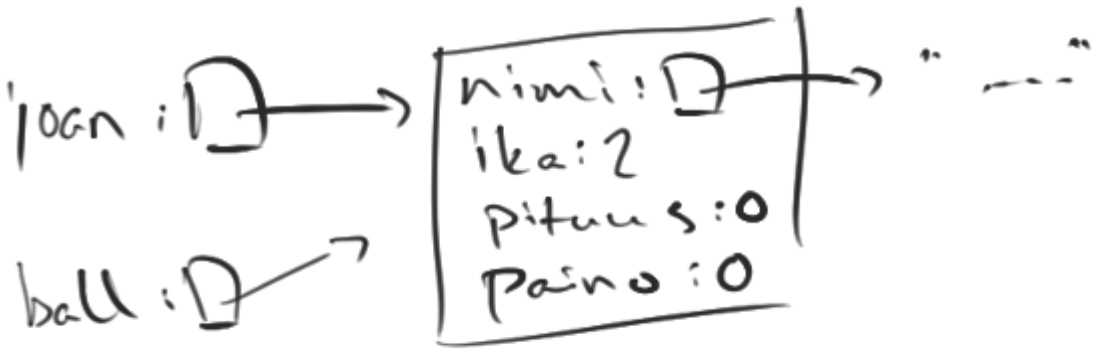
```
Person joan = new Person("Joan Ball");
System.out.println(joan);

Person ball = joan;
ball.growOlder();
ball.growOlder();

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

Sample output

```
Joan Ball, age 0 years
Joan Ball, age 2 years
```

Joan Ball — i.e. the Person object that the reference in the `joan` variable points at — starts as 0 years old. After this the value of the `joan` variable is assigned (so copied) to the `ball` variable. The Person object `ball` is aged by two years, and Joan Ball ages as a consequence!

An object's internal state is not copied when a variable's value is assigned. A new object is not being created in the statement `Person ball = joan;` — the value of the variable ball is assigned to be the copy of joan's value, i.e. a reference to an object.



Next, the example is continued so that a new object is created for the `joan` variable, and a reference to it is assigned as the value of the variable. The variable `ball` still refers to the object that we created earlier.

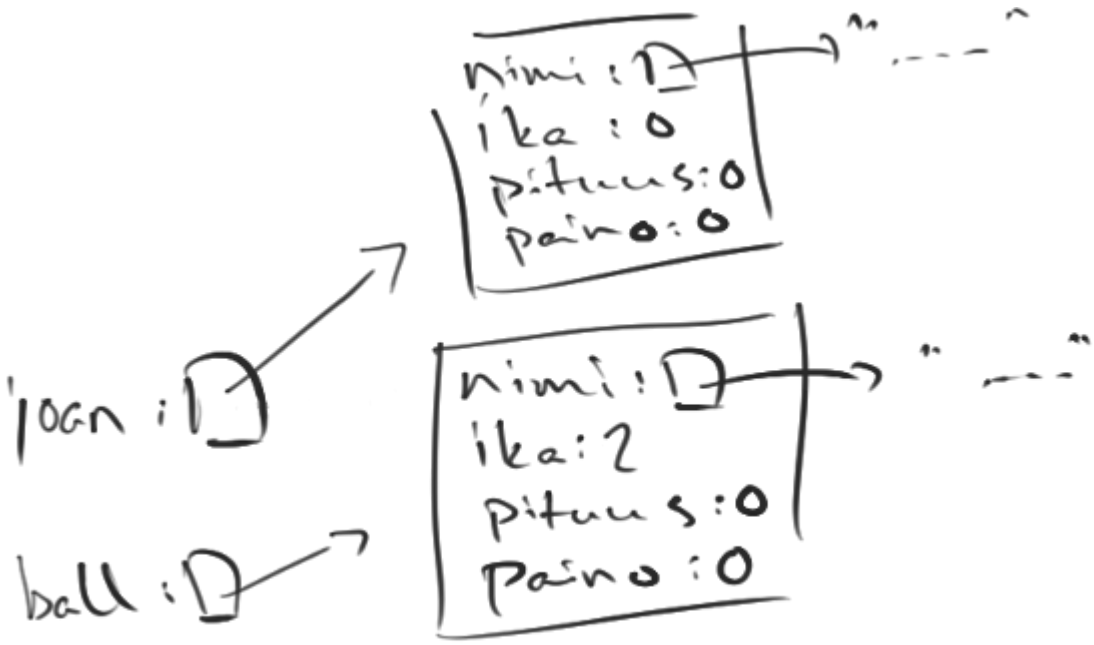
```
Person joan = new Person("Joan Ball");
System.out.println(joan);

Person ball = joan;
ball.growOlder();
ball.growOlder();

System.out.println(joan);

joan = new Person("Joan B.");
System.out.println(joan);
```

The following is printed:

Sample output

```
Joan Ball, age 0 years
Joan Ball, age 2 years
Joan B., age 0 years
```

So in the beginning the variable `joan` contains a reference to one object, but in the end a reference to another object has been copied as its value. Here is a picture of the situation after the last line of code.

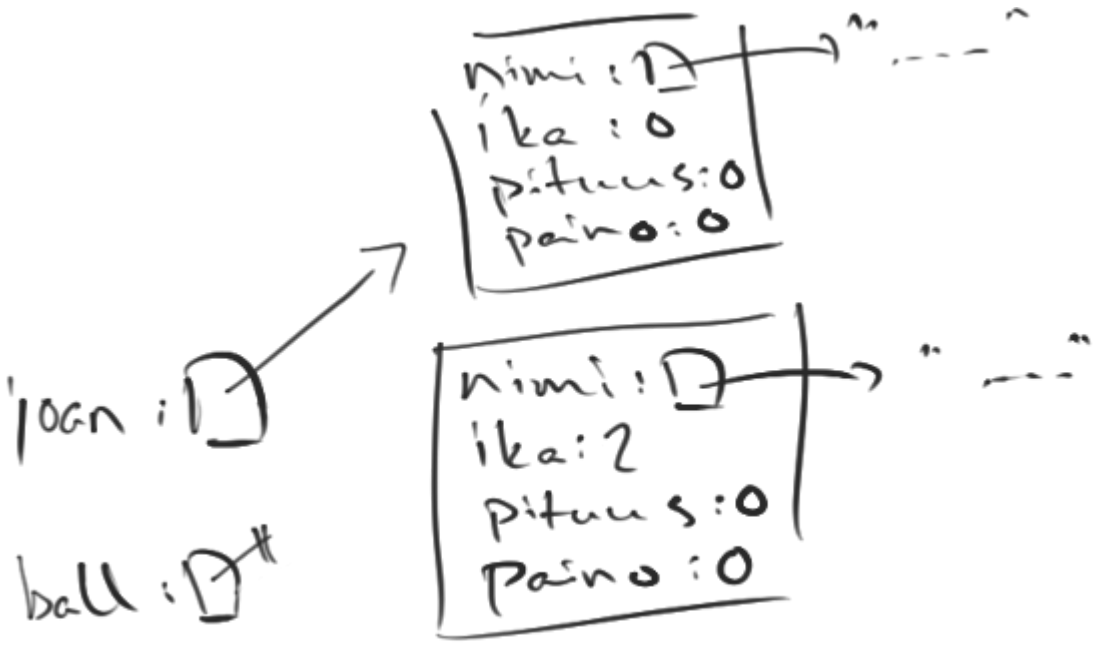


`null` value of a reference variable

Let's extend the example further by setting the value of the reference variable `ball` to `null`, i.e. a reference "to nothing". The `null` reference can be set as the value of any reference type variable.

```
Person joan = new Person("Joan Ball");
System.out.println(joan);

Person ball = joan;
ball.growOlder();
ball.growOlder();

System.out.println(joan);

joan = new Person("Joan B.");
System.out.println(joan);

ball = null;
```

The situation of the program after the last line is depicted below.



The object whose name is Joan Ball is referred to by nobody. In other words, the object has become "garbage". In the Java programming language the programmer need not worry about the program's memory use. From time to time, the automatic garbage collector of the Java language cleans up the objects that have become garbage. If the garbage collection did not happen, the garbage objects would reserve a memory location until the end of the program execution.

Let's see what happens when we try to print a variable that references "nothing" i.e. `null`.

```
Person joan = new Person("Joan Ball");
System.out.println(joan);

Person ball = joan;
ball.growOlder();
ball.growOlder();

System.out.println(joan);

joan = new Person("Joan B.");
System.out.println(joan);

ball = null;
System.out.println(ball);
```

Sample output

```
Joan Ball, age 0 years
Joan Ball, age 2 years
Joan B., age 0 years
null
```

Printing a `null` reference prints "null". How about if we were to try and call a method, say `growOlder`, on an object that refers to nothing:

```
Person joan = new Person("Joan Ball");
System.out.println(joan);

joan = null;
joan.growOlder();
```

The result:

Sample output

```
Joan Ball, age 0 years
Exception in thread "main" java.lang.NullPointerException
at Main.main(Main.java:(row))
Java Result: 1
```

Bad things follow. This could be the first time you have seen the text **NullPointerException**. In the course of the program, there occured an error indicating that we called a method on a variable that refers to nothing.

We promise that this is not the last time you will encounter the previous error. When you do, the first step is to look for variables whose value could be `null`. Fortunately, the error message is useful: it tells which row caused the error. Try it out yourself!

Programming exercise:
NullPointerException

Points
1/1

Implement a program that causes the NullPointerException error. The error should occur directly after starting the program — don't wait to read input from the user, for instance.

Exercise submission instructions

How to see the solution

Object as a method parameter

We have seen both primitive and reference variables act as method parameters. Since objects are reference variables, any type of object can be defined to be a method parameter. Let's take a look at a practical demonstration.

Amusement park rides only permit people who are taller than a certain height. The limit is not the same for all attractions. Let's create a class representing an amusement park ride. When creating a new object, the constructor receives as parameters the name of the ride, and the smallest height that permits entry to the ride.

```
public class AmusementParkRide {
    private String name;
    private int lowestHeight;

    public AmusementParkRide(String name, int lowestHeight) {
        this.name = name;
        this.lowestHeight = lowestHeight;
    }

    public String toString() {
        return this.name + ", minimum height: " + this.lowestHeight;
    }
}
```

Then let's write a method that can be used to check if a person is allowed to enter the ride, so if they are tall enough. The method returns `true` if the person given as the parameter is permitted access, and `false` otherwise.

Below, it is assumed that Person has the method `public int getHeight()` that returns the height of the person.

```
public class AmusementParkRide {
    private String name;
    private int lowestHeight;

    public AmusementParkRide(String name, int lowestHeight) {
        this.name = name;
        this.lowestHeight = lowestHeight;
    }

    public boolean allowedToRide(Person person) {
        if (person.getHeight() < this.lowestHeight) {
            return false;
        }

        return true;
    }

    public String toString() {
        return this.name + ", minimum height: " + this.lowestHeight;
    }
}
```

So the method `allowedToRide` of an AmusementParkRide object is given a `Person` object as a parameter. Like earlier, the value of the variable — in this case, a reference — is copied for the method to use. The method handles a copied reference, and it calls the `getHeight` method of the person passed as a parameter.

Below is an example main program where the amusement park ride method is called twice: first the supplied parameter is a person object `matt`, and then a person object `jasper`:

```
Person matt = new Person("Matt");
matt.setWeight(86);
matt.setHeight(180);
```

```
Person jasper = new Person("Jasper");
jasper.setWeight(34);
jasper.setHeight(132);

AmusementParkRide waterTrack = new AmusementParkRide("Water track", 140);

if (waterTrack.allowedToRide(matt)) {
    System.out.println(matt.getName() + " may enter the ride");
} else {
    System.out.println(matt.getName() + " may not enter the ride");
}

if (waterTrack.allowedToRide(jasper)) {
    System.out.println(jasper.getName() + " may enter the ride");
} else {
    System.out.println(jasper.getName() + " may not enter the ride");
}

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

The output of the program is:

Sample output

```
Matt may enter the ride
Jasper may not enter the ride
Water track, minimum height: 140
```

What if we wanted to know how many people have taken the ride?

Let's add an object variable to the amusement park ride. It keeps track of the number of people that were permitted to enter.

```
public class AmusementParkRide {
    private String name;
    private int lowestHeight;
    private int visitors;

    public AmusementParkRide(String name, int lowestHeight) {
        this.name = name;
        this.lowestHeight = lowestHeight;
        this.visitors = 0;
    }

    public boolean allowedToRide(Person person) {
```

```
        if (person.getHeight() < this.lowestHeight) {
            return false;
        }

        this.visitors++;
        return true;
    }

    public String toString() {
        return this.name + ", minimum height: " + this.lowestHeight +
            ", visitors: " + this.visitors;
    }
}
```

Now the previously used example program also keeps track of the number of visitors who have experienced the ride.

```
Person matt = new Person("Matt");
matt.setWeight(86);
matt.setHeight(180);

Person jasper = new Person("Jasper");
jasper.setWeight(34);
jasper.setHeight(132);

AmusementParkRide waterTrack = new AmusementParkRide("Water track", 140);

if (waterTrack.allowedToRide(matt)) {
    System.out.println(matt.getName() + " may enter the ride");
} else {
    System.out.println(matt.getName() + " may not enter the ride");
}

if (waterTrack.allowedToRide(jasper)) {
    System.out.println(jasper.getName() + " may enter the ride");
} else {
    System.out.println(jasper.getName() + " may not enter the ride");
}

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

The output of the program is:

Sample output

Matt may enter the ride
Jasper may not enter the ride
Water track, minimum height: 140, visitors: 1

Assisted creation of constructors, getters, and setters

Development environments can help the programmer. If you have created object variables for a class, creating constructors, getters, and setters can be done almost automatically.

Go inside the code block of the class, but outside of all the methods, and simultaneously press ctrl and space. If your class has e.g. an object variable `balance`, NetBeans offers the option to generate the getter and setter methods for the object variable, and a constuctor that assigns an initial value for that variable.

On some Linux machines, like on the ones on the Kumpula campus (University of Helsinki), this feature is triggered by simultaneously pressing ctrl, alt, and space.

Programming exercise:
Health station (3 parts)

Points
3/3

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

In the exercise base there is the class `Person`, which we are already quite familiar with. There is also an outline for the class `HealthStation`. Health station objects process people in different ways, they e.g. weigh and feed people. In this exercise we will construct a health station. The code of the Person class should not be modified in this exercise!

Part 1: Weighing people

In the outline of the Health station there is an outline for the method `weigh`:

```
public class HealthStation {

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

The method receives a person as a parameter, and it is meant to return to its caller the weight of that person. The weight information can be found by calling a suitable method of the person `person`. **So your task is to complete the code of the method!**

Here is a main program where a health station weight two people:

```
public static void main(String[] args) {
    // example main program for the first section of the exercise

    HealthStation childrensHospital = new HealthStation();

    Person ethan = new Person("Ethan", 1, 110, 7);
    Person peter = new Person("Peter", 33, 176, 85);

    System.out.println(ethan.getName() + " weight: " + childrensHospital.we
    System.out.println(peter.getName() + " weight: " + childrensHospital.we
}
```

The output should be the following:

Sample output

```
Ethan's weight: 7 kilos
Peter's weight: 85 kilos
```

Part 2: Feeding

It is possible to modify the state of the object that is received as a parameter. Write a method called `public void feed(Person person)` for the health station. It should increase the weight of the parameter person by one.

Following is an example where people are weighed first, and then ethan is fed three times in the children's hospital. After this the people are weighed again:

```
public static void main(String[] args) {
    HealthStation childrensHospital = new HealthStation();

    Person ethan = new Person("Ethan", 1, 110, 7);
    Person peter = new Person("Peter", 33, 176, 85);

    System.out.println(ethan.getName() + " weight: " + childrensHospital.we
    System.out.println(peter.getName() + " weight: " + childrensHospital.we

    childrensHospital.feed(ethan);
    childrensHospital.feed(ethan);
    childrensHospital.feed(ethan);

    System.out.println("");

    System.out.println(ethan.getName() + " weight: " + childrensHospital.we
    System.out.println(peter.getName() + " weight: " + childrensHospital.we
}
```

The output should reveal that Ethan's weight has increased by three:

Sample output

```
Ethan weight: 7 kilos
Peter weight: 85 kilos

Ethan weight: 10 kilos
Peter weight: 85 kilos
```

Part 3: Counting weighings

Create a new method called `public int weighings()` for the health station. It should tell how many weighings the health station has performed. *NB! You will need a new object variable for counting the number of weighings!*. Test main program:

```
public static void main(String[] args) {

    HealthStation childrensHospital = new HealthStation();

    Person ethan = new Person("Ethan", 1, 110, 7);
    Person peter = new Person("Peter", 33, 176, 85);

    System.out.println("weighings performed: " + childrensHospital.weighing

    childrensHospital.weigh(ethan);
    childrensHospital.weigh(peter);

    System.out.println("weighings performed: " + childrensHospital.weighing

    childrensHospital.weigh(ethan);
    childrensHospital.weigh(ethan);
    childrensHospital.weigh(ethan);
    childrensHospital.weigh(ethan);

    System.out.println("weighings performed: " + childrensHospital.weighing
}
```

The output is:

Sample output

```
weighings performed: 0
weighings performed: 2
weighings performed: 6
```

Exercise submission instructions

How to see the solution

Programming exercise:

Card payments (4 sections)

Points 4/4

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

Part 1: "Dumb" payment card

In a previous part we created a class called PaymentCard. The card had methods for eating affordably and heartily, and also for adding money to the card.

However, there was a problem with the PaymentCard class that is implemented in this fashion. The card knew the prices of the different lunches, and therefore was able to decrease the balance by the proper amount. What about if the prices are raised? Or new items are added to the list of offered products? A change in the pricing would mean that all the existing cards would have to be replaced with new cards that are aware of the new prices.

An improved solution is to make the cards "dumb"; unaware of the prices and products that are sold, and only keeping track of their balance. All the intelligence is better placed in separate objects, payment terminals.

Let's first implement the "dumb" version of the PaymentCard. The card only has methods for asking for the balance, adding money, and taking money. Complete the method `public boolean takeMoney(double amount)` in the class below (and found in the exercise template), using the following as a guide:

```
public class PaymentCard {
    private double balance;

    public PaymentCard(double balance) {
        this.balance = balance;
    }
```

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

    public void addMoney(double increase) {
        this.balance = this.balance + increase;
    }

    public boolean takeMoney(double amount) {
        // implement the method so that it only takes money from the card i
        // the balance is at least the amount parameter.
        // returns true if successful and false otherwise
    }
```

Test main program:

```
public class MainProgram {
    public static void main(String[] args) {
        PaymentCard petesCard = new PaymentCard(10);

        System.out.println("money " + petesCard.balance());
        boolean wasSuccessful = petesCard.takeMoney(8);
        System.out.println("successfully withdrew: " + wasSuccessful);
        System.out.println("money " + petesCard.balance());

        wasSuccessful = petesCard.takeMoney(4);
        System.out.println("successfully withdrew: " + wasSuccessful);
        System.out.println("money " + petesCard.balance());
    }
}
```

The output should be like below

Sample output

```
money 10.0
successfully took: true
money 2.0
successfully took: false
money 2.0
```

Part 2: Payment terminal and cash

When visiting a student cafeteria, the customer pays either with cash or with a payment card. The cashier uses a payment terminal to charge the card or to process the cash payment. First, let's create a terminal that's suitable for cash payments.

The outline of the payment terminal. The comments inside the methods tell the wanted functionality:

```
public class PaymentTerminal {
    private double money;  // amount of cash
    private int affordableMeals; // number of sold affordable meals
    private int heartyMeals;  // number of sold hearty meals

    public PaymentTerminal() {
        // register initially has 1000 euros of money
    }

    public double eatAffordably(double payment) {
        // an affordable meal costs 2.50 euros
        // increase the amount of cash by the price of an affordable meal a
        // if the payment parameter is not large enough, no meal is sold an
    }

    public double eatHeartily(double payment) {
        // a hearty meal costs 4.30 euros
        // increase the amount of cash by the price of a hearty meal and re
        // if the payment parameter is not large enough, no meal is sold an
    }

    public String toString() {
        return "money: " + money + ", number of sold afforable meals: " + a
    }
}
```

The terminal starts with 1000 euros in it. Implement the methods so they work correctly, using the basis above and the example prints of the main program below.

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

        double change = unicafeExactum.eatAffordably(10);
```

```
        System.out.println("remaining change " + change);

        change = unicafeExactum.eatAffordably(5);
        System.out.println("remaining change " + change);

        change = unicafeExactum.eatHeartily(4.3);
        System.out.println("remaining change " + change);

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

Sample output

```
remaining change: 7.5
remaining change: 2.5
remaining change: 0.0
money: 1009.3, number of sold afforable meals: 2, number of sold hearty meals: 1
```

Part 3: Card payments

Let's extend our payment terminal to also support card payments. We are going to create new methods for the terminal. It receives a payment card as a parameter, and decreases its balance by the price of the meal that was purchased. Here are the outlines for the methods, and instructions for completing them.

```
public class PaymentTerminal {
    // ...

    public boolean eatAffordably(PaymentCard card) {
        // an affordable meal costs 2.50 euros
        // if the payment card has enough money, the balance of the card is
        // otherwise false is returned
    }

    public boolean eatHeartily(PaymentCard card) {
        // a hearty meal costs 4.30 euros
        // if the payment card has enough money, the balance of the card is
        // otherwise false is returned
    }
```

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

NB: card payments don't increase the amount of cash in the register

Below is a main program to test the classes, and the output that is desired:

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

        double change = unicafeExactum.eatAffordably(10);
        System.out.println("remaining change: " + change);

        PaymentCard annesCard = new PaymentCard(7);

        boolean wasSuccessful = unicafeExactum.eatHeartily(annesCard);
        System.out.println("there was enough money: " + wasSuccessful);
        wasSuccessful = unicafeExactum.eatHeartily(annesCard);
        System.out.println("there was enough money: " + wasSuccessful);
        wasSuccessful = unicafeExactum.eatAffordably(annesCard);
        System.out.println("there was enough money: " + wasSuccessful);

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

Sample output

```
remaining change: 7.5
there was enough money: true
there was enough money: false
there was enough money: true
money: 1002.5, number of sold afforable meals: 2, number of sold hearty meals: 1
```

Part 4: Adding money

Let's create a method for the terminal that can be used to add money to a payment card. Recall that the payment that is received when

adding money to the card is stored in the register. The basis for the method:

```
public void addMoneyToCard(PaymentCard card, double sum) {
    // ...
}
```

A main program to illustrate:

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

        PaymentCard annesCard = new PaymentCard(2);

        System.out.println("amount of money on the card is " + annesCard.ba

        boolean wasSuccessful = unicafeExactum.eatHeartily(annesCard);
        System.out.println("there was enough money: " + wasSuccessful);

        unicafeExactum.addMoneyToCard(annesCard, 100);

        wasSuccessful = unicafeExactum.eatHeartily(annesCard);
        System.out.println("there was enough money: " + wasSuccessful);

        System.out.println("amount of money on the card is " + annesCard.ba

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

Sample output

money: 1000.0, number of sold afforable meals: 0, number of sold hearty meals: 0
amount of money on the card is 2.0 euros
there was enough money: false
there was enough money: true
amount of money on the card is 97.7 euros

money: 1100.0, number of sold afforable meals: 0, number of sold hearty meals: 1

Exercise submission instructions

How to see the solution

Object as object variable

Objects may contain references to objects.

Let's keep working with people, and add a birthday to the person class. A natural way of representing a birthday is to use a `Date` class. We could use the classname `Date`, but for the sake of avoiding confusion with the similarly named existing Java class, we will use `SimpleDate` here.

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

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

    public int getDay() {
        return this.day;
    }

    public int getMonth() {
        return this.month;
    }

    public int getYear() {
        return this.year;
    }

    @Override
```

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

Since we know the birthday, there is no need to store that age of a person as a separate object variable. The age of the person can be inferred from their birthday. Let's assume that the class `Person` now has the following variables.

```
public class Person {
    private String name;
    private SimpleDate birthday;
    private int weight = 0;
    private int length = 0;

// ...
```

Let's create a new Person constructor that allows for setting the birthday:

```
public Person(String name, SimpleDate date) {
    this.name = name;
    this.birthday = date;
}
```

Along with the constructor above, we could give Person another constructor where the birthday was given as integers.

```
public Person(String name, int day, int month, int year) {
    this.name = name;
    this.birthday = new SimpleDate(day, month, year);
}
```

The constructor receives as parameters the different parts of the date (day, month, year). They are used to create a date object, and finally the reference to that date is copied as the value of the object variable `birthday`.

Let's modify the `toString` method of the Person class so that instead of age, the method returns the birthday:

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

Let's see how the updated Person class works.

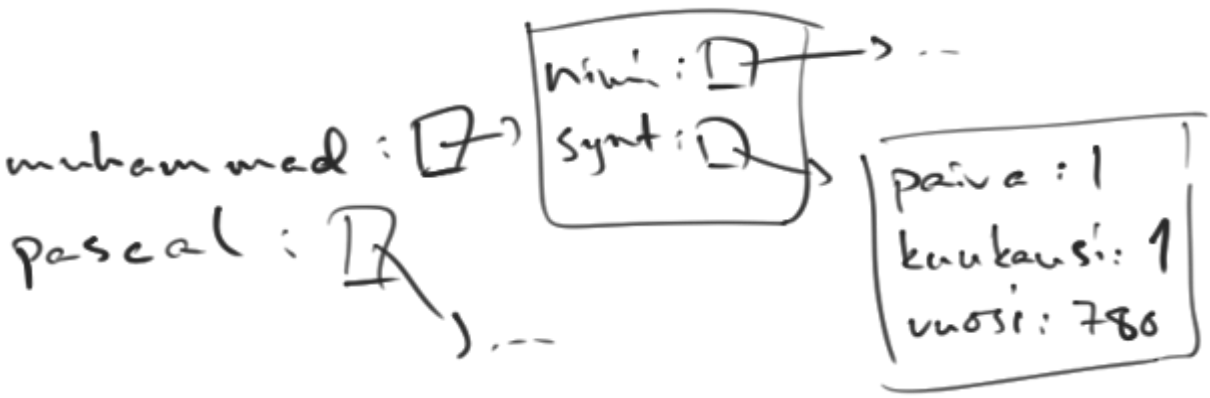
```
SimpleDate date = new SimpleDate(1, 1, 780);
Person muhammad = new Person("Muhammad ibn Musa al-Khwarizmi", date);
Person pascal = new Person("Blaise Pascal", 19, 6, 1623);

System.out.println(muhammad);
System.out.println(pascal);
```

Sample output

```
Muhammad ibn Musa al-Khwarizmi, born on 1.1.780
Blaise Pascal, born on 19.6.1623
```

Now a person object has object variables `name` and `birthday`. The variable `name` is a string, which itself is an object; the variable `birthday` is a SimpleDate object.

Both variables contain a reference to an object. Therefore a person object contains two references. In the image below, weight and height are not considered at all.



So the main program is connected to two Person objects by strands. A person has a name and a birthday. Since both variables are objects, these attributes exist at the other ends of the strands.

Birthday appears to be a good extension to the Person class. Earlier we noted that the object variable `age` can be calculated with birthday, so it was removed.

Date in Java programs

In the section above, we use our own class `SimpleDate` to represent date, because it is suitable for illustrating and practising the operation of objects. If you want to handle dates in your own programs, it's worth reading about the premade Java class LocalDate. It contains a significant amount of functionality that can be used to handle dates.

For example, the current date can be used with the existing `LocalDate` class in the following manner:

```
import java.time.LocalDate;

public class Example {

    public static void main(String[] args) {

        LocalDate now = LocalDate.now();
        int year = now.getYear();
        int month = now.getMonthValue();
        int day = now.getDayOfMonth();

        System.out.println("today is  " + day + "." + month + "." + year);

    }
}
```

Programming exercise:
Biggest pet shop

Points
1/1

Two classes, `Person` and `Pet`, are included in the exercise template. Each person has one pet. Modify the `public String toString` method of the `Person` class so that the string it returns tells the pet's name and breed in addition to the person's own name.

```
Pet lucy = new Pet("Lucy", "golden retriever");
Person leo = new Person("Leo", lucy);

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

Sample output

```
Leo, has a friend called Lucy (golden retriever)
```

Exercise submission instructions

How to see the solution

Object of same type as method parameter

We will continue working with the `Person` class. We recall that persons know their birthdays:

```
public class Person {

    private String name;
    private SimpleDate birthday;
    private int height;
    private int weight;

    // ...
}
```

We would like to compare the ages of two people. The comparison can be done in multiple ways. We could, for instance, implement a method called `public int ageAsYears()` for the Person class; in that case, the comparison would happen in the following manner:

```
Person muhammad = new Person("Muhammad ibn Musa al-Khwarizmi", 1, 1, 780);
Person pascal = new Person("Blaise Pascal", 19, 6, 1623);

if (muhammad.ageAsYears() > pascal.ageAsYears()) {
    System.out.println(muhammad.getName() + " is older than " + pascal.getName(
}
```

We are now going to learn a more "object-oriented" way to compare the ages of people.

We are going to create a new method `boolean olderThan(Person compared)` for the Person class. It can be used to compare a certain person object to the person supplied as the parameter based on their ages.

The method is meant to be used like this:

```
Person muhammad = new Person("Muhammad ibn Musa al-Khwarizmi", 1, 1, 780);
Person pascal = new Person("Blaise Pascal", 19, 6, 1623);

if (muhammad.olderThan(pascal)) {  //  same as muhammad.olderThan(pascal)==true
    System.out.println(muhammad.getName() + " is older than " + pascal.getName(
} else {
    System.out.println(muhammad.getName() + " is not older than " + pascal.getN
}
```

The program above asks if al-Khwarizmi is older than Pascal. The method `olderThan` returns `true` if the object that is used to call the method (`object.olderThan(objectGivenAsParameter)`) is older than the object given as the parameter, and `false` otherwise.

In practice, we call the `olderThan` method of the object that matches "Muhammad ibn Musa al-Khwarizmi", which is referred to by the variable `muhammad`. The reference `pascal`, matching the object "Blaise Pascal", is given as the parameter to that method.

The program prints:

Sample output

> Muhammad ibn Musa al-Khwarizmi is older than Blaise Pascal

The method `olderThan` receives a person object as its parameter. More precisely, the variable that is defined as the method parameter receives a copy of the value contained by the given variable. That value is a reference to an object, in this case.

The implementation of the method is illustrated below. Note that the **method may return a value in more than one place** — here the comparison has been divided into multiple parts based on the years, the months, and the days:

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

    public boolean olderThan(Person compared) {
        // 1. First compare years
        int ownYear = this.getBirthday().getYear();
        int comparedYear = compared.getBirthday().getYear();

        if (ownYear < comparedYear) {
            return true;
        }

        if (ownYear > comparedYear) {
            return false;
        }

        // 2. Same birthyear, compare months
        int ownMonth = this.getBirthday().getMonth();
        int comparedMonth = compared.getBirthday().getMonth();

        if (ownMonth < comparedMonth) {
            return true;
        }

        if (ownMonth > comparedMonth) {
            return false;
        }

        // 3. Same birth year and month, compare days
        int ownDay = this.getBirthday().getDay();
        int comparedDay = compared.getBirthday().getDay();

        if (ownDay < comparedDay) {
```

```
            return true;
        }

        return false;
    }
}
```

Let's pause for a moment to consider abstraction, one of the principles of object-oriented programming. The idea behind abstraction is to conceptualize the programming code so that each concept has its own clear responsibilities. When viewing the solution above, however, we notice that the comparison functionality would be better placed inside the `SimpleDate` class instead of the `Person` class.

We'll create a method called `public boolean before(SimpleDate compared)` for the class `SimpleDate`. The method returns the value `true` if the date given as the parameter is after (or on the same day as) the date of the object whose method is called.

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

    public SimpleDate(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;
    }

    // used to check if this date object (`this`) is before
    // the date object given as the parameter (`compared`)
    public boolean before(SimpleDate compared) {
        // first compare years
        if (this.year < compared.year) {
            return true;
        }

        if (this.year > compared.year) {
            return false;
        }
```

```
        // years are same, compare months
        if (this.month < compared.month) {
            return true;
        }

        if (this.month > compared.month) {
            return false;
        }

        // years and months are same, compare days
        if (this.day < compared.day) {
            return true;
        }

        return false;
    }
}
```

Even though the object variables `year`, `month`, and `day` are encapsulated (`private`) object variables, we can read their values by writing `compared.*variableName*`. This is because a `private` variable can be accessed from all the methods contained by that class. Notice that the syntax here matches calling some object method. Unlike when calling a method, we refer to a field of an object, so the parentheses that indicate a method call are not written.

An example of how to use the method:

```
public static void main(String[] args) {
    SimpleDate d1 = new SimpleDate(14, 2, 2011);
    SimpleDate d2 = new SimpleDate(21, 2, 2011);
    SimpleDate d3 = new SimpleDate(1, 3, 2011);
    SimpleDate d4 = new SimpleDate(31, 12, 2010);

    System.out.println(d1 + " is earlier than " + d2 + ": " + d1.before(d2));
    System.out.println(d2 + " is earlier than " + d1 + ": " + d2.before(d1));

    System.out.println(d2 + " is earlier than " + d3 + ": " + d2.before(d3));
    System.out.println(d3 + " is earlier than " + d2 + ": " + d3.before(d2));

    System.out.println(d4 + " is earlier than " + d1 + ": " + d4.before(d1));
    System.out.println(d1 + " is earlier than " + d4 + ": " + d1.before(d4));
}
```

Sample output

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

Let's tweak the method olderThan of the Person class so that from here on out, we take use of the comparison functionality that date objects provide.

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

    public boolean olderThan(Person compared) {
        if (this.birthday.before(compared.getBirthday())) {
            return true;
        }

        return false;

        // or return more directly:
        // return this.birthday.before(compared.getBirthday());
    }
}
```

Now the concrete comparison of dates is implemented in the class that it logically (based on the class names) belongs to.

Programming exercise:

Comparing apartments (3 parts)

Points 3/3

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

In the estate agent's information system, an apartment that is on sale is represented by an object that is instantiated from the following class:

```
public class Apartment {
    private int rooms;
    private int squares;
    private int pricePerSquare;

    public Apartment(int rooms, int squares, int pricePerSquare) {
        this.rooms = rooms;
        this.squares = squares;
        this.pricePerSquare = pricePerSquare;
    }
}
```

Your task is to create a few methods that can be used to compare apartments that are being sold.

Part 1: Comparing sizes

Create a method `public boolean largerThan(Apartment compared)` that returns true if the apartment object whose method is called has a larger total area than the apartment object that is being compared.

An example of how the method should work:

```
Apartment manhattanStudioApt = new Apartment(1, 16, 5500);
Apartment atlantaTwoBedroomApt = new Apartment(2, 38, 4200);
Apartment bangorThreeBedroomApt = new Apartment(3, 78, 2500);

System.out.println(manhattanStudioApt.largerThan(atlantaTwoBedroomApt));
System.out.println(bangorThreeBedroomApt.largerThan(atlantaTwoBedroomApt));
```

Part 2: Price difference

Create a method `public int priceDifference(Apartment compared)` that returns the price difference of the apartment object whose method was called and the apartment object received as the

parameter. The price difference is the absolute value of the difference of the prices (price can be calculated by multiplying the price per square by the number of squares).

An example of how the method should work:

```
Apartment manhattanStudioApt = new Apartment(1, 16, 5500);
Apartment atlantaTwoBedroomApt = new Apartment(2, 38, 4200);
Apartment bangorThreeBedroomApt = new Apartment(3, 78, 2500);

System.out.println(manhattanStudioApt.priceDifference(atlantaTwoBedroomApt)
System.out.println(bangorThreeBedroomApt.priceDifference(atlantaTwoBedroomA
```

Part 3: More expensive?

Write a method `public boolean moreExpensiveThan(Apartment compared)` that returns true if the apartment object whose method is called is more expensive than the apartment object being compared.

An example of how the method should work:

```
Apartment manhattanStudioApt = new Apartment(1, 16, 5500);
Apartment atlantaTwoBedroomApt = new Apartment(2, 38, 4200);
Apartment bangorThreeBedroomApt = new Apartment(3, 78, 2500);

System.out.println(manhattanStudioApt.moreExpensiveThan(atlantaTwoBedroomAp
System.out.println(bangorThreeBedroomApt.moreExpensiveThan(atlantaTwoBedroo
```

Exercise submission instructions

How to see the solution

Comparing the equality of objects (equals)

While working with strings, we learned that strings must be compared using the `equals` method. This is how it's done.

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

System.out.println("Enter two words, each on its own line.")
String first = scanner.nextLine();
String second = scanner.nextLine();

if (first.equals(second)) {
    System.out.println("The words were the same.");
} else {
    System.out.println("The words were not the same.");
}
```

With primitive variables such as `int`, comparing two variables can be done with two equality signs. This is because the value of a primitive variable is stored directly in the "variable's box". The value of reference variables, in contrast, is an address of the object that is referenced; so the "box" contains a reference to the memory location. Using two equality signs compares the equality of the values stored in the "boxes of the variables" — with reference variables, such comparisons would examine the equality of the memory references.

The method `equals` is similar to the method `toString` in the respect that it is available for use even if it has not been defined in the class. The default implementation of this method compares the equality of the references. Let's observe this with the help of the previously written `SimpleDate` class.

```
SimpleDate first = new SimpleDate(1, 1, 2000);
SimpleDate second = new SimpleDate(1, 1, 2000);
SimpleDate third = new SimpleDate(12, 12, 2012);
SimpleDate fourth = first;

if (first.equals(first)) {
    System.out.println("Variables first and first are equal");
} else {
    System.out.println("Variables first and first are not equal");
}

if (first.equals(second)) {
    System.out.println("Variables first and second are equal");
```

```
} else {
    System.out.println("Variables first and second are not equal");
}

if (first.equals(third)) {
    System.out.println("Variables first and third are equal");
} else {
    System.out.println("Variables first and third are not equal");
}

if (first.equals(fourth)) {
    System.out.println("Variables first and fourth are equal");
} else {
    System.out.println("Variables first and fourth are not equal");
}
```

Sample output

```
Variables first and first are equal
Variables first and second are not equal
Variables first and third are not equal
Variables first and fourth are equal
```

There is a problem with the program above. Even though two dates (first and second) have exactly the same values for object variables, they are different from each other from the point of view of the default `equals` method.

If we want to be able to compare two objects of our own design with the `equals` method, that method must be defined in the class. The method `equals` is defined as a method that returns a boolean type value — the return value indicates whether the objects are equal.

The `equals` method is implemented in such a way that it can be used to compare the current object with any other object. The method receives an `Object`-type object as its single parameter — all objects are `Object`-type, in addition to their own type. The `equals` method first compares if the addresses are equal: if so, the objects are equal. After this, we examine if the types of the objects are the same: if not, the objects are not equal. Next, the `Object`-type object passed as the parameter is converted to the type of the object that is being examined by using a type cast, so that the

values of the object variables can be compared. Below the equality comparison has been implemented for the SimpleDate class.

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

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

    public int getDay() {
        return this.day;
    }

    public int getMonth() {
        return this.month;
    }

    public int getYear() {
        return this.year;
    }

    public boolean equals(Object compared) {
        // if the variables are located in the same position, they are equal
        if (this == compared) {
            return true;
        }

        // if the type of the compared object is not SimpleDate, the objects ar
        if (!(compared instanceof SimpleDate)) {
            return false;
        }

        // convert the Object type compared object
        // into a SimpleDate type object called comparedSimpleDate
        SimpleDate comparedSimpleDate = (SimpleDate) compared;

        // if the values of the object variables are the same, the objects are
        if (this.day == comparedSimpleDate.day &&
            this.month == comparedSimpleDate.month &&
            this.year == comparedSimpleDate.year) {
            return true;
        }
```

```
        // otherwise the objects are not equal
        return false;
    }

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

Building a similar comparison functionality is possible for Person objects too. Below, the comparison has been implemented for Person objects that don't have a separate SimpleDate object. Notice that the names of people are strings (i.e. objects), so we use the `equals` method for comparing them.

```
public class Person {

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

    // constructors and methods


    public boolean equals(Object compared) {
        // if the variables are located in the same position, they are equal
        if (this == compared) {
            return true;
        }

        // if the compared object is not of type Person, the objects are not eq
        if (!(compared instanceof Person)) {
            return false;
        }

        // convert the object into a Person object
        Person comparedPerson = (Person) compared;

        // if the values of the object variables are equal, the objects are equ
        if (this.name.equals(comparedPerson.name) &&
            this.age == comparedPerson.age &&
            this.weight == comparedPerson.weight &&
            this.height == comparedPerson.height) {
            return true;
```

```
        }

        // otherwise the objects are not equal
        return false;
    }

    // .. methods
```

Programming exercise:

Song

Points 1/1

In the exercise base there is a class called `Song` that can be used to create new objects that represent songs. Add to that class the `equals` method so that the similarity of songs can be examined.

```
Song jackSparrow = new Song("The Lonely Island", "Jack Sparrow", 196);
Song anotherSparrow = new Song("The Lonely Island", "Jack Sparrow", 196);

if (jackSparrow.equals(anotherSparrow)) {
    System.out.println("Songs are equal.");
}

if (jackSparrow.equals("Another object")) {
    System.out.println("Strange things are afoot.");
}
```

Sample output

```
Songs are equal
```

Exercise submission instructions

How to see the solution

Programming exercise:

Identical twins

Points
1/1

In the exercise base you can find the `Person` class that is linked with an `SimpleDate` object. Add to the class Person the method `public boolean equals (Object compared)`, which can be used to compare the similarity of people. The comparison should take into account the equality of all the variables of a person (birthday included).

NB! Recall that you cannot compare two birthday objects with equality signs!

There are no tests in the exercise template to check the correctess of the solution. Only return your answer after the comparison works as it should. Below is some code to help test the program.

```
SimpleDate date = new SimpleDate(24, 3, 2017);
SimpleDate date2 = new SimpleDate(23, 7, 2017);

Person leo = new Person("Leo", date, 62, 9);
Person lily = new Person("Lily", date2, 65, 8);

if (leo.equals(lily)) {
    System.out.println("Is this quite correct?");
}

Person leoWithDifferentWeight = new Person("Leo", date, 62, 10);

if (leo.equals(leoWithDifferentWeight)) {
    System.out.println("Is this quite correct?");
}
```

Exercise submission instructions

How to see the solution

What is Object?

Every class we create (and every ready-made Java class) inherits the class Object, even though it is not specially visible in the program code. This is why an instance of any class can be passed as a parameter to a method that receives an Object type variable as its parameter. Inheriting the Object can be seen elsewhere, too: for instance, the `toString` method exists even if you have not implemented it yourself, just as the `equals` method does.

To illustrate, the following source code compiles successfully: `equals` method can be found in the Object class inherited by all classes.

```
public class Bird {
    private String name;

    public Bird(String name) {
        this.name = name;
    }
}
```

```
Bird red = new Bird("Red");
System.out.println(red);

Bird chuck = new Bird("Chuck");
System.out.println(chuck);

if (red.equals(chuck)) {
    System.out.println(red + " equals " + chuck);
}
```

Object equality and lists

Let's examine how the `equals` method is used with lists. Let's assume we have the previously described class `Bird` without any `equals` method.

```
public class Bird {
    private String name;
```

```
    public Bird(String name) {
        this.name = name;
    }
}
```

Let's create a list and add a bird to it. After this we'll check if that bird is contained in it.

```
ArrayList<Bird> birds = new ArrayList<>()
Bird red = new Bird("Red");

if (birds.contains(red)) {
    System.out.println("Red is on the list.");
} else {
    System.out.println("Red is not on the list.");
}

birds.add(red);
if (birds.contains(red)) {
    System.out.println("Red is on the list.");
} else {
    System.out.println("Red is not on the list.");
}


System.out.println("However!");

red = new Bird("Red");
if (birds.contains(red)) {
    System.out.println("Red is on the list.");
} else {
    System.out.println("Red is not on the list.");
}
```

Sample output

```
Red is not on the list.
Red is on the list.
However!
Red is not on the list.
```

We can notice in the example above that we can search a list for our own objects. First, when the bird had not been added to the list, it is not found

— and after adding it is found. When the program switches the `red` object into a new object, with exactly the same contents as before, it is no longer equal to the object on the list, and therefore cannot be found on the list.

The `contains` method of a list uses the `equals` method that is defined for the objects in its search for objects. In the example above, the `Bird` class has no definition for that method, so a bird with exactly the same contents — but a different reference — cannot be found on the list.

Let's implement the `equals` method for the class `Bird`. The method examines if the names of the objects are equal — if the names match, the birds are thought to be equal.

```
public class Bird {
    private String name;

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

    public boolean equals(Object compared) {
        // if the variables are located in the same position, they are equal
        if (this == compared) {
            return true;
        }

        // if the compared object is not of type Bird, the objects are not equa
        if (!(compared instanceof Bird)) {
            return false;
        }

        // convert the object to a Bird object
        Bird comparedBird = (Bird) compared;

        // if the values of the object variables are equal, the objects are, to
        return this.name.equals(comparedBird.name);

        /*
        // the comparison of names above is equal to
        // the following code

        if (this.name.equals(comparedBird.name)) {
            return true;
        }

        // otherwise the objects are not equal
```

```
        return false;
        */
    }
```

Now the contains list method recognizes birds with identical contents.

```
ArrayList<Bird> birds = new ArrayList<>()
Bird red = new Bird("Red");

if (birds.contains(red)) {
    System.out.println("Red is on the list.");
} else {
    System.out.println("Red is not on the list.");
}

birds.add(red);
if (birds.contains(red)) {
    System.out.println("Red is on the list.");
} else {
    System.out.println("Red is not on the list.");
}


System.out.println("However!");

red = new Bird("Red");
if (birds.contains(red)) {
    System.out.println("Red is on the list.");
} else {
    System.out.println("Red is not on the list.");
}
```

Sample output

```
Red is not on the list.
Red is on the list.
However!
Red is on the list.
```

Programming exercise:

Books

Points
1/1

There is a program in the exercise base that asks for books from the user and adds them to a list.

Modify the program so that books that are already on the list are not added to it again. Two books should be considered the same if they have the same name and publication year.

Example print

Sample output

```
Name (empty will stop):
Bossypants
Publication year:
2013
Name (empty will stop):
Seriously...I'm Kidding
Publication year:
2012
Name (empty will stop):
Seriously...I'm Kidding
Publication year:
2012
The book is already on the list. Let's not add the same book again.
Name (empty will stop):

Thank you! Books added: 2
```

Exercise submission instructions

How to see the solution

Programming exercise:

Archive (2 parts)

Points
2/2

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

In this exercise you get to implement a program that can be used to handle an archive. Several items can be added to it. When no more items are added, all the items in the archive are printed.

Part 1: Adding and listing items

The program should read items from the user. When all the items from the user have been read, the program prints the information of each item.

For each item, its identifier and name should be read. If the identifier or name is empty, the program stops asking for input, and prints all the item information.

Example print:

Sample output

```
Identifier? (empty will stop)
B07H8ND8HH
Name? (empty will stop)
He-Man figure
Identifier? (empty will stop)
B07H8ND8HH
Name? (empty will stop)
He-Man
Identifier? (empty will stop)
B07NQFMZYG
Name? (empty will stop)
He-Man figure
Identifier? (empty will stop)
B07NQFMZYG
```

```
Name? (empty will stop)
He-Man figure
Identifier? (empty will stop)

==Items==
B07H8ND8HH: He-Man figure
B07H8ND8HH: He-Man
B07NQFMZYG: He-Man figure
B07NQFMZYG: He-Man figure
```

The printing format of the items should be `identifier: name`.

NB! Don't print the colon (:) anywhere else in the output of the program.

Part 2: You only print once (per item)

Modify the program so that after entering the items, each item is printed at most once. Two items should be considered the same if their identifiers are the same (there can be variation in their names in different countries, for instance).

If the user enters the same item multiple times, the print uses the item that was added first.

Sample output

```
Identifier? (empty will stop)
B07H8ND8HH
Name? (empty will stop)
He-Man figure
Identifier? (empty will stop)
B07H8ND8HH
Name? (empty will stop)
He-Man
Identifier? (empty will stop)
B07NQFMZYG
Name? (empty will stop)
He-Man figure
Identifier? (empty will stop)
```

B07NQFMZYG
Name? (empty will stop)
He-Man figure
Identifier? (empty will stop)

==Items==
B07H8ND8HH: He-Man figure
B07NQFMZYG: He-Man figure

Hint! It is probably smart to add each item to the list at most once — compare the equality of the objects based on their identifiers.

Exercise submission instructions

How to see the solution

Object as a method's return value

We have seen methods return boolean values, numbers, and strings. Easy to guess, a method can return an object of any type.

In the next example we present a simple counter that has the method `clone`. The method can be used to create a clone of the counter; i.e. a new counter object that has the same value at the time of its creation as the counter that is being cloned.

```
public class Counter {
    private int value;

    // example of using multiple constructors:
    // you can call another constructor from a constructor by calling this
    // notice that the this call must be on the first line of the constructor
    public Counter() {
        this(0);
    }

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

```
    }

    public void increase() {
        this.value = this.value + 1;
    }

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

    public Counter clone() {
        // create a new counter object that receives the value of the cloned co
        Counter clone = new Counter(this.value);

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

An example of using counters follows:

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

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

Counter clone = counter.clone();

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

counter.increase();
counter.increase();
counter.increase();
counter.increase();

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

clone.increase();

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

Immediately after the cloning operation, the values contained by the clone and the cloned object are the same. However, they are two different objects, so increasing the value of one counter does not affect the value of the other in any way.

Similarly, a `Factory` object could also be used to create and return new `Car` objects. Below is a sketch of the outline of the factory — the factory also knows the makes of the cars that are created.

```
public class Factory {
    private String make;

    public Factory(String make) {
        this.make = make;
    }

    public Car procuceCar() {
        return new Car(this.make);
    }
}
```

Programming exercise:

Dating app (3 parts)

Points 3/3

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

With the exercise base the class `SimpleDate` is supplied. The date is stored with the help of the object variables `year`, `month`, and `day`:

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

    public SimpleDate(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 before(SimpleDate compared) {
        // first compare years
        if (this.year < compared.year) {
            return true;
        }

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

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

        return false;
    }
}
```

In this exercise set we will expand this class.

Part 1: Next day

Implement the method `public void advance()` that moves the date by one day. In this exercise we assume that each month has 30 day. NB! In *certain* situations you need to change the values of month and year.

Part 2: Advance specific number of days

Implement the method `public void advance(int howManyDays)` that moves the date by the number of days that is given. Use the method `advance()` that you implemented in the previous section to help you in this.

Part 3: Passing of time

Let's add the possibility to advance time to the `SimpleDate` class. Create the method `public SimpleDate afterNumberOfDays(int days)` for the class. It creates a **new** `SimpleDate` object whose date is the specified number of days greater than the object that the method was called on. You may still assume that each month has 30 days. Notice that the old date object must remain unchanged!

Since the method must create **a new object**, the structure of the code should be somewhat similar to this:

```
public SimpleDate afterNumberOfDays(int days) {
    SimpleDate newDate = new SimpleDate( ... );

    // do something..

    return newDate;
}
```

Here is an example of how the method works.

```
public static void main(String[] args) {
    SimpleDate date = new SimpleDate(13, 2, 2015);
    System.out.println("Friday of the examined week is " + pvm);

    SimpleDate newDate = date.afterNumberOfDays(7);
    int week = 1;
    while (week <= 7) {
        System.out.println("Friday after " + week + " weeks is " + newDate)
        newDate = newDate.afterNumberOfDays(7);

        week = week + 1;
    }


    System.out.println("The date after 790 days from the examined Friday is
    //    System.out.println("Try " + date.afterNumberOfDays(790));
}
```

The program prints:

Sample output

```
Friday of the examined week is 13.2.2015
Friday after 1 weeks is 20.2.2015
Friday after 2 weeks is 27.2.2015
Friday after 3 weeks is 4.3.2015
Friday after 4 weeks is 11.3.2015
Friday after 5 weeks is 18.3.2015
Friday after 6 weeks is 25.3.2015
Friday after 7 weeks is 2.4.2015
The date after 790 days from the examined Friday is ... try it out yourself!
```

NB! Instead of modifying the state of the old object we return a new one. Imagine that the `SimpleDate` class has a method `advance` that works similarly to the method we programmed, but it modifies the state of the old object. In that case the next block of code would cause problems.

```
SimpleDate now = new SimpleDate(13, 2, 2015);
SimpleDate afterOneWeek = now;
afterOneWeek.advance(7);

System.out.println("Now: " + now);
System.out.println("After one week: " + afterOneWeek);
```

The output of the program should be like this:

Sample output

```
Now: 20.2.2015
After one week: 20.2.2015
```

This is because a normal assignment only copies the reference to the object. So the objects `now` and `afterOneWeek` in the program now refer to the **one and same** `SimpleDate` object.

Exercise submission instructions

How to see the solution

Programming exercise:

Money (3 parts)

Points 3/3

NB! By submitting a solution to a part of an exercise which has multiple parts, you can get part of the exercise points. You can submit a part by using the 'submit' button on NetBeans. More on the programming exercise submission instructions: Exercise submission instructions.

In the Payment card exercise we used a double-type object variable to store the amount of money. In real applications this is not the approach you want to take, since as we have seen, calculating with doubles is not exact. A more reasonable way to handle amounts of money is create an own class for that purpose. Here is a layout for the class:

```
public class Money {

    private final int euros;
    private final int cents;

    public Money(int euros, int cents) {
        this.euros = euros;
        this.cents = cents;
    }

    public int euros() {
        return euros;
    }

    public int cents() {
        return cents;
    }

    public String toString() {
        String zero = "";
        if (cents <= 10) {
            zero = "0";
```

```
        }

        return euros + "." + zero + cents + "e";
    }
}
```

The word `final` used in the definition of object variables catches attention. The result of this word is that the values of these object variables cannot be modified after they have been set in the constructor. The objects of Money class are unchangeable so **immutable** — if we want to e.g. increase the amount of money, we must create a new object to represent that new amount of money.

Next we'll create a few operations for processing money.

Part 1: Plus

First create the method `public Money plus(Money addition)` that returns a new money object that is worth the total amount of the object whose method was called and the object that is received as the parameter.

The basis for the method is the following:

```
public Money plus(Money addition) {
    Money newMoney = new Money(...); // create a new Money object that has

    // return the new Money object
    return newMoney;
}
```

Here are some examples of how the method works.

```
Money a = new Money(10,0);
Money b = new Money(5,0);

Money c = a.plus(b);

System.out.println(a);  // 10.00e
System.out.println(b);  // 5.00e
System.out.println(c);  // 15.00e
```

```
a = a.plus(c);          // NB: a new Money object is created, and is placed
//  the old 10 euros at the end of the strand disappears and the Java garba

System.out.println(a);  // 25.00e
System.out.println(b);  // 5.00e
System.out.println(c);  // 15.00e
```

Part 2: Less

Create the method `public boolean lessThan(Money compared)` that returns true if the money-object on which the method is called on has a lesser value than the money object given as a parameter.

```
Money a = new Money(10, 0);
Money b = new Money(3, 0);
Money c = new Money(5, 0);

System.out.println(a.lessThan(b));  // false
System.out.println(b.lessThan(c));  // true
```

Part 3: Minus

Write the method `public Money minus(Money decreaser)` that returns a new money object worth the difference of the object whose method was called and the object received as the parameter. If the difference would be negative, the worth of the created money object is set to 0.

Here are examples of how the method works.

```
Money a = new Money(10, 0);
Money b = new Money(3, 50);

Money c = a.minus(b);

System.out.println(a);  // 10.00e
System.out.println(b);  // 3.50e
System.out.println(c);  // 6.50e

c = c.minus(a);       // NB: a new Money object is created, and is placed "
```

```
//  the old 6.5 euros at the end of the strand disappears and the Java garb


System.out.println(a);  // 10.00e
System.out.println(b);  // 3.50e
System.out.println(c);  // 0.00e
```

Exercise submission instructions

How to see the solution

You have reached the end of this section! Continue to the next section:

→ 5. Conclusion

Remember to check your points from the ball on the bottom-right corner of the material!

In this part:

Source code of the material

This course is created by the Agile Education Research -research group of the University of Helsinki.

Credits and about the material.

HELSINGIN YLIOPISTO
HELSINGFORS UNIVERSITET
UNIVERSITY OF HELSINKI

MASSIIVISET AVOIMET VERKKOKURSSIT
MASSIVE OPEN ONLINE COURSES · MOOC.FI