Programming exercise:

Items

Implement the class `Items` described here. **NB!** Don't modify the class `Item`.

Write a program that reads names of items from the user. If the name is empty, the program stops reading. Otherwise, the given name is used to create a new item, which you will then add to the `items` list.

Having read all the names, print all the items by using the `toString`method of the `Item` class. The implementation of the `Item` class keeps track of the time of creation, in addition to the name of the item.

An example of the working program is given below:

Sample output

```
Name: Hammer
Name: Collar
Name:

Hammer (created at: 06.07.2018 12:34:56)
Collar (created at: 06.07.2018 12:34:57)
```

Programming exercise:

Personal information

The program described here should be implemented in the class `PersonalInformationCollection`. **NB!** Do not modify the class `PersonalInformation`.

After the user has entered the last set of details (they enter an empty first name), exit the repeat statement.

Then print the collected personal information so that each entered object is printed in the following format: first and last names separated by a space (you don't print the identification number). An example of the working program is given below:

Sample output

```
First name: Jean
Last name: Bartik
Identification number: 271224
First name: Betty
Last name: Holberton
Identification number: 070317
First name:

Jean Bartik
Betty Holberton
```

Programming exercise:

Television programs

In the exercise template there is a ready-made class TelevisionProgram, representing a television program. The class has object variables name and duration, a constructor, and a few methods.

Implement a program that begins by reading television programs from the user. When the user inputs an empty string as the name of the program, the program stops reading programs.

After this the user is queried for a maximum duration. Once the maximum is given, the program proceeds to list all the programs whose duration is smaller or equal to the specified maximum duration.

Sample output

```
Name: Rick and Morty
Duration: 25
Name: Two and a Half Men
Duration: 30
Name: Love it or list it
Duration: 60
Name: House
Duration: 60

Program's maximum duration? 30
Rick and Morty, 25 minutes
Two and a Half Men, 30 minutes
```

Programming exercise:

Books

Write a program that first reads book information from the user. The details to be asked for each book include the title, the number of pages and the publication year. Entering an empty string as the name of the book ends the reading process.

After this the user is asked for what is to be printed. If the user inputs "everything", all the details are printed: the book titles, the numbers of pages and the publication years. However, if the user enters the string "name", only the book titles are printed.

It is probably worthwhile to implement a class called `Book` to represent a book. There are two points in total available for this exercise.

Sample output

```
Title: To Kill a Mockingbird
Pages: 281
Publication year: 1960
Title: A Brief History of Time
Pages: 256
Publication year: 1988
Title: Beautiful Code
Pages: 593
Publication year: 2007
Title: The Name of the Wind
Pages: 662
Publication year: 2007
Title:

What information will be printed? everything
To Kill a Mockingbird, 281 pages, 1960
A Brief History of Time, 256 pages, 1988
Beautiful Code, 593 pages, 2007
The Name of the Wind, 662 pages, 2007
```

Sample output

Title: To Kill a Mockingbird
Pages: 281
Publication year: 1960
Title: A Brief History of Time
Pages: 256
Publication year: 1988
Title: Beautiful Code
Pages: 593
Publication year: 2007
Title: The Name of the Wind
Pages: 662
Publication year: 2007
Title:

What information will be printed? name
To Kill a Mockingbird
A Brief History of Time
Beautiful Code
The Name of the Wind

Programming exercise:

Constructor Overload

The exercise template has a class `Product`, which represents a product in a shop. Every product has a name, location and weight.

Add the following three constructors to the `Product` class:

- `public Product(String name)` creates a product with the given name. Its location is set to "shelf" and its weight is set to 1.
- `public Product(String name, String location)` creates a product with the given name and the given location. Its weight is set to 1.
- `public Product(String name, int weight)` creates a product with the given name and the given weight. Its location is set to "shelf".

You can test your program with the following code:

```
Product tapeMeasure = new Product("Tape measure");
Product plaster = new Product("Plaster", "home improvement section");
Product tyre = new Product("Tyre", 5);

System.out.println(tapeMeasure);
System.out.println(plaster);
System.out.println(tyre);
```

Sample output

```
Tape measure (1 kg) can be found from the shelf
Plaster (1 kg) can be found from the home improvement section
Tyre (5 kg) can be found from the shelf
```

Programming exercise:

Overloaded Counter (2 parts)

Multiple constructors

Implement a class called `Counter`. The class contains a number whose value can be incremented and decremented. The class must have the following constructors:

- `public Counter(int startValue)` sets the start value of the counter to startValue.
- `public Counter()` sets the start value of the counter to 0.

And the following methods:

- `public int value()` returns the current value of the counter
- `public void increase()` increases the value by 1
- `public void decrease()` decreases the value by 1

Alternative methods

Implement versions which are given one parameter of the methods `increase` and `decrease`.

- `public void increase(int increaseBy)` increases the value of the counter by the value of increaseBy. If the value of increaseBy is negative, the value of the counter does not change.
- `public void decrease(int decreaseBy)` decreases the value of the counter by the value of decreaseBy. If the value of decreaseBy is negative, the value of the counter does not change.

Programming exercise:

Health station (3 parts)

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!

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

The output should be the following:

Sample output

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

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.weigh(ethan) + " kilos");
    System.out.println(peter.getName() + " weight: " +
 childrensHospital.weigh(peter) + " kilos");

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

    System.out.println("");

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

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

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 HealthInstitution();

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

    System.out.println("weighings performed: " + childrensHospital.weighings());

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

    System.out.println("weighings performed: " + childrensHospital.weighings());

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

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

The output is:

Sample output

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

Programming exercise:
Card payments (4 sections)

"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 if
        // 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
```

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 a meal and return the change
        // if the payment parameter is not large enough, no meal is sold...
    }

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

    public String toString() {
        return "money: " + money + ", number of sold afforable meals: " +
 affordableMeals + ", number of sold hearty meals: " + heartyMeals;
    }
}
```

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
```

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 card has enough money, the balance of the card is decreased...
        // otherwise false is returned
    }

    public boolean eatHeartily(PaymentCard card) {
        // a hearty meal costs 4.30 euros
        // if the card has enough money, the balance of the card is decreased...
        // 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
```

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.balance() + " euros");

        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.balance() + " euros");

        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
```

Programming exercise:

Biggest pet shop

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)
```

Programming exercise:

Comparing apartments (3 parts)

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.square = squares;
        this.pricePerSquare = pricePerSquare;
    }
}
```

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

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));
       // false
System.out.println(bangorThreeBedroomApt.largerThan(atlantaTwoBedroomApt));
  // true
```

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));
  //71600
System.out.println(bangorThreeBedroomApt.priceDifference(atlantaTwoBedroomApt));
   //35400
```

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(atlantaTwoBedroomApt));
  // false
System.out.println(bangorThreeBedroomApt.moreExpensiveThan(atlantaTwoBedroomApt));
   // true
```

Programming exercise:

Dating app (3 parts)

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.

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.

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.

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 " +date);

    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.

Programming exercise:

Money (3 parts)

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.

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 the corre

    // 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
//  the old 10 euros  disappears and the Java garbage collector takes care of it

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

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
```

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
//  the old 6.5 euros disappears and the Java garbage collector takes care of it


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