Programming exercise:

Room

Create a class named `Room`. Add the variables `private String code`and `private int seats` to the class. Then create a constructor `public Room(String classCode, int numberOfSeats)` through which values are assigned to the instance variables.

Programming exercise:

Whistle

Create a class named `Whistle`. Add the variable `private String sound` to the class. After that, create the constructor `public Whistle(String whistleSound)`, which is used to create a new whistle that's given a sound.

```
Whistle duckWhistle = new Whistle("Kvaak");
Whistle roosterWhistle = new Whistle("Peef");

duckWhistle.sound();
roosterWhistle.sound();
duckWhistle.sound();
```

Sample output

```
Kvaak
Peef
Kvaak
```

Programming exercise:
Door

Create a class named `Door`. The door does not have any variables. Create for it a constructor with no parameters (or use the default constructor). After that, create a `public void knock()` method for the door that prints the message "Who's there?" when called.

The door should work as follows.

```java
Door alexander = new Door();

alexander.knock();
alexander.knock();
```

Sample output

```
Who's there?
Who's there?
```

Programming exercise:

Product

Create a class `Product` that represents a store product. The product should have a price (double), a quantity (int) and a name (String).

The class should have:

- the constructor `public Product (String initialName, double initialPrice, int initialQuantity)`
- a method `public void printProduct()` that prints product information in the following format:

Sample output

```
Banana, price 1.1, 13 pcs
```

The output above is based on the product being assigned the name `banana`, with a price of `1.1`, and a quantity of `13` .

Programming exercise:

Decreasing counter (3 parts)

This exercise consists of multiple parts. Each part corresponds to one exercise point.

The exercise template comes with a partially executed class `decreasingCounter`:

```java
public class DecreasingCounter {
    private int value;   // a variable that remembers the value of the counter

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

    public void printValue {
        System.out.println("value: " + this.value);
    }

    public void decrement() {
        // write the method implementation here
        // the aim is to decrement the value of the counter by one
    }

    // and the other methods go here
}
```

The following is an example of how the main program uses the decreasing counter:

```java
public class MainPorgram {
    public static void main(String[] args) {
        DecreasingCounter counter = new DecreasingCounter(10);

        counter.printValue();

        counter.decrement();
        counter.printValue();

        counter.decrement();
        counter.printValue();
    }
}
```

The program above should have the following print output.

Sample output

```
value: 10
value: 9
value: 8
```

Implementation of the decrement() method

Implement the `decrement()` method in the class body in such a way that it decrements the `value` variable of the object it's being called on by one. Once you're done with the `decrement()` method, the main program of the previous example should work to produce the example output.

The counter's value cannot be negative

Implement the `decrement()` in such a way that the counter's value never becomes negative. This means that if the value of the counter is 0, it cannot be decremented. A conditional statement is useful here.

```java
public class MainProgram {
    public static void main(String[] args) {
        DecreasingCounter counter = new DecreasingCounter(2);

        counter.printValue();

        counter.decrement();
        counter.printValue();

        counter.decrement();
        counter.printValue();

        counter.decrement();
        counter.printValue();
    }
}
```

Prints:

Sample output

```
value: 2
value: 1
```

```
value: 0
value: 0
```

Resetting the counter value

Create the method `public void reset()` for the counter that resets the value of the counter to 0. For example:

```java
public class MainProgram {
    public static void main(String[] args) {
        DecreasingCounter counter = new DecreasingCounter(100);

        counter.printValue();

        counter.reset();
        counter.printValue();

        counter.decrement();
        counter.printValue();
    }
}
```

Prints:

Sample output

```
value: 100
value: 0
value: 0
```

Programming exercise:

Debt

Create the class `Debt` that has double-typed instance variables of `balance` and `interestRate`. The balance and the interest rate are passed to the constructor as parameters `public Debt(double initialBalance, double initialInterestRate)`.

In addition, create the methods `public void printBalance()` and `public void waitOneYear()` for the class. The method printBalance prints the current balance, and the waitOneYear method grows the debt amount.

The debt is increased by multiplying the balance by the interest rate.

The class should do the following:

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

        Debt mortgage = new Debt(120000.0, 1.01);
        mortgage.printBalance();

        mortgage.waitOneYear();
        mortgage.printBalance();

        int years = 0;

        while (years < 20) {
            mortgage.waitOneYear();
            years = years + 1;
        }

        mortgage.printBalance();
    }
}
```

The example above illustrates the development of a mortgage with an interest rate of one percent.

Prints:

Sample output

9 puslapis iš 24

```
120000.0
121200.0
147887.0328416936
```

When you've managed to get the program to work, check the previous example with the early 1900s recession interest rates as well.

Once you get the program to work, try out the previous example with the interest rates of the early 1990s recession when the interest rates were as high as 15-20% - try swapping the interest rate in the example above with `1.20` and see what happens.

Programming exercise:

Song

Create a class called `Song`. The song has the instance variables `name`(string) and `length` in seconds (integer). Both are set in the `public Song(String name, int length)` constructor. Also, add to the object the methods `public String name()`, which returns the name of the song, and `public int length()`, which returns the length of the song.

The class should work as follows.

```
Song garden = new Song("In The Garden", 10910);
System.out.println("The song " + garden.name() + " has a length of " +

 garden.length() +
" seconds.");
```

Sample output

```
The song In The Garden has a length of 10910 seconds.
```

Programming exercise:

Gauge

Create the class `Gauge`. The gauge has the instance variable `private int value`, a constructor without parameters (sets the initial value of the meter variable to 0), and also the following four methods:

- Method `public void increase()` grows the `value` instance variable's value by one. It does not grow the value beyond five.
- Method `public void decrease()` decreases the `value` instance variable's value by one. It does not decrease the value to negative numbers.
- Method `public int value()` returns the `value` variable's value.
- Method `public boolean full()` returns `true` if the instance variable `value` has the value five. Otherwise, it returns false.

An example of the class in use.

```java
Gauge g = new Gauge();

while(!g.full()) {
    System.out.println("Not full! Value: " + g.value());
    g.increase();
}

System.out.println("Full! Value: " + g.value());
g.decrease();
System.out.println("Not full! Value: " + g.value());
```

Sample output

```
Not full! Value: 0
Not full! Value: 1
Not full! Value: 2
Not full! Value: 3
Not full! Value: 4
Full! Value: 5
Not full! Value: 4
```

Programming exercise:

Agent

The exercise template defines an Agent class, having a first name and last name. A `print` method is defined for the class that creates the following string representation.

```
Agent bond = new Agent("James", "Bond");
bond.print();
```

Sample output

```
My name is Bond, James Bond
```

Remove the class' `print` method, and create a `public String toString()` method for it, which returns the string representation shown above.

The class should function as follows.

```
Agent bond = new Agent("James", "Bond");

bond.toString(); // prints nothing
System.out.println(bond);

Agent ionic = new Agent("Ionic", "Bond");
System.out.println(ionic);
```

Sample output

```
My name is Bond, James Bond
My name is Bond, Ionic Bond
```

Programming exercise:

Multiplier

Create a class Multiplier that has a:

- Constructor `public Multiplier(int number)`.
- Method `public int multiply(int number)` which returns the value `number` passed to it multiplied by the `number` provided to the constructor.

You also need to create an instance variable in this exercise.

An example of the class in use:

```java
Multiplier multiplyByThree = new Multiplier(3);

System.out.println("multiplyByThree.multiply(2): " + multiplyByThree.multiply(2));

Multiplier multiplyByFour = new Multiplier(4);

System.out.println("multiplyByFour.multiply(2): " + multiplyByFour.multiply(2));
System.out.println("multiplyByThree.multiply(1): " + multiplyByThree.multiply(1));
System.out.println("multiplyByFour.multiply(1): " + multiplyByFour.multiply(1));
```

Output

Sample output

```
multiplyByThree.multiply(2): 6
multiplyByFour.multiply(2): 8
multiplyByThree.multiply(1): 3
multiplyByFour.multiply(1): 4
```

Programming exercise:

Statistics (4 parts)

Count

Create a class `Statistics` that has the following functionality (the file for the class is provided in the exercise template):

- a method `addNumber` adds a new number to the statistics
- a method `getCount` tells the number of added numbers

The class does not need to store the added numbers anywhere, it is enough for it to remember their count. At this stage, the `addNumber`method can even neglect the numbers being added to the statistics, since the only thing being stored is the count of numbers added.

The method's body is the following:

```java
public class Statistics {
    private int count;

    public Statistics() {
        // initialize the variable numberCount here
    }

    public void addNumber(int number) {
        // write code here
    }

    public int getCount() {
        // write code here
    }
}
```

The following program introduces the class' use:

```java
public class MainProgram {
    public static void main(String[] args) {
        Statistics statistics = new Statistics();
        statistics.addNumber(3);
        statistics.addNumber(5);
        statistics.addNumber(1);
        statistics.addNumber(2);
        System.out.println("Count: " + statistics.getCount());
    }
}
```

The program prints the following:

Sample output

```
Count: 4
```

Sum and average

Expand the class with the following functionality:

- the `sum` method tells the sum of the numbers added (the sum of an empty number statistics object is 0)
- the `average` method tells the average of the numbers added (the average of an empty number statistics object is 0

The class' template is the following:

```
public class Statistics {
    private int count;
    private int sum;

    public Statistics() {
        // initialize the variables count and sum here
    }

    public void addNumber(int number) {
        // write code here
    }

    public int getCount() {
        // write code here
    }

    public int sum() {
        // write code here
    }

    public double average() {
        // write code here
    }
}
```

The following program demonstrates the class' use:

```java
public class Main {
    public static void main(String[] args) {
        Statistics statistics = new Statistics();
        statistics.addNumber(3);
        statistics.addNumber(5);
        statistics.addNumber(1);
        statistics.addNumber(2);
        System.out.println("Count: " + statistics.getCount());
        System.out.println("Sum: " + statistics.sum());
        System.out.println("Average: " + statistics.average());
    }
}
```

The program prints the following:

Sample output

```
Count: 4
Sum: 11
Average: 2.75
```

Sum of user input

Write a program that asks the user for numbers until the user enters -1. The program will then provide the sum of the numbers.

The program should use a `Statistics` object to calculate the sum.

NOTE: Do not modify the Statistics class in this part. Instead, implement the program for calculating the sum by making use of it.

Sample output

```
Enter numbers:
4
2
5
4
-1
Sum: 15
```

Multiple sums

Change the previous program so that it also calculates the sum of even and odd numbers.

NOTE: Define *three* Statistics objects in the program. Use the first to calculate the sum of all numbers, the second to calculate the sum of even numbers, and the third to calculate the sum of odd numbers.

For the test to work, the objects must be created in the main program in the order they were mentioned above (i.e., first the object that sums all the numbers, then the one that sums the even ones, and then finally the one that sums the odd numbers)!

NOTE: Do not change the Statistics class in any way!

The program should work as follows:

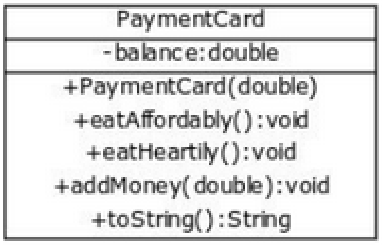
Sample output

```
Enter numbers:
4
2
5
2
-1
Sum: 13
Sum of even numbers: 8
Sum of odd numbers: 5
```

Programming exercise:

Payment Card (6 parts)

At the student canteen students pay for their lunch using a payment card. The final PaymentCard will look like the following as a class diagram:



In this exercise series, a class called `PaymentCard` is created which aims to mimic Unicafe's payment process

The class template

The project will include two code files:

The exercise template comes with a code file called `Main`, which contains the `main` method.

Add a new class to the project called `PaymentCard`. Here's how to add a new class: On the left side of the screen is the list of projects. Right-click on the project name. Select *New* and *Java Class* from the drop-down menu. Name the class as "PaymentCard".

First, create the `PaymentCard` object's constructor, which is passed the opening balance of the card, and which then stores that balance in the object's internal variable. Then, write the `toString` method, which will return the card's balance in the form "The card has a balance of X euros".

The following is the template of the `PaymentCard` class:

```java
public class PaymentCard {
    private double balance;

    public PaymentCard(double openingBalance) {
        // write code here
    }
    public String toString() {
        // write code here
    }
}
```

The following main program tests the class:

```
public class MainProgram {
    public static void main(String[] args) {
        PaymentCard card = new PaymentCard(50);
        System.out.println(card);
    }
}
```

The program should print the following:

Sample output

```
The card has a balance of 50.0 euros
```

Making transactions

Complement the `PaymentCard` class with the following methods:

```
public void eatAffordably() {
    // write code here
}

public void eatHeartily() {
    // write code here
}
```

The method `eatAffordably` should reduce the card's balance by € 2.60, and the method `eatHeartily` should reduce the card's balance by € 4.60.

The following main program tests the class:

```
public class MainProgram {
    public static void main(String[] args) {
        PaymentCard card = new PaymentCard(50);
        System.out.println(card);
        card.eatAffordably();
        System.out.println(card);

        card.eatHeartily();
        card.eatAffordably();
        System.out.println(card);
    }
}
```

The program should print approximately the following:

Sample output

```
The card has a balance of 50.0 euros
The card has a balance of 47.4 euros
The card has a balance of 40.199999999999996 euros
```

Non-negative balance

What happens if the card runs out of money? It doesn't make sense in this case for the balance to turn negative. Change the methods `eatAffordably` and `eatHeartily` so that they don't reduce the balance should it turn negative.

The following main program tests the class:

```
public class MainProgram {
    public static void main(String[] args) {
        PaymentCard card = new PaymentCard(5);
        System.out.println(card);

        card.eatHeartily();
        System.out.println(card);

        card.eatHeartily();
        System.out.println(card);
    }
}
```

The program should print the following:

Sample output

```
The card has a balance 5.0 euros
The card has a balance 0.40000000000000036 euros
The card has a balance 0.40000000000000036 euros
```

The second call to the method `eatHeartily` above did not affect the balance, since the balance would have otherwise become negative.

Topping up the card

Add the following method to the `PaymentCard` class:

```java
public void addMoney(double amount) {
    // write code here
}
```

The purpose of the method is to increase the card's balance by the amount of money given as a parameter. However, the card's balance may not exceed 150 euros. As such, if the amount to be topped up exceeds this limit, the balance should, in any case, become exactly 150 euros.

The following main program tests the class:

```java
public class MainProgram {
    public static void main(String[] args) {
        PaymentCard card = new PaymentCard(10);
        System.out.println(card);

        card.addMoney(15);
        System.out.println(card);

        card.addMoney(10);
        System.out.println(card);

        card.addMoney(200);
        System.out.println(card);
    }
}
```

The program should print the following:

Sample output

```
The card has a balance of 10.0 euros
The card has a balance of 25.0 euros
The card has a balance of 35.0 euros
The card has a balance of 150.0 euros
```

Topping up the card with a negative value

Change the `addMoney` method further, so that if there is an attempt to top it up with a negative amount, the value on the card will not change.

The following main program tests the class:

```java
public class MainProgram {
    public static void main(String[] args) {
        PaymentCard card = new PaymentCard(10);
        System.out.println("Paul: " + card);
        card.addMoney(-15);
        System.out.println("Paul: " + card);
    }
}
```

The program should print the following:

Sample output

```
Paul: The card has a balance of 10.0 euros
Paul: The card has a balance of 10.0 euros
```

Multiple cards

Write code in the `main` method of the `MainProgram` class that contains the following sequence of events:

- Create Paul's card. The opening balance of the card is 20 euros
- Create Matt's card. The opening balance of the card is 30 euros
- Paul eats heartily
- Matt eats affordably
- The cards' values are printed (each on its own line, with the cardholder name at the beginning of it)
- Paul tops up 20 euros
- Matt eats heartily
- The cards' values are printed (each on its own line, with the cardholder name at the beginning of it)
- Paul eats affordably
- Paul eats affordably
- Matt tops up 50 euros

- The cards' values are printed (each on its own line, with the cardholder name at the beginning of it)

The main program's template is as follows:

```java
public class Main {
    public static void main(String[] args) {
        PaymentCard paulsCard = new PaymentCard(20);
        PaymentCard mattsCard = new PaymentCard(30);

        // write code here
    }
}
```

The program should produce the following print output:

Sample output

```
Paul: The card has a balance of 15.4 euros
Matt: The card has a balance of 27.4 euros
Paul: The card has a balance of 35.4 euros
Matt: The card has a balance of 22.799999999999997 euros
Paul: The card has a balance of 30.199999999999996 euros
Matt: The card has a balance of 72.8 euros
```