Part 4

Introduction to object-oriented programming

Learning Objectives

- You're familiar with the concepts of class, object, constructor, object methods, and object variables.
- You understand that a class defines an object's methods and that the values of instance (object) variables are object-specific.
- You know how to create classes and objects, and know how to use objects in your programs.

We'll now begin our journey into the world of object-oriented programming. We'll start with focusing on describing concepts and data using objects. From there on, we'll learn how to add functionality, i.e., methods to our program.

Object-oriented programming is concerned with isolating concepts of a problem domain into separate entities and then using those entities to solve problems. Concepts related to a problem can only be considered once they've been identified. In other words, we can form abstractions from problems that make those problems easier to approach.

Once concepts related to a given problem have been identified, we can also begin to build constructs that represent them into programs. These constructs, and the individual instances that are formed from them, i.e., objects, are used in solving the problem. The statement "programs are built from small, clear, and cooperative objects" may not make much

sense yet. However, it will appear more sensible as we progress through the course, perhaps even self-evident.

Classes and Objects

We've already used some of the classes and objects provided by Java. A **class** defines the attributes of objects, i.e., the information related to them (instance variables), and their commands, i.e., their methods. The values of instance (i.e., object) variables define the internal state of an individual object, whereas methods define the functionality it offers.

A **Method** is a piece of source code written inside a class that's been named and has the ability to be called. A method is always part of some class and is often used to modify the internal state of an object instantiated from a class.

As an example, `ArrayList` is a class offered by Java, and we've made use of objects instantiated from it in our programs. Below, an `ArrayList` object named `integers` is created and some integers are added to it.

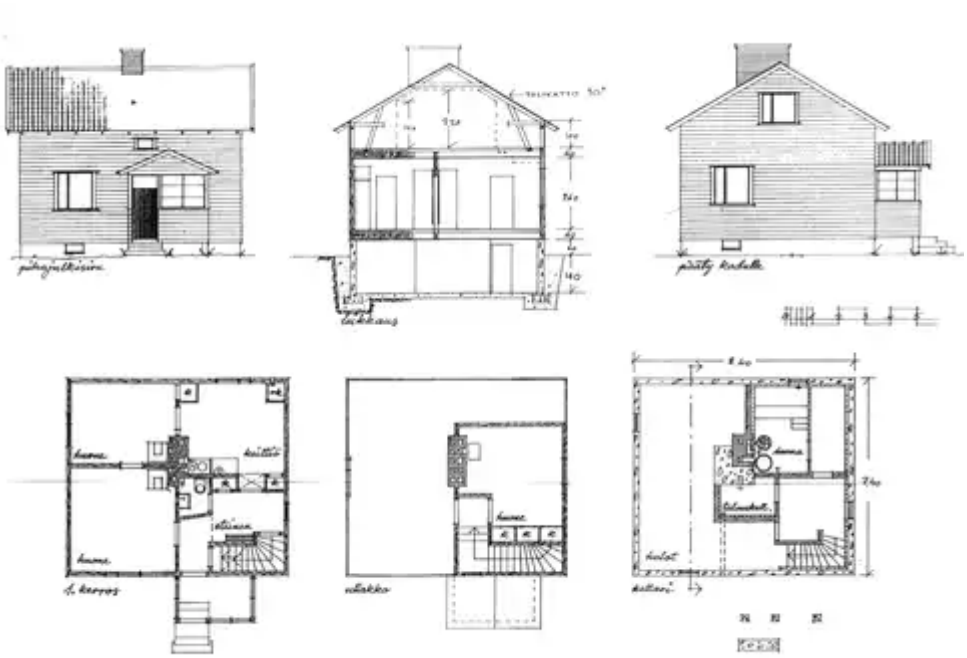
```
// we create an object from the ArrayList class named integers
ArrayList<Integer> integers = new ArrayList<>();

// let's add the values 15, 34, 65, 111 to the integers object
integers.add(15);
integers.add(34);
integers.add(65);
integers.add(111);

// we print the size of the integers object
System.out.println(integers.size());
```

An object is always instantiated by calling a method that created an object, i.e., a **constructor** by using the `new` keyword.

The Relationship Between a Class and an Object

A class lays out a blueprint for any objects that are instantiated from it. Let's draw from an analogy from outside the world of computers. Detached houses are most likely familiar to most, and we can safely assume the existence of drawings somewhere that determine what

exactly a detached house is to be like. A class is a blueprint. In other words, it specifies what kinds of objects can be instantiated from it:



Individual objects, detached houses in this case, are all created based on the same blueprints - they're instances of the same class. The states of individual objects, i.e., their attributes (such as the wall color, the building material of the roof, the color of its foundations, the doors' materials and color, ...) may all vary, however. The following is an "object of a detached-house class":

Programming exercise:

Your first account

Points
1/1

The exercise template comes with a ready-made class named `Account`. The `Account` object represents a bank account that has balance (i.e. one that has some amount of money in it). The accounts are used as follows:

```
Account artosAccount = new Account("Arto's account", 100.00);
Account artosSwissAccount = new Account("Arto's account in Switzerland", 10

System.out.println("Initial state");
System.out.println(artosAccount);
System.out.println(artosSwissAccount);

artosAccount.withdraw(20);
System.out.println("The balance of Arto's account is now: " + artosAccount.
artosSwissAccount.deposit(200);
System.out.println("The balance of Arto's other account is now: " + artosSw
```

```
System.out.println("End state");
System.out.println(artosAccount);
System.out.println(artosSwissAccount);
```

Write a program that creates an account with a balance of 100.0, deposits 20.0 in it, and finally prints the balance. **NB!** Perform all the operations in this exact order.

Exercise submission instructions

How to see the solution

Programming exercise:

Your first bank transfer

Points 1/1

The `Account` from the previous exercise class is also available in this exercise.

Write a program that:

1. Creates an account named `"Matthews account"` with the balance 1000
2. Creates an account named `"My account"` with the balance 0
3. Withdraws 100.0 from Matthew's account
4. Deposits 100.0 to "my account"
5. Prints both the accounts

Exercise submission instructions

How to see the solution

Quiz:
Calling a method

Points:
1/1

Examine the program below.

```
public class Student {

    private int credits;

    public Student() {
        this.credits = 0;
    }

    public void play() {
        this.credits = this.credits - 8;
    }
}

-------------------------------

public static void main(String[] args) {
    Student matt = new Student();
    // Code is placed here
}
```

Which of the following alternatives does not cause an error when the "//Code is placed here" comment is replaced with it?

Select the correct answer

- Student.play();
- ✓ matt.play();
- matt.play;
- Student.play;

```
play();
```

The answer is correct

Answered The number of tries is not limited

Creating Classes

A class specifies what the objects instantiated from it are like.

- The **object's variables (instance variables)** specify the internal state of the object
- The **object's methods** specify what the object does

We'll now familiarize ourselves with creating our own classes and defining the variable that belong to them.

A class is defined to represent some meaningful entity, where a "meaningful entity" often refers to a real-world object or concept. If a computer program had to process personal information, it would perhaps be meaningful to define a seperate class `Person` consisting of methods and attributes related to an individual.

Let's begin. We'll assume that we have a project template that has an empty main program:

```
public class Main {

    public static void main(String[] args) {

    }
}
```

Creating a New Class

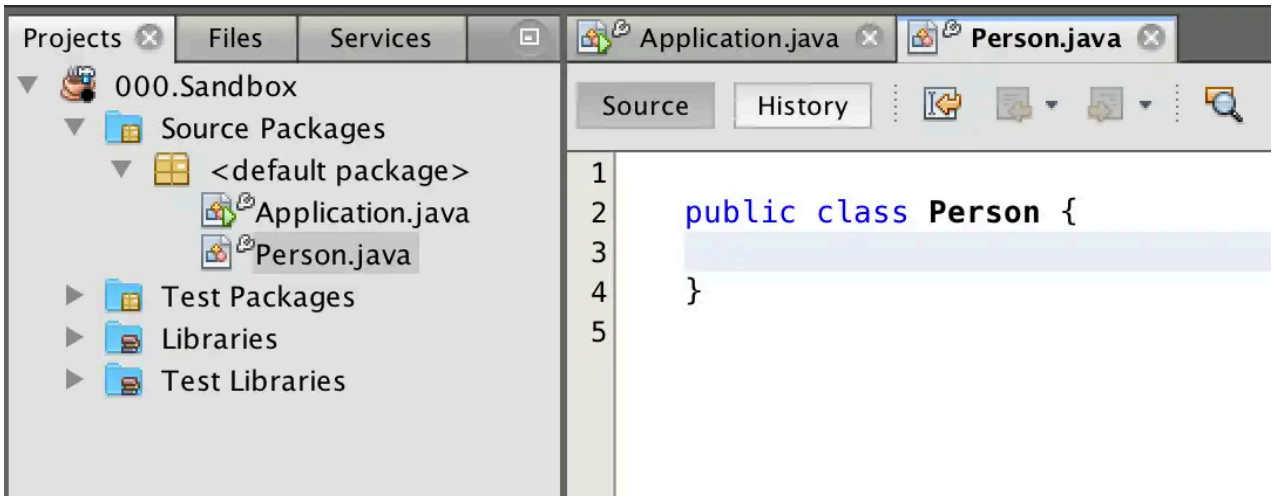
In NetBeans, a new class can be created by going to the *projects* section located on the left, right-clicking *new*, and then *java class*. The class is provided a name in the dialog that opens.

> As with variables and methods, the name of a class should be as descriptive as possible. It's usual for a class to live on and take on a different form as a program develops. As such, the class may have to be renamed at some later point.

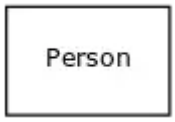
Let's create a class named `Person`. For this class, we create a separate file named `Person.java`. Our program now consists of two separate files since the main program is also in its own file. The `Person.java` file initially contains the class definition **public class Person** and the curly brackets that confine the contents of the class.

```
public class Person {

}
```

After creating a new file in NetBeans, the current state is as follows. In the image below, the class `Person` has been added to the SandboxExercise.



You can also draw a class diagram to depict a class. We'll become familiar with its notations as we go along. An empty person-named class looks like this:



A class defines the attributes and behaviors of objects that are created from it. Let's decide that each person object has a name and an age. It's

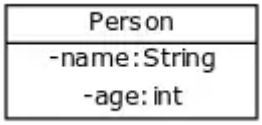
natural to represent the name as a string, and the age as an integer. We'll go ahead and add these to our blueprint:

```
public class Person {
    private String name;
    private int age;
}
```

We specify above that each object created from the `Person` class has a `name` and an `age`. Variables defined inside a class are called **instance variables**, or object fields or object attributes. Other names also seem to exist.

Instance variables are written on the lines following the class definition `public class Person {`. Each variable is preceded by the keyword private. The keyword **private** means that the variables are "hidden" inside the object. This is known as **encapsulation**.

In the class diagram, the variables associated with the class are defined as "variableName: variableType". The minus sign before the variable name indicates that the variable is encapsulated (it has the keyword private).



We have now defined a blueprint — a class — for the person object. Each new person object has the variables `name` and `age`, which are able to hold object-specific values. The "state" of a person consists of the values assigned to their name and age.

Programming exercise:

Dog attributes

Points 1/1

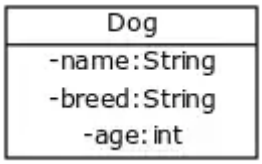
In this exercise, you'll practice creating a class.

A new class can be added in NetBeans the following way: On the left side of the screen is the Projects list. Right-click on the project's name,

which for this exercise is `Part04_03.DogAttributes`. From the drop-down menu, select *New* and *Java Class*. NetBeans will then ask for the class name.

Name the class `Dog` in this exercise, and press the finish button.

You have now created a class called `Dog`. Add the variables `private String name`,`private String breed` and `private int age` to the class. As a class diagram, the class looks like this:



The class doesn't do much yet. However, practicing this step is valuable for what is to come.

Exercise submission instructions

How to see the solution

Defining a Constructor

We want to set an initial state for an object that's created. Custom objects are created the same way as objects from pre-made Java classes, such as `ArrayList`, using the `new` keyword. It'd be convenient to pass values to the variables of that object as it's being created. For example, when creating a new person object, it's useful to be able to provide it with a name:

```
public static void main(String[] args) {
    Person ada = new Person("Ada");
    // ...
}
```

This is achieved by defining the method that creates the object, i.e., its constructor. The constructor is defined after the instance variables. In the

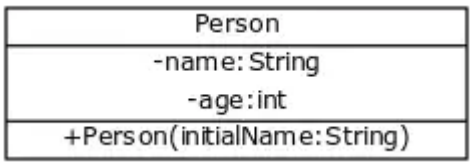
following example, a constructor is defined for the Person class, which can be used to create a new Person object. The constructor sets the age of the object being created to 0, and the string passed to the constructor as a parameter as its name:

```
public class Person {
    private String name;
    private int age;

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

The constructor's name is always the same as the class name. The class in the example above is named Person, so the constructor will also have to be named Person. The constructor is also provided, as a parameter, the name of the person object to be created. The parameter is enclosed in parentheses and follows the constructor's name. The parentheses that contain optional parameters are followed by curly brackets. In between these brackets is the source code that the program executes when the constructor is called (e.g., `new Person ("Ada")`).

Objects are always created using a constructor.

A few things to note: the constructor contains the expression `this.age = 0`. This expression sets the instance variable `age` of the newly created object (i.e., "this" object's age) to 0. The second expression `this.name = initialName` likewise assigns the string passed as a parameter to the instance variable `name` of the object created.

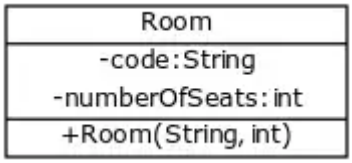


Programming exercise:

Room

Points 1/1

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.



This class doesn't do much either. However, in the following exercise the object instantiated from our class is already capable of printing text.

Exercise submission instructions

How to see the solution

Default Constructor

If the programmer does not define a constructor for a class, Java automatically creates a default one for it. A default constructor is a constructor that doesn't do anything apart from creating the object. The object's variables remain uninitialized (generally, the value of any object references will be `null`, meaning that they do not point to anything, and the values of primitives will be `0`)

For example, an object can be created from the class below by making the call `new Person()`

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

```
    private int age;
}
```

If a constructor has been defined for a class, no default constructor exists. For the class below, calling `new Person` would cause an error, as Java cannot find a constructor in the class that has no parameters.

```
public class Person {
    private String name;
    private int age;

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

Defining Methods For an Object

We know how to create an object and initialize its variables. However, an object also needs methods to be able to do anything. As we've learned, a **method** is a named section of source code inside a class which can be invoked.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }

    public void printPerson() {
        System.out.println(this.name + ", age " + this.age + " years");
    }
}
```

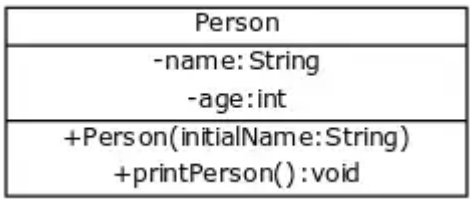
A method is written inside of the class beneath the constructor. The method name is preceded by `public void`, since the method is intended to be visible to the outside world (`public`), and it does not return a value (`void`).

Objects and the Static Modifier

We've used the modifier `static` in some of the methods that we've written. The `static` modifier indicates that the method in question does not belong to an object and thus cannot be used to access any variables that belong to objects.

Going forward, our methods will not include the `static` keyword if they're used to process information about objects created from a given class. If a method receives as parameters all the variables whose values it uses, it can have a `static` modifier.

In addition to the class name, instance variables and constructor, the class diagram now also includes the method `printPerson`. Since the method comes with the `public` modifier, the method name is prefixed with a plus sign. No parameters are defined for the method, so nothing is put inside the method's parentheses. The method is also marked with information indicating that it does not return a value, here `void`.



The method `printPerson` contains one line of code that makes use of the instance variables `name` and `age` — the class diagram says nothing about its internal implementations. Instance variables are referred to with the prefix `this`. All of the object's variables are visible and available from within the method.

Let's create three persons in the main program and request them to print themselves:

```
public class Main {

    public static void main(String[] args) {
        Person ada = new Person("Ada");
        Person antti = new Person("Antti");
        Person martin = new Person("Martin");

        ada.printPerson();
        antti.printPerson();
        martin.printPerson();
    }
}
```

Prints:

Sample output

```
Ada, age 0 years
Antti, age 0 years
Martin, age 0 years
```

This as a screencast:

Programming exercise:

Whistle

Points
1/1

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.

Then create the method `public void sound()` that prints the whistle's sound.

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

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

Sample output

```
Kvaak
Peef
Kvaak
```

Exercise submission instructions

How to see the solution

Programming exercise:

Door

Points
1/1

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.

```
Door alexander = new Door();

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

Sample output

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

Exercise submission instructions

How to see the solution

Programming exercise:

Product

Points

1/1

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` .

Exercise submission instructions

How to see the solution

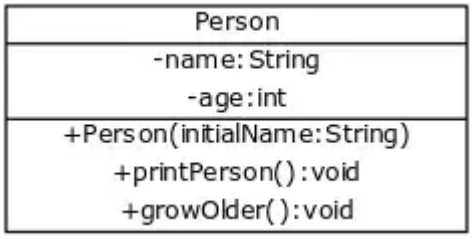
Changing an Instance Variable's Value in a Method

Let's add a method to the previously created person class that increments the age of the person by a year.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }

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

    // growOlder() method has been added
    public void growOlder() {
        this.age = this.age + 1;
    }
}
```

The method is written inside the `Person` class just as the `printPerson` method was. The method increments the value of the instance variable `age` by one.

The class diagram also gets an update.



Let's call the method and see what happens:

```
public class Main {

    public static void main(String[] args) {
        Person ada = new Person("Ada");
        Person antti = new Person("Antti");

        ada.printPerson();
        antti.printPerson();
        System.out.println("");

        ada.growOlder();
        ada.growOlder();

        ada.printPerson();
        antti.printPerson();
    }
}
```

The program's print output is as follows:

Sample output

```
Ada, age 0 years
Antti, age 0 years

Ada, age 2 years
Antti, age 0 years
```

That is to say that when the two objects are "born" they're both zero-years old (`this.age = 0;` is executed in the constructor). The `ada` object's `growOlder` method is called twice. As the print output demonstrates, the age of Ada is 2 years after growing older. Calling the method on an object corresponding to Ada has no impact on the age of the other person object since each object instantiated from a class has its own instance variables.

The method can also contain conditional statements and loops. The growOlder method below limits aging to 30 years.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }

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

    // no one exceeds the age of 30
    public void growOlder() {
        if (this.age < 30) {
            this.age = this.age + 1;
        }
    }
}
```

Programming exercise:

Decreasing counter (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.

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

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

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

```
    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:

```
public class MainProgram {
    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
```

Part 1: 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.

Part 2: 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.

```
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
```

Part 3: Resetting the counter value

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

```
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
```

Exercise submission instructions

How to see the solution

Programming exercise:

Debt

Points 1/1

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:

```
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

```
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.

Exercise submission instructions

How to see the solution

Returning a Value From a Method

A method can return a value. The methods we've created in our objects haven't so far returned anything. This has been marked by typing the keyword *void* in the method definition.

```
public class Door {
    public void knock() {
        // ...
    }
}
```

The keyword **void** means that the method does not return a value.

If we want the method to return a value, we need to replace the `void` keyword with the type of the variable to be returned. In the following example, the Teacher class has a method `grade` that always returns an integer-type (`int`) variable (in this case, the value 10). The value is always returned with the **return** command:

```
public class Teacher {
    public int grade() {
        return 10;
    }
}
```

The method above returns an `int` type variable of value 10 when called. For the return value to be used, it needs to be assigned to a variable. This happens the same way as regular value assignment, i.e., by using the equals sign:

```
public static void main(String[] args) {
    Teacher teacher = new Teacher();

    int grading = teacher.grade();

    System.out.println("The grade received is " + grading);
}
```

Sample output

```
The grade received is 10
```

The method's return value is assigned to a variable of type `int` value just as any other int value would be. The return value could also be used to form part of an expression.

```
public static void main(String[] args) {
    Teacher first = new Teacher();
    Teacher second = new Teacher();
    Teacher third = new Teacher();

    double average = (first.grade() + second.grade() + third.grade()) / 3.0;

    System.out.println("Grading average " + average);
}
```

Sample output

```
Grading average 10.0
```

All the variables we've encountered so far can also be returned by a method. To sum:

- A method that returns nothing has the `void` modifier as the type of variable to be returned.

```
public void methodThatReturnsNothing() {
    // the method body
}
```

- A method that returns an integer variable has the `int` modifier as the type of variable to be returned.

```
public int methodThatReturnsAnInteger() {
    // the method body, requires a return statement
}
```

- A method that returns a string has the `String` modifier as the type of the variable to be returned

```
public String methodThatReturnsAString() {
    // the method body, requires a return statement
}
```

- A method that returns a double-precision number has the `double` modifier as the type of the variable to be returned.

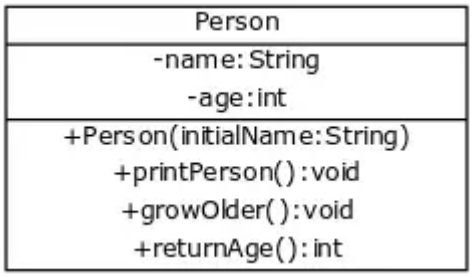
```
public double methodThatReturnsADouble() {
    // the method body, requires a return statement
}
```

Let's continue with the Person class and add a `returnAge` method that returns the person's age.

```
public class Person {
    private String name;
    private int age;

    public Person(String initialName) {
        this.age = 0;
        this.name = initialName;
    }

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

    public void growOlder() {
```

```
        if (this.age < 30) {
            this.age = this.age + 1;
        }
    }
    // the added method
    public int returnAge() {
        return this.age;
    }
```

The class in its entirety:



Let's illustrate how the method works:

```
public class Main {

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

        pekka.growOlder();
        pekka.growOlder();

        antti.growOlder();

        System.out.println("Pekka's age: " + pekka.returnAge());
        System.out.println("Antti's age: " + antti.returnAge());
        int combined = pekka.returnAge() + antti.returnAge();

        System.out.println("Pekka's and Antti's combined age " + combined + " y
    }
}
```

Sample output

Pekka's age 2
Antti's age 1

Pekka's and Antti's combined age 3 years

Quiz:

Which number will be printed?

Points: 1/1

Examine the following program.

```
public class Counter{

    private int value;

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

    public void changeValue(int modifier) {
        this.value = this.value - modifier;
    }

    public int getValue() {
        return value;
    }

}

--------------------------------

public static void main(String[] args) {
    Counter counter = new Counter(7);
    counter.changeValue(5);
    System.out.println(counter.getValue());
}
```

Which number will be printed?

Select the correct answer

✓ 2

7

5

-2

12

The answer is correct

Answered The number of tries is not limited

Programming exercise:

Song

Points 1/1

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 " + gard
```

Sample output

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

Exercise submission instructions

How to see the solution

Programming exercise:

Film

Points

1/1

Create a film class with the instance variables `name` (String) and `ageRating` (int). Write the constructor `public Film(String filmName, int filmAgeRating)` for the class, and also the methods `public String name()` and `public int ageRating()`. The first of these returns the film title and the second returns its rating.

Below is an example use case of the class.

```
Film chipmunks = new Film("Alvin and the Chipmunks: The Squeakquel", 0);

Scanner reader = new Scanner(System.in);

System.out.println("How old are you?");
int age = Integer.valueOf(reader.nextLine());

System.out.println();
if (age >= chipmunks.ageRating()) {
    System.out.println("You may watch the film " + chipmunks.name());
} else {
    System.out.println("You may not watch the film " + chipmunks.name());
}
```

Sample output

```
How old are you?
7

You may watch the film Alvin and the Chipmunks: The Squeakquel
```

Exercise submission instructions

How to see the solution

As we came to notice, methods can contain source code in the same way as other parts of our program. Methods can have conditionals or loops, and other methods can also be called from them.

Let's now write a method for the person that determines if the person is of legal age. The method returns a boolean - either `true` or `false`:

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

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

        return true;
    }

    /*
     The method could have been written more succinctly in the following way:

    public boolean isOfLegalAge() {
        return this.age >= 18;
    }
    */
}
```

And let's test it out:

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

    int i = 0;
    while (i < 30) {
        pekka.growOlder();
        i = i + 1;
```

```
    }

    antti.growOlder();

    System.out.println("");

    if (antti.isOfLegalAge()) {
        System.out.print("of legal age: ");
        antti.printPerson();
    } else {
        System.out.print("underage: ");
        antti.printPerson();
    }

    if (pekka.isOfLegalAge()) {
        System.out.print("of legal age: ");
        pekka.printPerson();
    } else {
        System.out.print("underage: ");
        pekka.printPerson();
    }
}
```

Sample output

```
underage: Antti, age 1 years
of legal age: Pekka, age 30 years
```

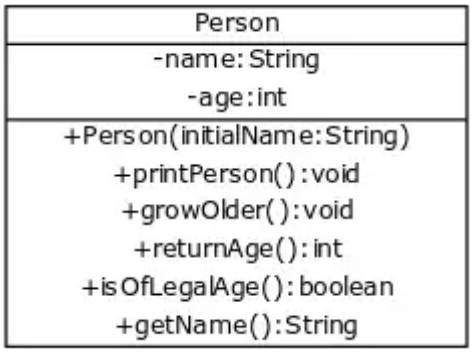
Let's fine-tune the solution a bit more. In its current form, a person can only be "printed" in a way that includes both the name and the age. Situations exist, however, where we may only want to know the name of an object. Let's write a separate method for this use case:

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

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

The `getName` method returns the instance variable `name` to the caller. The name of this method is somewhat strange. It is the convention in Java to

name a method that returns an instance variable exactly this way, i.e., `getVariableName`. Such methods are often referred to as "getters".

The class as a whole:



Let's mould the main program to use the new "getter" method:

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

    int i = 0;
    while (i < 30) {
        pekka.growOlder();
        i = i + 1;
    }

    antti.growOlder();

    System.out.println("");

    if (antti.isOfLegalAge()) {
        System.out.println(antti.getName() + " is of legal age");
    } else {
        System.out.println(antti.getName() + " is underage");
    }

    if (pekka.isOfLegalAge()) {
        System.out.println(pekka.getName() + " is of legal age");
    } else {
        System.out.println(pekka.getName() + " is underage ");
    }
}
```

The print output is starting to turn out quit neat:

Sample output

Antti is underage
Pekka is of legal age

Programming exercise:

Gauge

Points 1/1

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.

```
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
```

Exercise submission instructions

How to see the solution

A string representation of an object and the toString-method

We are guilty of programming in a somewhat poor style by creating a method for printing the object, i.e., the `printPerson` method. A preferred way is to define a method for the object that returns a "string representation" of the object. The method returning the string representation is always `toString` in Java. Let's define this method for the person in the following example:

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

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

The `toString` functions as `printPerson` does. However, it doesn't itself print anything but instead returns a string representation, which the calling method can execute for printing as needed.

The method is used in a somewhat surprising way:

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

    int i = 0;
```

```
    while (i < 30) {
        pekka.growOlder();
        i = i + 1;
    }

    antti.growOlder();

    System.out.println(antti); // same as System.out.println(antti.toString());
    System.out.println(pekka); // same as System.out.println(pekka.toString());
```

In principle, the `System.out.println` method requests the object's string representation and prints it. The call to the `toString` method returning the string representation does not have to be written explicitly, as Java adds it automatically. When a programmer writes:

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

Java extends the call at run time to the following form:

```
System.out.println(antti.toString());
```

As such, the call `System.out.println(antti)` calls the `toString` method of the `antti` object and prints the string returned by it.

We can remove the now obsolete `printPerson` method from the Person class.

The second part of the screencast:

Lisää olioista: toString-metodi

Programming exercise:

Agent

Points
1/1

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
```

Exercise submission instructions

How to see the solution

Method parameters

Let's continue with the `Person` class once more. We've decided that we want to calculate people's body mass indexes. To do this, we write methods for the person to set both the height and the weight, and also a method to calculate the body mass index. The new and changed parts of the Person object are as follows:

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

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

    public void setHeight(int newHeight) {
```

```
        this.height = newHeight;
    }

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

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

    // ...
}
```

The instance variables `height` and `weight` were added to the person. Values for these can be set using the `setHeight` and `setWeight` methods. Java's standard naming convention is used once again, that is, if the method's only purpose is to set a value to an instance variable, then it's named as `setVariableName`. Value-setting methods are often called "setters". The new methods are put to use in the following case:

```
public static void main(String[] args) {
    Person matti = new Person("Matti");
    Person juhana = new Person("Juhana");

    matti.setHeight(180);
    matti.setWeight(86);

    juhana.setHeight(175);
    juhana.setWeight(64);

    System.out.println(matti.getName() + ", body mass index is " + matti.bodyMa
    System.out.println(juhana.getName() + ", body mass index is " + juhana.body
}
```

Prints:

Sample output

```
Matti, body mass index is 26.54320987654321
Juhana, body mass index is 20.897959183673468
```

A parameter and instance variable having the same name!

In the preceding example, the `setHeight` method sets the value of the parameter `newHeight` to the instance variable `height`:

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

The parameter's name could also be the same as the instance variable's, so the following would also work:

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

In this case, `height` in the method refers specifically to a parameter named *height* and `this.height` to an instance variable of the same name. For example, the following example would not work as the code does not refer to the instance variable *height* at all. What the code does in effect is set the `height` variable received as a parameter to the value it already contains:

```
public void setHeight(int height) {
    // DON'T DO THIS!!!
    height = height;
}
```

```
public void setHeight(int height) {
    // DO THIS INSTEAD!!!
    this.height = height;
}
```

Programming exercise:

Multiplier

Points
1/1

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:

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

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

Multiplier multiplyByFour = new Multiplier(4);

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

Output

Sample output

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

Exercise submission instructions

How to see the solution

Calling an internal method

The object may also call its methods. For example, if we wanted the string representation returned by toString to also tell of a person's body mass index, the object's own `bodyMassIndex` method should be called in the `toString` method:

```
public String toString() {
    return this.name + ", age " + this.age + " years, my body mass index is " +
}
```

So, when an object calls an internal method, the name of the method and this prefix suffice. An alternative way is to call the object's own method in the form `bodyMassIndex()`, whereby no emphasis is placed on the fact that the object's own bodyMassIndex method is being called:

```
public String toString() {
    return this.name + ", age " + this.age + " years, my body mass index is " +
}
```

The screencast's third part:

Lisää olioista: oman metodin kutsuminen

Programming exercise:

Statistics (4 parts)

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: 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:

```
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:

```
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
```

Part 2: 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:

```
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
```

Part 3: 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
```

Part 4: 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
```

Exercise submission instructions

How to see the solution

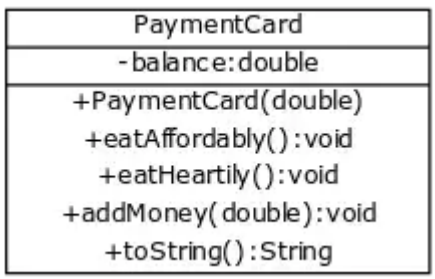
Programming exercise:

Payment Card (6 parts)

Points 6/6

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.

At the University of Helsinki student canteen, i.e. Unicafe, 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

Part 1: 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:

```
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
```

Part 2: 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
```

Part 3: 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.

Part 4: Topping up the card

Add the following method to the `PaymentCard` class:

```
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:

```
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
```

Part 5: 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:

```
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
```

Part 6: 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:

```
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
```

Exercise submission instructions

How to see the solution

Rounding errors

You probably noticed that some of the figures have rounding errors. In the previous exercise, for example, Pekka's balance of 30.7 may be printed as `30.700000000000003`. This is because floating-point numbers, such as `double`, are actually stored in binary form. That is, in zeros and ones using only a limited number of numbers. As the number of floating-point numbers is infinite — (in case you're wondering "how infinite?", think how many floating-point or decimal values fit between the numbers 5 and 6 for instance). All of the floating-point numbers simply cannot be represented by a finite

number of zeros and ones. Thus, the computer must place a limit on the accuracy of stored numbers.

Normally, account balances, for instance, are saved as integers such that, say, the value 1 represents one cent.

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

→ 2. Objects in a list

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