# Java programming

We are going to start from scratch. So, you don’t have to have any kind of basic coding skills before this course. Most of you *should* already know the basics of coding.

Almost all material is in English, but the lectures will be held in Finnish.

This is the first course of five coding courses where we use Java, and I am probably going to teach you all of them. I have planned them in a way that the things we are studying are presented in a logically proceeding manner. Here are the names of all the courses:

1. Java programming (Java-ohjelmointi)
2. Algorithmic programming (Algoritminen ohjelmointi)
3. Heuristics (Heuristiikat)
4. Intelligent algorithms (Älykkäät algoritmit)
5. Algorithm project (Algoritmiprojekti)

The information given in one course may overlap with another course. This means that I may present things in a course that could be presented instead on another course. Therefore, you really should take all these courses. I can assure you that if you study these courses hard you will have a solid ground for becoming an excellent coder.

Contents

[Java programming 1](#_Toc46756502)

[Contents 1](#_Toc46756503)

[1. Introduction 2](#_Toc46756504)

[About programming and this course 2](#_Toc46756505)

[2. JDK and NetBeans 2](#_Toc46756506)

[3. Coding, compiling and running a program. 3](#_Toc46756507)

[4. Algorithmic thinking. 3](#_Toc46756508)

[5. Variables and memory 4](#_Toc46756509)

[Characters 4](#_Toc46756510)

[Numbers 5](#_Toc46756511)

[Objects 5](#_Toc46756512)

[Memory handling 5](#_Toc46756513)

[Addresses 8](#_Toc46756514)

[Example of memory access and handling 8](#_Toc46756515)

[6. The control statements 9](#_Toc46756516)

[7. The Loops 9](#_Toc46756517)

[8. String handling 10](#_Toc46756518)

[9. Arrays 10](#_Toc46756519)

[10. Subroutines 10](#_Toc46756520)

[11. The rest of the course (separate documents or web-sources). 11](#_Toc46756521)

[12. References 11](#_Toc46756522)

Jump to chapter 11 before getting into the introduction.

## 1. Introduction

Java is a general-purpose programming language. It means it can be used for almost any kind of coding. The language is object-oriented and not dependent of the platform (operating system). It means you can always use the same code for different platforms – you just have to compile the program. Some references say that you don’t even need to compile the program, but I would not suggest it.

Since the year 2000 Java has been among the three most popular programming languages. See: <https://www.tiobe.com/tiobe-index/>, https://madnight.github.io/githut/#/pull\_requests/2021/2 and <https://statisticstimes.com/tech/top-computer-languages.php>. As a curiosity check out the “dead languages” C and C++. I would say that if you only want to learn one language properly, choose Java. It is the most popular language if you combine search engines, github-projects, and job openings.

As said, Java is an object-oriented language but can also be used in a functional manner. Functional programming concepts were added to the language in version 8. We will take a quick look at functional programming later.

### About programming and this course

People tend to think that all code is already done, and coding is just joining pieces of code together or simply finding the code that does the trick. This couldn’t be more wrong. The world of computer programs is getting more complicated as time goes by. When I started coding, different programs did not communicate with each other. Nowadays they do. They really do. See for example the email-program Outlook. It can be attached to almost anything – it is a calendar, meeting announcer, conference organizer, and many other things. In the beginning it was only used for sending and receiving email. The idea that you should reach many other programs from a single program is supposed to make things easier. But is it making things easier? Maybe for the users, but definitely not for the programmers.

I made a turn in my programming teaching two years ago. Before 2019 I focused much on how elegant the program was i.e. I tried to make beautiful code and tried to follow some design principles (DbC, MVC, Modularity, …). The last twelve years has taught me that I usually don’t have the time to think about design principles. I barely have enough time to make the program work. So, what is more important: 1) the program works as intended or 2) it is beautifully coded using some fancy design principle?

Moreover, some design principles can make the program run too slow. For example, if you try to make your program modular (or use DbC) you might run into big performance issues very soon. More on this on the lectures.

## 2. JDK and NetBeans

My intention was that we would do this together on the first lecture, but we can’t do that because you probably don’t have two screens or computers. So, you can’t watch and do at the same time. You must install the programs on your own.

First, we install the JDK, because NetBeans sniffs for it. If you already have some Java version and/or NetBeans, contact the teacher if you don’t manage to install everything properly. It though, doesn’t matter what version of the IDE you use and the JDK is new enough if it is at least version 8.

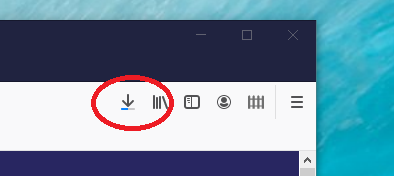
I am using version 15 at the time, but I am going to update to version 17 in September when the new version is published.

Go to:

<https://www.oracle.com/java/technologies/javase-downloads.html>

Click on JDK Download. Choose the right operating system and the installer version from the next page. Install the JDK to the suggested place. Version 11 has long-term support (LTS) but it does not matter what version you install.

In Firefox the downloaded file can be found (looks a bit different on this day):



Then we install the NetBeans IDE. Go to: <https://netbeans.apache.org/download/index.html>

Click *Download* on the latest version. As of August 16th, it is version 12, but it really doesn’t matter what version you use. Anything from 8 or above is sufficient. I use version 12.2 (12.0 with updates).

Choose (under Installers) the Windows version (choose linux or macosx if you have one of those operating systems). Choose any site.

I suggest you install it with the options the installer suggests. You might run into problems otherwise.

(We are almost ready to start coding. I don’t know what happens when we launch netbeans so our only option is to try. We will figure it out since we are doing this on the first lecture.)

## 3. Coding, compiling and running a program.

We will make a small program after we complete section 2. I’ll show you how to write, compile and run the program. And where to find the actual files. You will find this example in javaprogramming.zip.

package javaprogramming;

import java.util.Scanner;

public class Hello {

public static void main ( String[] args ) {

Scanner reader = new Scanner ( System.in );

System.out.print ("Gimme your name: ");

String name = reader.next();

System.out.println ("Your name is probably: " + name );

}

}

## 4. Algorithmic thinking.

This subject has its own course, but this is a thing we should understand in the very beginning of our coding journey. On this course I will only give you a short introduction to the subject. This introduction is however crucial to becoming a good coder. So, pay attention.

Your life is filled with algorithms although you might not be aware of it. Almost everything you do in real life can be formed as an algorithm. Let’s take a few examples and derive them into algorithms:

* Making coffee.
* Follow some recipe in food making.
* Change the tires of a car.
* Having a romantic evening with your lady/man.
* Cleaning the house.
* Driving from location A to location B.

You can begin by making the algorithm broad which means that it is not too specific. After that you can make it more specific i.e., add details to make it fine-grained.

In programming, the algorithm is the key to everything. If you can’t get it right, your program won’t work. Nowadays some languages or programming paradigms are trying to hide the algorithm behind operations, and that is dangerous. In my opinion, you should use such operations only after you truly understand the inner working behind them. A good example in Java are *streams*.

See SanallinenAlgoritmi.docx.

## 5. Variables and memory

In Java you have 3 kinds of types of variables: characters, numbers and objects. Characters and numbers are so called *simple* variables, also often called *built in/primitive types.* They can only store one value at a time. No matter how you handle built-in types, Java makes copies of them. If you assign a value from one variable to another variable, a copy is made. If you pass a value to a subroutine, a copy is passed. If you return a value from a function, a copy is returned. That is why they are very easy to handle.

The story is totally different when you handle objects. Objects never gets copied. When you assign an object to another object only the address of the object gets copied. So, after the assignment both variables *refer to the same memory location.* If you pass an object to a subroutine, only the address gets copied. I will (try my best to) make this point very clear in the forthcoming chapters.

### Characters

Java has two kind of character types: *char and String.* A char can only hold one single character, but the character can be “whatever” character, for example a, b, u, 9, “, #, ^. According to the Java specification “a char data type can hold a 16-bit Unicode character”. Unfortunately, the Unicode Standard encoding takes at most 21 bits nowadays. We will return to this subject later. For now, we are satisfied with the fact that a char can hold any character you can type from somewhere (usually the keyboard).

When saving a character to a char variable you have to enclose it with single quotation marks (‘). This is because the computer must be able to distinguish a char from any other type. A few examples:

char sign = ‘s’;

char smallA = ‘a’;

char atSign = ‘@’;

char plusSign = ‘+’;

A String is a is a sequence of characters, but it can also be empty or include only one character. Just like the char type the string type is based on UTF-16 but that is a subject of later times. A few examples:

String anyChars = “This is perfectly valid string with a few specials @£$”;

String emptyString = “”;

String onlyOneCharString = “c”;

You might have recognized that the type *char* started with a lowercase character and the String with an uppercase character. That is because the primitive types always start with lowercase characters and the objects always starts with an uppercase character. You should follow this “rule” when making your own objects.

### Numbers

Java has 7 number types: 4 integers, 2 floating point numbers and 1 boolean. The integers can store different sized integers – *byte* (8-bit), *short* (16-bit), *int* (32-bit) and *long* (64-bit). You can use byte and short to save memory, but if you don’t have to save memory, don’t use them. Use int 99 % of the time and long when you really need to save a large integer. Java converts all numeric integer values to int. So, f.ex., when you type number 12, Java takes it as an integer.

The 2 floating-point numbers are *float* (32-bit) and *double* (64-bit). Use float only if you need to save memory. Java converts all numeric floating-point numbers to double.

The last numeric type is boolean. It is a type that can have only two values: false or true. These values are traditionally coded as 0 (false) and 1 (true) and that is why they should be taken as numeric values.

Let’s look at Basics\BuiltIns.java.

### Objects

Every instance of every class is handled as an object in Java. When you use built-in object types (i.e. those classes that comes with the language or are included in some library package) or declare your own classes they are always objects when you instantiate them.

When handling objects, they are always handled with their address. It means, that the address is copied but not the thingies that the object is referring to.

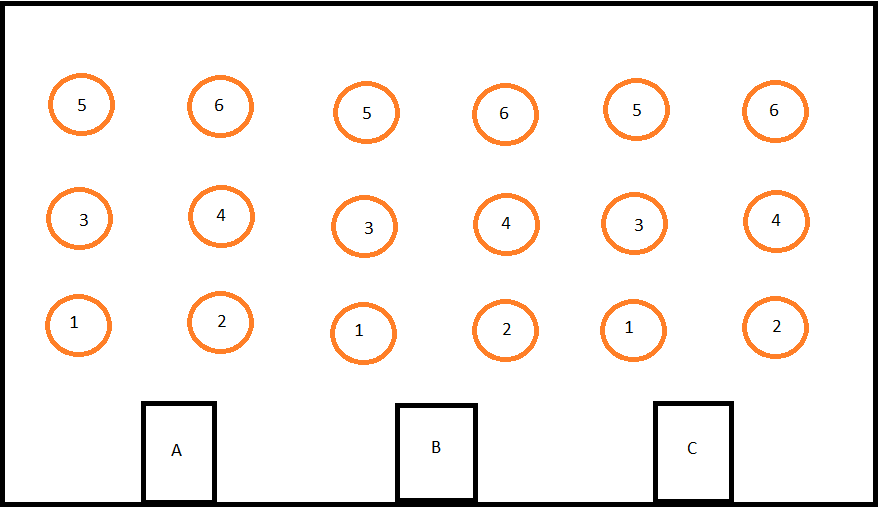
The most important thing to remember when handling objects is that they refer to a memory location. So, when you make an assignment:

SomeObject so = someOtherObject;

the object *so* starts to refer to same memory location as someOtherObject. Too many coders tend to forget this, and they seem to think that this assignment makes a copy of someOtherObject and assigns that copy to the variable so. But that is not the case. In some languages this kind of assignment makes a copy but not in Java. In Java, if you want a copy, you must make a copy by hand.

### Memory handling

Let’s start with a beautiful picture of a house, with three stairwells, each containing 6 one-room flats. This construction is a very simple representation of the memory of a computer. To refer to a flat you must type the name of the stairwell + the number of the flat. So, for example, B4 refers to flat number 4 in stairwell B and C1 refers to flat number 1 in stairwell C.



The name of the stairwell + the number of the flat is the *address* of the flat. (In real life you would need the name of the street and the name of the town to know the flats’ location.)

Our memory layout contains only 18 addresses, but a real computer’s memory contains millions of addresses. What is nice is that *usually* you don’t have to worry about these addresses. You simply make a *variable,* and the computer *allocates* the right amount of memory *from somewhere in the memory* for you.

So, we have a house with three stairwells each containing 6 one-room flats. Let’s assume that one person can occupy one flat at a time. So, there can never be more than one person in a flat. Let us define that the flat can hold a *string* representing the name of the occupant. Defining

String person1 = “Osmo”;

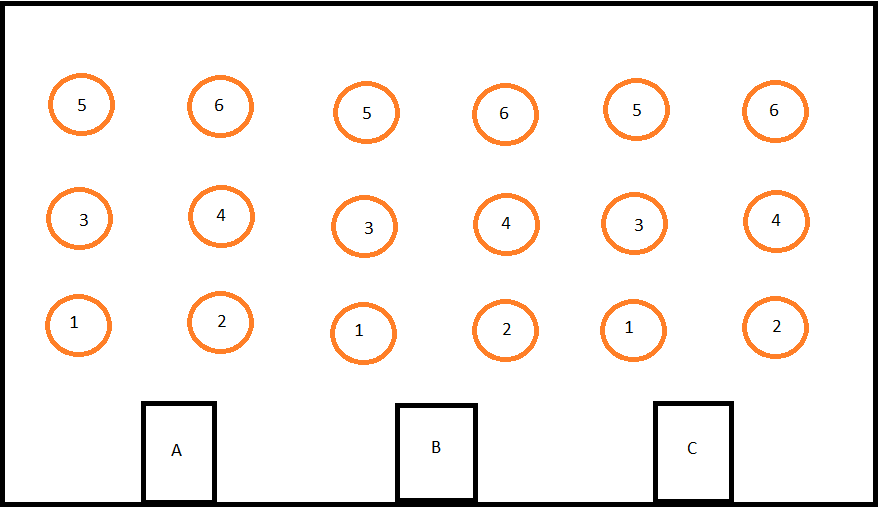
causes the computer to *assign* the value “Osmo” to a *variable* named *person1.* The computer places this variable in *some memory location.* Usually, you don’t have to worry about the memory location, but you must understand the concept.

Let’s assume that the computer puts the variable person1 in address A4 (stairwell A, flat 4). Now we have four things we know about this variable:

* The *type* of the variable is *String*.
* The *name* of the variable is *person1.*
* The *value* of the variable is *“Osmo”*.
* The *address* of the variable is *A4*.

Of these four things the programmer always decides the type, name, and the value of a variable. The *type* must be such that the variable can store the wanted *value* in the variable. The *name* should/must be such that it describes the thing you save into the variable. The address is determined by the computer, and you usually don’t have to worry about it.

One last thing to remember is that a *simple* variable can *never store more than one value.* This means that if you assign a new value to a variable, you will lose the old value. So, if you would need the old value in the future, you must save it in another variable. We will shortly take an example of this situation.

Because the picture is so beautiful, let’s copy it here once more.

Above we defined a variable person1 to include the string “Osmo”. Let’s define another variable:

String person2 = “Olli”;

Let’s say the computer puts this variable in memory location C2. So Osmo (person1) is located in address A4, and Olli (person2) is located in address C2.

What should we do if Osmo and Olli would like to change apartments (memory locations)? Remember, that a flat (memory location) can only hold one resident (value) at a time. I.e., if we assign a new value to a variable, we lose the old value. So, what happens if we try to change the values of the variables this way:

person1 = person2;

person2 = person1;

First, we assign the value of person2 to person1. The computer does that by making a *copy* of the value in person2 and assigns that copied value to person1. Therefore, after the first line of code we have the value “Olli” in both variables. The original value “Osmo”, that was located in variable person1, is lost.

What should we do? The problem is that a variable can only hold one value at a time. Every time you assign a new value to a variable, the old value is lost. If we however need the old value, we must save it somewhere.

Every time we need to assign a value to a variable and save the old value at the same time, we need to declare a new variable. In this case we need one new variable. Let’s name it *temp*, because it is a temporary variable, used only for a short time:

String temp = person2; // temp=”Olli”, person1=”Osmo”, person2=“Olli”

person2 = person1; // temp=”Olli”, person1=”Osmo”, person2=“Osmo”

person1 = temp; // temp=”Olli”, person1=”Olli”, person2=“Osmo”

On the first line we declare a new String variable temp. Remember that this variable also gets located in the memory somewhere. We don’t know the exact location, but we know that it is put in the memory. Then the computer copies the value of person2 into temp.

On the second line the computer copies the value of person1 into person2. On the last line the computer copies the value of temp into person1.

And voila: we have changed the values of two variables using one *helper* variable. The variable temp is not needed after this operation, and it can be forgotten. In Java this variable will be destroyed after we leave the *block* where the variable was introduced. But that is a story we will get back to in a few moments.

### Addresses

You cannot get/show the address of a simple variable i.e., a built-in type (actually you can, but it is a bit hard and usually very unnecessary). You can however get/show the address of an object. Or better said – you can get the object’s virtual address. This means that the address of an object does not show exactly where the object is located in memory but the address for different objects is *distinct*. This in turn means that if two objects have the same address, *they refer to the same memory location.* It is not important where the object exactly resides in memory, but it might be very important to know if two (or more) objects refers to the same memory location.

Nowadays the problem of two variables referring to the same memory location has been tackled with *functional programming*. In functional programming one of the main principles is that you *copy everything* when handling objects. We shall look at this *paradigm* later, but for now I can tell you that it has not solved the problem. And furthermore: it will never solve it. Why, you might ask. Well, because we need efficient programs.

When we use copying it means that we use copies of objects. Every time we handle an object, we first make a copy of it. Something like this:

Object o = new Object();

// Must update some fields in the object. So, we make a copy.

// Usually this is done under the hood i.e., automatically.

Object o2 = o.makeCopy(); // Copying it by hand.

Object o2 = o.doSomething(); // Making a copy under the hood.

Making copies is very time consuming. The bigger the object, the more time it takes to copy it. The lesson is: do not make unnecessary copies if you want your program to be efficient. On the other hand – let the program make copies if the performance of the program is not too important or the copying occurs in rare places.

(Functional enthusiasts are very eager in telling you that this paradigm changes everything. Usually that is *mainly* because functional code never mutates data (which is bullshit) and never saves the same region of memory in two different variables (which is also bullshit). These two “rules” can (maybe) make programs easier to read and maintain but are unrealistic in real life. If you for example need to update an object all the time, only a total idiot would make a copy of the object in every handling.)

### Example of memory access and handling

Let’s look at Basics\AddressBasics.java.

## 6. The control statements

This is to be presented in lecture.

if ( condition ) {

do\_something;

…

}

[else if ( someOtherCondition ) {

do\_something;

…

}

…

else {

do\_something;

…

}]

switch ( condition ) {

case value1 : do\_something;

…

break;

case value1 : do\_something;

…

break;

…

default: do\_something;

…

[break;]

}

Comparison (relational) operators: ==, !=, >, >=, <, <=

Logical operators: !, &, &&, |, ||, ^

## 7. The Loops

This is to be presented in lecture.

do {

do\_something;

…

} while ( condition );

while ( condition ) {

do\_something;

…

}

for ( initializeLoopVariable; condition; doAfterBodyRun ) {

body;

}

Let’s look at Basics\ControlLoops.java.

Let’s look at Basics\ControlLoops2.java. This may be hard to understand so hopefully we remember to get back to this example later.

## 8. String handling

In Java a String is a sequence of *any* characters. It is officially an object, but it behaves like a built-in type. It means that you cannot change the inner representation of a String – when using any method on a String a copy is made.

We will walk through this subject with the help of the documentation. This has two reasons – I am too lazy to write things down and we must learn how to use the Java documentation. The link to the String class: <https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/String.html>.

We will code some of the examples, so open NetBeans.

Let’s look at Basics\Stringing.java.

## 9. Arrays

An array is simply a variable that can store multiple values. So, an int can store one integer number, but int[] can store several integer numbers. The amount of numbers an array can store is determined by its size.

An array is always treated as an object in Java. Therefore, it must (always) be introduced with the *new* clause. A few examples:

int[] ints = new int[10]; // Stores 10 integers.

double[] floats = new double[7]; // Stores 7 double values.

String[] strings = new String[3]; // Stores 3 Strings.

Object[] objs = new Object[100]; // Stores 100 objects of any *kind.*

Most of the times you define one-dimensional arrays, like the above ones. But you can have as many dimensions on the array as you please. Just insert more brackets after the introduction of the variable (i.e. the type). A few examples:

int[][] ints = new int[2][2]; // A two-dimensional array (2x2).

double[][][][] = new double[2][4][8][16]; // A four-dimensional array (2x4x8x16).

The first array has 4 places (two in a “row” and two in every “column”). The second has 2\*4\*8\*16 = 1024 places. You usually don’t want to have more than three dimensions in your arrays. They get tricky to handle with even three dimensions. I usually don’t make more than two-dimensional arrays.

The values are addressed with numbers corresponding to the arrays’ indexes. For example, the above int[] tables’ values can be addressed this way: ints[0] (the first place), ints[1] (the second place), …, ints[9] (the last place). The number of the index can, of course, be stored in a variable. The index starts from 0 and ends with length-1.

A two-dimensional array is indexed in the same way – we only have two indexes. The above int[][] tables’ values are addressed this way: ints[0][0], ints[0][1], ints[1][0] and ints[1][1].

Let’s look at Basics\ArrayHandling.java.

## 10. Subroutines

Subroutines are used for two purposes: 1) dividing the code into smaller, more manageable portions 2) make non-repetitive code i.e., not solving the same problem more than once.

Here is a description of a subroutine:

[return\_type|void] name ( [type parameter1, type parameter2, … ]) {

do\_the\_things;

[return return\_type;]

}

The subroutine can return a value of a specified type. If it does, the type must be specified. If the subroutine doesn’t return anything the keyword *void* is used to imply it.

The *name* of the subroutine should be descriptive and in Java it should always start with a lowercase letter. Do not use the underscore character – it does not make the code look cool. It makes you look stupid.

The subroutine can receive parameters. If it does the types and names of those parameters must be specified. There is an exception to this rule, but we will not deal with it at this point.

Then there is the block of code that does the things the subroutine is supposed to do. The block is enclosed with parenthesis ({}).

Usually at the end of the subroutine there is a return-clause which return the value computed in the code. The return-clause can though be anywhere in the code and there can be multiple return-clauses.

A couple of examples:

int max ( int val1, int val2 ) {

if ( val1 >= val2 ) return val1;

else return val2;

}

int max ( int val1, int val2 ) {

int retVal = -1;

if ( val1 >= val2 ) retVal = val1;

else retVal = val2;

return retVal;

}

Let’s look at Basics\SimpleMethods.java, MathMethods.java, Methods.java and MoreMethods.java

## 11. The rest of the course (separate documents or web-sources).

* Classes
* Inheritance
* Static thingys
* Interfaces
* Exceptions
* Generics
* Threads
* (Functional programming)
* Data structures

## 12. References

* Support roadmap: <https://www.oracle.com/java/technologies/java-se-support-roadmap.html>
* We use version 11 because it has long-term support (LTS). But you can use whichever version of Java you like. In September a new version having LTS is published.
* Documentation (you’ll find everything you need): <https://docs.oracle.com/en/java/javase/11/index.html>
* Tutorials: <https://mkyong.com/>
* Generics: <http://www.angelikalanger.com/>