Lecture 2

`if`, `for`, `while`

Lecture 1 recap

```java
import static javax.swing.JOptionPane.*;

public class Hello {
    public static void main(String[] args) {
        String name;
        name = showInputDialog(null, "Vad heter du?");
        showMessageDialog(null, "Hej " + name + "!");
    }
}
```

Lecture 1 recap

Make `showInputDialog`/`showMessageDialog` available

```java
import static javax.swing.JOptionPane.*;

public class Hello {
    public static void main(String[] args) {
        String name;
        name = showInputDialog(null, "Vad heter du?");
        showMessageDialog(null, "Hej " + name + "!");
    }
}
```

Lecture 1 recap

```
import static javax.swing.JOptionPane.*;

public class Hello {
    public static vo d main(String[] args) {
        String name;
        name = showIn   Dialog(null, "Vad heter du?");
        showMessageDia      ull, "Hej " + name + "!");
    }
}
```

Curly braces are used to group things into *blocks*.

Lecture 1 recap

A *class*: a collection of data and operations.
Java programs are built from these.

```
import static javax.swing.JOptionPane.*;

public class Hello {
    public static void main(String[] args) {
        String name;
        name = showInputDialog(null, "Vad heter du?");
        showMessageDialog(null, "Hej " + name + "!");
    }
}
```

Lecture 1 recap

The `main` method.
Executed when the program starts.

```
import static javax.swi OptionPane.*;

public class Hello {
    public static void main(String[] args) {
        String name;
        name = showInputDialog(null, "Vad heter du?");
        showMessageDialog(null, "Hej " + name + "!");
    }
}
```

Lecture 1 recap

```java
import static javax.swing.JOptionPane.*;

public class Hello {
    public static void main(String[] args) {
        String name;
        name = showInputDialog(null, "Vad heter du?");
        showMessageDialog(null, "Hej " + name + "!");
    }
}
```

A *variable* of type `String`, called `name`.
A named location for storing text data.

Lecture 1 recap

The `showInputDialog` method is *called*.

The *arguments* to `showInputDialog`: `null` and the string “`Vad heter du?`”

```
import static java   ing.JOpt

public class Hello {
    public static void main(String[] args) {
        String name;
        name = showInputDialog(null, "Vad heter du?");
        showMessageDialog(null, "Hej " + name + "!");
    }
}
```

The *return value* is assigned to the variable `name`.

Lecture 1 recap

Semicolons terminate *statements*.

```
import static javax.swing.JOptionPane.

public class Hello {
    public static void main(String[] args) {
        String name;
        name = showInputDialog(null, "Vad heter du?");
        showMessageDialog(null, "Hej " + name + "!");
    }
}
```

Expressions have types and values.

Administrative stuff

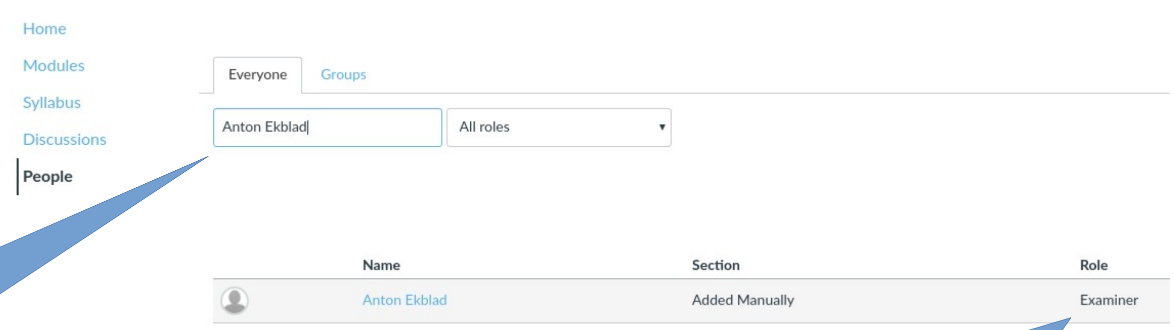
Still looking for student representatives!

(There will be cake.)

About the exam

- When? **16/12** at **08.30 – 11-30**
- Where? **SB-Multisal**
- Registration **not** through Studieportalen
  - Registered on course = registered on exam
  - Make sure you’re registered in Canvas!

Go to “People”

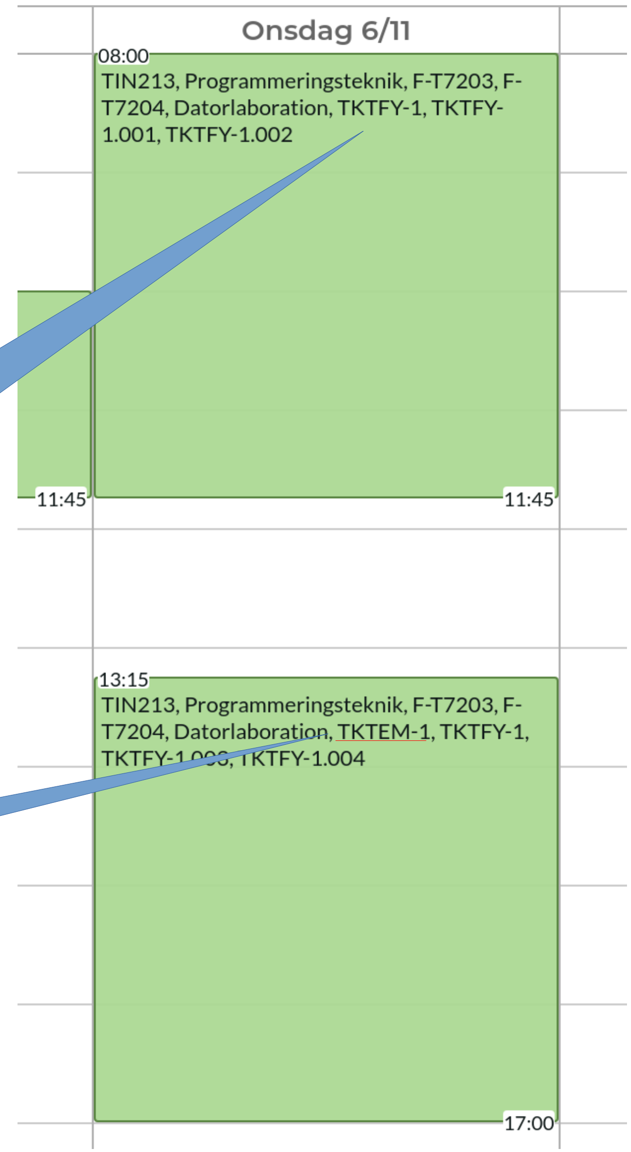


Search your name

Role should be “Student”

Lab groups

- All TM students have the same group for labs
- Still two TM groups for exercises



No TM!

Yes TM!

Lab groups

- Omreg/other groupless without partners:
  - Find a lab partner
  - Join their group

Lab Matchmaking

- Matchmaking during the break
- If you have not found a lab partner after that, try posting a “looking for partner” message on the discussion board.
- If you *still* can’t find one, email me (`antonek@chalmers.se`)

Enough with the boring stuff,
let’s code!

Symbols Used in This Course

When we give instructions on how to write Java:

Syntax

The rules of Java
If you do not do it this way, it will not work!

Best Practice

There are several ways that will work.
This way is almost always better than the others.

Conventions

There are several ways that will work. None is best.
This is the way everyone does it. It was an arbitrary choice.

Common Mistakes

Be Boring

Follow the best practices and the conventions unless there is a **very** good reason not to.

It makes your code easier to read and understand.

“Principle of Least Surprise”

Example
Mobile phone calculator

Given:

- the price per minute and number of minutes used in a month,
- the price per GB and number of GB used in a month,

calculate my phone bill this month.

If I buy more than 100 GB, I get a 10% discount on the whole bill

Conditional Statements

```
if ( condition ) {
    block
}

if ( condition ) {
    block
} else {
    block
}
```

The boolean expression is evaluated.

If it is `true`, the first block is executed.

If it is `false` and there is an `else` block, the `else` block is executed.

Conditional Statements

```
if ( condition ) {
    block
}
```

```
if ( condition ) {
    block
} else {
    block
}
```

If there is only one statement in the block, the braces `{ }` are optional.

Include them anyway

Conditional Statements

```
if(userIsAdmin)
    doSensitiveOperation();
```

If there is only one statement in the block, the braces `{ }` are optional.

Include them anyway

Conditional Statements

```
if(userIsAdmin)
    doSensitiveOperation();
    doAnotherSensitiveOperation();
```

If there is only one statement in the block, the braces `{  }` are optional.

Include them anyway

Conditional Statements

```
if(userIsAdmin)
    doSensitiveOperation();
    doAnotherSensitiveOperation();
```

This line is always executed!

If there is only one statement in the block, the braces `{ }` are optional.

Include them anyway

Conditional Statements

```
if(userIsAdmin) {
    doSensitiveOperation();
    doAnotherSensitiveOperation();
}
```

Much better!

If there is only one statement in the block, the braces `{ }` are optional.

Include them anyway

A Common Pattern

```
if ( condition1 ) {
    block 1
} else if ( condition 2 ) {
    block 2
} else if ( condition 3 ) {
    block 3
} else if …
…
… else {
    block n
}
```

If the number of conditions gets too large, consider reorganising your code.

Example
Projectile range calculator

We are firing a projectile at a given initial speed and angle.
Assume no air resistance and level ground.
What is the range of the projectile? (How far along the ground has it travelled when it hits the ground again?)

We will make it easy for ourselves
Just use the formula:

$$r = \frac{2 v^2 \sin\theta \cos\theta}{g}$$

If it flies more than 40 meters, say “nice throw”
Otherwise, say “try again”

Example

Every month I invest 1000 kr in a bank account with 3% interest.

I am rich if I own 1.000.000 kr

When will I become rich?

`while`-loops

- `while (` condition `) {`
  block
  `}`

If the condition is true, execute the block
Keep executing the block until the condition becomes false
Note: No semicolon at the end!

- `do {`
  block
  `} while (` condition `);`

Execute the block
If the condition is true, execute the block again
Keep executing the block until the condition becomes false
(Always executes the block at least once.)
Note: yes semicolon at the end!

Example

The user enters an integer `k`

Calculate the sum of all the integers from `1` to `k`

`for`-loops

```
for ( initialisation; condition; increment ) {
  block
}
```

Execute the initialisation statement
If the condition is true, then execute the block and the increment
Keep executing (block then increment) until the condition becomes false

Note: If the condition is false immediately after initialisation, then block is never executed

Examples of using for

- `for (int i = 1; i <= 10; i++)`
  counts from 1 to 10

- `for (int i = 10; i >= 1; i--)`
  counts down from 10 to 1

- `for (int i = 2; i <= 10; i+=2)`
  counts 2, 4, 6, 8, 10

`for` and `while` are equivalent

```
for ( initialisation; condition; increment ) {
  block
}
```

does the same as

```
{
 initialisation;
 while ( condition ) {
  block
  increment;
 }
}
```

`for` and `while` are equivalent

```
while ( condition ) {
    block
}
```

does the same as

```
for (;condition;) {
    block
}
```

`for` and `while` are equivalent

- Use `for` when you know in advance how many times the block will be executed

  Use `while` when the condition depends on something that will be changed inside the block

Rules for Naming Variables

Syntax A variable name may be any combination of lower-case letters, upper-case letters, numerals and the underscore (_) and dollar (`$`) sign,

but they must not start with a numeral.

Convention Variables start with a lower case letter and use “camel case” (`numberOfCustomers`)

Best Practice

Please do not use Swedish letters (ä, å, ö) in your code.
(I know the textbook does this, but please don’t!)

When these are uploaded to Fire then downloaded, they can become corrupted.

If you want to write Swedish, use
“ae” for “ä”, “aa” for “å”, “oe” for “ö”

Types

- Primitive types

  `boolean`, `byte`, `short`, **`int`**, `long`, `float`, **`double`**, **`char`**

- Types defined in library

  `String`, ...

- Define your own types

Built-in Types

- Types for integers:

Type	Values	Memory Used
`byte`	-128 to 127	1 byte
`short`	-32768 to 32767	2 bytes
`int`	-2147483648 to 2147483647	4 bytes
`long`	-9223372036854775808 to 9223372036854775807	8 bytes

- Types for floating-point numbers:

Type	Values	Memory Used
`float`	$\pm 3.40282347 \times 10^{38}$ to $\pm 1.40239846 \times 10^{-45}$	4 bytes
`double`	$\pm 1.79769313486231570 \times 10^{308}$ to $\pm 4.94065645841246544 \times 10^{-324}$	8 bytes

Best Practice Use `int` and `double` unless there is a very good reason not to.

Built-in Types

- Other

Type	Values	Memory
`boolean`	`true` and `false`	1 byte
`char`	one character	2 bytes

Numeric Literals

A *literal* is an expression denoting a fixed value.

- `int` literals
  - Decimal numerals: `26`, `0`, `−15`
  - Hexadecimal numerals: `0x1a`
  - Binary numerals: `0b11010`
- `double` literals
  - Numeral with a decimal point: `135.7`, `26.0`
  - Scientific notation: `3.0e5`, `2e−4`, `1.9E7`
  - These can be followed by `d`: `135.7d, 2e−4D`
- `long` literals
  - `int` literal followed by `l`: `99l`, `−24L`
- `float` literals
  - double literal followed by `f`: `135.7f`, `1.9E7F`

Literals

- `char` literals
  - Character in single quotes: `'a', ' ', '$', '0'`
  - Unicode code: `'\u0275'`
  - Escape character
    - `'\n'`: Newline
    - `'\''`: Single quote
    - `'\"'`: Double quote
    - `'\\'`: Backslash
- String literals
  - Sequence of characters in double quotes: `"Hello World", "a", "", " "`
  - Same escape sequences as for characters
  - Can mix regular characters and escape sequences: `"Hello World!\nMy name is \"Anton\"\n"`
- boolean literals
  - `true/false`

Arithmetic Operators

On the *numeric types:*

`byte`, `short`, `int`, `long`, `float`, `double`

We can build up expressions using *operators*.

- Addition: `x + y`
- Subtraction: `x - y`
- Multiplication: `x * y`
- Division: `x / y`
- Modulus: `x % y` – Remainder when dividing `x` by `y`

  Example: `12 % 5` returns `2`

  - Most common use: `if (x % 2 == 0)` tests whether `x` is even

Integer division returns an integer!

`double a = 12 / 5` will set `a` to be `2.0`

Logical Operators

On the types `int` and `boolean`, we can use `==` to test equality:

`x == y` is `true` if `x` and `y` have equal values

`x != y` is `true` if `x` and `y` have unequal values

On the numeric types, we can write:

```
x < y    x > y    x <= y    x >= y
```

Rounding Errors

Whenever you do arithmetic using floating point numbers, expect rounding errors.

Therefore, never use == to compare two floating point numbers.

Instead, use `Math.abs(x - y) < 0.001`

Choose this constant wisely!

Logical Operators

We can use these operators on the type `boolean`:

- `x && y` – "`x` and `y`".
  True if x and y are both true
  False if x is false or y is false or both
- `x || y` – "x or y"
  True if x is true or y is true or both (*inclusive* or)
  False if x and y are both false
- `! x` – "not x"
  True if x is false
  False if x is true

Logical Operators

Don't compare `boolean` values to `true`/`false`:

- `x == true` is equivalent to `x`
- `x == false` is equivalent to `!x`
- Good:
  - `while(condition)`
  - `if(!condition)`
- Bad:
  - `while(condition == true)`
  - `if(condition == false)`

Other Operators

- `x++` - increments x (adds one to x) and returns the value after increment
- `++x` – increments x and returns the value before increment
- `x--` - decrements x (subtracts one from x) and returns the value after decrement
- `--x` – decrements x and returns the value after decrement
- `-x` – returns the negation of x (-1 multiplied by x)
- `x += y` – equivalent to x = x + y
- `x -= y` – equivalent to x = x – y
- `x *= y` – equivalent to x = x * y
- `x /= y` – equivalent to x = x / y
- `cond ? x : y` – evaluates cond. If cond is true, returns x. If cond is false, returns y

Priority

`3 + 4 * 5 % 6` will be evaluated as `3 + ((4 * 5) % 6)`

Full rules for priority in appendix A of *Java Direkt*

If in doubt – use brackets

Use brackets if the reader might have a hard time working out the priorities,
even if the compiler does not need them

```
3 + (4 * 5) % 6
```

Typecasting

When we want to change an `int` into a `double`:

```
int a = 3;
```

The expression `(double)a` returns the double value `3.0`

We have cast or typecast an integer to a double.

Typecasting

```
double a = 1 / 2;
```

This sets `a` to be `0.0`. Not what we wanted. Instead:

`double a = 1.0 / 2;` or `double a = 1 / 2.0;`
or `double a = 1.0 / 2.0;`

```
int a = 1;
int b = 2;
double c = a / b;
```

This sets `a` to be `0.0`. Not what we wanted. Instead:

`double c = (double) a / b;`
or `double c = a / (double) b;`

Reading this week:

- *Java Direkt med Swing* 1.8-1.13, 2.4, 3.4, 5.2
- *Code Complete* Chapter 8

Exercises this week:

- *Java Direkt med Swing* section 1.16 (excluding exercises about graphical programs)