(2) Java Introduction: Imperative Programming – Part 1

Nico Ludwig (@ersatzteilchen)

TOC – Part 1

TOC – Part 2

- Cited literature:
  - Just Java, Peter van der Linden
  - Thinking in Java, Bruce Eckel

Initial Words

Yes, my slides are heavy.

I do so, because I want people to go through the slides at their own pace w/o having to watch an accompanying video.

On each slide you'll find the crucial information. In the notes to each slide you'll find more details and related information, which would be part of the talk I gave.

Have fun!

Starting to write Programs

- Well, we discussed a lot of theory up to now. And now it is time to get our feet wet!
  - Assume this Java program, which just prints the text "Hello, World!" to the console:

```
// <Program.java>
public class Program {
	public static void main(String[] args) throws Exception {
		System.out.println("Hello, World!");
	}
}
```

  - The text, which is the program's code, looks weird:
    - What do the different English words do?
    - What do the different parentheses mean?
    - And what is the meaning of colors?

- We'll learn, that there are rules in the Java programming language, which are not difficult to understand.
  - These rules span two things: the syntax of the language and the semantics of the language.
  - The first hurdle to get is getting our code compiled, therefor we have to learn how to write correct or valid Java programs.

- In this lecture, we'll get a feeling for the syntax and gradually understand the semantics of syntactic elements.
  - Keep in mind, that HLLs like Java are specially designed to be human-readable.

Code Snippets

- Hence we'll begin using code snippets (or simply called "snippets") as examples
  - Java code is written into ordinary text files, they just have the suffix .java.
  - A Java program also brings a lot of "fluff", which has no meaning to us right now.
  - Therefor we will strip Java code down to snippets:

```
// <Program.java>
// A full Java program.
public class Program {
        public static void main(String[] args) throws Exception {
                System.out.println("Hello, World!");
        }
}
```

→

```
// <Program.java>
// The pale text represents "fluff".
public class Program {
        public static void main(String[] args) throws Exception {
                System.out.println("Hello, World!");
        }
}
```

→

```
// This is a code snippet:
System.out.println("Hello, World!");
```

- In Java snippets, we'll get rid off the surrounding class definition and the surrounding *main()*-method.
  - We'll learn what classes and methods are for in future lectures.
  - Then only the "essence" of the code remains.
  - But: A snippet is no runnable program code by itself, it is still required to put its code into a *main()*-method of a class definition.
  - => i.e. for running Java programs we used a class definition and a *main()*-method, without knowing why it is required.
- We'll use mostly snippets, instead of fully blown Java programs in upcoming lectures!

6

- Java code is written in text files. Sometimes the code is called "source code" or programmers just call the whole source code as "the sources".
- An important thing to keep in mind is that the execution of a Java program starts in the static method *main()*. – Hence we accept that we will leave the explicit definition of *main()* away in the code examples of this course.
- Esp. in these these very small code snippets, the surrounding definition of the *main()* method and the need to put yet another class definition, really looks like a lot of fluff! The scripting language Groovy, which is derived from Java, does allow to get rid of exactly such fluff. – So if you will some of the following Java code snippets are more or less compatible to Groovy.
  - Of course the *main()*-method and the class definition are not "fluff"! – But we'll not concentrate on those right now.

Compile and Run our Program on the Console

- Before we can put the presented Java program into effect, it needs to be compiled.
  - Compilation means, that the symbolic Java code is transformed into byte code.
  - Let's assume our program code resides in Program.java.
  - A compiler is a program, which performs this transformation. The Java compiler of the JDK is javac.

- Compile the Java program in Program.java, the byte machine code is put into a binary file named Program.class:

```
// <Program.java>
// A full Java program.
public class Program {
        public static void main(String[] args) throws Exception {
                System.out.println("Hello, World!");
        }
}
```

```
Terminal
NicosMBP:src nico$ javac Program.java
NicosMBP:src nico$ ls
Program.class Program.java
NicosMBP:src nico$
```

  - If we compile multiple java-files, javac will compile them in the correct order.

- After the program is compiled, we can execute it with the Java interpreter, the standard Java interpreter is java.
  - The Java interpreter awaits a class-file as argument. The name of the file must be passed without the .class-suffix:

```
Terminal
NicosMBP:src nico$ java Program
Hello, World!
NicosMBP:src nico$
```

Highlighting Code in Colors

- If code in a snippet produces command line output, it will occasionally be shown in the code with the "// >"-notation:

```
System.out.println("Hello, World!");
```

→

```
System.out.println("Hello, World!");
// >Hello, World!
```

Good to know
The symbol "/" is usually called (forward) "slash" or "whack" (but "whack" is virtually the term for the backslash). Sometimes it is also called solidus. However, "slant" is the official ASCII name for "/".

- As can be seen, the text line starting with "// " was highlighted in green color.
  - In Java a line starting with two slashes "//" is a Java-comment.

- We use colors in the snippets to highlight elements in the code: brown for text, blue for keywords and green for comments.
  - We use colors in the code only to highlight different elements of the language.
  - We'll discuss the meaning of texts, keywords and comments in the upcoming slides.

- The highlighting (i.e. coloring) of language elements could look completely different in the code editor you use:

```
public class Program {
    public static void main(String[] args) throws Exception {
        System.out.println("Hello, World!");
        // >Hello, World!
    }
}
```

↔

```
public class Program {
    public static void main(String[] args) throws Exception {
        System.out.println("Hello, World!");
        // >Hello, World!
    }
}
```

  - All code editors allow to configure the coloring of language elements to fit our need.

Console Output

- We have to discuss the command *System.out.println()* because we will use it often: it writes output to the console.
- When we compile and run this snippet of code (of course it must be enclosed in *main()* etc.), we get this output:

```
System.out.println("Hello, World!");
```

```
Terminal
NicosMBP:src nico$ java Program
Hello, World!
NicosMBP:src nico$
```

- Alternatively, we get the same console output with this code:

```
System.out.println("Hello, "
                   + "World!");
```

```
Terminal
NicosMBP:src nico$ java Program
Hello, World!
NicosMBP:src nico$
```

  - *System.out* is an object, which represents the console and with the *println()*-method we send stuff out to the console.
  - *println()* expects an argument, which is set between the parentheses, e.g. textual data, which is written in double quotes.
  - Actually, we can mix the +-operator with any textual and non textual data to build a text to write to the console.

- Each call of *System.out.println()* writes a text and a newline.
  - If we want to put a newline between "Hello, " and "World" we can call *System.out.println()* twice:

```
System.out.println("Hello, ");
System.out.println("World!");
```

```
Terminal
NicosMBP:src nico$ java Program
NicosMBP:src nico$
Hello,
World!
NicosMBP:src nico$
```

Syntax and Semantics

- To learn a programming language its syntax and its semantics must be learned.
- For example the syntax of this code snippet:

```
System.out.println("Hello, World!");  // line (1)
// >Hello, World!                    // line (2)
```

  - On line (1) we have the words *System*, *out* and *println* separated by dots.
  - *println* is followed by a pair of parentheses, which encloses the text "Hello, World!" written in quotes.
  - The line ends with a semicolon.
  - Line (2) starts with // and contains the text ' >Hello World!'.
- What we've described here is the structure of the code, its words and individual characters. This structure is called syntax.
  - Therefor, the (colored) highlighting of the code's structure is also called syntax highlighting.
- What this structure really does, when the program runs is not obvious. The meaning of the structure is called semantics.

Syntax: structure of the code ↔ **Semantics**: meaning of the syntax

- The special quality of HLLs is, that their syntax allows to guess its semantics pretty reliably.

10

- Esp. the need to write semicolons to terminate statements scares Java newbies.

Java Syntax Cornerstones – Part 1 – Imperative Elements

- We'll begin by discussing imperative programming in Java.

> **Definition**
> *Imperative programming means to program with statements being executed sequentially to change the state of the program.*

- Imperative programming is a general programming paradigm.
  - Other paradigms: procedural programming, functional programming and object-oriented programming.

> **Good to know**:
> Imperative from latin *imperare* – to command someone/something.

- Imperative programming includes following (imperative) elements in general:
  - Values – values, which represent the state or possible states of the program, which have a type and consume memory
  - Variables – hold the state of the program, they are abstract locations (or "cells") in memory with names
  - Operators – connect values and variables to express an operation
  - Expressions – formulate a combination of values, variables and operators that can be evaluated, yielding another value
  - Statements – executable instructions, that execute expressions, which typically change the state (i.e. the contents of variables) of the program
  - Conditional branches – execute statements depending on a condition
  - Unconditional branches – jumps between statements unconditionally
  - Loops – execute statements repeatedly
  - Input and output – communicate with the "world outside of the program" (the user, the file system or a network)

Java Syntax Cornerstones – Part 2 – Values

- Values represent a very important, yet simple concept in any programming language.
  - E.g. in the "Hello, World!" program we saw the value "Hello, World!".

```
System.out.println("Hello, World!");   // The literal value "Hello, World!"
                                       "Hello, World!"
```

- Values simply represent data in a program.
  - All values in Java have a certain type and a certain memory consumption.
    - The text "Hello, World!" is of type *String*, which is the type to hold textual data in Java.
    - The memory consumption of "Hello, World!" is relevant, but not important to be given in concrete numbers, however, it depends on the *String*'s length.
- If we write a value directly in our source code, e.g. "Hello, World!", we call such a value literal value, or just literal.
  - Java's primitive types can have literal values. E.g. writing a text into double quotes makes it a literal *String* value or *String*-literal.
  - In this course, we highlight *String*-literals in brown color.
- However, values of primitive types have an important limitation: Those values can not be modified in Java!
- Sometimes, values are also called objects in Java, but this is a story we have yet to clarify.
- To do something useful with unmodifiable values, we have to understand the ideas of expressions and variables.

- In assembly languages literals are sometimes called immediate constants. In opposite to most HLLs, immediate constants are written immediately into the instruction stream. In HLLs, literals are usually stored into a kind of data segment, from which the literal values are loaded.

Java Syntax Cornerstones – Part 3 – Expressions

- Expressions represent another important, but simple concept in any programming language.
  - E.g. consider this expression:

```
// Expression:
3 + 4
```

- Obviously, expressions just represent "calculations" in a program, in this case an addition calculation.
  - Java's +-operator does expectedly add values, as we know it from mathematics.
  - Java has to evaluate an expression (i.e. "do the addition" in this case), to get its resulting value (i.e. the "sum" in this case).
    - To talk about the value of an expression we say the expression "evaluates to a value" or it "yields a value", or that the expression "returns a value".
  - This expression evaluates to the value 7. – Notice, that "the 7" is not written as literal value, it is rather a computed value.
  - On the other hand, we used two literals, 3 and 4, to formulate this expression.
  - 3 and 4 are int-literals. Java's primitive type int, which is a type to hold integer numbers.

- I didn't call expressions "mathematical terms", because sometimes they aren't.
  - Java's expressions can yield values and modify the state of a program. The latter concept is basically unknown in mathematics.
  - Java uses types and operators, which have no counterpart in ("elementary") mathematics.

- Elementary facts about expressions:
  - A literal value is also an expression. A literal is an expression evaluating to itself.
  - An expression can be built up from other expressions.

```
3 // this is an int-literal and an expression
```

```
3 + 4 - 7 // multiple expressions
```

Java Syntax Cornerstones – Part 4 – Values: Output, Intermediate Results and Variables

- From the "Hello, World!" program we know, that *System.out.println()* is a syntax to output information to the outside world:
  - We know how to output textual values, e.g. the *String*-literal "Hello, World!" to the command line from a Java program:

```
System.out.println("Hello, World!");
// >Hello, World!
```

    - A new syntactic need is, that we have to write a ; (semicolon) at the end of *System.out.println()*, that executes the output of "Hello, World!".
  - Well, it is also possible to output numeric values, e.g. the int-literal 42 to the command line:

```
System.out.println(42);
// >42
```

  - We can also output the immediate result of an expression to the command line:

```
System.out.println(45 * 3 + 21 + 5 + 45 * 3);
// >296
```

    - The operator * means multiplication! When we progress seeing more Java code, we'll see more of Java's operators.

- Above, the expression "45 * 3" is used in the calculation for two times. Java allows storing intermediate results in variables.
  - We can calculate the expression "45 * 3" and store its intermediate result in a variable named *product*:

```
int product = 45 * 3;
```

  - Then we can use *product* in further expressions to get the effective result:

```
System.out.println(product + 21 + 5 + product);
// >296
```

  - Here we can clearly see, that the line calculating *product* must be executed sequentially before *product* is used in the next line!
  - As can be seen, we've defined a variable *product* with the keyword int and initialized it with the result of the expression 45 * 3.
  - Using the keyword "int" defines a variable being of type integer. Integer variables can only hold integer values!

Java Syntax Cornerstones – Part 5 – Keywords, Statements, Syntax vs Semantics and Errors

- Java reserves symbols for its grammar, these symbols are called keywords.
  - In upcoming snippets all keywords are written in blue color. We already saw some keywords: int, class etc.
- We see ; all over in Java code. They must be written to execute expressions. Expression terminated with ; are called statements.
- Initialization and assignment statements are excellent examples to consider syntax versus semantics.
  - Initialization and assignment statements have similar Java-syntax, but their meaning is different: they have different semantics:

```
int product = 45 * 3; // Initialization
```

```
product = 42;          // Assignment
```

- Compile time errors versus run time errors:

```
49 = 3 + 5; // Invalid initialization/assignment to a constant value
```

  - This code is no correct Java-syntax! The compiler will reject it, issue a compile time error and abort the compilation-process.
    - An important function of the Java-compiler is to check the syntax for syntax errors. Syntax errors result in compile time errors and abort compilation.
    - The syntax error here: a constant literal value can not be assigned! Sounds logical...

```
int zero = 0;
int oddResult = 42/zero;
```

  - Both statements are ok for the compiler.
    - But the last one throws an *ArithmeticException* (java.lang.ArithmeticException: / by zero) at run time and terminates the program!

15

- What keywords does the audience know from any programming languages?
- Syntax versus semantics; what's that?
  - Grammar/keywords versus their meaning.
- Programming errors can occur on different "times" in development. Can anybody explain this statement?
- There are link time errors in the C-family languages, but this kind of errors isn't present in Java where linking happens at run time.
  - The time during programming is often called "design time".

Java Syntax Cornerstones – Part 6 – Variables

- Besides values, the most elementary syntax cornerstones are variables. Variables hold data/values/state in a program.
  - In Java, variable definitions make variables applicable in the code.
  - Variables need to be typed on definition, this is called static typing.
  - Java doesn't have the concept of (variable) declarations.
  - We can define multiple variables of the same type in one statement as a comma-separated list in a compound declaration:

```
int width = 32;
```

variable type | variable name | initialization

```
int | width | = 32;
```

```
int width = 32, height = 67;
```

- A defined variable has a data/value/state and consumes memory.
  - Sometimes, the compound of a portion of memory and the data/value/state it holds is called object.
- Besides definition, variables can be initialized, assigned to and read.
  - Initialization sets initial values to a variable.
    - Uninitialized variables have a strictly defined default value in Java! We'll discuss this topic in short.
  - Assignment sets a variable to a new value.
  - Local variables need to be effectively initialized by initialization or assignment before usage:
  - Reading retrieves the value of a variable.

```
int age = 19; // Initialization
```

```
age = 36; // Assignment
```

```
int diameter;
// Invalid! Variable diameter might not have been initialized
System.out.println(diameter);
```

```
int diameter;
diameter = 56;
// Fine! diameter will be effectively initialized!
System.out.println(diameter);
```

```
int hisAge = age; // read age and initialize hisAge.
```

16

- The same variable should only be used for exactly one purpose!
- After definition variables are referenced just by their name (prefixes or sigils (like the '$' sigil) as found in scripting languages are not used).
- The requirement to initialize variables before usage was a lesson learned from C/C++, where uninitialized variable are a nasty source of bugs!
- The rule that effective initialization of local variables is required does also accept conditional assignment:

```
double diameter;
if (weHaveMonday) {
        diameter = 4.56;
} else {
        diameter = 87.933;
}
// Fine! diameter will be effectively initialized!
System.out.println(diameter);
```

Java Syntax Cornerstones – Part 7 – Variables – Local Type Inference

- The type of local variable needs not to be written explicitly in most cases! Instead we can use the var keyword for the type:

```
// Explicitly typed local:
int width = 32;
```

→

```
// Implicitly typed local:
var width = 32;
```

  - The Java compiler infers the type of the local by analyzing the initializer expression. This is called implicit typing or type inference.
    - Here 32 is a literal int, so the compiler infers *width* to be an int variable.
  - The compiler can also infer the type from more complex initializer expressions:

```
var height = 12.5 * 4;
```

    - Here the expression evaluates to type double so the compiler infers *height* to be of type double.

- But implicitly typed variables are still statically typed! I.e we cannot assign a value of another type to *width*:

```
width = 32.5; // Invalid! incompatible type
```

  - Local type inference does not introduce dynamic typing into Java: they do not function like var in JavaScript!

- var cannot be used with compound declarations:

```
var x = 9.3, y = 24.89; // Invalid! java: 'var' is not allowed in a compound declaration
```


- The keyword var is similar to C++11's "additional meaning" of the auto keyword.

Java Syntax Cornerstones – Part 8 – Variables – Local Type Inference

- (Info for Java experts) The compiler is not able to infer all types.
  - It must be an initialization expression (inferred is the type of the expression right from =).
  - The initializer must not be a null literal w/o cast (a typeless expression).
  - The compiler cannot infer the type of lambdas or method references (also typeless expressions).

- Problems with the var keyword in practical usage:
  - It can make code more difficult to read; declaration and usage should be near.
  - Its usability is much depending on the code editor's quality.

Java Syntax Cornerstones – Part 9 – Variables – Local Type Inference

- Pros
  - Implicitly typed variables must be initialized, which may reduce coding errors.
  - It can improve readability, when the inferred type is obvious.
  - Less imports are needed.
  - Code can be modified easier.
- Cons
  - Can be misunderstood by VB/COM/JavaScript developers. (No duck typing here!)
  - If the right hand side expression is complex, developers can't infer the type ;).
- When in doubt mind that code is more often read than written!
  - Usually, implicit typing should not be used while learning Java, thus it'll be avoided in this course.

- Another con: The var keyword hurts the interface segregation principle (after SOLID). – It enforces a style, where on the left hand side of the assignment a non-interface type, but a very concrete type is inferred.
- A further con: confusion. Confusingly the var keyword is another way to express implicitly typed variables in Java (other way: the arguments of generic methods).

Java Syntax Cornerstones – Part 10 – Syntactic Style The free Form

- The Java syntax allows a lot of freedom as far as formatting is concerned.

- Spacing: Java allows to add as many whitespaces around syntactic elements as we want to, the semantics stays the same.
  - A whitespace is usually just a blank space, i.e. the character we get, when we hit the space-key.
  - In a broader sense esp. vertical tab and linefeed are also considered as being whitespace characters.
  - In sounds academic, but it boils down to this: All three statements are equivalent for the compiler, i.e. have the same semantics:

```
System.out.println(3 + 4);
```
≙
```
System    .out  .println( 3 +    4  );
```
≙
```
System
      .out  .println( 3 + 4 ) ;
```
  - Also these statements are all equivalent for the compiler:

```
if (answerIsOk) {
      System.out.println(3 + 4);
}
```
≙
```
if      (   answerIsOk)       { System.out.println(3 + 4); }
```
≙
```
if      (   answerIsOk) {
      System.   out.println   (3 + 4);
}
```
  - We will discuss the meaning of the if-statement and the usage of braces (so called blocks) in short.

Good to know
Whitespaces, esp. spaces are sometimes called "blanks".

- Java is a so called free form language: The same syntax with the same semantics but different formatting.

- Freedom is good! But chaos is not! If each programmer would follow its taste on free formatting, we'll end in chaos!
  - Just compare the snippets above, were I really went crazy on formatting freedom...
  - On the bottom-line, programmers agree upon so-called coding conventions to define rules for free from languages.

Java Syntax Cornerstones – Part 11 – Syntactic Style

- An expression is like a mathematical term: "something, that yields a value".

```
// Expression:
3 + 4
```

- A statement is a set of expressions to take effect, it doesn't need to yield a value.
  - Statements are like phrases, sentences or commands.
  - Individual statements need to be terminated with a semicolon.

```
// Statement:
System.out.println(3 + 4);
```

- A block is a set of statements within curly braces ({}).
  - Java code is written in blocks, this is a main Java style feature.

```
if (answerIsOk) {
        System.out.println(3 + 4);
}
```

- Blocks fringe method and type definitions, scopes and control structures.
  - Control structures must be cascaded to code meaningful programs.
    - I.e. control structure blocks must be cascaded!
  - Cascaded blocks should be indented to enhance readability!
    - We should use common conventions for indenting and bracing!

```
// BSD style:
if (answerIsOk)
{
        // In the block.
}
```

```
// We'll use the 1TBS style:
if (answerIsOk) {
        if (answerIsOk) {
                // In the cascaded
                // if-block.
        }
}
```

- So, blocks are important for control structures.
  - Now we're going to see blocks in action with if/else- and switch-statements.

21

- Blocks:
  - It is absolutely required to indent cascading blocks to make the code readable!
  - Empty blocks should be used instead of empty statements to denote empty control structures:

```
// Empty statement:
// (Not so good.)
if (answerIsOk);
```

→

```
// Empty statement, pair of empty braces:
// (We'll use this syntax in this course.)
if (answerIsOk) {
        // pass
}
```

- Bracing styles:
  - BSD (Berkley Software Distribution) style: should generally be used, when beginning programing, as it has a very good readability.
  - 1TSB style: "Only true bracing style", it is used in this course, as this style saves some lines (it is a "line-saver" style). It is the only style allowing to write correct JavaScript code to avoid mistakes because of semicolon insertion.
  - There exist many more bracing styles, which is often a source of coding wars among programmers, therefor we need coding conventions to cool down the minds.
- Within blocks we can use a free syntax in Java, but please adhere to coding conventions.
  - In Java, blocks are mandatory for do-loops, try-blocks and catch-clauses.

Java Syntax Cornerstones – Part 12 – Sequences

- Now we are going to discuss control structures to write conditional code.

- Java code is executed synchronously, execution is performed from one statement to the next in the order they are written.
  - Synchronous execution means, that one statement needs to complete execution, before the next can start execution.

```
// Writing some names to the console:
System.out.println("Hello Grace!");
System.out.println("Hello Bonnie!");
System.out.println("Hello James!");
```

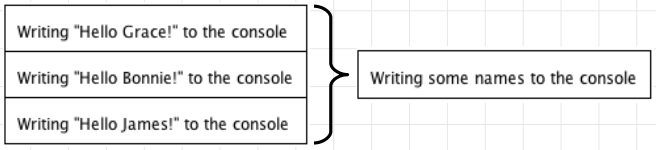
  - Java allows writing multiple statements into a single line, but each statement needs an individual semicolon at its end!

```
System.out.println("Hello Grace!"); System.out.println("Hello Bonnie!"); System.out.println("Hello James!");
```

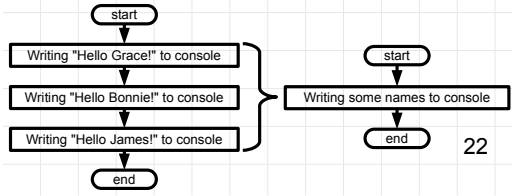
- As a matter of fact, what we see here is the most fundamental control structure: the sequence (of statements).

- Sequences (sequence of statements, or "operations") are written as simple boxes in NSDs and flowchart diagrams.

NSD (sequence)



Flowchart (sequence)



Java Syntax Cornerstones – Part 13 – Conditional Code

- Control structures are expressed as syntactic elements, which define how the logic flow of the algorithm executes.
  - If we do not use any syntactic element to define this flow, we'd end up with the just presented control structure of a sequence.
  - Now its time to discuss the conditional flow/execution of code.

- if statements allow to execute statements or sequences under a certain condition:

```
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
        System.out.println("Hello Grace!");
}
System.out.println("Hello Bonnie!");
System.out.println("Hello James!");
```

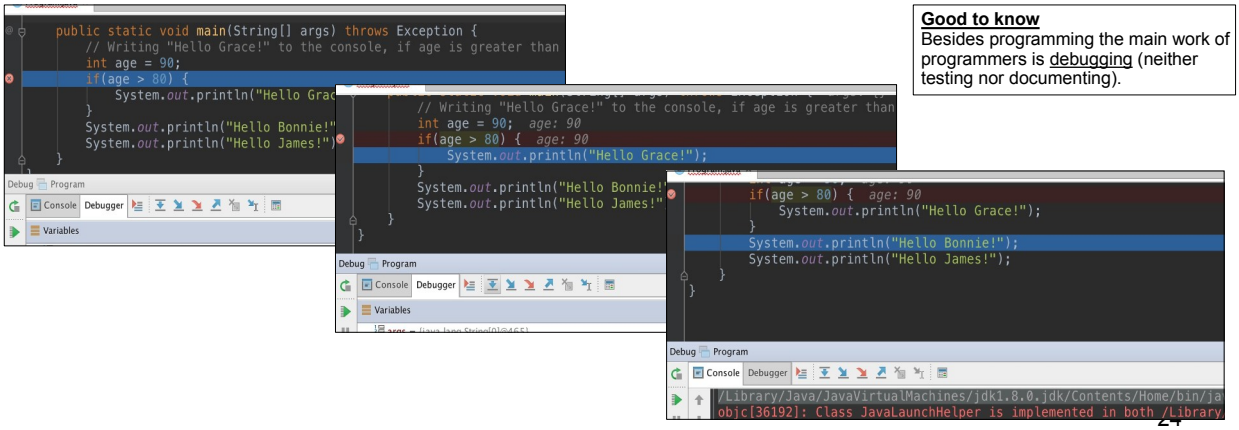
  - An if statement awaits a conditional expression in parentheses (*age* > 80) and a sequence of statements written in a block.
    - The if statement branches the logic of the code, so that the code in the block is executed conditionally.
    - In this case "Hello Grace!" will be printed to the console always, because the value of *age* is 90, which is greater than 80!
    - To distinguish the conditional block in a clear way from the unconditional code, the conditional block's statement (or sequence) is indented.
  - The statements, following the if statement's block will not be executed conditionally.
    - In this case "Hello Bonnie!" and "Hello James!" will be executed always, the value of *age* doesn't matter!
  - An important point to mention here: Java's if statements read very similar to a english prosaic text.
    - Here we clearly see the quality of Java as HLL: the if-syntax exactly matches the semantics of a spoken language!

- What we see here is another fundamental control structure: branching statements.

Java Syntax Cornerstones – Part 14 – Conditional Code

- Java supports two syntactical basic constructs: sequence and cascade (using blocks).

- Most IDEs support to execute a program stepwise among so called breakpoints, this called debugging.
  - This is a very neat way to understand how sequential and cascaded program flow is going forward.



Good to know
Besides programming the main work of programmers is debugging (neither testing nor documenting).

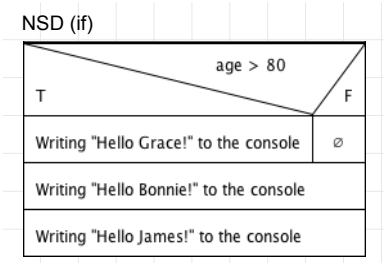
  - During debugging, the line which is selected, is the next line, which will be executed in the program!

Java Syntax Cornerstones – Part 15 – Conditional Code

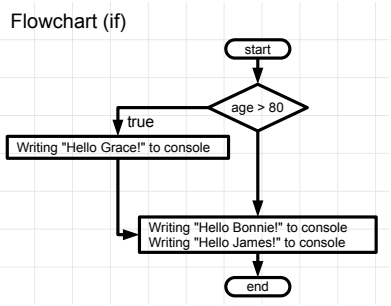
```
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
        System.out.println("Hello Grace!");
}
System.out.println("Hello Bonnie!");
System.out.println("Hello James!");
```

- if statements are written as tip-down triangles in NSDs and as diamonds in statechart diagrams:

NSD (if)



Flowchart (if)



  - These notations show, how the flow of control during program execution is virtually forked following the condition.
  - The "wing" of the decision elements, which the flow of controls follows, if the decision is met is marked with "T"/"true".

Java Syntax Cornerstones – Part 16 – Conditional Code

- In order to widen our understanding of if statements, we'll take a look at cascaded if statements:

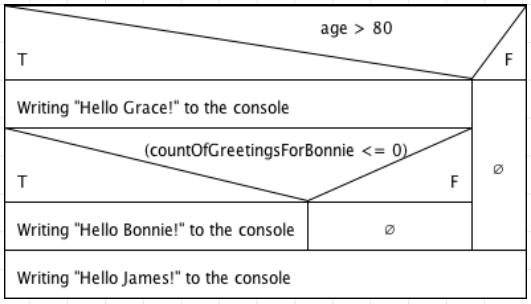
```
int age = 90;
int countOfGreetingsForBonnie = 0;
if (age > 80) {
        System.out.println("Hello Grace!");
        if (countOfGreetingsForBonnie <= 0) {
                System.out.println("Hello Bonnie!");
        }
}
System.out.println("Hello James!");
```

  - The snippet writes "Hello Grace!" if *age* is greater than 80 and then also "Hello Bonnie!" if *countOfGreetingsForBonnie* is less than or equals 0.
  - As can be seen, the cascaded relation of the if statements is expressed by the cascaded blocks and by cascaded block-indentation.
  - In Java we'll have to write a lot of code in this cascading style using cascaded blocks!

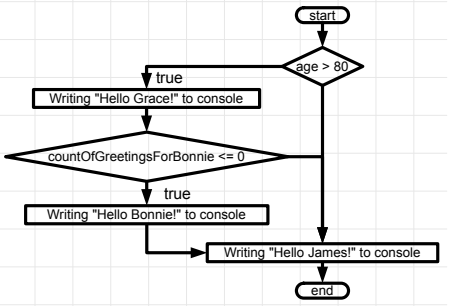
Good to know
In Java we notate the mathematical ≤ condition as "<=" and ≥ as ">=".

- The cascaded if statements are reflected as cascaded tip-down triangles and diamonds in NSDs and statechart diagrams:

NSD (cascaded ifs)



Flowchart (cascaded if)



Java Syntax Cornerstones – Part 17 – Conditional Code

- This code writes "Hello Grace!" to the console only conditionally, but "Hello Bonnie!" and "Hello James!" always:

```
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
	System.out.println("Hello Grace!");
}
System.out.println("Hello Bonnie!");
System.out.println("Hello James!");
```

- If we want to print "Hello Bonnie!", only if *age* is not greater than 80, we could program it like so:

```
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
	System.out.println("Hello Grace!");
}
// Writing "Hello Bonnie!" to the console, if age is not greater than 80:
if (age <= 80) {
	System.out.println("Hello Bonnie!");
}
System.out.println("Hello James!");
```

  - "Hello James!" is always written to the console, because it is unconditional code, i.e. it is not contained in an if-block!

- Well, this solution is somewhat cumbersome, because we have to code the negated ("if not") condition of greater than.

- In programming a conditional pair "if"/"if not" is a very common situation, therefor Java provides if/else statements.

Java Syntax Cornerstones – Part 18 – Conditional Code

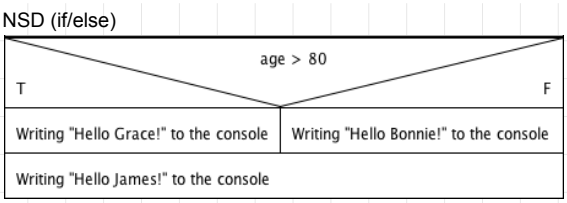
- Now it's time to re-formulate our last conditional code example with if/else statements:

```
// Writing "Hello Grace!" to the console, if age is greater than 80:
int age = 90;
if (age > 80) {
        System.out.println("Hello Grace!");
}
// Writing "Hello Bonnie!" to the console, if age is not greater than 80:
if (age <= 80) {
        System.out.println("Hello Bonnie!");
}
System.out.println("Hello James!");
```

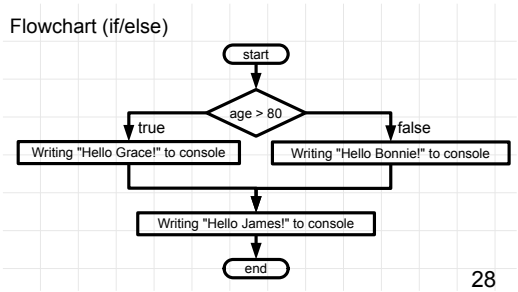
```
// Writing "Hello Grace!" to the console, if age is greater than 80,
// else write "Hello Bonnie!":
int age = 90;
if (age > 80) {
        System.out.println("Hello Grace!");
} else {
        System.out.println("Hello Bonnie!");
}
System.out.println("Hello James!");
```

- Conditional code with if/else statements can also be expressed as NSDs and statechart diagrams:

NSD (if/else)



Flowchart (if/else)



  - This time the "wings" of the decision elements each are explicitly marked with the values "T"/"true" and "F"/"false".

Java Syntax Cornerstones – Part 19 – Conditional Code

- Let's discuss yet another constellation:
  - write "Hello Grace!" if *age* is greater than 80,
  - if not, write "Hello Bonnie!" if *greetingsForBonnie* is less than or equals 0,
  - if not, write "Hello James!".

- Meanwhile we can write code to express this constellation with a set of cascaded if and else statements:

```
int age = 90;
int greetingsForBonnie = 0;
if (age > 80) {
        System.out.println("Hello Grace!");
} else {
        if (greetingsForBonnie <= 0) {
                System.out.println("Hello Bonnie!");
        } else {
                System.out.println("Hello James!");
        }
}
```

- However, there is one new aspect in this cascaded code: we cascaded an if statement in another else block.
  - I.e. no if statement in another if block.

- Such constellations appear so often, that Java programmers usually condense the cascading to if/else if/else.
  - Now we'll discuss how this condensed syntax works.

Java Syntax Cornerstones – Part 20 – Conditional Code

- Here it is. We can condense another if statement within an else block to an else if statement:

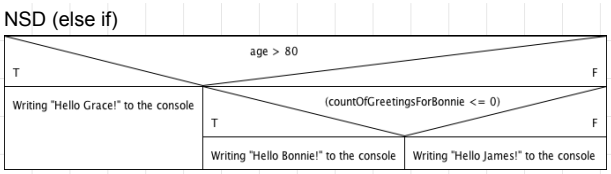
```
int age = 90;
int greetingsForBonnie = 0;
if (age > 80) {
        System.out.println("Hello Grace!");
} else {
        if (greetingsForBonnie <= 0) {
                System.out.println("Hello Bonnie!");
        } else {
                System.out.println("Hello James!");
        }
}
```

→

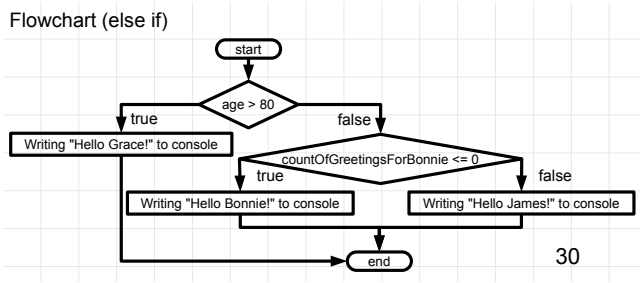
```
int age = 90;
int countOfGreetingsForBonnie = 0;
if (age > 80) {
        System.out.println("Hello Grace!");
} else if (greetingsForBonnie <= 0) {
        System.out.println("Hello Bonnie!");
} else {
        System.out.println("Hello James!");
}
```

- Conditional code with else if statements can also be expressed as NSDs and statechart diagrams with cascading forks:

NSD (else if)



Flowchart (else if)



Java Syntax Cornerstones – Part 21 – Conditional Code

- For some control structures, e.g. if/else, Java allows to leave braces away, if a belonging to block has only one statement.

```
// (1) meets our conventions: recommendable!
if (age > 80) {
        System.out.println("Hello Grace!");
} else if (greetingsForBonnie <= 0) {
        System.out.println("Hello Bonnie!");
} else {
        System.out.println("Hello James!");
}
```

≙

```
// (2) omitted braces: not recommendable!
if (age > 80)
        System.out.println("Hello Grace!");
else if (greetingsForBonnie <= 0)
        System.out.println("Hello Bonnie!");
else
        System.out.println("Hello James!");
```

≙

```
// (3) omitted braces and line breaks: not recommendable!
if (age > 80) System.out.println("Hello Grace!");
else if (greetingsForBonnie <= 0) System.out.println("Hello Bonnie!");
else System.out.println("Hello James!");
```

- A word of warning from the trenches: Generally use braces/blocks, even if it is not required (only one statement in a block)!
  - Bracing allows the best readability, while omitting braces is very error prone! Also experienced programmers can fail!
  - Many coding conventions explicitly forbid to leave braces away!
  - => In this course: We'll generally use braces!

- There is one exception: this recommendable formatting of if/else plus else if does already omit braces actually:

```
// (1) meets our conventions, recommendable!
if (age > 80) {
        System.out.println("Hello Grace!");
} else {
        if (greetingsForBonnie <= 0) {
                System.out.println("Hello Bonnie!");
        } else {
                System.out.println("Hello James!");
        }
}
```

≙

```
// (2) else if, maybe better readability, also recommendable!
if (age > 80) {
        System.out.println("Hello Grace!");
} else if (greetingsForBonnie <= 0) {
        System.out.println("Hello Bonnie!");
} else {
        System.out.println("Hello James!");
}
```


- According to if/else: We should always use blocks! Matching if/else pairs are clear to the reader when we use blocks. This avoids the "dangling else" problem.

```
if (a == 1)
        if (b == 1)
                a = 42;
else
        b = 42;
```

  - Some developers think, that the else belongs to *if (a == 1)*, but this is not the case! – It belongs to its nearest if (i.e. *if (b == 1)*). Consequent usage of braces and indentation avoids the dangling else "optical illusion".
- The programming language Perl explicitly disallows leaving away braces on any control flow statement. – This is a bit strange restriction in Perl, because Perl allows a lot of (error prone) freedom in other places of the language.

Java Syntax Cornerstones – Part 22 – Conditional Code

- Assume this constellation:
  - if *billsAge* is less than or equal to 10, Bill will get a birthday gift worth 50€, else he'll get a birthday gift worth 75€.
  - No problem with if/else:

```
int billsAge = 12;
int birthdayGiftPrize = 0;
if (billsAge <= 10) {
        birthdayGiftPrize = 50;
} else {
        birthdayGiftPrize = 75;
}
```

- There is a clever way to abbreviate such a if/else statement with a compact expression, the conditional expression:

```
int billsAge = 12;
int birthdayGiftPrize = (billsAge <= 10) ? 50 : 75; // (1)
```

  - When statement (1) is executed and *billsAge* is less than or equal to 10 *birthdayGiftPrize* is initialized to 50, otherwise to 75.

- Conditional expressions apply the conditional operator **?:**.

```
// Conditional expression:
condition ? expression1 : expression2;
```

  - The conditional operator is the only ternary operator, i.e. it accepts three arguments.
  - The arguments are *condition*, *expression1* and *expression2*.
  - If *condition* evaluates to true *expression1* is evaluated, else *expression2*.

- The operator ?: is more difficult to debug than if/else, because it does not consist of alternative statements to be executed.

Java Syntax Cornerstones – Part 23 – Conditional Code

- Conditional expressions allow writing conditional code without statements, i.e. without blocks and cascading.
  - But, sometimes programmers want the indented style of if/else with the conditional operator. So, let's just reformat the code:

```
int birthdayGiftPrize = (billsAge <= 10) ? 50 : 75;
```

→

```
int birthdayGiftPrize = (billsAge <= 10)
                        ? 50
                        : 75;
```

- In opposite to if-statements, it is not required to put parentheses around the condition:

```
int birthdayGiftPrize = (billsAge <= 10) ? 50 : 75;
```

→

```
int birthdayGiftPrize = billsAge <= 10 ? 50 : 75;
```

  - But in most cases, parentheses are not bad and should be kept for clarity.

Java Syntax Cornerstones – Part 24 – Conditional Code

- Now let's assume, that we have to deal with more variants of ages and prizes for birthday gifts:
  - (1) If the person's age is 10 years, the gift may cost 50€ and for an age of 15 years it may cost 75€.
  - (2) For all other ages, the gift may only cost 25€. We can code this with a simple piece of conditional code:

```
int personsAge = 10;
int birthdayGiftPrize = 0;
if (personsAge == 10) {
        birthdayGiftPrize = 50;
} else if (personsAge == 15) {
        birthdayGiftPrize = 75;
} else {
        birthdayGiftPrize = 25;
}
```

  - The specialty of this code: the comparison is only based on the equality of int constants.
  - In Java we use the comparison operator "==" to express equality expressions, not "="!

- Alternatively we can use the switch statement to execute statements conditionally on mutually exclusive int constants:

```
switch (personsAge) {      // Switch over the personsAge variable ...
        case 10:           // against the constant 10 or ...
                birthdayGiftPrize = 50;
                break;
        case 15:           // the constant 15 or ...
                birthdayGiftPrize = 75;
                break;
        default:           // all other ages.
                birthdayGiftPrize = 25;
}
```

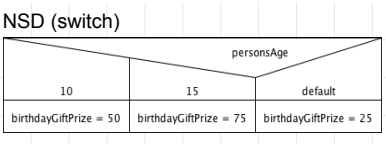

- According to switch:
  - switch statements are very popular in Win32 programming. Esp. traditionally programmed Win32 message loops make excessive use of switch statements.
  - (Opinion of [NLu]: Don't use switch! It looks temptingly simple, but is rather unstructured and can introduce goofy bugs. Also it leads to longer function bodies. It should be used only in the most low level code.)

Java Syntax Cornerstones – Part 25 – Conditional Code

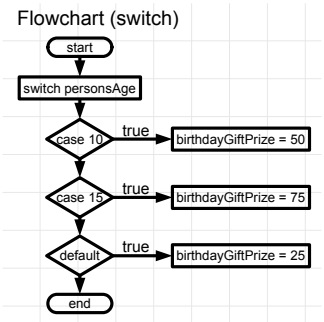
- Before we continue discussing the switch statement, let's show its NSD and flowchart representation.

```
switch (personsAge) {  // Switch over the personsAge variable ...
      case 10:              // against the constant 10 or ...
            birthdayGiftPrize = 50;
            break;
      case 15:              // the constant 15 or ...
            birthdayGiftPrize = 75;
            break;
      default:              // all other ages.
            birthdayGiftPrize = 25;
}
```

NSD (switch)



Flowchart (switch)



Java Syntax Cornerstones – Part 26 – Conditional Code

- The readability of switch statements is really good and the mechanics should be clear as well, but is has downsides.

```
switch (personsAge) {  // Switch over the personsAge variable ...
        case 10:               // against the constant 10 or ...
                birthdayGiftPrize = 50;
                break;
        case 15:               // the constant 15 or ...
                birthdayGiftPrize = 75;
                break;
        default:               // all other ages.
                birthdayGiftPrize = 25;
}
```

- The switch statement
  - does switch among a set of branched code blocks,
  - equality compares the statement's input, which is passed as parameter to switch (*personsAge*), against a set of constants,
  - and jumps to a labeled case section, often called case-label, if the evaluated switch expression matches to the case constant.
  - the (optional) break statement limits the conditional code belonging to case-label, similar to an if's block.
  - If none of the case-labels met, the code after the optional default-label will be executed.
  - The order of case-labels doesn't matter, because the candidate will only match one of them.
  - The switch statement works with int values, *String* values and enum values (*String*s will be discussed in short).

- In fact, switch looks temptingly simple, but one can introduce goofy errors, because of switch's dangerous parts.
  - We'll discuss these parts now.

Java Syntax Cornerstones – Part 27 – Conditional Code

- Let's assume another variant of the *birthdayGiftPrize* calculation: if the person gets 10 or 15 years old it may cost 75€.
  - We can modify the switch statement to a switch statement with case-fall through to cover this variant:

```
switch (personsAge) {  // Switch over the personsAge variable ...
      case 10:             // against the constant 10 or ...
            birthdayGiftPrize = 50;
            break;
      case 15:             // the constant 15 or ...
            birthdayGiftPrize = 75;
            break;
      default:             // all other ages.
            birthdayGiftPrize = 25;
}
```

→

```
switch (personsAge) {  // Switch over the personsAge variable ...
      case 10:             // against the constant 10 or ...
      case 15:             // the constant 15 or ...
            birthdayGiftPrize = 75;
            break;
      default:             // all other ages.
            birthdayGiftPrize = 25;
}
```

- Problems with switch: actually it is not simple at all and the readability is also not so good! – switch is a noisy statement.
  - switch statements with case-fall through are potentially dangerous:
    - Fall throughs are introduced by just leaving away the break statement, which formally belonged to a specific case statement.
    - Where the break statement is left away, case-labelled code will fall through. I.e. a break statement then belongs to more than one case.
    - From another perspective, if one unintentionally forgets to write the break statement, the control flow will surprisingly fall through!
    - This is a common source of bugs in Java and C/C++.
  - Other problems with switch statements:
    - Actually they are more complicated than if/else: they use case, break and default statements and allow application of fall throughs.
    - Inflexible: (1) Only equality comparison is supported, i.e, no other conditional expressions. (2) Only int-, *String*- and enum-values can be handled.

- => The switch statement will be avoided in the examples used in this course!

- Changing code from using switch to if/else can be error prone.
- Good, when used with only returns instead of any breaks in the case sections.
- Future Java versions introduce some enhancements to switch:
  - Simple variants of switch work without break.
  - A way to specify multiple labels for a specific case.
  - Additionally the switch expression was introduced. – The switch expression relates to the switch statement like ?: relates to the if/else statement.
  - Maybe, even more advanced switch-feature might allow more complex expressions than just comparison against constants – a feature called pattern matching. With pattern matching the order of case-"labels" will matter, because the candidate could match to more than only one of them and are no longer mutually exclusive.
    - By default, the order of case-labels in switch doesn't matter, they are mutually exclusive.

Java Syntax Cornerstones – Part 28 – Scopes

- The region, in which a variable has a meaning and can be used is called scope.
  - Java limits the scope of a variable by the positioning in the code and by blocks.
- Example: the scope of a variable in an if-statement's block:

y's scope

```
int x = 78; // Define x in the outer scope.
if (x == 78) {
        System.out.print(x); // Fine! x is defined before if's scope/block, i.e. in the outer scope.
        int y = 56; // Define y in the inner scope.
        System.out.print(y);  // Fine! y is defined in if's scope/block, i.e. in the inner scope.
}
System.out.print(y); // Invalid! y is not known/has no meaning in the outer scope, it was defined in if's scope!
```

- Example: the scope of a variable in a scope-limiting block:

y's scope

```
int x = 78; // Define x in the outer scope.
{
        System.out.print(x); // Fine! x is defined before the scope-limit block, i.e. in the outer scope.
        int y = 56; // Define y in the inner scope.
        System.out.print(y);  // Fine! y is defined in the scope-limit block, i.e. in the inner scope.
}
System.out.print(y); // Invalid! y is has no meaning in the outer scope, it was defined in the scope-limit block!
```

- Example: the definition of a variable must be unique in a scope, if not we get a name clash, resulting in a compile time error:

```
if (x == 78) {
        int y = 23;
        int y = 50; // Invalid! variable y is already defined …
}
```

```
if (x == 78) {
        int y = 23;
        if (z == 8) {
                int y = 50; // Invalid! variable y is already defined …
        }
}
```

38

- What is a scope?
  - A scope defines an area of code, in which a variable has certain meaning.
  - Other example: The scope of a variable in JavaScript is the scope of the function, in which the variable is defined. – There are no other sub scopes (e.g. curly braces). Instead JavaScript's variables are said to be hoisted from a nested scope to the function's scope.

Java Syntax Cornerstones – Part 29 – Types

- Values, e.g. those held in variables, have types: All values in Java have a certain type! This is a profound concept of Java.

- Primitive types are types, which are "built into" Java, whose values can be represented with literals.
  - We already know the primitive type int. int variables can be created with integer literals:

```
int intValue = 23;
```

  - Java's primitive types have own keywords: int, double, boolean etc.
  - We'll learn Java's primitive types in short.

- Reference types are types, that refer to non-primitive complex types (a topic, we'll discuss in future lectures):
  - Complex types are just types, which are built from other types.
  - 1. Arrays of primitive and other reference types.
  - 2. User defined types (UDTs): classes, interfaces, enums and @interfaces
  - (Complex type is no official term from the Java language specification, instead all non-primitive types are called reference types.)

- What are primitive types?
  - These types are integrated into Java. Most of them are represented by keywords, the prominent exception is the type *String*.
  - What is a literal?
    - A literal is a value of specific type that can be written out in source code directly.
    - In a sense a literal is the opposite of a symbol.

Java Syntax Cornerstones – Part 30 – Identifiers

- We already mentioned variables
  - Java's variables are used like those in algebra, but if we use only short names like 'x' and 'y' we'll run out of variable names soon
  - We'll have to deal with the more variables, the bigger a program gets. Therefor it is needed to give variables meaningful names.

- Names of variables are also called identifiers of variables.
  - Identifiers are case sensitive: *aValue* is not the same identifier as *aVaLuE*!
  - Java keywords mustn't be used as identifiers.
  - Special characters (except _ (underscore)) are not allowed in identifiers!
  - The $ (dollar sign) is allowed as first character of an identifier.
  - Digits are not allowed as first character.
  - Identifiers may contain all other characters, which are defined by unicode, also e.g. umlauts and ideograms.

Good to know
The _ (underscore) is sometimes also called "underbar" or just "under" among developers.

```
// Valid Java identifiers:
int $n = 23;          // with a dollar sign (first character)
int count_ = 58;      // with an underscore
int      = 566;       // with a Chinese ideogram
int nKäufer = 2;      // with an umlaut
```

```
// Invalid Java identifiers:
int n*c = 23;         // invalid special character * used
int 2gone = 58;       // starts with a digit
int void = 17;        // Java keyword cannot be identifier
```

  - The identifiers of variables must be unique in the same scope:

```
if (x == 78) {
      int y = 23;
      int y = 50; // Invalid! variable x is already defined …
}
```

    - Yes, we already mentioned this, when we discussed "scopes".

40

- All identifiers are case sensitive.
  - What's an identifier?
  - Use PascalCase for types.
  - private, protected, public and local identifiers should be written in camelCase.
  - For identifiers of variables, there also exist the Hungarian notation (HN). HN mandates to put a specific type-dependent prefix in front of a variable name, f for float (farea), i for int and (e.g. iage) sz (e.g. szname) for "string zero terminated". HN is called Hungarian notation, because its inventor, Charles Simonyis is Hungarian. HN is used when programming the Win32 SDK, but esp. the .NET Framework Design Guidelines prohibits using HN for other (non-Microsoft) developers (however, the guidelines make no statement about private fields). The problem using HN is, that the name of such variables must be changed every time their types change, and when developing oo, changing of a variable's type is quite common. But HN can be useful:
    - It could make sense, when using a dynamically typed language (but I doubt this).
    - With Win32 programming it makes sense, because a lot of handle-types and constants are just ints with a typedef, whose semantics might not be understood by functions (it is just an int…). HN can help here, because one can spot passing wrong arguments at least visually, if variables carry prefixes to tell handle-types from ints.
  - For compile time constants, we should use SCREAMING_SNAKE_CASE, sometimes also called MACRO_CASE, is a variant of snake_case. – We will discuss constants in short.
- Concerning the underscore:
  - With Java 9 the underscore as single character cannot be used as valid standalone identifier.
  - Nevertheless in Java it is highly discouraged to use underscores in identifiers of non-constant symbols or as prefix for private fields.
- What makes up a variable?
  - A name, a type and a value.
  - Variable names should not be short names like in maths, rather use meaningful names. – Alas in the examples of this course often short variable names will be used. In maths, variables often apply identifiers following some rules:
    - Most identifies are reduced to only one letter of the alphabet.
    - The first letters of the alphabet are often used to name known values. E.g. *a*, *b*, *c* and *d* to name the sides of a 4-polygon.
    - The last letters of the alphabet are often used to name unknown values. E.g. *x*, *y* and *z* to name parameters of a function.
    - The letters "in between" are often used to name natural numbers, esp. *n* and *m*, but also *i* and *j* to name indexes.
    - Angles are named using greek lower case letters.
- Definition and initialization:
  - What's that?
  - A definition reserves memory for a value of a symbol; a definition of a symbol mustn't be repeated in code!
  - What is a symbol?
    - Symbols are identifiers with a special meaning in code (e.g. identifiers of variables, constants or functions).
  - An assignment sets the value of a symbol of a variable; assignments of variables can be repeated in the code.
  - An initialization combines the definition of a symbol with the assignment of a value. An initialization of a symbol mustn't be repeated in the code!

Java Syntax Cornerstones – Part 31 – Identifiers

- Because of all the freedom we have with identifier-naming, we need to tame the freedom somewhat with some conventions:
  - Convention, or coding convention means, that we've to agree upon a common notation for the identifiers we introduce in programs.
  - The most important identifiers in Java are those for variables, methods and type names.

- For variable names, we use *camelCase* for naming as convention! Camelcase means:
  - (1) names of variables consist of full words
  - (2) if the name consists of more than one word, all words are jammed together and each word starts with an upper case letter, but
  - (3) the very first word is always written in lower case.
  - (4) No separators like '_' are allowed.

```
// Examples for the camelCase notation:
int age = 90;
int birthdayGiftPrize = 75;
int countOfGreetingsForBonnie = 0;
```

Java Syntax Cornerstones – Part 32 – Comments

- In most of the examples we have already used comments.
- Comments allow to write all kinds of human readable annotations into the code, without introducing syntax errors.
  - Comments can be applied everywhere in code. They are simple to spot in our examples, because they're highlighted in green.
  - But besides the highlighting, there is of course a syntax concept behind comments in Java, there are even two sorts of comments:
    - (1) Single line comments, which make text from the beginning of "//" until the end of the line a comment:

```
// comment
System.out.println(r * r * 3.14); // Explanation: Prints the area of the circle to the console.
```

    - (2) Comment brackets "/*" and "*/", which makes all enclosing text, even if spanning multiple lines, a comment:

```
/* comment
 Explanation: Prints the area of the circle to the console:*/ System.out.println(r * r * 3.14);
```

- Esp. when beginning programming, we should use comments to be able to understand code when looking at it after a while!
  - E.g. the explanation of the console output in the examples above, explaining the formula, can prove useful.
- Technically, comments are ignored by the compiler. The compiler handles comments like whitespaces.
  - "Ignored" means, that the compiler will leave comments in the code (the compiler never modifies the code), but ignores its content.
  - => Seen from another perspective, this means, that valid Java code can be "deactivated" by making it a comment:

```
// This statement will not be compiled and thus also never be executed:
//System.out.println(r * r * 3.14);
```


- Esp. programming newbies should use comments very often to remember what the code does. Later in the life as programmer, produced code should document itself.
- Multiline comments are often called C-style comments, whereas single line comments are often called C++-style comments (they were introduced with C++).
- Multiline comments act like a set of parentheses that enclose the commented text. Comments will be converted into whitespaces before the preprocessing phase starts.
- Prefer: Single line comments concern only one line.
- Avoid multiline comments as they can not be cascaded. The first occurrence of /* is always matched with the first occurrence of */.
- Virtually, comments are not that good, because they never change!
  - Writing good comments is difficult, a comment should not just repeat what the code in a verbose manner.
  - We as programmers have to change comments along with changes in the code.
  - Better than comments is self describing code!
  - Also commented code can be a problem for refactoring. Because in most cases commented identifiers are not refactored, when identifiers are renamed via a refactoring tool.

Constants – Part 1

- Sometimes, we have to deal with the same value over and over again!

- 1. We can use literal values, e.g. the value 3.14 (pi) calculating a circle's area:

```
double a = r * r * 3.14;
```

  - Here we can't use int values/variables, because pi is a floating point number! Floating point numbers are of type double in Java!
    - So, 3.14 is double literal, whereby the '.' represents the decimal point (not the ',', as we find it in, e.g., Germany).
  - Will we remember the meaning of the literal 3.14 in that expression/formula? Such literals are called magic numbers.
  - Magic numbers should be avoided, as we could forget their meaning and other readers may not understand their meaning!

- 2. We can use a variable for pi to give the value's variable a memorable name/identifier:

```
// The double variable PI:
double PI = 3.14;
```

```
// Better memorable, eh?
double a = r * r * PI;
```

  - But we can not prevent programmers from assigning to variables, replacing the value 3.14 with something different!

```
PI = 4; // Ouch! Changes the meaning of PI!
```

- But we can solve these problems with the introduction of constants.

Constants – Part 2

- 3. In Java we define constants as variables with the final keyword. Java's constants are just called final variables.

```
// Constant double (compile time constant) PI:
final double PI = 3.14;
```

```
// This line remains valid:
double a = r * r * PI;
```

  - Such objects that can not be modified are called constants.

```
PI = 4; // Invalid! PI is a constant, not a variable!
```

Good to know

```
// In Java we do not need to define PI explicitly. The constant
// PI is already defined in the class Math:
double a = r * r * Math.PI;
```

- Constants prevent us from doing coding errors and provide self documentation.
  - Constants are like variables, which cannot be assigned to more than once, they replace magic numbers perfectly.

Good to know

const is a reserved word in Java, but it is not a keyword with a meaning, esp. it cannot be used as identifier. Maybe there is a future use for const...

- The constant *PI* is a so called compile time constant. A … What?
  - Well, the constant value of *PI* is known to the compiler.
    - => *PI* was initialized with the literal 3.14, and the value of that literal is known at compile time.
  - The value of a compile time constant must be explicitly set exactly once in its lifetime, if not we'll get a compile time error.
    - => After *PI* was set to 3.14, it can never be set to another value.
  - For compile time constants, we use *SCREAMING_SNAKE_CASE* for naming as convention, e.g.:

```
// Approximate speed of light in km/s in vacuum:
final double SPEED_OF_LIGHT = 300000.0;
```

```
// Type inference also works for constants:
final var SPEED_OF_LIGHT = 300000.0;
```

Constants – Part 3

- We can also define run time constants in Java.
  - Those are constants, whose values are "fixed" at run time, i.e. not already at compile time.
    - (Mind, compile time happens before run time.)
  - A run time constant is defined, by using a final variable as before, but initializing it with a value, which is not known at compile time.
  - It sounds complicated, but is really easy to get:

```
// rectangleArea as run time constant:
final double rectangleArea  = a * b;
```

    - In this statement, the variables *a* and *b* have no constant values, maybe they were input by the user.

- Sometimes (compile time) constants are called symbolic constants to separate the wording from literals.

```
// 3.14 is just a literal (constant) here:
double a = r * r * 3.14;
```

```
// PI is now a symbolic constant:
double a = r * r * PI;
```

- Effective initialization
  - final variables need to be effectively initialized by initialization or assignment before usage:

```
final double diameter;
// Invalid! Variable diameter might not have been initialized
System.out.println(diameter);
```

```
final double diameter;
diameter = 4.56;
// Fine! diameter will be effectively initialized!
System.out.println(diameter);
```

Good to know

```
// We can also use if/else statements
// for the initialization for constants:
final double diameter;
if (weHaveMonday) {
        diameter = 4.56
} else {
        diameter = 87.933;
} // … the final must just be eventually
  // initialized!
```

- Another way to define constants are enumerations (enums), which we will discuss in a future lecture.

Java's Primitive Integral Datatypes

- int, long, short, byte and char

```
// Definition and initialization of an int:
int numberOfEntries = 16;
```

Good to know
The term "integer" is for "value of integrity", i.e. a non-divisible value.

- int, long, short, byte are signed in Java, i.e. its values can have positive and negative values.
  - These types use the two's complement representation in memory.
    - So the range of the value domain is broken into a negative and a positive wing.
    - The 0 (zero) counts as positive value.
  - The only "unsigned" integral type in Java is char.
    - Java isn't suited for hardware-near programming, because larger unsigned types would be needed.

Good to know
The 0 is sometimes written as Ø (slashed zero) or 0̟ (zero with point). In earlier days of computing it was required to tell the letter O from the digit 0, but the display and printer resolution was not very high and fonts were monospaced, therefor slashed zero and zero with point were introduced.

- The default value of uninitialized integral variables (fields) is always 0!

- Literals and sizes:
  - int {42, -0x2A, 052, 0b101010, 1_200, default value: 0}; size: 4B
  - char {'A', 65, 0x41, '\u0041', 0101, 0b10001, default value: '\u000'}; size: 2B
  - byte {'A', 65, 0x41, '\u0041', 0101, 0b10001, default value: 0}; size: 1B
  - short {'A', 65, 0x41, '\u0041', 0101, 0b10001, 1_200, default value: 0}; size: 2B
  - long {42L, -0x2al, 052L, 0b10001, 0b10001L, 1_200L, default value: 0L}; size: 8B

Good to know
The bits and bytes a certain type occupies can be found as fields of Java's wrapper types for primitive types: e.g. *Integer.SIZE* and *Integer.BYTES*.
Java's runtime libraries provide a set of useful constants to represent the minimum and maximum values of types:
for int: *Integer.MIN_VALUE*, *Integer.MAX_VALUE*,
for char: *Character.MIN_VALUE*, *Character.MAX_VALUE*,
for short: *Short.MIN_VALUE*, *Short.MAX_VALUE*,
and for long: *Long.MIN_VALUE*, *Long.MAX_VALUE*.

- Integral data seems to be the most basic and important data type in programming. – Most basic programming lectures usually start using integral data.
- What are "integral" types?
- What is the two's complement?
  - Negative ints could be represented with the one's complement. The one's complement of a positive number is just the inversion of all the bits, which represent the number. – The problem with this approach is, that we'll end up with two representations of the 0, one for -0 and one for +0. To overcome this problem, we just add 1 to the one's complement, which makes the sign of the number clear, the result os the two's complement. – We will discuss this in more depth in a future lecture.
- We should always use int as our default integral type.
- There exist 3rd party libraries (joou) that add support for unsigned types into Java.
- In literals, underscores can be used in any place and also multiple underscores can be written in direct sequence.
- Literals of type long should always be written with an upper case L to avoid "optical illusions", because a lower case l could be confused with the digit 1 depending on the editor's font.
- Big numbers, such as durations in milliseconds are often specified as long values.

Java's boolean Type

- When an expression evaluates to a truth value in Java, its result can be stored in a variable of type boolean.
  - E.g. the result of a conditional expression used in an if statement can be extracted like so:

```
// An if statement as we know it:
if (age >= 18) {
        // pass
}
```

```
// An if statement having the conditional expression extracted:
boolean isAdult = age >= 18;
if (isAdult) {
        // pass
}
```

  - In opposite to, e.g., C/C++, numeric results like int do not evaluate to boolean:

```
// No, this will not work/compile:
boolean truthValueA = 1; // int cannot be converted to boolean
boolean truthValueB = 0; // int cannot be converted to boolean
```

- The default value of uninitialized boolean variables (fields) is always false!

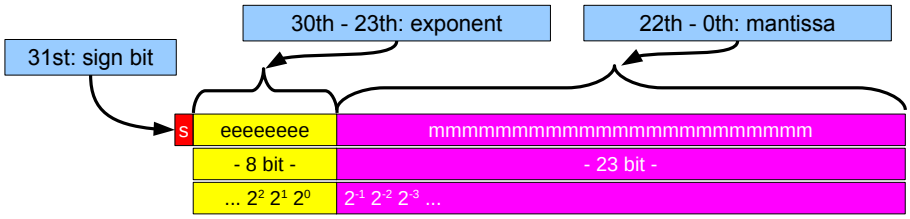
- Literals and sizes:
  - boolean {true, false, default value: false}; size: not precisely defined

Java Primitive Floating Point Datatypes

- double and float

```
// Definition and initialization of a double:
double approxDistance = 35.25;
```

- Those types are signed, but w/o two's complement representation. They use IEEE 754 (here single (float)) representation:



- The default value of uninitialized floaty variables (fields) is always 0.0!
  - Mind, that the '.' represents the decimal point (not the ',', as we find it in, e.g., Germany).

- Literals and sizes:
  - double {3.14, 12.3D, 12.3d, .5 (0.5) , 10. (10.0), 1_200.3_9F, 123E-1, default value: 0.0}; size: 8B
  - float {3.14F, 12.3f, 10.f, 1_200.3_9F, 123E-1f, default value: 0.0f}; size: 4B
  - Hexadecimal, octal and binary literals cannot be used in floating point literals!

Good to know
Java's runtime libraries provide a set of useful constants to represent the minimum and maximum values of types. For double: *Double.MIN_VALUE*, *Double.MIN_VALUE*, *Double.MAX_VALUE*, for float: *Float.MIN_VALUE*, *Float.MAX_VALUE*, *Float.MAX_EXPONENT*.

- What does the term "double" exactly mean?
  - The double precision of float, the Java type float has single precision.
- A double can represent bigger and more precise numbers than float.
- Floating point types try to display rational numbers of various precision.
- Institute of Electrical and Electronics Engineers (IEEE, pronounced as "i triple e").
- In literals, underscores can be used in any place and also multiple underscores can be written in direct sequence.
- We should always use double as our default floating point type!
  - The hardware (FPU) is typically optimized for doubles.
  - APIs for embedded systems do often not support double precision. E.g. Open GL ES (ES for Embedded Systems) provides only float in its interfaces.
- We have to be aware that we have to use "." instead of "," in the floating point literals! -> Therefore these datatypes are called floating point datatypes!

Implicit Type Conversions

- Type conversion: Under certain circumstances a value of a specific type can be used to create a value of another type.
- Java supports standard conversions, which are done implicitly, when values are assigned (or passed to methods).
  - Implicit conversions can be done from a smaller to a larger integral type and from a less precise to a more precise floaty type.

```
// Some standard integral (implicit) conversions:
char c = 'k';
// c = 'k'
int i = c;
// i = 107
long l = i;
// l = 107L
```

```
// Some standard floaty (implicit) conversions:
float f = 16.74F;
// f = 16.74F
double d = f;
// d = 16.74
```

- Standard conversions work also from integral to floaty types:

```
// Some standard conversions from integral to floaty types:
long ll = 16_746_395L;
// ll = 16_746_395L
float ff = ll ;
// ll = 16_746_395.0F
double dd = ll ;
// dd = 16_746_395.0
```

- The idea of standard conversion simply means, that conversions from "smaller" types to "larger" types are possible.

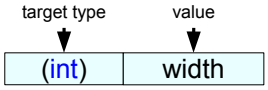
Explicit Type Conversions

- What about conversion from a larger to a smaller type?
  - Java supports such conversions as well, but those must be done explicitly.
- Java forces us as programmers to write the desired conversion into the code explicitly.

```
// Some explicit conversions:
int i = 107;
// i = 107
char c = (char)i; // using the cast operator
// c = 'k'
long l = Long.MAX_VALUE;
// l = 9223372036854775807L
int ii = (int)l;
// ii = -1 // Overflow, this long value is too large for int!
```

```
// Some explicit conversions:
float f = 16.74F;
// f = 16.74F
int i = (int)f;
// i = 16
double d = f;
// d = 16.739999771118164
int ii = (int)d;
// ii = 16
float ff = (float)d;
// f = 16.74F
```

- We're using a new syntax here featuring the cast operator.



- It makes a lot of sense that Java forces these conversions to be done explicitly with the cast operator syntax.
  - Some conversions, e.g. from long to int can have strange results. This effect results from overflow effects.
  - Conversions from floaty to integral type lead to clipping digits right from the decimal point! So, no rounding is taking place!
  - => All these conversions can be lossy, a programmer needs to know what she does, therefor they have to be done explicitly!

- Between reference types and value types, so called boxing conversions can be done. – This is a topic for future lectures.
- Overflow effects are discussed in the C++ lessons.

Integral Division and Division by 0

- Multiplication of integers is no problem: the largest type is the resulting type.
- The division of integers does not have a floaty result!

```
int x1 = 5;
int x2 = 3;
double result = x1 / x2; // result will just contain an int value expressed as double
// result = 1.0
```

  - The result's value is of the type of the largest contributed integer.
  - Even if the result variable's declared type is floaty the result is integral (*result* is declared to be a double above).
  - Also the result won't be rounded, instead the places after the period will be clipped!
- To correct it we must explicitly convert any of the integral operands to a floaty type with the cast operator:

```
int x1 = 5;
int x2 = 3;
double result = (double)x1 / x2;
// result = 1.6666666666666667
```

- Division by 0:
  - Floaty values divided by 0 result in "infinity".
  - Integral values divided by 0 result in an *ArithmeticException* "/ by zero".
  - => We should always check the divisor for being not 0 in our code before dividing.

```
double result = 5.0 / 0;
// result = Double.POSITIVE_INFINITY
int result2 = 5 / 0;
// Will throw ArithmeticException "/ by zero"
```


- The integral result of the integral division makes sense, because how should the integral division know something about floaty types?
- Clipping: the result of 5 : 2 is $1.\overline{6}$, but the part after the period will be clipped away.
- The division by 0 is not "not allowed" in maths, instead it is "just" undefined.

Textual Data: Strings – Part 1

- In some examples we already saw the application of text data, e.g. to write text to the command line.
- The type, which represents textual data in Java is *String*, more specifically the class *String*.
  - E.g. let's define the variable *hello* of type *String*, which has the value "Hello, World!".

```
String hello = "Hello, World!";
```

  - As can be seen, the type *String* isn't written in blue color. This is because *String* is a reference type!
- The concept of using *String*s is very important in Java. Dealing with textual data is generally very important in programming.
- *String* literals and values:
  - *String* literals have to be written as a text enclosed in double quotes!
    - The examples of this course set *String* literals in brown color.
  - A *String* value can hold any unicode-character, e.g. also umlauts and ideograms.

```
String withUnicodeCharacters = "数nKäufer";
```

    - We'll not discuss the meaning of unicode in this course, just keep in mind, that *String*s can hold data of any spoken language basically.
  - Comments within *String*-literals become part of the *String*-literals.

```
String otherString = "Hello, World! // comment";
```

  - *String* values can have 2^{31} - 1 chars at maximum.
  - But a *String* literal can only have up to 2^{16} - 1 letters!

52

- Concerning the term "string" mind the German term "Zeichenkette", which means "string of characters".
- Another difference between primitive types having keywords and *String*, is that *String* is a reference type and the other primitive types are value types. We'll discuss reference and value types in a future lecture.
- 2^{31}-1 is about 2 billions
- 2^{16}-1 is about 65k

Textual Data: Strings – Part 2

- Initialization of *String* variables:
  - We use = to initialize *String* variables, as with other types.
  - The default value of uninitialized *String* variables is null!
  - We also use = to assign *String* variables, as with other types.

```
String name = "Arthur";   // initialization of name
String otherName;         // the uninitialized otherName has the value null
otherName = "Jamie";      // assigning a value
```

```
String otherName = null; // explicit initialization to null
```

- Wait! What is null?
  - As mentioned, *String*s are complex types and the default value of variables of complex types is null in Java.
  - Because null is obviously a valid value for each thinkable complex type, of which type is null? Well, null has no type at all!
  - Let's just assume, that null is a constant, esp. a literal, representing the absence of a value of complex type.

- Besides using *String*s to output text, initializing and assigning variables, the next important operation is *String* concatenation.
  - *String*s can be "glued together", programmers call this operation *String* concatenation. In Java it can be done with the +-operator:

```
String name = "Arthur";
String fullName = "Arthur" + " " + "Dent";              // Concatenation of literals with the +-operator (does also work over multiple lines)
// fullName = "Arthur Dent"
String anotherFullName = name + " and" + " Ford"; // Concatenation of a String variable and two literals
// anotherFullName = "Arthur and Ford"
```

  - If values of other types (int, double, boolean etc.) are concatenated to a *String*, the values will be copied and converted to *String*s:

```
String ageText = name + " is of age" + 42 + "."; // The int 42 will be copied and converted to the String "42" and then be concatenated.
// ageText = "Arthur is of age 42."
```

Strings, Characters and Methods – Part 1

- A *String* can be interpreted as a sequence of characters, like a text is a sequence of letters.
  - An individual character of a *String* is represented by a value of type char.
  - We can get an individual character of a *String* by calling *String*'s method *charAt()*:

```
String aName = "Arthur";
char firstLetter = aName.charAt(0);
// firstLetter = 'A'
```

Good to know
char is pronounced [ˈtʃɒr] not [ˈkɒr]!

- What is a "method"?
  - Up to now, we've dealt with mathematical operators to work with values, but as a complex type, *String* goes far beyond operators!
  - Basically like an operator, a method performs an action on the value it is called, but its syntax is different.
  - As can be seen we call *charAt()* by writing a period and the method's name next to the variable *aName* holding the value.
  - Mind that the name of the method *charAt()* does also obey the camelCase naming convention!
  - We already saw the method *println()*. We called it on *System.out*, this variable is responsible for outputting data to the console.
    - The variable *System.out* is not of type *String*, but of another complex type, namely *PrintStream*.

- What does the method "*charAt()*" do?
  - *charAt()* accepts an argument, which is the position of the respective character in the *String aName*.
    - A "position in a *String*" is usually called index by programmers. Indexes in Java-*String*s are 0-based, thus the first char is at index 0, not at index 1!
  - The method *charAt()* returns the char value found on the accepted index in the *String* as result of its call.
    - In the example above we consequently assigned the result of the *charAt()*-call to a variable of type char (*firstLetter*).

- If the index passed to *String.charAt()* exceeds the *String*'s bounds, a *java.lang.StringIndexOutOfBoundsException* will be thrown.

Strings, Characters and Methods – Part 2

- Using *String*s does generally involve the usage of many methods, which can be called on *String*s.
  - That *String* is a more complex type, than, e.g. int, means, that it keeps more information than a primitive type, e.g. the length of text:

```
String name = "Arthur";
int namesLength = name.length();
// namesLength = 6
```

  - Obviously, there is no mathematical operator for "length of text", instead we have to use *String*'s method *length()*.
    - The method *length()* does not accept any arguments. – We just want to get the *String*'s length, where no further arguments are required!
    - The method *length()* returns the *String*'s length as a result of its call. The length of a *String* is an int value.

- Having fun with *charAt()* and *length():*
  - Task: get the last character of a *String*. – We can solve this task by combining the methods *charAt()* and *length()*!

```
char lastLetter = name.charAt(name.length() - 1); // We have to subtract 1, because indexes are 0-based!
// lastLetter = 'r'
```

- In future lectures, we'll see a lot more complex types and methods. Calling methods is also a simple Java-concept.

- But for now let's introduce another important application of *String*'s methods: *String* comparison.

Comparison of Strings

- The comparison of *String*s needs to be done with methods and not with operators, esp. not with "=="!
  - The most important comparison of *String*s is to check for equality of two *String*s. It is done with *String*'s method *equals()*:

```
// WRONG way to compare two Strings
String herName = "Gwen";
String hisName = "Peter";

if (herName == hisName){
        System.out.println("Em, sorry?");
}
```

```
// Correct way to compare two Strings
String herName = "Gwen";
String hisName = "Peter";

if (herName.equals(hisName)){
        System.out.println("Em, sorry?");
}
```

- *String*s are compared case-sensitively, i.e. the case of the compared *String*s does matter!
  - We can use *String*'s method *equalsIgnoreCase()* to compare *String* in a case-insensitive manner:

```
String herName1 = "Helena";
String herName2 = "heLeNA";
// Comparing Strings case-sensitively:
boolean result = herName1.equals(herName2);
// result = false
```

```
String herName1 = "Helena";
String herName2 = "heLeNA";
// Comparing Strings case-insensitively:
boolean result = herName1.equalsIgnoreCase(herName2);
// result = true
```

- To compare *String*s for their relative order (a *String* is "greater than" or "less than" another one), we've to use *compareTo()*:
  - The *Strings a* and *b* are compared for their lexicographic order.
  - *compareTo()* returns an int! A boolean result makes no sense:
    - If *a* is "less than" *b*, *compareTo()* returns a value less than zero.
    - If *a* is "greater than" *b*, *compareTo()* returns a value greater than zero.
    - If *a* is equal to *b*, *compareTo()* returns zero.
  - Tip: There also exists *compareToIgnoreCase()*.

```
String a = "Athens";
String b = "Olympia";

if (0 > a.compareTo(b)) { // compareTo() returns an int!
        System.out.println(a + " lexicographically before " + b);
        // >Athens lexicographically before Olympia
}
```


- The exact numeric result of *String.compareTo()*/*String.compareToIgnoreCase()* has no meaning! Only whether their result is less than, greater than or equal to zero is relevant.

String to int and String to double – Part 1

- Java has no implicit conversions or cast operations from primitive values (e.g. int or double) to *String*s and vice versa.

```
String anInteger = (String)125;      // Invalid!
String aDouble = (String)3.14;       // Invalid!
```

  - Such conversion operations must be done with methods.

- There exist two ways to convert primitive values to *String*s:
  - (1) The method *String.valueOf()* converts the passed primitive value (e.g. int or double) to a *String*:

```
String anInteger = String.valueOf(125);
// anInteger = "125"
String aDouble = String.valueOf(3.14);
// aDouble = "3.14"
```

  - (2) Each primitive type in Java has a so called wrapper class.
    - E.g. for int there exists the wrapper class *Integer*, double's wrapper class is *Double*.
    - All wrapper classes support the special method *toString()*, which converts the passed primitive value (e.g. int or double) to a *String*:

```
String anInteger = Integer.toString(125);
// anInteger = "125"
String aDouble = Double.toString(3.14);
// aDouble = "3.14"
```

String to int and String to double – Part 2

- Converting a *String* to a primitive value is tricky, because the *String* in question needs to be read and interpreted.

- Therefor programmers usually call all conversion operations, which result in *String* values parsing.
  - The term parsing underscores the fact, that *String*-conversion may involve sophisticated analysis operations.

- The conversion from *String*s to primitive values is as simple as the conversion the other way:
  - Each wrapper class supports a method *parse**TypeName**()*, e.g. *Integer.parse**Int**()* or *Double.parse**Double**()*:

```
int primitiveInt = Integer.parseInt("125");
// primitiveInt = 125
double primitiveDouble = Double.parseDouble("3.14");
// primitiveDouble = 3.14
```

- Parsing a *String* to a primitive value can be dangerous!
  - The problem: a text representing a *String* value could contain unexpected characters!
  - The programmer must exactly know, of which primitive type the content of the *String* is meant to be!
  - E.g. assume we try to parse an int from a *String*, but that *String* contains prosaic text instead of an int value:

```
int primitiveIntInDoubt = Integer.parseInt("Good morning!");
// Throws a NumberFormatException: For input string: "Good morning!"
```

    - So, if we do this is Java (in most cases unintentionally :)), the Java runtime throws a *NumberFormatException*
  - Alas Java provides no way to check the content of a *String*, before parsing takes place!

Thank you!