UNIVERSITÄT
ZU KÖLN

Session 2: Syntax, Variables, Data types, Operators

Softwaretechnologie: Java I

Nils Reiter
`nils.reiter@uni-koeln.de`

October 16, 2024

INSTITUT FÜR
DIGITAL HUMANITIES
UNIVERSITÄT ZU KÖLN

```java
public class Demo {
  public static void main(String[] args) {
    System.out.println("Willkommen an der Universität zu Köln!");
  }
}
```

Section 1

Syntax and Basic Components of Java Programs

Statements

- Sequences of statements: A program
- Each statement ends with a semicolon `;`
- Different kinds of statements
    - Function call: `System.out.println("Welcome ...")`
        - A pre-defined subroutine/mini program that does something
- Statements can be grouped into code blocks with curly braces `{ ... }`

Statements

Function Calls

```
System.out.println("Welcome ...")
```

- Three identifiers, joined with a period
- Round braces
- A literal value

➔ A function call with a single argument

Java Syntax

- Identifiers: Names for all kinds of things
  - Case-sensitive
  - Only letters, underscore and digits
    - Cannot start with a digit
  - We will define identifiers ourselves

hallo

Hallo

HALLO

_5Freunde

Java Syntax

- Identifiers: Names for all kinds of things
    - Case-sensitive
    - Only letters, underscore and digits
        - Cannot start with a digit
    - We will define identifiers ourselves
    - Which of the following is a legal identifier?
        - `hello`
        - `hällö`
        - `this_is_an_identifier_or_is_it?`
        - `king-charles-5`
        - `3doorsdown`

x

y

x1

Java Syntax

- Identifiers: Names for all kinds of things
  - Case-sensitive
  - Only letters, underscore and digits
    - Cannot start with a digit
  - We will define identifiers ourselves
  - Which of the following is a legal identifier?
    - `hello`
    - `hällö`
    - `this_is_an_identifier_or_is_it?`
    - `king-charles-5`
    - `3doorsdown`
- Literals: Values that we write into the code
  - E.g., `"Welcome ..."`

Formatting

- Java does not care about indentation or line breaks
- This:

```
public class Demo { public static void main(String[] args) { System.out.println("Welc
```

is a perfectly fine Java program

Formatting

- Java does not care about indentation or line breaks
- This:

  ```java
  public class Demo { public static void main(String[] args) { System.out.println("Welc
  ```

  is a perfectly fine Java program
- Human programmers care about indentation and line breaks
- Programming: Dealing with complexity
  - Sensible formatting is one aspect

Formatting

- Java does not care about indentation or line breaks
- This:

  ```java
  public class Demo { public static void main(String[] args) { System.out.println("Welc
  ```

  is a perfectly fine Java program
- Human programmers care about indentation and line breaks
- Programming: Dealing with complexity
  - Sensible formatting is one aspect

➔ Format your code such that it reflects the logic of the code

Variables

- Placeholders for values
- Identifier as name, but must be unique
- Value can change over time
- Are typed: They can only hold values of one type
- Need to be declared before they can be used
- Can be used instead of literal values

"Welcome ..."

System.out.println(...)

myWelcomeMessage

Variables

- Placeholders for values
- Identifier as name, but must be unique
- Value can change over time
- Are typed: They can only hold values of one type
- Need to be declared before they can be used
- Can be used instead of literal values
- New kinds of statements
    - Declaration of a variable: `String s;`
    - Assignment of a value to a variable: `s = "Welcome ...";`

Variables

- Placeholders for values
- Identifier as name, but must be unique
- Value can change over time
- Are typed: They can only hold values of one type
- Need to be declared before they can be used
- Can be used instead of literal values
- New kinds of statements
    - Declaration of a variable: `String s;`
    - Assignment of a value to a variable: `s = "Welcome ...";`

```java
String s; // Declaration
s = "Welcome ..."; // Assignment
System.out.println(s);
```

Variables

- Placeholders for values
- Identifier as name, but must be unique
- Value can change over time
- Are typed: They can only hold values of one type
- Need to be declared before they can be used
- Can be used instead of literal values
- New kinds of statements
  - Declaration of a variable: `String s;`
  - Assignment of a value to a variable: `s = "Welcome ...";`
  - Declaration and assignment in one statement: `String s = "Welcome ..."`

```java
String s; // Declaration
s = "Welcome ..."; // Assignment
System.out.println(s);
```

Variables

- Placeholders for values
- Identifier as name, but must be unique
- Value can change over time
- Are typed: They can only hold values of one type
- Need to be declared before they can be used
- Can be used instead of literal values
- New kinds of statements
  - Declaration of a variable: `String s;`
  - Assignment of a value to a variable: `s = "Welcome ...";`
  - Declaration and assignment in one statement: `String s = "Welcome ..."`

```
String s; // Declaration
s = "Welcome ..."; // Assignment
System.out.println(s);
```

```
String s = " ..."; // Declaration
                   // + Assignment
System.out.println(s); // Func. call
```

Assignment Statements

```
s = "Welcome ...";
```

Assignment Statements

```
s = "Welcome ...";
```

- Assign some value to some variable
- Right-hand side (RHS): The value
  - E.g., a literal value: `String s = "Welcome ...";`

Assignment Statements

```java
s = "Welcome ...";
```

- Assign some value to some variable
- Right-hand side (RHS): The value
  - E.g., a literal value: `String s = "Welcome ...";`
  - In general, RHS is an expression

Expressions

- Structure of an assignment statement: IDENTIFIER = EXPRESSION ;
- Different kinds of expressions
  - Literal values are expressions (`"Welcome"` is an expression)
  - Variables are expressions (`s` is an expression (if declared before))

Expressions

- Structure of an assignment statement: IDENTIFIER = EXPRESSION ;
- Different kinds of expressions
    - Literal values are expressions (`"Welcome"` is an expression)
    - Variables are expressions (`s` is an expression (if declared before))
    - Expressions combined with operators are expressions
        - Operators: comparison, mathematical computation, and many more
        - E.g., `"Hi"+"Everyone"` is an expression
          (because the plus-operator is defined for strings)

More Expressions

All the following are expressions:

- `5 + 7`
- `5 + i` (if a variable `i` is defined and of the correct type)
- `5 + 5 * i` (if a variable `i` is defined and of the correct type)
    - Mathematical operator precedence (“Punkt vor Strich”)
- `(5 + 5) * i` (if a variable `i` is defined and of the correct type)
    - We can use parentheses to influence the order in which things are computed

- Expressions can be executed and yield a (single, clearly defined) value
- The result of the expression is assigned (if the expression is part of an assignment)

demo

Section 2

Data Types

Data Types

- Java: Strong typing
- All variables and literals in Java have types
- Types are known at compile-time
  - (i.e., when we write a program)
- Benefit: Compiler can prevent type-related errors
  - E.g., it's a compile error to subtract a String

Primitive Data Types and Objects

Two kinds of types

- Primitive data types: Built into the language
    - Type names are reserved keywords in Java
        - I.e., it's not allowed to use them as an identifier
    - Convention: Lower cased

Primitive Data Types and Objects

Two kinds of types

- Primitive data types: Built into the language
    - Type names are reserved keywords in Java
        - I.e., it's not allowed to use them as an identifier
    - Convention: Lower cased
- Non-primitive data types ("reference types"): Established in the library
    - Type names are defined by library authors
    - Convention: Upper cased
    - Reference types can also be defined by us (in the form of classes, to be discussed later)

String

Primitive Data Types

Keyword	Full name	Values
`boolean`	Binary value	`true`, `false`
`byte`	1 Byte (= 8 bit)	-128 to 127
`short`	short integer (16 bit)	$-32\,768$ to $32\,767$
`int`	Integer (32 bit)	$-2\,147\,483\,648$ to $2\,147\,483\,647$
`long`	long integer (64 bit)	$-9\,223\,372\,036\,854\,775\,808$ to $9\,223\,372\,036\,854\,775\,807$
`char`	Character in UTF-16	`'\u0000'` to `'\uffff'` ($65536 = 2^{16}$ symbols)
`float`	Decimal numbers (32 bit)	$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
`double`	Decimal numbers (64 bit)	$\pm 4.9 \times 10^{-324}$ to $\pm 1.8 \times 10^{308}$

Table: All primitive data types in Java

Bits and Bytes

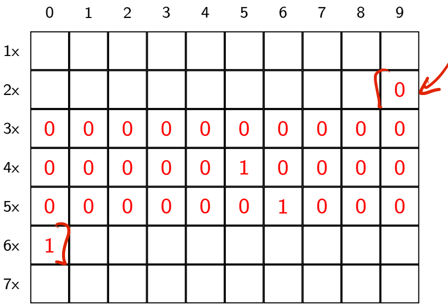
- 1 Bit: 0 or 1
- 1 Byte = 8 Bit: 0,0,0,0,0,0,0,0 – 1,1,1,1,1,1,1,1

Primitive Types in Memory

	0	1	2	3	4	5	6	7	8	9
1x										
2x										
3x										
4x										
5x										
6x										
7x										

```
int myIntVariable = 32785;
// compiler knows that myIntVariable
// goes from bits 29 to 60
byte myByteVariable = 4;
// myByteVariable: bits 61 to 68
```

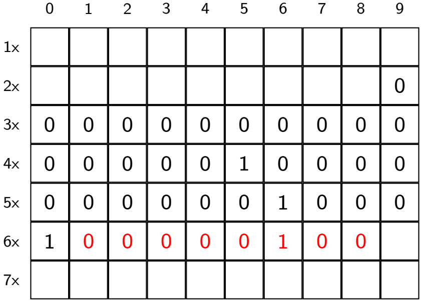
Primitive Types in Memory



	0	1	2	3	4	5	6	7	8	9
1x										
2x										0
3x	0	0	0	0	0	0	0	0	0	0
4x	0	0	0	0	0	1	0	0	0	0
5x	0	0	0	0	0	0	1	0	0	0
6x	1									
7x										

```java
int myIntVariable = 32785;
// compiler knows that myIntVariable
// goes from bits 29 to 60
byte myByteVariable = 4;
// myByteVariable: bits 61 to 68
```

Primitive Types in Memory



	0	1	2	3	4	5	6	7	8	9
1x										
2x										0
3x	0	0	0	0	0	0	0	0	0	0
4x	0	0	0	0	0	1	0	0	0	0
5x	0	0	0	0	0	0	1	0	0	0
6x	1	0	0	0	0	0	1	0	0	
7x										

```java
int myIntVariable = 32785;
// compiler knows that myIntVariable
// goes from bits 29 to 60
byte myByteVariable = 4;
// myByteVariable: bits 61 to 68
```

Data Types for Boolean Values

Keyword	Full name	Values
`boolean`	Binary value	`true`, `false`

- Occupies a single bit
- Comparison operators that produce a boolean value:
  - `5 < 7 //yields true`
  - `9 == 5 //yields false`
  - `5 <= 6 //yields true`

Data Types for Boolean Values

R x ← 5;

Keyword	Full name	Values
`boolean`	Binary value	`true`, `false`

- Occupies a single bit
- Comparison operators that produce a boolean value:
  - `5 < 7 //yields true`
  - `9 == 5 //yields false`
  - `5 <= 6 //yields true`

boolean b = true;

⚠ `=` and `==` are not the same thing

Data Types for Natural Numbers

Keyword	Full name	Values
`byte`	1 Byte (= 8 bit)	-128 to 127
`short`	short integer (16 bit)	$-32\,768$ to $32\,767$
`int`	Integer (32 bit)	$-2\,147\,483\,648$ to $2\,147\,483\,647$
`long`	long integer (64 bit)	$-9\,223\,372\,036\,854\,775\,808$ to $9\,223\,372\,036\,854\,775\,807$

- Integral data types defined over the number of bits they occupy
- Shorter types consume less memory
- I.e.: If you know a value will never be higher than 127, use a byte
  - E.g., To store calendar weeks, human age in years(?), ...

Integral Data Types

Literals

- By default: literal numbers are of type `int` (e.g., in an expression)

```java
int myIntValue = 27; // literal int value assigned to an int variable
byte myByteValue = 27; // literal int value assigned to a byte variable
long myLongValue = 27; // literal int assigned to a long variable

long myLargeLongValue = 2700000000000000000L;
                      // append L to enforce a long literal
long mySmallLongValue = 27L; // also works for small numbers
```

Integral Data Types

Literals

- By default: literal numbers are of type `int` (e.g., in an expression)

```java
int myIntValue = 27; // literal int value assigned to an int variable
byte myByteValue = 27; // literal int value assigned to a byte variable
long myLongValue = 27; // literal int assigned to a long variable

long myLargeLongValue = 2700000000000000000L;
                      // append L to enforce a long literal
long mySmallLongValue = 27L; // also works for small numbers
```

- Why can we assign an int literal to a byte/long/short variable?
  ➔ Implicit casting (next week)!

Character Data

Keyword	Full name	Values
`char`	Character in UTF-16	`'\u0000'` to `'\uffff'` ($65536 = 2^{16}$ symbols)

- Characters are represented in computers by enumerating them
- American Standard Code for Information Interchange (ASCII) Wikipedia: ASCII
  - 128 characters, including control symbols for telegraphy
  - No German Umlauts etc.
- Unicode: A single standard to represent *all* characters from all languages unicode.org
  - $149\,186$ characters, including CJK ideographs Unicode 15.0 charts
  - Complex enumeration scheme

Character Data

char data type

- `char` represents a single character in two bytes (16 bit)
- Literal char values are written with single quotes: `char ch = 'a';`
- Unicode code points can also be used: `char ch = '\u1A0A'; //"BUGINESE LETTER NA"`
  - $1A0A_{b=16} = 6666_{b=10}$ = ᨊ
- Integer values also possible: `char ch = 121;` (implicit cast)
- `char` is not the same as `String`

Decimal Numbers

- Real numbers challenging for computers
- Floating-point arithmetic developed in Mesopotamia (ca. 700 BCE!)
- First used in computer by Zuse in 1937/1941

Decimal Numbers

- Real numbers challenging for computers
- Floating-point arithmetic developed in Mesopotamia (ca. 700 BCE!)
- First used in computer by Zuse in 1937/1941
- Naive idea: Two integer values, before and after decimal point
  - Wasteful and complex to implement math

Decimal Numbers

- Real numbers challenging for computers
- Floating-point arithmetic developed in Mesopotamia (ca. 700 BCE!)
- First used in computer by Zuse in 1937/1941
- Naive idea: Two integer values, before and after decimal point
  - Wasteful and complex to implement math
- Better idea: Represent number in scientific notation, store digits and exponent separately
  - E.g.: $123.345 = 123345 * 10^{-3}$ (there are many details left out here)

Decimal Numbers

- Real numbers challenging for computers
- Floating-point arithmetic developed in Mesopotamia (ca. 700 BCE!)
- First used in computer by Zuse in 1937/1941
- Naive idea: Two integer values, before and after decimal point
  - Wasteful and complex to implement math
- Better idea: Represent number in scientific notation, store digits and exponent separately
  - E.g.: $123.345 = \underline{123345} * \boxed{10^{-3}}$ (there are many details left out here)

⚠ Floating point numbers are *approximations*, not all values can be represented

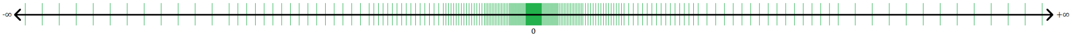


Figure: Representable numbers in floating point representation

Decimal Numbers in Java

Keyword	Full name	Values
`float`	Decimal numbers (32 bit)	$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
`double`	Decimal numbers (64 bit)	$\pm 4.9 \times 10^{-324}$ to $\pm 1.8 \times 10^{308}$

Table: Floating point types

Decimal Numbers in Java

Keyword	Full name	Values
`float`	Decimal numbers (32 bit)	$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
`double`	Decimal numbers (64 bit)	$\pm 4.9 \times 10^{-324}$ to $\pm 1.8 \times 10^{308}$

Table: Floating point types

- By default: Decimal numbers interpreted as double

Decimal Numbers in Java

Keyword	Full name	Values
`float`	Decimal numbers (32 bit)	$\pm 1.4 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$
`double`	Decimal numbers (64 bit)	$\pm 4.9 \times 10^{-324}$ to $\pm 1.8 \times 10^{308}$

Table: Floating point types

- By default: Decimal numbers interpreted as double

```java
float myFloatVariable = 3.0; // literal double, no implicit cast: compile error!
double myDoubleVariable = 3.0; // literal double
float myExplicitFloatVariable = 5.0f; // literal float value
double myDoubleVariable = 5.0f; // literal float casted into a double
```

Division, again

- Dividing two `int` numbers yields unexpected results
- If one number is a floating-point-number, we get decimal division

```
int a = 7;
int bInt = 14;
System.out.println(a / bInt); // prints 0

double bFloat = 14.0;
System.out.println(7 / bFloat); // prints 0.5
```

Operators

division

modulo

- Math operators: `+`, `-`, `*`, `/`, `%`
  - `/` behaves differently in decimal or natural mode
- Comparison operators: `<`, `<=`, `==`, `>=`, `>`
  - Yield boolean values

Operators

- Math operators: `+`, `-`, `*`, `/`, `%`
  - `/` behaves differently in decimal or natural mode
- Comparison operators: `<`, `<=`, `==`, `>=`, `>`
  - Yield boolean values
- Logical operators: `&&`, `||`, `!`
  - Yield boolean values, expect boolean input
- All operators: `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html`
  - Some will be introduced later in the semester

More int-Operators

`+`	Addition	`5 + 5 //10`
`-`	Subtraction	`5 - 5 //0`
`*`	Multiplication	`5 * 5 //25`
`/`	Integer Division	`5 / 5 //1`
		`5 / 4 //1`
		`4 / 5 //0`
`%`	Modulo	`5 % 5 //0`
		`5 % 4 //1`
		`4 % 5 //4`

More int-Operators

`+`	Addition	`5 + 5 //10`
`-`	Subtraction	`5 - 5 //0`
`*`	Multiplication	`5 * 5 //25`
`/`	Integer Division	`5 / 5 //1`
		`5 / 4 //1`
		`4 / 5 //0`
`%`	Modulo	`5 % 5 //0`
		`5 % 4 //1`
		`4 % 5 //4`

All these operators operate on two `int`-values and yield an `int`-value

Comparison Operators

Symbol	Description	Example
`<`	less than	`3 < 5 //true`
`>`	greater than	`3 > 5 //false`
`==`	equal	`3 == 5 //false`

Comparison Operators

Symbol	Description	Example
`<`	less than	`3 < 5 //true`
`>`	greater than	`3 > 5 //false`
`==`	equal	`3 == 5 //false`

- Important difference
  - `==`: Comparison operator
  - `=`: Assignment operator

Comparison Operators

Symbol	Description	Example
`<`	less than	`3 < 5 //true`
`>`	greater than	`3 > 5 //false`
`==`	equal	`3 == 5 //false`

- Important difference
  - `==`: Comparison operator
  - `=`: Assignment operator
- New type: `boolean`
  - Only two possible values: `true` or `false`

Comparison Operators

Symbol	Description	Example
`<`	less than	`3 < 5 //true`
`>`	greater than	`3 > 5 //false`
`==`	equal	`3 == 5 //false`

- Important difference
  - `==`: Comparison operator
  - `=`: Assignment operator
- New type: `boolean`
  - Only two possible values: `true` or `false`

More operators

Variables and Scope

- Most variables have limited lifetime: Their scope
- Code blocks `{ }` define scope boundaries
- Scope is nested: We can access upwards, but not downwards

Variables and Scope

- Most variables have limited lifetime: Their scope
- Code blocks `{ }` define scope boundaries
- Scope is nested: We can access upwards, but not downwards

```java
public class Scope {

  public static void main(String[] args) {
    int a = 5;
    int b = 17;

    int c = a + 45;
    System.out.println(c);
    {
      int d = b - 10;
      System.out.println(d);

    }
    int d = b - 10;
    System.out.println(d);

  }
}
```

Section 3

Exercise