Programming Basics – WiSe21/22

Fundamental language concepts

Prof. Dr Silke Lechner-Greite

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Table of contents - overall

Technische Hochschule Rosenheim

Chapter 2: Fundamental language concepts

2.1 Data types

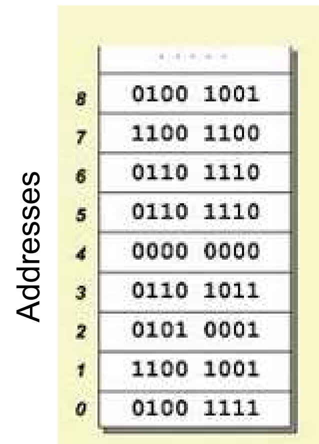
2.2 Variables and assignments

2.3 Expressions and operators

How does a computer store information?

- Computer memory can store any bit patterns
  - Meaning must be clearly defined
  - Definition of different schemas for use on a sequence of bits

- Data type
  - is a schema for the use of bits to represent values
  - Values are not just numbers, but any kind of data that a computer can process



Main Memory

What data can be processed in Java programmes?

Technische Hochschule Rosenheim

- Java distinguishes between two categories:
  - Primitive types
    - Simple types for numbers, (Unicode) characters and truth values (a.k.a. logical values or Boolean values)

later

  - Non-primitive (reference) types
    - Management of object references
      (e.g. strings, dialogues or data structures)

Overview of primitive data types

Technische Hochschule Rosenheim

Source: Ullenboom (2012): Java ist auch eine Insel

8 primitive data types

- 6 different types for the representation of numbers
- 1 type for the representation of characters
- 1 type for the representation of boolean values

Numerical primitive data types

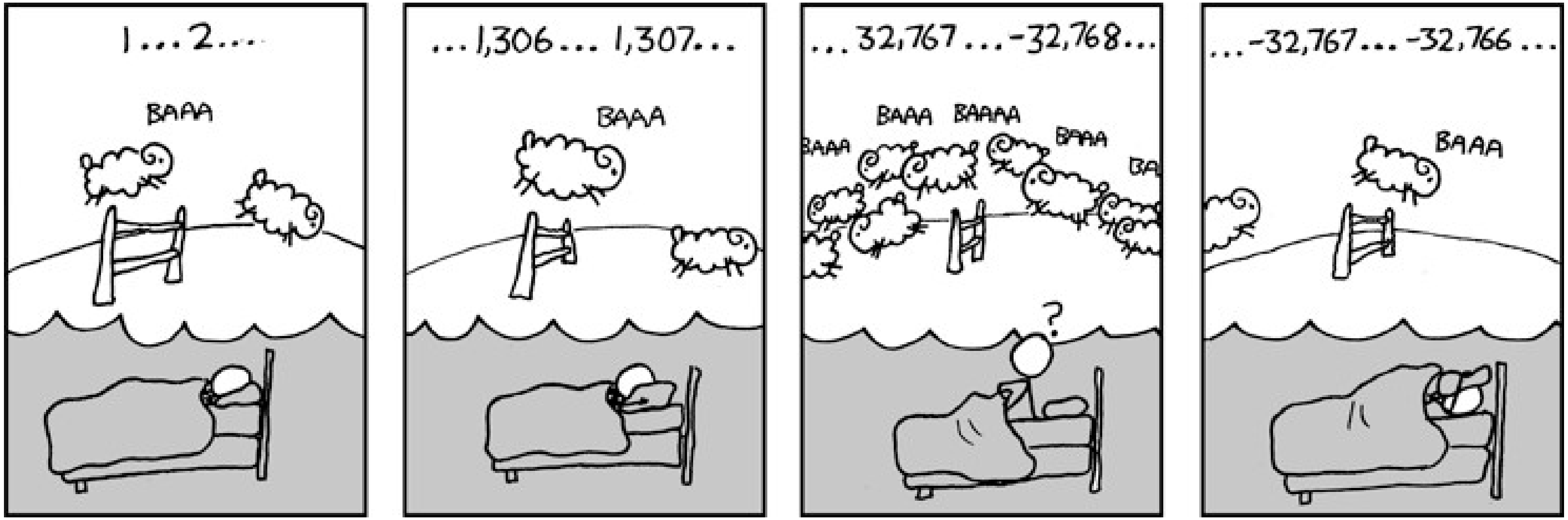
Integer primitive data types		
Type	**Size**	**Range of values**
byte	8 Bit	-128 **to** +127
short	16 Bit	-32.768 **to** +32.767
int	32 Bit	**approx. -2 billion to +2 billion**
long	64 Bit	**approx. -10e18 to +10e18**

Primitive floating point types		
Type	**Size**	**Range of values**
float	32 Bit	**-3.4e38 to +3.4e38**
double	64 Bit	**-1.7e308 to +1.7e308**

Legend: e stands for "powers of ten"

The larger the value range, the more bits are required

Digression: number overflows



Literals

- For programming we do not use the bit patterns.

- We use literals:
  - Specific values of the respective primitive data type
  - Examples of integer literals: 122, 16 or -32
  - Examples of floating-point literals: 123.0, -19823.234, 0.00000321

Primitive data type `char`

- Representation of characters using 16 bits
- Application of the Unicode method
  - Each character is assigned a bit pattern (digital code)
  - Extract:

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	LF	11	VT	12	FF	13	CR	14	SO	15	SI
16	DLE	17	DCI	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	|	125	}	126	~	127	DEL

Character literals

- In a programme, a character literal is enclosed by simple quotation marks:
  'a' 'm' 'A'

- Control characters are also possible, e.g.
  '\n' new line
  ' \t' tab character

You'll learn more on this topic in the Chapter about Characters and Strings!

Primitive data type `boolean`

- Type for truth values
- Only two values possible => two `boolean` literals:
  - `true` = true, yes, applicable
  - `false` = false, no, not applicable

- `boolean` is not a numerical type, it's incompatible with `int` or `double`

Exercise – Data types

- Live exercise
  - Complete Task 1 on the live exercises sheet “Fundamental language concepts”
  - You have 5 minutes.

Chapter 2: Fundamental language concepts

Fundamental concept variable - motivation

Technische Hochschule Rosenheim

- Main memory: storage of data as well as machine commands

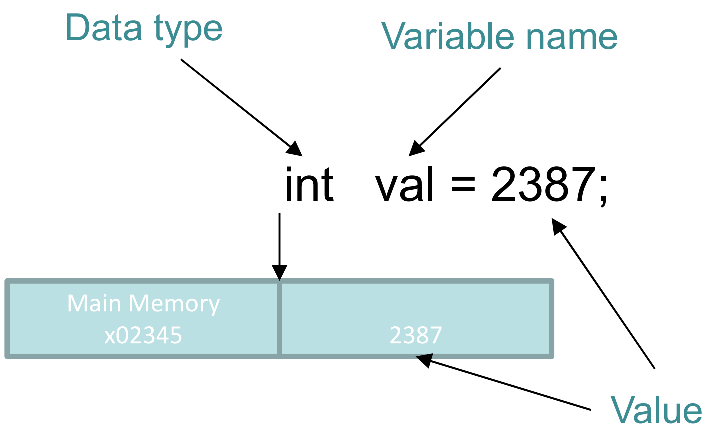
- To put data in memory and then retrieve it later, a programme must have a name for each memory section it uses.

Variable

- Name for a memory location in the main memory
  - Use of a specific data type
  - Value of the variable is stored in memory
  - Value can be read (retrieved) and changed during programme execution

- Containers in which specific data values can be stored for later use

- Data type determines
  - size of the memory area and
  - what kind of data can be stored



Variable names

- Variable names are designators/identifiers
- By convention, variable names begin with lower case letters

!!!

- Examples of variable names:

```
salary
i
counter
track2
```

Identifiers - different types

Technische Hochschule Rosenheim

- In many places in the source code, names (designators, identifiers) can be freely selected by the programmer

```
public class Hello        Class name
{                                      Method name
  public static void main (String[] args)  Variable name
  {
    System.out.println ("Hello World!");
  }
}
```

- Names must comply with the syntax
  - Only upper and lower case letters, numbers and underscore (_) are allowed
  - The first character cannot be a number
  - None of the approx. fifty reserved words (keywords) allowed

Identifiers (1)

- Examples:
  - `counter`
  - `colourDepth`
  - `iso9660`
  - `xmlProcessor`
  - `MAX_VALUE`

- Not allowed:

`1stTry`	first letter cannot be a number
`queen of hearts`	spaces not allowed in name
`const`	reserved word
`muenchen-erding`	hyphen not allowed in name

Identifiers (2)

https://en.wikipedia.org/wiki/Camel_case

Recommendations for notation:

		
➢ Lower case letters for variables	`COUNTER`	`counter`
➢ Upper case letters for constants	`max_value`	`MAX_VALUE`
➢ New parts of words with upper case letters	`gettoken`	`getToken`
➢ Whole words	`c`	`counter`
➢ Meaningful names	`o0O0oO`	`counter`
➢ Write confusing abbreviations out in full	`bup`	`binaryUpload`
➢ Common acronyms in upper case letters	`Html`	`HTML`

Java keywords

`abstract`	`continue`	`float`	`native`	`super`
`assert`	`default`	`for`	`new`	`switch`
`boolean`	`do`	`goto (*)`	`package`	`synchronized`
`break`	`double`	`if`	`private`	`this`
`byte`	`else`	`implements`	`protected`	`throw`
`case`	`enum`	`import`	`public`	`throws`
`catch`	`extends`	`instanceof`	`return`	`transient`
`char`	`false`	`int`	`short`	`try`
`class`	`final`	`interface`	`static`	`void`
`const (*)`	`finally`	`long`	`strictfp`	`volatile`
				`while`

(*) Although `const` and `goto` are reserved keywords, they are not used in Java .

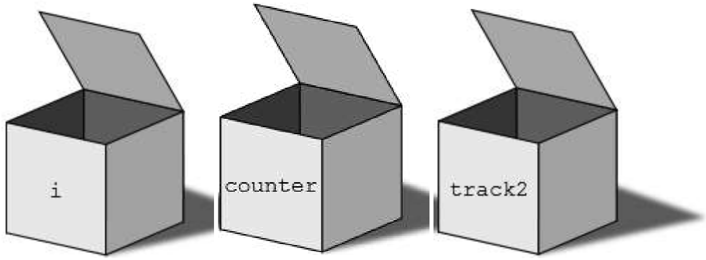
Definition of a variable

- Variables must be defined (i.e. declared)

```
Data type Variable name ;
```

- Examples:

```
int i;
int counter;
int track2;
```



- Only one definition per variable
- Definition == statement (instruction)

Initialisation of variables

- Immediately assign an initial value to the variable when it is defined = initialisation

```
int fahrenheit = 91;
```

Short form for
definition + initialisation

- Separate definition and value assignment …

```
int start;
start = 11;
```

… is equivalent to

```
int start = 11;
```

- Initialise variables whenever possible! (IDEs point you to it, too.)

Uninitialised variables

- Compiler monitors the use of variables
- Example of an error:

```java
public static void main(String[] args) {
    int length;
    int breadth;
    breadth = 2*length;
}
}
```

```
Variable 'length' might not have been initialized
Initialize variable 'length'   Alt+Umschalt+Eingabe     More actions...   Alt+Eingabe
int length
```

Compiler Error:

java: variable length might not have been initialized

  - A newly defined variable has no value (uninitialised)
  - You can assign a value to an uninitialised variable (write)
  - However, it cannot be read from an uninitialised variable

- Compiler does not translate the programme!

Fixed variables - constants

- Some variables (values) should not change after they are assigned => constants
- Protection against change with modifier `final`

```
final int speedOfLight;
speedOfLight = 299792458;
```

- `final` allows only one value assignment

```
final int speedOfLight;
speedOfLight = 299792458;
speedOfLight = 0;            // Error!
```

Exercise – Definition of variables

- Live exercise
  - Complete Tasks 2 and 3 on the live exercises sheet “Fundamental language concepts”
  - You have 10 minutes.

Example programme (1)

Technische Hochschule Rosenheim

- Simple output of the value of a variable

```
public class Example {

    public static void main(String[] args) {

        int workingSalary = 1200;
        System.out.println("Earnings: " + workingSalary);

    }

}
```

Example programme (2)

- Use of variables

```
public class Example {

   public static void main(String[] args)    {

      int workingHours = 40;
      double hourlySalary = 10.0;

      System.out.println("Hours worked: " + workingHours);
      System.out.println("Gross salary: " + (workingHours * hourlySalary));

   }

}
```

Important concept: to use the value stored in a variable, simply use the name of the variable.

Assignment instructions (1)

- Syntax:

```
variable name = expression;
```

  - Equal sign "=" means "assignment“ or “initialisation”
  - `expression` is a collection of characters that returns a value

- Example assignments (assumption: variables have already been defined):

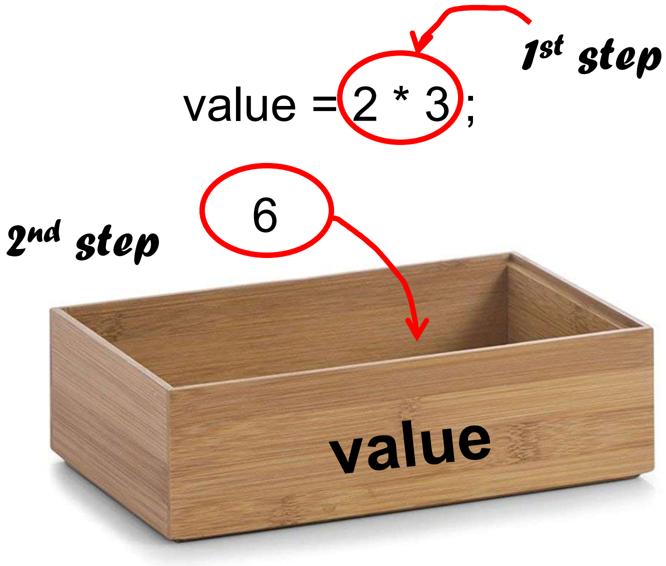
```
total =  3 + 8;
price =  12.99;
tax = price * 0.05;
```

Assignment instructions (2)

Technische Hochschule Rosenheim

- Semantics: two steps
  1. Calculation TO THE RIGHT of the equal sign
     -> value of the expression on the right is calculated or
     -> value is only used
  2. Contents of the variable TO THE LEFT of the equal sign is replaced by the result of step 1

- Example: `value =  2 * 3;`
  1. Calculation 2 * 3 results in 6
  2. 6 is placed in the variable named `value`



Types of instructions

Technische Hochschule Rosenheim

- So far, you know the following types of instructions:
  - Definition / Declaration
    ```
    int counter;
    ```
  - Value assignment / Initialisation
    ```
    counter = 1;
    ```
  - Output
    ```
    System.out.println(counter);
    ```
- Programme = list of instructions
- Sequence of instructions in the programme is arbitrary, but with regard to a variable, attention must be paid to
  1. Definition
  2. Writing (left in a value assignment)
  3. Reading (as an expression or part of an expression)

Exercise – Value assignment

- Live exercise
  - Complete Task 4 on the live exercises sheet “Fundamental language concepts”
  - You have 5 minutes.

Chapter 2: Fundamental language concepts

2.1 Data types

2.2 Variables and assignments

2.3 Expressions and operators

Expressions (1)

- Simply put, an expression is a combination of:
  - Operands - variables, constants / literals
  - Operators – symbols that link values with each other, e.g. "+" for addition or "*" for multiplication
  - Brackets - "(" and ")"

- Examples:
  - Correct expression `(44 – x) / (y +8)`
  - Incorrect expression `*z 88`

Expressions (2)

- Properties:
  - Every expression can be calculated/evaluated.
  - Every expression has a type and value, that results after the expression is evaluated.

- If an expression contains multiple operators, then precedences determine the order in which the operators are performed (cf. mathematics: BODMAS rule for order of operations)
- *BODMAS: B*rackets, *O*rder, *D*ivision/*M*ultiplication, *A*ddition/*S*ubtraction

- Different execution order through brackets (expressions in brackets are evaluated first)

Operands – integers (1)

- Numerical constants in the source code
- Notation:
  - Sequence of decimal digits
  - Positive and negative values: + or – sign before the number
  - No sign before the number = + (sign before the number optional)
  - Usually decimal, i.e. base 10

```
0
28
+28
-3888
-0
+123456789
```

Operands – integers (2)

- Syntax for other number bases:
  - Hexadecimal (base 16): Prefix "0x"

```
0x28
0x1000
```

  - Octal (base 8): Leading digit "0"

```
023
01000
```

Danger - source of errors!

  - Binary (base 2): Prefix "0b"

```
0b1000
0b1_010_000
```

Operands - floating-point numbers

Technische Hochschule Rosenheim

- Often used:
  - numbers with fractions (such as π = 3.141592…)
  - very large or very small values (such as 10^{23}, 10^{-34})

- Notation:

Digits after the decimal point	Powers of ten (E or e)		Suffix (D or d)
`3.14`	`1E23`	$(1 \cdot 10^{23})$	`D`
`0.001`	`1e-34`	$(1 \cdot 10^{-34})$	`-234d`
`-123.04`	`6.670E-11`	$(6.670 \cdot 10^{-11})$	`0.001D`
`21500.0`	`-4.17e-4`	$(-4.17 \cdot 10^{-4})$	`1e-34d`

Multiple notations are possible for the same value: `20.5` or `0.0205E3` or `205000E-4`

Arithmetic operators

- Selection:

Operator	Meaning	Precedence
-	Unary minus	Highest
+	Unary plus	Highest
*	Multplication	Middle
/	Division	Middle
%	Modulo operation	Middle
+	Addition	Lowest
-	Subtraction	Lowest

All operators are defined for integers and floating-point numbers

Numerical expressions – uniformly integer (1)

Technische Hochschule Rosenheim

- Notation is similar to mathematics – but no superscript, subscript or numbers on top of each other
  - Mathematical expressions must be "flattened" into a linear string: $\frac{3}{4}$ becomes `3/4`
  - Always write out multiplication operators: 5a becomes `5*a`
- Fundamental arithmetic operations:
  - + Addition
  - - Subtraction
  - * Multiplication
  - / Division

```
5 + 2 * 7
3 * 4 + 4 * 5
50 – 10 + 20
+2*+8
-5–3
17 / 4
```

Integer division!!!

Numerical expressions – uniformly integers (2)

- Integer division
  - truncates the digits after the decimal point of the result
  - there is no rounding

```
17 / 4    ->     4  (not 4.25)
-17 / 4   ->    -4  (not -4.25)
```

- Remainder (modulus) operator returns division remainder (%)

```
11 % 4    ->     3
 7 % 2    ->     1
18 % 18   ->     0
 1 % 18   ->     1
```

Negative operands

```
11 % -4    ->     3
-11 % 4    ->    -3
-11 % -4   ->    -3
```

Sign before the number of the result = Sign before the number of the left (first) operand

Numerical expressions – uniformly floating-point numbers

Technische Hochschule Rosenheim

- Floating-point arithmetic is mathematically more precise

```
20.0 / 8.0 -> 2.5
```

- Reasons for `int` arithmetic:
  - `double` arithmetic is slower
  - `double` values require more space
  - `double` arithmetic makes rounding errors hard to predict

`int` when possible, `double` when necessary

```
System.out.println(1000.0/50.0*50.0);   // 1000.0
System.out.println(1000.0/60.0*60.0);   // 1000.0000000000001
```

Numerical expressions – different data types

Technische Hochschule Rosenheim

- Implicit type conversion = automatic conversion of one type to another
- Example:
  - Two operands of the same type: Result type = operand type

```
1   + 2   → 3   (int)
1.0 + 2.0 → 3.0 (double)
```

  - Mixed operand type: `double` result

```
1.0  + 2    → 3.0    (double)
1    + 2.0  → 3.0    (double)
```

Implicit type conversion
(`int` -> `double`)

Steps:

1. Convert `int` operand to `double`
2. Calculate result

```
1.0  + 2   → 1.0 + 2.0
           → 3.0
```

Implicit type conversion (`double` -> `int`)

Technische Hochschule Rosenheim

- There is an equivalent `double` value for every `int` value

```
5 -> 5.0
-5000000 -> -5E9
```

- But: for many `double` values, there is no equivalent `int` value
- Therefore: no implicit type conversion of `double` -> `int`
  - Allowed:

```
double d = 5;
// first convert 5 -> 5.0, then assign
```

  - Error:

```
int i = 5.0;
// Error!
```

Explicit type conversion (type cast)

Technische Hochschule Rosenheim

- "Forced" type conversion = explicit type conversion

```
(type) expression
```

```
(int)2.5 * 3 -> 2 * 3 = 6
```

  - `(type)` formally a unary operator
  - "larger" data types can be transferred to "smaller" data types
  - Please note: loss of information!

```
double x = 3.89;
int y;
y = (int) x;            // y is assigned a value of 3
```

Minimise type casts wherever possible or avoid them entirely

Exercise – Analysing expressions

- Live exercise
  - Complete Task 5 on the live exercises sheet “Fundamental language concepts”
  - You have 5 minutes.

Evaluating expressions (1)

- The sequence of the calculation (= semantics) must still be defined
- Order is decisive:

```
2 + 3 * 4   -> 5 * 4   -> 20
2 + 3 * 4   -> 2 + 12 -> 14
```

- Order follows precedence of operators
  - Multiplicative operators (*,/,%) have higher precedence than additive operators (+,-)
- Brackets explicitly define the order

```
(2 + 3) * 4   -> 5 * 4   ->  20
```

- Unary (= single operand) operators have higher precedence than binary operators

```
-3+-4   -> (-3)+(-4)   ->  -7
```

- Precedence for different operators – associativity for operators with equal precedence

Evaluating expressions (2)

- Example:

```
8 - 3 - 2   -> 5 - 2 ->  3
8 - 3 - 2   -> 8 - 1 ->  7
```

- Associativity (direction of execution)
  - left-associative: operators are evaluated from left to right
  - right-associative: operators are evaluated from right to left

- All binary arithmetic operators are left-associative!
- For information on individual operators, see the operator table

Operator table

Operator	Rank	Type	Description
++, --	1	Arithmetical	Increment and Decrement
+, -	1	Arithmetical	unary plus and minus
~	1	Integral	bitwise complement
!	1	Boolean	logical complement
(Typ)	1	Any	Cast
*, /, %	2	Arithmetical	Multiplication, division, remainder
+, -	3	Arithmetical	Addition and subtraction
+	3	String	String concatenation
<<	4	Integral	Shift left
>>	4	Integral	Right shift with sign extension
>>>	4	Integral	Right shift without sign extension
<, <=, >, >=	5	Arithmetical	numerical comparisons
instanceof	5	Object	Type comparison
==, !=	6	Primitive	Equality/inequality of values
==, !=	6	Object	Equality/inequality of references
&	7	Integral	bitwise And
&	7	Boolean	logical And
^	8	Integral	bitwise XOR
^	8	Boolean	logical XOR
|	9	Integral	bitwise Or
|	9	Boolean	logical Or
&&	10	Boolean	logical conditional And, Short circuit
||	11	Boolean	Logical conditional Or, Short circuit
?:	12	Any	Condition Operator
=	13	Any	Assignment
*=, /=, %=, +=, =, <<=, >>=, >>>=, &=, ^=, |=	14	Arithmetical	Assignment with Operation
+=	14	String	Assignment with string concatenation

Translated from :
https://openbook.rheinwerk-verlag.de/javainsel/02_004.html

Exercise – Evaluating expressions

- Live exercise
  - Complete Task 6 on the live exercises sheet “Fundamental language concepts”
  - You have 5 minutes.

Relational operators

- Relational operators expect numerical operands, result = truth value

Operator	Meaning
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

- Evaluation analogous to arithmetic expressions according to precedence and associativity

- Relational operands only take effect after arithmetic operators – "bind weakly"

```
2 + 3 < 2 * 3
2 + 3 < 6
    5 < 6
    true
```

Exercise – Relational operators

- Live exercise
  - Complete Task 7 on the live exercises sheet “Fundamental language concepts”
  - You have 5 minutes.

Logical operators (1)

- Logical operators link truth values
- Overview of logical operators:

Operator	# operands	Name	Meaning	Result is true exactly when
&&	2	AND	Logical And	… both operands are true
||	2	OR	Inclusive logical Or	… at least one operand is true
^	2	XOR	Exclusive logical Or	… exactly one operand is true
!	1	NOT	Logical Not	… the operand is false

boolean a	boolean b	! a	a && b	a || b	a ^ b
`true`	`true`	`false`	`true`	`true`	`false`
`true`	`false`	`false`	`false`	`true`	`true`
`false`	`true`	`true`	`false`	`true`	`true`
`false`	`false`	`true`	`false`	`False`	`false`

Logical operators (2)

Technische Hochschule Rosenheim

- Logical operators can be used to formulate compound conditions
- Example:
  - Mathematics: $-8 \leq x < 8$
  - Text: x is greater than or equal to -8 and less than 8
  - Java: `(x >= -8) && (x < 8)`
- Precedence: binary, logical operators `&&`, `||`, `^` bind weaker than arithmetic and relational operators
- Like all unary operators, not `!` binds stronger than binary operators
- Example:

```
x > 6 - 11 && x + 1 < 2 * 3
(x > (6 - 11)) && ((x + 1) < (2 * 3))
```

Operator groups

- So far: three operator groups

Group	Operators	Types
Arithmetical	+ - * / %	Numerical -> numerical
Relational	< > <= => == !=	Numerical -> boolean
Logical	&& || ^ !	Boolean -> boolean

- Another possibility:
  - `boolean` values can also be checked with `==` and `!=`

`boolean` variables

- Variables of type `boolean` are allowed

```
boolean isOK;
isOK = true;
```

or

```
boolean isOK = true;
```

- Logical expressions can be assigned to `boolean` variables

```
boolean ice = temperature < 0;
boolean steam = temperature > 100;
boolean water = !ice && !steam;
```

Exercise – `boolean` variables and logical expressions

- Live exercise
  - Complete Task 8 on the live exercises sheet “Fundamental language concepts”
  - You have 5 minutes.

Runtime library (1)

- Runtime library = collection of functions, e.g.
  - input and output
  - mathematical functions
  - . . .

- Functional scope of the runtime library is referred to as the Java API (application programming interface)
  - defines the interface between user code and specified system parts
  - http://docs.oracle.com/javase/8/docs/api/

Runtime library (2)

Technische Hochschule Rosenheim

- For example: mathematical functions (`java.lang.Math`)
- Call scheme:

```
Math.function(args)
```

- Selection of functions:

Function	Mathematical	Java
Square root	$\sqrt{x}$	Math.sqrt(x)
Natural logarithm	$\ln(x)$	Math.log(x)
Logarithm to base 10	$log_{10}(x)$	Math.log10(x)
exponential function	e^x	Math.exp(x)
Sinus	$\sin(x)$	Math.sin(x)
Arcus tangent	$\arctan(x)$	Math.atan(x)

- Predefined constants: pi π, Euler's number *e*

```
Math.PI and Math.E
```