3. Java - Language Constructs I

Names and Identifiers, Variables, Assignments, Constants, Datatypes, Operations, Evaluation of Expressions, Type Conversions

Educational Objectives

- You know the basic blocks of the programming language Java
- You understand the use of *variables* in a program and you can use them properly
- You know how *values* are defined in the source code (*literals*)
- You are able to read and interpret simple *arithmetic expressions*
- You understand the reasons for a *type system* and are able to determine the type of an expression

Definition: *Names and Identifiers*

Names denote things in a program like variables, constants, types, methods, or classes.

Book, on page 21

Names and Identifiers

A program (that is, a class) needs a name

```
public class SudokuSolver { ...
```

- Convention for class names: use *CamelCase* → *Words are combined into one word, each starting with a capital letter*

Allowed names for “entities” in a program:

- Names begin with a *letter* or _ or *$*
- Then, sequence of *letters*, *numbers* or _ or *$*

Names - what is allowed

`_myName`

`TheCure`

`__AN$WE4_1S_42__`

`$bling$`

`me@home`

`strictfp` ?!

`49ers`

`side-swipe`

`Ph.D's`

Keywords

The following words are already used by the language and cannot be used as names:

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


Definition: *Variables*

Variables are buckets for values and have a specified type. *Variables need to be* declared *before first use.*

Book, on page 23

Variables

- Variables are *buckets* for a value
- Have a *data type* and a *name*
- The data type determines what kind of values are allowed in the variable

`int` x	`int` y	`float` f	`char` c
23	42	0.0f	'a'

Declaration in Java:

```
int x = 23, y = 42;
float f;
char c = 'a';
```

↑ Initialization

Definition: *Constants*

Constants are variables that are initialized upon declaration and may not change their value later on.

Book, on page 35

Constants

- Keyword `final`
- The value of the variable can be set exactly once

Example

```
final int maxSize = 100;
```

Hint: Always use `final`, unless the value actually needs to change over time.

Definition: *Types*

A Type defines a set of values that belong to the type as well as a set of operations that can be performed with the values of the type.

Book, on page 24

Definition: *Standard Types*

Java provides several predefined types for various numeric ranges as well as boolean values and strings.

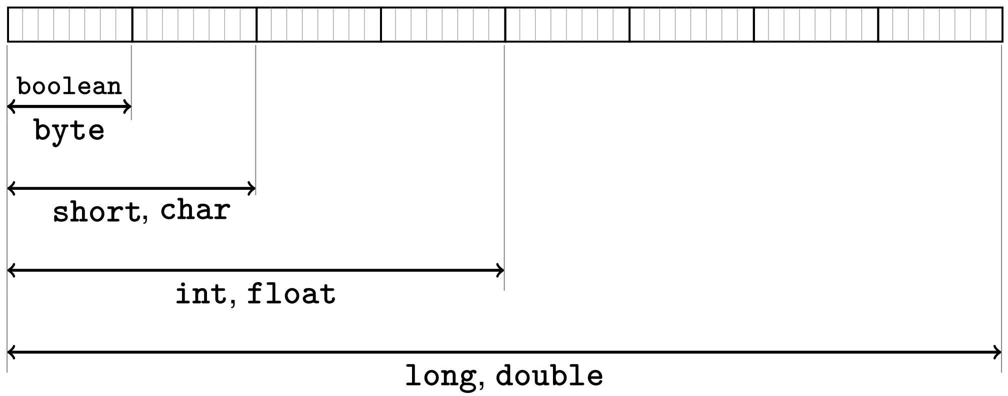
Book, on page 24

Standard Types

Data Type	Definition	Value Range	Initial Value
`byte`	8-bit integer	$-128, \ldots, 127$	0
`short`	16-bit integer	$-32'768, \ldots, 32'767$	0
`int`	32-bit integer	$-2^{31}, \ldots, 2^{31}-1$	0
`long`	64-bit integer	$-2^{63}, \ldots, 2^{63}-1$	`0L`
`float`	32-bit floating point	$\pm 1.4E^{-45}, \ldots, \pm 3.4E^{+38}$	`0.0f`
`double`	64-bit floating point	$\pm 4.9E^{-324}, \ldots, \pm 1.7E^{+308}$	`0.0d`
`boolean`	logical value	`true`, `false`	`false`
`char`	unicode-16 character	`'\u0000'`,...,`'a'`,`'b'`,...,`'\uFFFF'`	`'\u0000'`
`String`	string	∞	`null`

Types and Memory Usage

Reminder: Memory cells contain 1 Byte = 8 bit



Definition: *Literals*

Representation of a value of a standard type in the source code.

Book, on page 22 - 23

Literals: Integer Numbers

- Type `int` (or `short`, `byte`)

  `12` : value 12

  `-3` : value -3

- Type `long`

  `25_872_224L` : value $25'872'224$

  Hint: Underscores between digits are allowed!

Literals: Floating Point Numbers

are different from integers by providing

- decimal comma
  - `1.0` : type **`double`**, value 1
  - `1.27f` : type **`float`**, value 1.27
- and / or exponent.
  - `1e3` : type **`double`**, value 1000
  - `1.23e-7` : type **`double`**, value $1.23 \cdot 10^{-7}$
  - `1.23e-7f` : type **`float`**, value $1.23 \cdot 10^{-7}$

1.23e-7f

integer part

fractional part

exponent

Literals: Characters and Strings

- Individual characters:
  - `'a'` : Type **`char`**, value 97
- Strings:
  - `"Hello There!"` : Type **`String`**
  - `"a"` : Type **`String`**

Mind: Characters and Strings are two different things!

Character: In ASCII Table

0	<NUL>	32	<SPC>	64	@	96	`	128	Ä	160	†	192	¿	224	‡
1	<SOH>	33	!	65	A	97	a	129	Å	161	°	193	¡	225	·
2	<STX>	34	"	66	B	98	b	130	Ç	162	¢	194	¬	226	‚
3	<ETX>	35	#	67	C	99	c	131	É	163	£	195	√	227	„
4	<EOT>	36	$	68	D	100	d	132	Ñ	164	§	196	ƒ	228	‰
5	<ENQ>	37	%	69	E	101	e	133	Ö	165	•	197	≈	229	Â
6	<ACK>	38	&	70	F	102	f	134	Ü	166	¶	198	∆	230	Ê
7	<BEL>	39	'	71	G	103	g	135	á	167	ß	199	«	231	Á
8	<BS>	40	(	72	H	104	h	136	à	168	®	200	»	232	Ë
9	<TAB>	41	)	73	I	105	i	137	â	169	©	201	…	233	È
10	<LF>	42	*	74	J	106	j	138	ä	170	™	202		234	Í
11	<VT>	43	+	75	K	107	k	139	ã	171	´	203	À	235	Î
12	<FF>	44	,	76	L	108	l	140	å	172	¨	204	Ã	236	Ï
13	<CR>	45	-	77	M	109	m	141	ç	173	≠	205	Õ	237	Ì
14	<SO>	46	.	78	N	110	n	142	é	174	Æ	206	Œ	238	Ó
15	<SI>	47	/	79	O	111	o	143	è	175	Ø	207	œ	239	Ô
16	<DLE>	48	0	80	P	112	p	144	ê	176	∞	208	–	240	
17	<DC1>	49	1	81	Q	113	q	145	ë	177	±	209	—	241	Ò
18	<DC2>	50	2	82	R	114	r	146	í	178	≤	210	“	242	Ú
19	<DC3>	51	3	83	S	115	s	147	ì	179	≥	211	”	243	Û
20	<DC4>	52	4	84	T	116	t	148	î	180	¥	212	‘	244	Ù
21	<NAK>	53	5	85	U	117	u	149	ï	181	µ	213	’	245	ı
22	<SYN	54	6	86	V	118	v	150	ñ	182	∂	214	÷	246	ˆ
23	<ETB>	55	7	87	W	119	w	151	ó	183	∑	215	◊	247	˜
24	<CAN>	56	8	88	X	120	x	152	ò	184	∏	216	ÿ	248	¯
25	<EM>	57	9	89	Y	121	y	153	ô	185	π	217	Ÿ	249	˘
26	<SUB>	58	:	90	Z	122	z	154	ö	186	∫	218	⁄	250	˙
27	<ESC>	59	;	91	[	123	{	155	õ	187	ª	219	€	251	˚
28	<FS>	60	<	92	\	124	|	156	ú	188	º	220	‹	252	¸
29	<GS>	61	=	93	]	125	}	157	ù	189	Ω	221	›	253	˝
30	<RS>	62	>	94	^	126	~	158	û	190	æ	222	ﬁ	254	˛
31	<US>	63	?	95	_	127	<DEL>	159	ü	191	ø	223	ﬂ	255	ˇ

Definition: *Assignments*

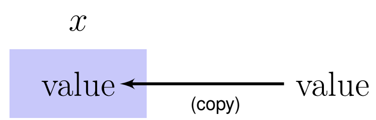
An assignment is used to store a (computed) value into a variable.

Book, on page 27

Value Assignment

Copies a value into variable x

- In pseudo code: $x \leftarrow \text{value}$
- In Java: x = value



"=" is the assignment operator *and not a comparison!*

Therefore, `int y = 42` is both a declaration + an assignment.

Value Assignment

Examples

```java
int a = 3;
double b;

b = 3.141;

int c = a = 0;

String name = "Inf";
```

A *nested* assignment:
The expression `a = 0` stores the value `0` into variable `a`. *and then returns the value*

Definition: *Arithmetic Expressions*

An arithmetic expression consists of operands and operators and computes a numeric value of a given type.

Book, on page 28

Arithmetic Binary Operators

Infix notation: $x \text{ op } y$ with the following operators

op: + − ∗ / %

modulo (↑ %)

- **Precedence**: Multiplication, division, and modulo first, then addition and subtraction
- **Associativity**: Evaluation from left to right

Arithmetic Binary Operators

- Division x / y: Integer division if x and y are integer.
- Division x / y: Floating-point division if x *or* y is a floating-poing number!

Examples

Integer division and modulo

- `5 / 3` evaluates to `1`   `−5 / 3` evaluates to `−1`
- `5 % 3` evaluates to `2`   `−5 % 3` evaluates to `−2`

Arithmetic Assignment

```
x = x + y
   ⇕
 x += y
```

Analogous for $-$, $*$, /, %

Examples:

```
x -= 3;      // x = x - 3
name += "x"  // name = name + "x"
num *= 2;    // num = num * 2
```


Arithmetic Unary Operators

Prefix notation: + x or $-$ x

Precedence: Unary operators bind stronger than binary operators

Examples

Assuming `x` is 3

- `2 * −x` evaluates to `−6`
- `−x − +1` evaluates to `−4`

Increment/Decrement Operators

Increment operators `++x` and `x++` have the same effect: $x \leftarrow x + 1$.But different return values:

- *Prefix operator* `++x` returns the *new* value:
  `a = ++x;` $\iff$ `x = x + 1; a = x;`
- *Postfix operator* `x++` returns the *old* value:
  `a = x++;` $\iff$ `temp = x; x = x + 1; a= temp;`

Precedence: Increment and decrement operators bind stronger than unary operators

Analogous for `x−−` and `−−x`.

Increment/Decrement Operators

Examples

Assuming `x` is initially set to 2

- `y = ++x * 3` evaluates to: `x` is 3 and `y` is 9
- `y = x++ * 3` evaluates to: `x` is 3 and `y` is 6

Expressions

- represent *computations*
- are either *primary*
- or *composed* . . .
- . . . from other expressions, using *operators*
- are statically typed

Analogy: Construction kit

Expressions

Examples

primary: "`-4.1d`" or "`x`" or `"Hi"`

composed: "`x + y`" or "`f * 2.1f`"

The type of "`12 * 2.1f`" is `float`

Celsius to Fahrenheit

```
public class Main {

    public static void main(String[] args) {
        Out.print("Celsius: ");
        int celsius = In.readInt();
        float fahrenheit = 9 * celsius / 5 + 32;
        Out.println("Fahrenheit: " + fahrenheit);
    }
}
```

Example: 15° Celsius are 59° Fahrenheit

Celsius to Fahrenheit - Analysis

```
9 * celsius / 5 + 32
```

- Arithmetic expression,
- contains three literals, one variable, three operator symbols

Where are the brackets in this expression?

Rule 1: Precedence

Multiplicative operators (*, /, %) have a higher precedence ("bind stronger") than additive operators (+, -).

Example

`9 * celsius / 5 + 32`

means

`(9 * celsius / 5) + 32`

Rule 2: Associativity

Arithmetic operators (*, /, %, +, -) are left-associative: in case of the same precedence, the evaluation happens from left to right.

Example

`9 * celsius / 5 + 32`

means

`((9 * celsius) / 5) + 32`

Rule 3: Arity

Unary operators +, - before binary operators +, -.

Example

`9 * celsius / + 5 + 32`

means

`9 * celsius / (+5) + 32`

Bracketing

Any expression can be bracketed unambiguously using the

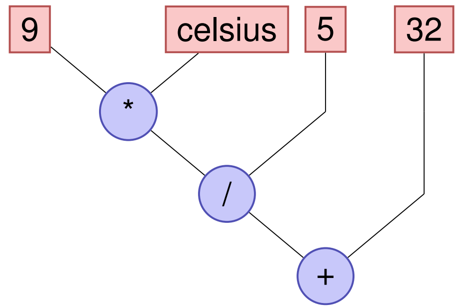
- associativities
- precedences
- arities (number of operands)

of the involved operators.

Expression Trees

Bracketing leads to an expression tree

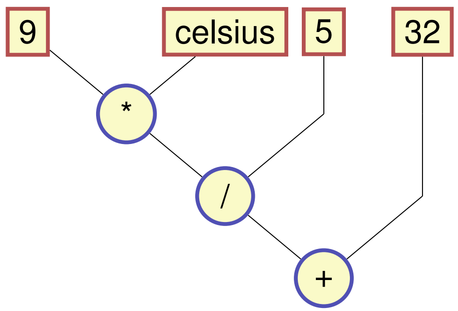
```
(((9 * celsius) / 5) + 32)
```



Evaluation Order

“From leafs to the root” in the expression tree

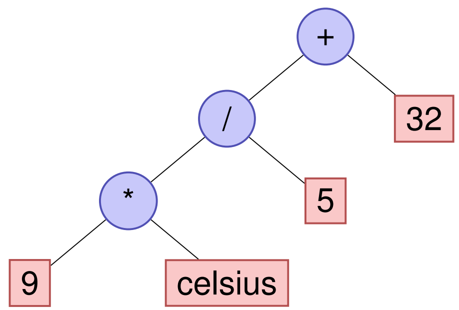
```
9 * celsius / 5 + 32
```



Expression Trees – Notation

Usual notation: root on top

```
9 * celsius / 5 + 32
```



Definition: *Type System*

A type system is a set of rules that are applied to the different constructs of the language.

Book, on page 24

Type System

Java festures a *static* type system:

- All types must be declared
- If possible, the compiler checks the typing . . .
- . . . otherwise it's checked at run-time

Advantages of a static type system

- *Fail-fast* Bugs in the program are often found already by the compiler
- Understandable code

Type errors

Example

```
int pi_ish;
float pi = 3.14f;

pi_ish = pi;
```

Compiler error:

```
./Root/Main.java:12: error: incompatible types: possible lossy conversion from float to int
        pi_ish = pi;
                 ^
```


Explicit Type Conversion

Example

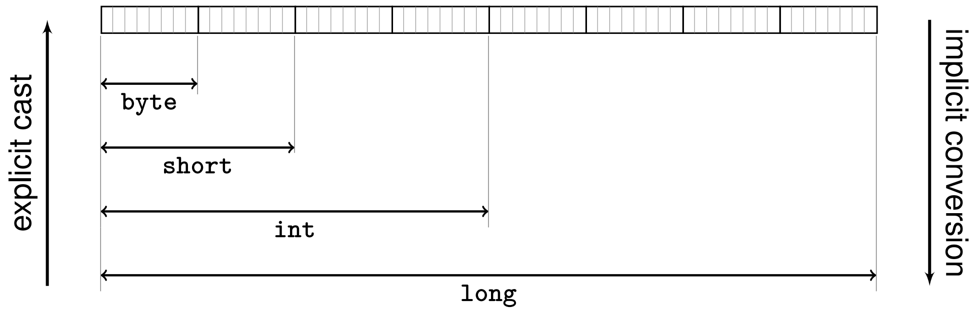
```
int pi_ish;
float pi = 3.14f;

pi_ish = (int) pi;
```

Explicit type conversion using casts (type)

- Statically type-correct, compiler is happy
- Run-time behavior: depends on the situation
  Here: loss of precision: 3.14 ⇒ 3
- Can crash a program at run-time

Type Conversion - Visually for Integer Numbers



Potential loss of information when casting explicitly, because less memory available to represent the number

Definition: *Mixed Expressions*

A mixed expression consists of operands of different types.

Book, on page 70

Mixed Expressions, Conversion

- Floating point numbers are more general than integers.
- In mixed expressions integers are converted to floating point numbers.

```
9 * celsius / 5  + 32
```


Type Conversions for Binary Operations

Numeric operands in a binary operation are being converted according to the following rules:

- If both operands have the same type, no conversion will happen
- If one operand is `double`, the other operand is converted to `double` as well
- If one operand is `float`, the other operand is converted to `float` as well
- If one operand is `long`, the other operand is converted to `long` as well
- Otherwise: Both operands are being converted to `int`