3 Java as an example of a simple programming language

From Bits to Numbers

- A computer represents all data in binary form (using transistors, i.e., electrical switches, values: on/off)
- Interpretation: switch on $=$ **1**, switch off $=$ **0**
- Instead of single bits, groups of bits are read and set
  - 1 bit = 1 'switch', 1 byte = 8 bits
- The main memory of a computer is a list of bytes, e.g.,:

address	...	**40**	**41**	**42**	**43**	...
content	...	**01100111**	**00010010**	**00000000**	**00000000**	...

- Recent CPUs usually process 32 bit or 64 bit at once.
- Memory accesses thus always start addresses divisibe by 4 (or 8)
- In the example: 32bit-access to address 40 yields the bit group

00000000 00000000 00010010 01100111

(usually shown with descending addresses ('little endian'))

To ease readability:

- groups of 4 bits are often represented by a single character ('nibble', 'half byte') and denoted 'hexadecimally'
-

nibble	**0000**	**0001**	**0010**	**0011**	**0100**	**0101**	**0110**	**0111**
hexadecimal	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
decimal value	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**

nibble	**1000**	**1001**	**1010**	**1011**	**1100**	**1101**	**1110**	**1111**
hexadecimal	**8**	**9**	***A***	***B***	***C***	***D***	***E***	***F***
decimal value	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

- Example: **00000000 00000000 00010010 01100111**
  hexadecimal representation **00 00 12 67**
- To indicate hexadecimal representation, Java uses a leading **0*x***, e.g., '`a=0x00001267`'
- Less frequently used: groups of 3 bits, 'octal' representation, in Java indicated by leading 0,
  e.g.,: '`a=0715`' for the octal number '**111 001 101**'
  (decimal value: $\mathbf{7 \cdot 8^2 + 1 \cdot 8 + 5 = 461}$)

Binary representation of natural numbers:

- Numbers are usually represented with 32 or 64 bits:
  Example: the bit pattern `0x00001267`, binary:

0 0 0 0 0 0 0 0	0 0 0 0 0 0 0 0	0 0 0 1 0 0 1 0	0 1 1 0 0 1 1 1
31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

  can be interpreted as a (binary) number as follows :

  $$\begin{aligned} & 1 \cdot 2^{12} + 1 \cdot 2^9 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\ = \; & 4096 + 512 + 64 + 32 + 4 + 2 + 1 \\ = \; & 4711 \end{aligned}$$

- alternatively: evaluate **0*x*00001267** directly (with base 16):

  $$\begin{aligned} & 1 \cdot 16^3 + 2 \cdot 16^2 + 6 \cdot 16^1 + 7 \cdot 16^0 \\ = \; & 4096 + 512 + 16 + 7 = 4711 \end{aligned}$$

Binary representation of natural numbers/integers:

- The k-bit pattern $b_{k-1} \ldots b_2 b_1 b_0$ represents the **natural number**

$$\sum_{i=0}^{k-1} b_i \cdot 2^i$$

- Usually, however, the '**two's complement** representation is used, where $b_{k-1} \ldots b_2 b_1 b_0$ corresponds to the **integer**

$$\sum_{i=0}^{k-2} b_i \cdot 2^i \quad - b_{k-1} \cdot 2^{k-1}$$

- Examples for $k = 8$, i.e., interpretation of a byte as an integer:

byte	computation	value
0 000 0000	$0 - 0$	**0**
0 000 0001	$1 - 0$	**1**
0 000 0101	$5 - 0$	**5**
0 101 0101	$85 - 0$	**85**
0 111 1111	$127 - 0$	**127**

byte	computation	value
1 000 0000	$0 - 128$	-128
1 000 0001	$1 - 128$	-127
1 000 0101	$5 - 128$	-123
1 101 0101	$85 - 128$	-43
1 111 1111	$127 - 128$	-1

Primitive Java data types for integers

With k bits one can represent

- negative integers from -2^{k-1} to -1,
- positive integers from 1 to $2^{k-1}-1$, (asymmetric!)
- and the 0.

Java uses two's complement for integers, with the following number ranges:

Java type			minimal value	maximal value
`byte`	1 byte	8 bit	-128	127
`short`	2 bytes	16 bits	-32768	32767
`int`	4 bytes	32 bits	-2147483648	2147483647
`long`	8 bytes	64 bits	-9223372036854775808	9223372036854775807

Remark: $$\sum_{i=0}^{k-2} b_i \cdot 2^i \;-\; b_{k-1} \cdot 2^{k-1} = \sum_{i=0}^{k-1} b_i \cdot 2^i \;-\; b_{k-1} \cdot 2^k$$

Therefore a compute can use the same(!) hardware for addition/subtraction/multiplication of integers and for natural numbers!

- Attention: the computer uses only the last bits of a number, all other bits are **discarded without warning**!
- This corresponds to computing modulo $\mathbf{2^k}$ (with sign according to two's complement)
- Example: addition of the decimal values **1 000 000 000** and **2 000 000 000** with `int`, i.e., $\boldsymbol{k = 32}$:

$$\begin{array}{rl} & 00111011\ 10011010\ 11001010\ 00000000 \\ + & 01110111\ 00110101\ 10010100\ 00000000 \\ = & 10110010\ 11010000\ 01011110\ 00000000 \end{array}$$

  The result is **negative** (two's complement!) with value

$$\mathbf{3\,000\,000\,000 - 2^{32} = -1\,294\,967\,296}$$

- The programmer must take care that no such '**overflow**' happens!

Conversion rules for integers in Java:

- Java has integer types **byte**, **short**, **int**, and **long**
- Numbers are always stored in two's complement!
- **byte** and **short** numbers are always converted to **int** before any computation!
- However, results are not automatically converted to **byte** and **short**!

```
    byte b1,b2;
    b1=101;  b2=102;

    int  i;
    i = b1 + b2;
    System.out.println("as int: " + i);  // result: 203

    byte b;
//  b = b1 + b2;          // wrong! -> compiler indicates error
    b = (byte) (b1 + b2); //  ok with 'cast' (see later)
    System.out.println("as byte: " + b);  // result: -53
```

Identifying and dealing with integer overflows is a task of the programmer!

3 Java as an example of a simple programming language

property	32 bit / `float`	64 bit / `double`
greatest positive (finite) number	$\mathbf{2^{128} - 2^{104} \approx 2^{128} \approx 3.4 \cdot 10^{38}}$	$\mathbf{2^{1024} - 2^{971} \approx 2^{1024} \approx 1.8 \cdot 10^{308}}$
smallest possible normalized number	$\mathbf{2^{-126} \approx 1.2 \cdot 10^{-38}}$	$\mathbf{2^{-1022} \approx 2.2 \cdot 10^{-308}}$
smallest possible denormalised number	$\mathbf{2^{-150} \approx 7 \cdot 10^{-46}}$	$\mathbf{2^{-1075} \approx 4.94 \cdot 10^{-324}}$

Floating point numbers as primitive data types:

```
public class K3B17E_double_float {
  public static void main(String[] args) {
    float  xf;
    double xd;
    xf = 123456.78901234567890f;
    xd = 0.01234567890123456789 0d;
    System.out.println( xf );
    System.out.println( xd );
    System.out.println( 0.12345e-5 );
    System.out.println( 1.0e200 * 1.0e200 );
    System.out.println(  1.0 / 0.0 );
    System.out.println( -1.0 / 0.0 );
    System.out.println(  0.0 / 0.0 );
  }
}
```

```
123456.79
0.012345678901234568
1.2345E-6
Infinity
Infinity
-Infinity
NaN
```

Comparing **float** (or **double**) numbers is dangerous:

```java
public class K3B18E_inexact {
  public static void main(String[] args) {

    float xf = 1.0f, yf = 1.0f;
    xf = xf / 3 * 100000 * 3 / 100000;  // value = 1 , or...?
    System.out.print ( "xf: " + xf + "\nyf: " + yf + "\n");
    if (xf == yf) System.out.println ("xf and yf are equal }");
    else          System.out.println ("xf and yf are not equal");

    double xd = 1.0, yd = 1.0;
    xd = xd / 3 * 100000 * 3 / 100000;  // value = 1 , or...?
    System.out.print ( "xd: " + xd + "\nyd: " + yf + "\n");
    if (xd == yd) System.out.println ("xd and yd are equal");
    else          System.out.println ("xd and yd are not equal");
  }
}
```

```
xf: 1.0000001
yf: 1.0
xf and yf are not equal
xd: 0.9999999999999999
yd: 1.0
xd and yd are not equal
```

‘Tolerant’ comparison of **float** numbers (or **double** numbers)

```java
public class K3B19E_Tolerance {
  public static void main(String[] args) {

    double x = 1.0, y = 1.0, tolerance = 0.001;

    x = x / 3 * 100000 * 3 / 100000;

    System.out.print ("x: " + x + "\ny: " + y + "\n");
    if (Math.abs (x - y) < tolerance) // absolute value of (x—y)
      System.out.println("x and y are almost equal");
    else
      System.out.println("x and y are probably not equal");
  }
}
```

```
x: 0.9999999999999999
y: 1.0
x and y are almost equal
```

3 Java as an example of a simple programming language

- Programming languages
- Lexical structure of Java programs
- Variable – Name – Value – Type
- Literals and Constants
- Truth values
- Assignments and Expressions
- Basic control structures in Java
- Integer Numbers
- Floating Point Numbers
- Characters and Strings
- Additional Operators
- Methods

- Purpose of the type **`char`**: storing single characters
- character set: Unicode,
  represented as groups of 2 bytes
- characters can be given in hexadecimal form `\u0000` to `\uFFFF` (anywhere in the source code!)
- 'readable' character literals with apostroph (`'`)

in source code	meaning
`char character = 'a'`	`\u0061`, ASCII, printable
`char percent = '%'`	`\u0025`, ASCII, printable
`char newline = '\n'`	`\u000A`, ASCII 'control character'
`char c_return = '\r'`	`\u000D`, ASCII 'control character'
`char backslash = '\\'`	`\u005C`, ASCII, printable
`char quote = '\"'`	`\u0060`, ASCII, printable
`char omega = '\u03a9'`	**Ω**, Unicode hexadecimal
`char sigma = 'Σ'`	**Σ**, Unicode direct

- terminology:
  `\n`, `\r`, `\\`, `\'` etc. are called 'escape sequences'
  `\u03a9` etc. are called 'Unicode escape sequences

- first 128 characters `\u0000` to `\u007F` in the Unicode charset:

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	*nul*	*soh*	*stx*	*etx*	*eot*	*enq*	*ack*	*bel*	*bs*	*ht*	*nl*	*vt*	*ff*	*cr*	*so*	*si*
1	*dle*	*dc1*	*dc2*	*dc3*	*dc4*	*nak*	*syn*	*etb*	*can*	*em*	*sub*	*esc*	*fs*	*gs*	*rs*	*us*
2	*sp*	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	U	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	*del*

- `\u0000` to `\u007F` are exactly the first 127 characters in the 7bit '*ASCII*' charset (*American Standard Code for Information Interchange*, 1963)...
- ... and also in the 8bit '*ISO LATIN 1*' charset (ISO-8859-1)

- Character string literals are assigned to the type **`String`**.
- **`String`** is not a primitive data type, but can partly be used like one (later more on the class **`String`**).

source code	meaning
`"This is a string"`	String with 16 characters
`"this"+"string"`	concatenated String
`"\nString"`	String with linefeed at the beginning
`""`	empty String
`"\""`	String that contains only the character `"`

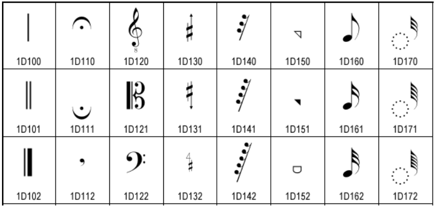
Example program:

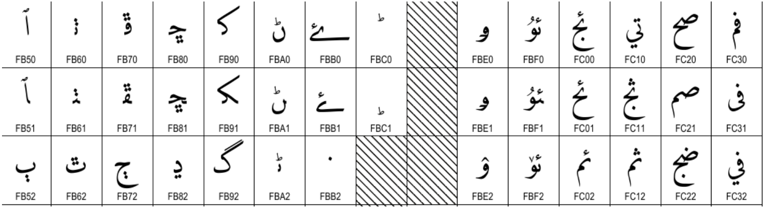
```
String chain = "This is a String";
int i = 27;
chain = chain + "␣of length " + i;
/*    i is automatically converted           */
/*    to a String                            */
/*    value of chain: This is a String of length 27 */
```

To allow for more than $\mathbf{2^{16} = 65536}$ different characters in a String:

- After a `char` from `\uD800` – `\uDBFF` in a String only a `char` from `\uDC00` - `\uDFFF` can follow
- These four bytes are always interpreted together!
- binary: values from `\uD800` – `\uDBFF`: `1101 10xx xxxx xxxx`
- binary: values from `\uDC00` – `\uDFFF`: `1101 11xx xxxx xxxx`
- The first 5 bits `1101 1` indicate if this is a part of a ‘surrogate’ pair!
- This yields 20 bits for up to **1048576** characters
- Example: Unicode character `U+1D160`
  binary: `0001 1101 0001 0110 0000`
  split in two groups of 10 bits: `00 0111 0100`, `01 0110 0000`
  surrogate pair: `\uD874 \uDD60`
- More information on Unicode at `http://www.unicode.org/`

Examples for Unicode tables:





Increment and Decrement

expression	operation	value of the expression
`count++`	add 1	= value *before* the addition
`++count`	add 1	= value *after* the addition
`count--`	subtract 1	= value *before* the subtraction
`--count`	subtract 1	= value *after* the subtraction

```
int x = 0, count = 0;
x = count++;        //  after instruction:  x: 0, count: 1
x = ++count;        //  after instruction:  x: 2, count: 2
x = count--;        //  after instruction:  x: 2, count: 1
x = --count;        //  after instruction:  x: 0, count: 0
```

Examples:

-
  ```
  x = i++
  ```
  is equivalent to
  ```
  x = i; i= i+1;
  ```

-
  ```
  x = ++i
  ```
  is equivalent to
  ```
  i= i+1; x = i;
  ```

-
  ```
  if (i ++ == 1) sequence
  ```
  is equivalent to
  ```
  if (i==1) {i=i+1; sequence} else {i= i+1;}
  ```

What does the following program?:

```
public class K3B20E_Increment {
   public static void main(String[] args) {

      int x = 0, count = 0;

      ++count;

      x = 5*(count++) + count--;
      //allowed, but bad style...

   System.out.println("x: " + x + "  count: "+count);
   }
}
```

```
> java K3B20E_Inkrement
x: 7  count: 1
```

Assignment operators

operator	meaning	example	equivalent to
`=`	assignment	`x=y`	
`+=`	addition, then assignment	`x+=y`	`x=x+y`
`+=`	concatenation, then assignment	`x+=y`	`x=x+y`
`-=`	subtraction, then assignment	`x-=y`	`x=x-y`
`*=`	multiplication, then assignment	`x*=y`	`x=x*y`
`/=`	division, then assignment	`x/=y`	`x=x/y`
`%=`	remainder, then assignment	`x%=y`	`x=x%y`

Similar to increment and decrement, assignment operators are only shortcut notations (in the style of `C` and `C++`).

Conditional operator

- Syntax:

```
condition ? expression1 : expression2
```

- Semantics: comparable to

```
if (condition) sequence1 else sequence2
```

  but at the level of expressions!
- The conditional operator is (only) meaningful in expressions.
- **x = (y>0 ? y : −y)** operates as

```
if (y>0) x = y; else x = −y;
```

- **x = y + (z > y ? 2 * y + z : y + z)** operates as

```
if (z > y)
   w = 2 * y + z;
else
   w = y + z;
x = y + w;
```

Bit operators

- Read and write access to single bits.
- Defined only for integer types and char.

Java operator	spoken as	precedence	order
unary, i.e., one operand			
$\sim$	bitwise 'not'	1	***R*** $\rightarrow$ ***L***
binary, i.e., two operands			
`&`	bitwise 'AND'	3	***L*** $\rightarrow$ ***R***
`|`	bitwise 'OR'	5	***L*** $\rightarrow$ ***R***
$\hat{}$	bitwise 'XOR'	4	***L*** $\rightarrow$ ***R***

- Operations on individual bits analogously to logic truth tables:

`a`	`~a`
`1`	`0`
`0`	`1`

`a`	`b`	`a&b`	`a|b`	`a^b`
`1`	`1`	`1`	`1`	`0`
`1`	`0`	`0`	`1`	`1`
`0`	`1`	`0`	`1`	`1`
`0`	`0`	`0`	`0`	`0`

- **`a^b`** named as ‘exclusive or’
- this results, for example, in operations on bytes:

`A`	`00110011`
`B`	`01010101`
`~A`	`11001100`
`A&B`	`00010001`
`A|B`	`01110111`
`A^B`	`01100110`

Example: conversion of numbers into binary form using bit operators

```
public class K3B21E_Binary {
   public static void main(String[] args) {

      byte a, b = (byte) 1, c;
      a = (byte) Integer.parseInt(args[0]);

      System.out.println("decimal: " + a);

      String binary= "";
      while (true) {
         c = (byte) (a & b);
         if (c != 0) binary = "1" + binary;
         else        binary = "0" + binary;
         if (b == -128) break;
         b = (byte) (b * 2);
      }

      System.out.println ("binary:   "+binary);
   }
}
```

`b` serves as a mask to access the individual bits of `a` from right to left.

Example: execution of the loop for $\boldsymbol{a = 52}$:

```
while (true) {
   c = (byte) (a & b);
   if (c != 0) binary = "1" + binary;
   else        binary = "0" + binary;
   if (b == -128) break;
   b = (byte) (b * 2);
}
```

step	a	b	c	binary
1	00110100	00000001	00000000	0
2	00110100	00000010	00000000	00
3	00110100	00000100	00000100	100
4	00110100	00001000	00000000	0100
5	00110100	00010000	00010000	10100
6	00110100	00100000	00100000	110100
7	00110100	01000000	00000000	0110100
8	00110100	10000000	00000000	00110100

Shift operators

- 'shifting' access to a group of bits.
- defined only for integer types and char, together with a natural number

Java operator	spoken as	precedence
`<<`	left shift	2
`>>`	right shift with sign	2
`>>>`	right shift with zero fill	2

shift operations on bytes:

`bits`	`00110110`
`bits << 1`	`01101100`
`bits >> 1`	`00011011`
`bits >> 5`	`00000001`
`bits >>> 1`	`00011011`
`bits >>> 5`	`00000001`

`bits`	`10011011`
`bits << 1`	`00110110`
`bits >> 1`	`11001101`
`bits >> 5`	`11111100`
`bits >>> 1`	`01001101`
`bits >>> 5`	`00000100`

- With ‘`>>`’ the sign is preserved,
- With ‘`>>>`’ negative numbers turn positive...

Example for shift operators on **`int`**:

```
public class K3B22E_ShiftInt {
   public static void main(String[] args) {

      int a, b, c, d;
      a = Integer.parseInt(args[0]);


      b = a << 2;
      c = a >> 2;
      d = a >>> 2;

      System.out.println
                ("a:␣␣␣␣␣␣␣" + a
              + "\na␣<<␣2:␣␣" + b
              + "\na␣>>␣2:␣␣" + c
              + "\na␣>>>␣2:␣" + d);
   }
}
```

for example with input $\boldsymbol{a = -17}$:

binary	decimal
1111 .... 1110 1111	−17
1111 .... 1011 1100	−68
1111 .... 1111 1011	−5
0011 .... 1111 1011	1073741819

Example for shift operators on **`char`** (unsigned!):

```
public class K3B23E_ShiftChar {
   public static void main(String[] args) {

      char ch;  short a, b, c, d;
      a = (short) Integer.parseInt(args[0]);

      ch = (char) a;
      b = (short) (ch << 2);
      c = (short) (ch >> 2);
      d = (short) (ch >>> 2);

      System.out.println
                  ( "a:␣␣␣␣␣␣␣␣␣" + a
                + "\nch␣<<␣2␣␣:␣" + b
                + "\nch␣>>␣2␣␣:␣" + c
                + "\nch␣>>>␣2␣:␣" + d);
   }
}
```

for example with input $a = -17$:

binary	decimal
1111 1111 1110 1111	-17
1111 1111 1011 1100	-68
0011 1111 1111 1011	16379
0011 1111 1111 1011	16379

Bit operators **&** and **|** can also be applied to the type **boolean**.

```
public class K3B24E_BitOp {
   public static void main(String[] args) {

      int i = 3, j = 2;
      if ((++i < ++j) && (i++ > j++))
         {}
      else
         System.out.println("&&␣:␣i:␣" + i + "␣j:␣" + j);

      i = 3; j = 2;
      if ((++i < ++j) & (i++ > j++))
         {}
      else
         System.out.println("&␣␣:␣i:␣" + i + "␣j:␣" + j);
   }
}
```

- With **&&** the execution terminates as soon as the result is fixed!
- With **&** the expression is always evaluated completely!
- (see later, when we have discussed methods...)