Programmieren II

Arrays & Control Structures

Alexander Fraser

fraser@cl.uni-heidelberg.de

(Material from T. Bögel, K. Spreyer, S. Ponzetto, M. Hartung)

April 30, 2014

Outline

Outline

Last week

Types and variables

- Strong & static typing
- (Primitive) data types for numbers, characters and truth values
- Declaring variables
- Assigning values to variables
- Arithmetic operations

Java syntax

- Syntax for a class definition
- Signature of the main method
- Printing to the standard output
- Compiling and running a java program

Brief recap

Java's primitive data types

- Numbers
  - Whole numbers: `byte`, `short`, `int`, `long`
  - Floating point numbers: `float`, `double`
- Characters: `char`[a]
- Truth values: `boolean`

[a] chars are actually numbers. Conceptually, however, they represent characters – at least for us in this course.

Brief recap

Variables

`type name [ = expr`$_{\text{type}}$ `];`

- Widening conversion: `name`$_{\text{greater\_type}}$ `=` `expr`$_{\text{smaller\_type}}$`;`
- Strong typing (cast):
  `name`$_{\text{smaller\_type}}$ `= (smaller type)` `expr`$_{\text{greater\_type}}$`;`
- Static typing: once a type, always a type!

Operators

$i++$ means $i = i + 1$
$i+ = 2$ means $i = i + 2$

Today

Overview

- Reference types (1): arrays
- Code in detail: recap & control structures
- (Control structures – formal)
- Multi-dimensional arrays

- Next time: Reference types (2): classes!

Outline

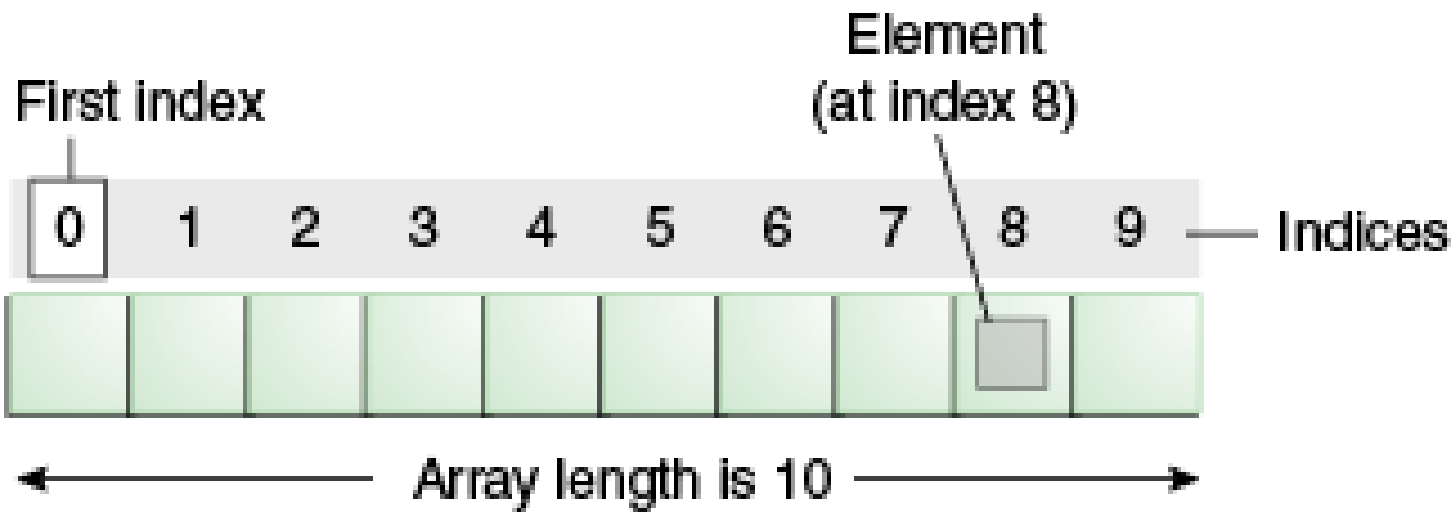
Arrays

What is an array?

- Arrays are containers for multiple values
  - ordered: (like a list in python)
  - uniform: all elements need to be of the same type! (unlike python)
- In Java:

```
int[] intArray;         /* could contain 2, 23, 42,
                           and 432,999 */
String[] stringArray; // could contain “Bruce” and “Wayne”
String[] args;        // look familiar?
```

Graphical illustration



source: `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/arrays.html`

Arrays

How to create arrays?

- Arrays are of a fixed size
  - is specified upon initialization
  - length stored in `length` field variable

```
int importantNumbers[] = new int[4];    // array for 4 elems

// init only legal in declaration statements!!
String[] progLanguages = { “Java”, “Python”, “$PHP” };

System.out.println( progLanguages.length );  // prints 3
```

Arrays

How to create arrays?

- Arrays are of a fixed size
  - is specified upon initialization
  - length stored in `length` field variable

```
int importantNumbers[] = new int[4];    // array for 4 elems

// init only legal in declaration statements!!
String[] progLanguages = { “Java”, “Python”, “$PHP” };

System.out.println( progLanguages.length );  // prints 3
```

Arrays

How to create arrays?

- Arrays are of a fixed size
  - is specified upon initialization
  - length stored in `length` field variable

```
int importantNumbers[] = new int[4];    // array for 4 elems

// init only legal in declaration statements!!
String[] progLanguages = { “Java”, “Python”, “$PHP” };

System.out.println( progLanguages.length );  // prints 3
```

Arrays

How to create arrays?

- Arrays are of a fixed size
  - is specified upon initialization
  - length stored in `length` field variable

```
int importantNumbers[] = new int[4];    // array for 4 elems

// init only legal in declaration statements!!
String[] progLanguages = { “Java”, “Python”, “$PHP” };

System.out.println( progLanguages.length );  // prints 3
```

Arrays

How to create arrays?

- Arrays are of a fixed size
  - is specified upon initialization
  - length stored in `length` field variable

```
int importantNumbers[] = new int[4];    // array for 4 elems

// init only legal in declaration statements!!
String[] progLanguages = { “Java”, “Python”, “$PHP” };

System.out.println( progLanguages.length );  // prints 3
```

Arrays

How to create arrays?

- Arrays are of a fixed size
  - is specified upon initialization
  - length stored in `length` field variable

```
int importantNumbers[] = new int[4];    // array for 4 elems

// init only legal in declaration statements!!
String[] progLanguages = { “Java”, “Python”, “$PHP” };

System.out.println( progLanguages.length );  // prints 3
```

Arrays

Accessing array elements

- Elements are indexed by their position within an array
  - starting with 0 (not 1; cf. python's lists)
  - first index: 0, last index: `length-1`
- Access elements with operator `[]`:

```
System.out.println( progLanguages[0] );  // prints “Java”

progLanguages[3];   // runtime exception!!

// [] can also be used in assignments
progLanguages[0] = “Whitespace”;
progLanguages[1] = “LOLCODE”;
```

Arrays

Accessing array elements

- Elements are indexed by their position within an array
  - starting with 0 (not 1; cf. python's lists)
  - first index: 0, last index: `length-1`
- Access elements with operator `[]`:

```
System.out.println( progLanguages[0] );  // prints “Java”

progLanguages[3];   // runtime exception!!

// [] can also be used in assignments
progLanguages[0] = “Whitespace”;
progLanguages[1] = “LOLCODE”;
```

Arrays

Accessing array elements

- Elements are indexed by their position within an array
  - starting with 0 (not 1; cf. python's lists)
  - first index: 0, last index: `length-1`
- Access elements with operator `[]`:

```
System.out.println( progLanguages[0] );  // prints "Java"

progLanguages[3];   // runtime exception!!

// [] can also be used in assignments
progLanguages[0] = "Whitespace";
progLanguages[1] = "LOLCODE";
```

Arrays

Accessing array elements

- Elements are indexed by their position within an array
  - starting with 0 (not 1; cf. python's lists)
  - first index: 0, last index: `length-1`
- Access elements with operator `[]`:

```
System.out.println( progLanguages[0] );  // prints “Java”

progLanguages[3];   // runtime exception!!

// [] can also be used in assignments
progLanguages[0] = “Whitespace”;
progLanguages[1] = “LOLCODE”;
```

Elements of a simple program

```
public class Multiply {
     public static void main( String[] args)  {
          int x = Integer.parseInt(args[0]);
          int y = Integer.parseInt(args[1]);
          int z = x * y;
          System.out.println(“ Result: ”+z);
      }
}
```

Elements of a simple program

```
public class Multiply {
     public static void main( String[] args)  {
          int x = Integer.parseInt(args[0]);
          int y = Integer.parseInt(args[1]);
          int z = x * y;
          System.out.println(“ Result: "+z);
      }
}
```

- Class name

Elements of a simple program

```
public class Multiply {
     public static void main( String[] args)   {
          int x = Integer.parseInt(args[0]);
          int y = Integer.parseInt(args[1]);
          int z = x * y;
          System.out.println(“ Result: "+z);
       }
}
```

- Class name
- Definition of the main method

Elements of a simple program

```
public class Multiply {
     public static void main( String[] args)  {
          int x = Integer.parseInt(args[0]);
          int y = Integer.parseInt(args[1]);
          int z = x * y;
          System.out.println(“ Result: ”+z);
      }
}
```

- Class name
- Definition of the main method
- Method name (main method is always called `main`)

Elements of a simple program

```
public class Multiply {
     public static void main( String[] args)  {
          int x = Integer.parseInt(args[0]);
          int y = Integer.parseInt(args[1]);
          int z = x * y;
          System.out.println(“ Result: ”+z);
      }
}
```

- Class name
- Definition of the main method
- Method name (main method is always called `main`)
- Data type of the return value (`void`, if no value is returned)

Elements of a simple program

```
public class Multiply {
     public static void main( String[] args)  {
          int x = Integer.parseInt(args[0]);
          int y = Integer.parseInt(args[1]);
          int z = x * y;
          System.out.println(“ Result: ”+z);
      }
}
```

- Class name
- Definition of the main method
- Method name (main method is always called `main`)
- Data type of the return value (`void`, if no value is returned)
- List of function parameters (form: `data_type variable (, data_type variable)*`);
  in this case: string-array as the only parameter

Elements of a simple program

```
public class Multiply {
     public static void main( String[] args)  {
          int x = Integer.parseInt(args[0]);
          int y = Integer.parseInt(args[1]);
          int z = x * y;
          System.out.println(“ Result: ”+z);
      }
}
```

- Class name
- Definition of the main method
- Method name (main method is always called `main`)
- Data type of the return value (`void`, if no value is returned)
- List of function parameters (form: `data_type variable (, data_type variable)*`);
  in this case: string-array as the only parameter
- Accessing the first element of the array `args`

Elements of a simple program

```
public class Multiply {
     public static void main( String[] args)  {
          int x = Integer.parseInt(args[0]);
          int y = Integer.parseInt(args[1]);
          int z = x * y;
          System.out.println(“ Result: ”+z);
      }
}
```

- Class name
- Definition of the main method
- Method name (main method is always called `main`)
- Data type of the return value (`void`, if no value is returned)
- List of function parameters (form: `data_type variable (, data_type variable)*`);
  in this case: string-array as the only parameter
- Accessing the first element of the array `args`
- Conversion from `Strings` to `Integer` (`Integer` is a wrapper for the primitive data type `int`)

Outline

Control structures

- At some point in your life as a programmer: desire to repeatedly execute statements under certain conditions...
- Control structures
  - Conditionals (selection, branching)
  - Loops (iteration)
- Control structures $\rightarrow$ complex statements

Control structures

Overview

- `if-else`
- Loops
  - `while`
  - `for`
  - (“for-each”)
- Others
  - `switch`
  - `do-while`
  - ternary operator `?:` (not a statement, but an assignment!)

if-else

if-else

selectively execute statements depending on conditions

Syntax

```
if ( expr )        ⇐ expr needs to be a boolean
     statement1
[ else
     statement2
]
```

Meaning

If `expr` is `true`, execute `statement1`, otherwise execute `statement2` (if there is an else branch)

Excursion: blocks

Blocks of statements

- Problem: execute multiple statements under certain conditions
- Use { and } to group statements to a block.
- Example:

```
if ( n >= 0 ) {
      a=n;
      System.out.println(a);
}
else System.out.println("negative");
```

if-else

if-else cascades

- Syntax:

```
if ( expr1 )
     statement1;
else if ( expr2)
     statement2;
else if ( expr3)
     statement3;
...
else
     statements;
```

- Expressions are evaluated in order. Execute corresponding statement if an expression is `true`.

Loops: `while`

Loops: multiple executions of statements

Syntax

```
while ( expr )      ⇐ expr needs to be a boolean
     statement;
```

Meaning

1. Condition `expr` is evaluated
2. If it is `true`, `statement` is executed repeatedly
3. Otherwise: exit the loop

Loops: while

Example

```
int i = 10;
while ( i > 0 ) {
    System.out.println( i );
    i--;
}
```

Loops: while

Example

```
int i = 10;
while ( i > 0 ) {
    System.out.println( i );
    i--;
}
```

Output:

```
10
9
[...]
1
```

Loops: for

Syntax

```
for ( init; expr; update)
     statement
```

Meaning

1. Execute assignment `init` (usually: declaration & assignment)
2. if `expr` is `true`
   a. Execute `statement`
   b. Execute `update`
   c. Jump back to step 2
3. Otherwise: exit the loop

Loops: for

Example

```
for ( int i = 10; i > 0; i-- ) {
    System.out.println( i );
}
```

Loops: for

Example

```
for ( int i = 10; i > 0; i-- ) {
    System.out.println( i );
}
```

Output (equivalent to previous example):

```
10
9
[...]
1
```

Loops: for

for loop with multiple variables

Multiple variables can be used in a single for loop:

```
for ( int a=1, b=10 ; a <= b; a++, b--) {
      System.out.println("a: " + a + " b: " + b);
}
```

Loops: for

for loop with multiple variables

Multiple variables can be used in a single for loop:

```
for ( int a=1, b=10 ; a <= b; a++, b--) {
      System.out.println("a: " + a + " b: " + b);
}
```

Output:

```
a: 1 b: 10
a: 2 b: 9
[...]
a: 5 b: 6
```

Jump statements

Jump statements

alter the execution flow within a loop

- `break` exits the surrounding loop immediately
- `continue` re-evaluates the condition of the loop (skips the remaining part of the loop)

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . . // in one
      while ( condition2 ) {
            . . . // in two
            break;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

`break` vs. `continue`

```
while ( condition1 ) {
      . . . // in one
      while ( condition2 ) {
            . . . // in two
            break;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . . // in one
      while ( condition2 ) {
            . . . // in two
            break;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

`break` vs. `continue`

```
while ( condition1 ) {
      . . . // in one
      while ( condition2 ) {
            . . . // in two
            break;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . . // in one
      while ( condition2 ) {
            . . . // in two
            break;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . . // in one
      while ( condition2 ) {
            . . . // in two
            break;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . . // in one
      while ( condition2 ) {
            . . . // in two
            break;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . . // in one
      while ( condition2 ) {
            . . . // in two
            break;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .   // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Example:

break vs. continue

```
while ( condition1 ) {
      . . .  // in one
      while ( condition2 ) {
            . . . // in two
            continue;
            . . . // more statements
      }
      . . . // after two
}
. . . // after one
```

Jump statements

Jump statements are always embedded within a conditional (`if-else`).

Otherwise: subsequent statements are never executed. Cf. line 6 in the previous example ("*... // more statements*")

Compiler output: `unreachable statement`

By using **labels**, nested loops can be interrupted, however, this is usually bad coding style and will not be discussed (but if you are interested, see here: `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/branch.html`, google: java tutorial branch)

Loops: do-while

- Syntax:

```
do
        statement
while ( expr );
```

- Meaning:
  1. Execute `statement` first
  2. Then: evaluate `expr` (type: `boolean`)
  3. If `expr` is `true`, go to 1
  4. Else: exit the loop
- Basically the same as `while`, but executes `statement` at least once.

Loops: Overview

- All loops can be written as `while`-loops
- `for` is just an abbreviation
- `do-while` executes a statement first and then evaluates the condition
- All loops can be interrupted with `break` and continued with `continue`

Exercises: for-loop

Exercise 1

Write a loop to compute the sum of all even numbers between 30 and 130 (both inclusive).

Exercise 2

Write a loop to compute the product of all even numbers between 2 and 50 (both inclusive) and all odd numbers between 149 and 101.

Exercise: for-loops

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Foo" instead of the number and for the multiples of five print "Bar". For numbers which are multiples of both three and five print "FooBar".

Let's enhance the for-loop I

Enhanced for-loop (foreach)

- Additional version of the for-loop to iterate through Collections (later) and Arrays.
- Much more readable
- Much less complicated
- $\rightarrow$ Use it *whenever* possible!

Syntax

```
for (type variable : array) {
        statement(variable)
}
```

Let's enhance the for-loop II

Example

```
double[] scores = {1.2, 3.0, 0.8};
double sum = 0.;

for (double d : scores) {  // d gets successively each
                           // value in scores
    sum += d;
}
```

A few limitations...

- Read-only!
- Iteration over *one* structure only
- One element of the Array/Collection during each iteration
- One direction! (forward)
- Requires Java version ≥ 5

switch

- Similar to cascaded `if-else`
- Multi-branched selection depending on the value of an expression
- Syntax:

```
switch ( expr )   ⇐ expr type byte, int, short, char, String!
      case value01:
           statement1;
      case value02:
           statement2;
...
[ default:  //optional !
           statement3; ]
```

switch

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

        int dayOfWeek = Integer.parseInt( args[0] );
  String result;
        switch ( dayOfWeek ) {
            case 1:  result = "Monday"; break;
            case 2:  result = "Tuesday"; break;
            case 3:  result = "Wednesday"; break;
            case 4:  result = "Thursday"; break;
            case 5:  result = "Friday"; break;
            case 6:  result = "weekend!"; break;
            case 7:  result = "weekend!"; break;
            default:  result = "invalid day"; break;
        }
  System.out.println( result );
    }
}
```

code/Switch.java

switch

- Value that is checked needs to be of the type (`expr`), needs to be of type `byte`, `short`, `int`, `char`, or `String`. (Can also be an `enumerated` type, to be discussed later)
- Each value (`value`) in the `case` statements needs to be compatible with the type of `expr`
- All `value` expressions need to be literals (no variables) and disjunct!

switch

- By default, switch executes all cases from the first matching one ("fall-through")
- In most cases, only one case should be executed.
- `break` exits the `switch` statement - be sure to put a `break` after each case

Outline

Array data type

- array is a data type, denoted by square brackets `[]`
- e.g., `int[]`, `boolean[]`, `String[]`, ...
- arrays are homogenous: they contain elements of the same data type
- declaration:

```
int[] a; // a is of type array of int
boolean[] b; // b is of type array of boolean
```

Array initialization

Assume we have declared `int[] a;`

- `a = new int[5];`
  - new `int[5]` creates new `int`-array of length 5
  - initial value of a's elements is zero
- `a = new int[]{ 2, 3, 5 };`
  - creates new array of length 3 with initial elements 2, 3 and 5

- `int[] c = { 2, 3, 5 };`
  - only in declaration

Array length

- the length of an array is the number of elements it contains
- length is determined upon creation and cannot be changed afterwards
- every array has an attribute `length` which has an int-value representing the size of the array:

```
int[] a = new int[15];
int aLen = a.length;  // value of aLen is 15
boolean[] b = { true, true, false };
int bLen = b.length;  // value of bLen is ...
```

Multidimensional arrays

- arrays can be nested
  - `int[][]` – array of int arrays (2-dimensional)
  - `String[][][]` – array of arrays of String arrays (3-dimensional
  - ...

Creating multidimensional arrays

- ```
  int[][] products = { { 0, 0, 0 },
                       { 0, 1, 2 },
                       { 0, 2, 4 } }
  ```

- ```
  int[][] product = new int[3][];
  for ( int i = 0; i < 3; i++ ) {
    product[i] = new int[3];
  }
  ```

- ```
  int[][] product = new int[10][10];
  ```
  ```

Exercise: arrays & control structures[1]

Given an array of ints, return `true`*, if they are in strict increasing order, such as {2,5,11}, or {5,6,7}, but not {6,5,7} or {5,5,7}. However, with the exception that if "equalOk"is* `true`*, equality is allowed, such as {5,5,7} or {5,5,5}.*

[1] source: `http://codingbat.com/prob/p140272`

Exercise: while loops

Sum up all the integers in a multidimensional array. The computation should stop once one of the values equals zero.

Sample input

```
{{1,3,402,201},{1,2},{34,591,9294,294},{203,0,10},...}
```

Exercise: nested for-loops

Use nested for loops to generate a list of all the two digit numbers which are less than or equal to fifty-six, and the sum of whose digits is greater than ten.

Literature

Java language basics - operators, flow and arrays
`http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html`, `flow.html`, *and* `arrays.html`.

Sierra, K. & Bates, B.
Head First Java, Ch. 2 & 4
O'Reilly Media, 2005.

Ullenboom, Ch.
Java ist auch eine Insel. (Ch. 2.2-2.7, 3.7)
Galileo Computing, 2012.