

BERUFSAKADEMIE MANNHEIM
Information Technology International

Lecture Notes

Programming I
Part 2b The Java Programming Language

Dipl.-Betriebswirt (BA) VOLKAN YAVUZ
{xmachina GmbH

November 2001–February 2002
Classes: TIM01AGR && TIT01EGR
Term: 1

Contents

1	Structured Programming, 2nd Part	1
1.1	Branching Statements	1
1.1.1	The break Statement	2
1.1.2	The continue Statement	2
1.1.3	The return Statement	3
1.2	Top-Down Approach	3
1.3	Bottom-Up Approach	3
1.4	Functions and Methods	3
1.5	Input, Process, Output	4
2	Exercises and Examples	5
2.1	Top-Down Approach	5
2.2	Mathematical	6
2.3	Functions	6
2.4	From Structured Programming to Object Oriented Programming	6
	References	14

1 Structured Programming, 2nd Part

1.1 Branching Statements

Branching statements are used to “break out” from the iteration- and selection statements. Unlike statements like ‘goto’, they do not allow jumps to non-enclosing blocks, thus preventing the problems caused by non-local jumps that leave the strictly hierarchical order of blocks.

1.1.1 The break Statement

The ‘**break**’-statement can be used in an *unlabeled* and a *labeled* form.

Using the *unlabeled* form, the ‘**break**’-statement causes termination of the innermost enclosing ‘**switch**’-statement, ‘**for**’-, ‘**while**’-, or ‘**do-while**’-loop. Execution continues at the statement immediately following the enclosing statement.

Using the *labeled* form, the ‘**break**’-statement causes termination of an outer statement which is identified by the label specified in the ‘**break**’-statement. Execution continues at the statement immediately following the labeled statement. Note that it is not possible to “break-out” to non-enclosing blocks (also referred to as *non-local* jumps).

In the following example, the first ‘**break**’-statement causes the ‘**for**’-loop to be terminated, continuing execution with *<statements-3>*. The second ‘**break**’-statement causes the immediately enclosing ‘**while**’-loop to be terminated, continuing with *<statements-2>* and the next iteration of the ‘**for**’-loop.

```
1 search:
2   for (<for-expression>) {
3       while (<while-expression>) {
4           if (<if-expression>) {
5               <statements>
6               break search;
7           } else {
8               <statements>
9               break;
10          }
11      }
12      <statements-2>
13  }
14  <statements-3>
```

1.1.2 The continue Statement

The ‘**continue**’-statement can be used in an *unlabeled* and a *labeled* form. Using the *unlabeled* form, the ‘**continue**’-statement causes execution to skip the remaining statements of the current iteration of the immediately enclosing ‘**for**’, ‘**while**’, or ‘**do-while**’-loop and starts the next iteration.

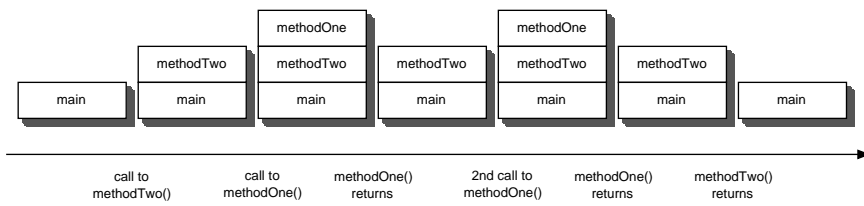
The *labeled*-form of the ‘**continue**’-statement skips the current iteration of an outer loop marked with the given label.

In the following example, the first ‘**continue**’-statement causes execution to continue with the next iteration of the ‘**for**’-loop, thus skipping *<statements-1>* and *<statements-3>*.

The second ‘**continue**’-statement causes execution to skip *<statements-2>* and continuing with the next iteration of the ‘**while**’-loop.

```
15 search:
16   for (<for-expression>) {
17       while (<while-expression>) {
18           if (<if-expression>) {
19               continue search;
20               <statements-1>
21           } else {
22               continue;
23               <statements-2>
24           }
25       }
26       <statements-3>
27   }
```

Figure 1: Return statement and runtime stack



1.1.3 The return Statement

The `return`-statement is used to return from the current method, and continues execution at the statement immediately following the original method call. The `return`-statement can optionally return a value.

The *return-address* is stored in a data structure called *stack*. New elements can only be put on top of the stack (*push*). Existing elements can only be taken from the top of the stack (*pop*). Execution of the following example and its effects on the *runtime stack* are shown as depicted in figure 1.

```

28 public static void methodOne(int value) {
29     if (<{if-expression}>) {
30         <{statements-1}>;
31         return;
32     } else {
33         <{statements-2}>;
34         return;
35     }
36 }
37
38 public static int methodTwo(int value)
39 {
40     methodOne(<{int-value}>);
41     methodOne(<{int-value}>);
42 }
43
44 public static void main(String[] args)
45 {
46     int value = methodTwo(<{int-value}>);
47 }
    
```

1.2 Top-Down Approach

To solve a large problem, that problem is broken down into several problems that are more easily solved. This process of breaking down the process is continued until the problem is eventually small enough to be solved directly without breaking it down further (see figure 2 on the next page).

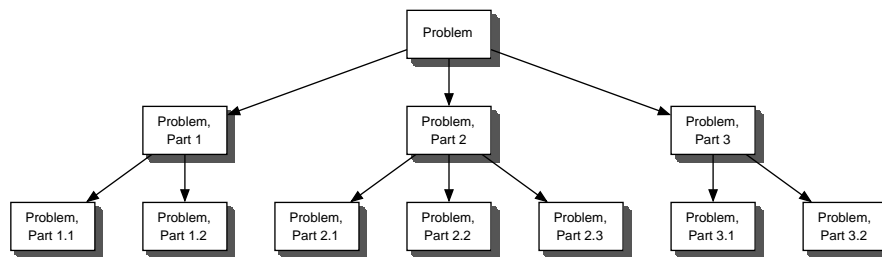
1.3 Bottom-Up Approach

The *bottom-up approach* in contrast starts with small pieces of code that implement parts of the desired functionality, and then assembles those pieces together until the complete functionality is available.

1.4 Functions and Methods

The building blocks of structured programming (sequence, iteration and selection) allowed control over the program flow, and are sufficient to implement any algorithm.

Figure 2: Top-Down Approach



Apart from the concepts delivered by object-oriented programming, the notion of a *function* is the next building block of a program. Functions¹ help in many ways in implementing and maintaining a computer program.

Multiple Usage As soon as some code like a complicated algorithm is used in more than one place within a program, functions can be used. This helps avoid *code redundancy*, leading to programs that are shorter (the code implementing the algorithm appears only once), concise (implementation of the algorithm and its use are kept apart) and better to maintain (changes to the algorithm's implementation need only be done in one place, instead of in all places where the algorithm is used).

Decomposition Even if some code is used only once, functions support a well defined *decomposition* of programs, allowing a more easy form of building a program, while improving readability. "Well defined" means that functions form natural units of a program, implementing a self-contained functionality, interconnected with the rest of the program (i.e. other functions) only through explicitly defined interfaces (formal parameter lists). All details of a function, its actual implementation of an algorithm and its usage of locally declared variables, are encapsulated from the rest of the program.

Reuse Functions support reuse of functionality in a way that already implemented algorithms can be used by other programs and in other systems. This leads to the building of libraries of generally usable functions (like the mathematical libraries that are available for the most programming languages).

1.5 Input, Process, Output

Another important principle is called *Input, Process, Output*². Any computer program is divided into the three phases

Input The phase in which all necessary input is retrieved, irrespective of *how*.

Process The phase in which the data gathered in the previous phase is processed, either yielding intermediate or final results.

Output The phase, in which the results are either output in human-readable form, or maybe made available in a form suitable for the next program, starting the next "iteration of IPO".

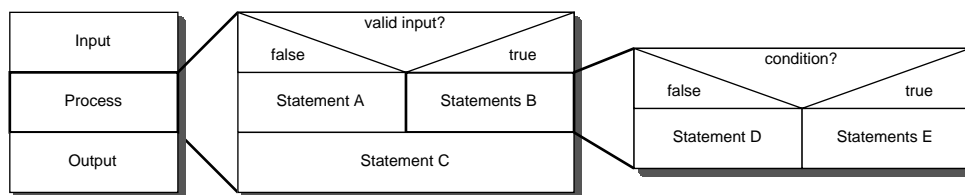
The IPO-principle can either be applied to complete programs, that read and output data in human-readable form, to methods that also accept input in form of actual parameters, process them, and return a result, and even to any single statement.

A combination of the principles of structured programming and IPO is shown in figure 3 on the following page.

¹ In terms of object-orientation, the term *method* is used. In other contexts, the same concept is called *procedure* or *sub-routine*.

² EVA-Prinzip (Eingabe, Verarbeitung, Ausgabe).

Figure 3: Input-Process-Output



2 Exercises and Examples

2.1 Top-Down Approach

Exercise 1 (Triangles) Write a program that reads in the three lengths of a triangle (‘int’-values in ascending order). The program should output whether those three lengths form

- no triangle at all,
- an equilateral triangle (gleichseitiges Dreieck),
- an isosceles triangle (gleichschenkliges Dreieck), or
- a scalene triangle (ungleichseitiges Dreieck)

Additionally, the program should output, in case those three lengths form a triangle, its area. Some hints about the implementation:

To interactively read in values, use a ‘BufferedReader’, defined in the package ‘java.io’. The method ‘readLine’ throws an ‘IOException’, so it has to be declared. ‘readLine’ is a ‘public’ method and returns a ‘string’.

```
48 public String readLine() throws IOException
```

To convert a read in ‘String’ into an ‘int’, use ‘Integer.parseInt’. ‘parseInt’ is a ‘public static’ method, accepts a ‘String’ and returns an ‘int’.

```
49 public static int parseInt(String s) throws NumberFormatException
```

Three lines can only form a triangle, if the sum of the two shorter lines is more than the longest line. As the lines a, b, c of the triangle are in ascending order (aufsteigender Ordnung), the condition “it’s not a triangle” can be written as ‘ $(a + b) \leq c$ ’.

The area A of a triangle can be calculated using the following formulae:

$$s = \frac{1}{2}(a + b + c)$$

$$A = \sqrt{s(s - a)(s - b)(s - c)}$$

To compute the square root (Quadratwurzel), use the method ‘sqrt’ defined in class ‘java.lang.Math’. The method signature is as follows:

```
50 public static double sqrt(double a)
```

As a starting point, use the code given.

```
51 import java.io.*;
52 import java.lang.Math;
53
54 class Triangle
55 {
56     public static void main(String[] args) throws IOException
```

```
57     {
58         BufferedReader input =
59             new BufferedReader(new InputStreamReader(System.in));
60
61         int a = Integer.parseInt(input.readLine());
62         ...
63         double area = Math.sqrt(...);
```

2.2 Mathematical

Exercise 2 (Factorial) Write a program that reads in an ‘int’-value ≥ 0 and computes its factorial³. The factorial $f(n)$ of n is defined as:

$$f(n) = \begin{cases} 1 & \text{if } n = 0 \\ \prod_{i=1}^n i & \text{if } n > 0 \end{cases}$$

For example, $f(6) = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 \cdot 6 = 720$.

Exercise 3 (Minimum) Write a program that reads in an series ‘int’-values terminated by a ‘0’-value. The program should then output the smallest number entered.

2.3 Functions

Exercise 4 (Primes) Write a program that reads in a positive ‘int’-value, and outputs the closest (either greater or less) prime number⁴. If there are two primes in equal distance, both should be output (e. g. for the input 9, both primes 7 and 11 should be output). If the input number already is a prime number the number should be output.

One group should implement a method that checks whether a given number is prime, the other group should implement the program that uses the method. For this to work, the “interface” of the method should be defined and agreed upon.

A positive integer p is prime if $p > 1$ is true and there exists no integer a with $2 \leq a < p$ that p can be divided by. Thus, we check for each number $a = 2, 3, \dots, p - 1$ whether p can be divided by a . An integer p can be divided by an integer a , if ‘ $p \% a == 0$ ’ is true.⁵

2.4 From Structured Programming to Object Oriented Programming

Example 1 (Students) Write a program that reads in up to 5 tuples, containing the following information:

- The name of the student
- The term the student is in currently
- Whether the student has passed the term, or not

□

In case a name is empty, the program should stop reading in information. For each student that has passed the term, the term stored should be incremented. Finally, the list of students (i. e. their names and the terms they’re in) should be output *in reverse order*. Use the following code fragments as a starting point:

³ Fakultät

⁴ Primzahl

⁵ ‘%’ is the modulo operator.

```
64 import java.io.*;
65
66 class StudentApplication
67 {
68     public static void main(String[] args) throws IOException
69     {
70         String[] names = new String[5];
71         int [] terms = new int[5];
72
73         ...
74
75         do {
76             System.out.println("Please enter name");
77             BufferedReader input =
78                 new BufferedReader(new InputStreamReader(System.in));
79
80             ...
81
82         } while (...);
83         ...
84         for (...) {
85             // output information
86         }
87     }
88 }
```

Example 2 (Students, second Version) There is a problem in the first implementation of the program. The *literal* '5' is present in too many places within the program. Replace the literal '5' with a *constant*.

```
89 private static final int MAX_STUDENTS = 5;
```

Example 3 (Students, third Version) There is still a problem in the second implementation. Access to central *data structure* is scattered all around the program. This can be improved by *isolating* all access to the data structure in a few functions:

```
90 import java.io.*;
91
92 class StudentApplication
93 {
94     ...
95
96     private static void setEntry(int pos, String name, int term)
97     {
98         ...
99     }
100
101     private static String getName(int pos)
102     {
103         ...
104     }
105
106     private static int getTerm(int pos)
107     {
108         ...
109     }
110 }
```

```
111     public static void main(String[] args) throws IOException
112     {
113         ...
114         do {
115             ...
116             setEntry(counter, new_name, new_term);
117             ...
118         } while ((counter < MAX_STUDENTS) && (!terminated));
119         ...
120         for (int i = counter; i > -1; i--) {
121             ...
122             System.out.println("name=" + getName(i) + ", term=" + getTerm(i));
123         }
124     }
125 }
```

Example 4 (Students, forth Version) There is still room for improvement in the third version. All information concerning a *student* relevant for this application can be grouped together in a *data structure*, implemented by defining a class 'Student' and further using instances of that class.

```
126 import java.io.*;
127
128 class Student
129 {
130     public String name;
131     public int term;
132 }
133
134 class StudentApplication
135 {
136     private static final int MAX_STUDENTS = 5;
137     private static Student[] students = new Student[MAX_STUDENTS];
138
139     private static void setEntry(int pos, Student student)
140     {
141         ...
142     }
143
144     private static Student getEntry(int pos)
145     {
146         ...
147     }
148
149     public static void main(String[] args) throws IOException
150     {
151         ...
152         do {
153             ...
154             Student new_student = new Student();
155             new_student.name = new_name;
156             new_student.term = new_term;
157             ...
158             setEntry(counter, new_student);
159         } while ((counter < MAX_STUDENTS) && (!terminated));
160         ...
161         for (int i = counter; i > -1; i--) {
162             Student student = getEntry(i);
163             System.out.print("Entry " + i + ": ");
```

```
164         System.out.println("name=" + student.name + ", term=" + student.term);
165     }
166 }
167 }
```

Example 5 (Students, fifth Version) A student in the context of this application not only possesses *state*, but *behavior* also. An operation that can be validly performed is “pass the current term”. An instance is also able to print information about itself (as implemented in the method ‘`toString()`’). By creating this *abstract data type* (ADT), all state is accessible only through explicitly defined methods, grouping them both together.

```
168 import java.io.*;
169
170 class Student
171 {
172     private String name;
173     private int term;
174
175     public Student(String name, int term)
176     {
177         this.name = name;
178         this.term = term;
179     }
180
181     public void pass()
182     {
183         this.term += 1;
184     }
185
186     public String toString()
187     {
188         return "name=" + this.name + ", term=" + this.term;
189     }
190 }
191
192 class StudentApplication
193 {
194     ...
195
196     public static void main(String[] args) throws IOException
197     {
198         ...
199         do {
200             ...
201             Student new_student = new Student(new_name, new_term);
202
203             if (passedString.equals("yes"))
204                 new_student.pass();
205             ...
206         } while ((counter < MAX_STUDENTS) && (!terminated));
207         ...
208         for (int i = counter; i > -1; i--) {
209             Student student = getEntry(i);
210             System.out.print("Entry " + i + ": ");
211             System.out.println(student.toString());
212         }
213     }
214 }
```

214 }

Example 6 (Students, sixth Version) Looking closer at the application, we can identify another piece of functionality that could be isolated from the rest: the data structure containing the list of students.

```
215 import java.io.*;
216
217 class Students
218 {
219     public static final int MAX_STUDENTS = 5;
220     private Student[] students = new Student[MAX_STUDENTS];
221     private int count;
222
223     public Students()
224     {
225         ...
226     }
227
228     public int addStudent(Student student)
229     {
230         ...
231     }
232
233     public boolean hasMoreStudents()
234     {
235         ...
236     }
237
238     public Student getStudent()
239     {
240         ...
241     }
242 }
243
244 class StudentApplication
245 {
246     public static void main(String[] args) throws IOException
247     {
248         Students students = new Students();
249         ...
250         do {
251             ...
252             counter = students.addStudent(new_student);
253         } while ((counter < Students.MAX_STUDENTS)
254                && (!terminated));
255         ...
256         while (students.hasMoreStudents()) {
257             Student student = students.getStudent();
258             System.out.println(student.toString());
259         }
260     }
261 }
```

Example 7 (Students, seventh Version) The implementation of the data structure can be improved further: by allowing the data structure to grow (and shrink) dynamically. ‘addStudent’ and ‘getStudent’ are the only methods that have an effect on the array size, so we make sure to call the appropriate ‘private’ methods.

```
262     public int addStudent(Student student)
263     {
264         ...
265         growArray();
266         ...
267     }
268
269     public Student getStudent()
270     {
271         ...
272         shrinkArray();
273         ...
274     }
```

First, we define some constants that will be used in the implementation. Upon creation of an instance, we also create an array of size 'INITIAL_SIZE'.

```
275     class Students
276     {
277         private static final int INITIAL_SIZE = 5;
278         private Student[] students = new Student[INITIAL_SIZE];
279
280         private int count;
281         private int current_size = INITIAL_SIZE;
282
283         ...
```

First, we check whether it is necessary to grow the array. We create a new array with one more element. Then, all elements of the "old" array are copied into the new one (actually, only the references to the elements are copied, not the elements themselves). Finally, the "old" array is discarded (it will be destroyed automatically by the garbage collector) by assigning 'students = new_students'.

```
285         private void growArray()
286         {
287             if (count >= current_size) {
288                 int new_size = count + 1;
289                 Student[] new_students = new Student[new_size];
290
291                 System.err.println("(old=" + current_size + ", new=" + new_size + ")");
292                 System.arraycopy(students, 0, new_students, 0, current_size);
293                 current_size = new_size;
294                 students = new_students;
295             }
296         }
```

First, we check whether we really need to shrink the array. If that's the case, we shrink the array to 'count + 1' elements. Then, we make sure that the internal array shrinks no less than 'INITIAL_SIZE'. Creation of the new array, copying and discarding the old array work the same as in 'growArray'.

```
298         private void shrinkArray()
299         {
300             if (count < current_size) {
301                 int new_size = count + 1;
302                 if (new_size < INITIAL_SIZE)
303                     new_size = INITIAL_SIZE;
304
305                 Student[] new_students = new Student[new_size];
306                 System.err.println("(old=" + current_size + ", new=" + new_size + ")");
307                 System.arraycopy(students, 0, new_students, 0, count);
308                 current_size = new_size;
```

```
309     students = new_students;
310     }
311 }
```

Additionally, we correct an error in the previous versions (by placing the definition of ‘input’ *before* the ‘do’-loop). We would also like to be able to redirect input from a file via ‘STDIN’ into the program.

```
312 class StudentApplication
313 {
314
315     public static void main(String[] args) throws IOException
316     {
317         ...
318         System.out.println("Enter any number of pairs of name and term.");
319         ...
320         BufferedReader input =
321             new BufferedReader(new InputStreamReader(System.in));
322         do {
323             ...
324         } while (!terminated);
```

After having done these corrections, we can create a file with the following contents, and call our application like ‘java StudentApplication < input.data’.

```
1 Marty McFly
2 13
3 yes
4 Lorraine Baines
5 10
6 yes
7 George McFly
8 9
9 yes
10 Biff Tannen
11 8
12 no
13 Jennifer Parker
14 8
15 no
16 Dr. Emmet L. Brown
17 13
18 yes
19 Mr. Strickland
20 14
21 no
```

Example 8 (Students, eighth Version) There’s a problem with the current implementation of the container. The *strategy* to keep the internal array the size needed is not very efficient. With every addition or removal of an element, the size is adjusted, which in turn makes it necessary to copy the whole array into a new one with the adjusted size. This can be improved by a more intelligent sizing of the newly created array. Upon growing, the array is not only grown to hold exactly just the one newly needed element, but to provide enough space for several new elements. The same principle is applied while shrinking the array. It is not shrunk by every removal of an element, but only when just a fraction of the available space is occupied.

We define some constants that will be used to determine

- *by which rate* to grow the array, and
- *when* to shrink the array:

□

```
325 class Students
326 {
327     ...
328     private static final float GROWTH_RATE = 0.5f;
329     private static final float SHRINK_THRESHOLD = 0.25f;
```

As growth and shrinkage of the internal array are only performed in exactly two methods, we just have to inspect those methods in detail:
Just the calculation of the new size has to be adjusted.

```
330     private void growArray()
331     {
332         if (count >= current_size) {
333             int new_size = current_size + (int)(current_size * GROWTH_RATE);
```

The changes to shrink the array are minimal, also:

```
334     private void shrinkArray()
335     {
336         if ((current_size > INITIAL_SIZE) &&
337             (count < (int)(current_size * SHRINK_THRESHOLD)) ) {
```

Example 9 (Students, ninth Version) This (hopefully) final version of our application should be improved so that it can read the data from a file with a given format, and accepts that filename as a given argument from the command line.

```
22 $ java StudentApplication
23 Usage: StudentApplication: {-h | -f <filename> }
```

The following example gives an example of an input file to be recognized by our application:

```
24 Marty McFly;13;yes
25 Lorraine Baines;10;yes
26 George McFly;9;yes
27 Biff Tannen;8;no
28 Jennifer Parker;8;no
29 Dave McFly;5;yes
30 Linda McFly;7;yes
31 Sam Baines;4;no
32 Stella Baines;8;no
33 Dr. Emmet L. Brown;13;yes
34 Mr. Strickland;14;no
```

The following method accepts a ‘String’ argument (in this case a line read from a file), and uses a ‘StringTokenizer’ to split it into three parts at the separator ‘;’.

```
338 import java.util.StringTokenizer;
339 import java.io.*;
340
341 /* Reads in (from a file ) triples of a student's name, her current term
342 * and whether the term was passed successfully , and outputs all
343 * student's names and the new term in reverse order. */
344
345 class StudentApplication
346 {
347     ...
348     private void processLine(String line)
349     {
350         StringTokenizer tokenizer = new StringTokenizer(line, ";");
351
352         String name = tokenizer.nextToken();
```

```

353     int term = Integer.parseInt(tokenizer.nextToken());
354     String passed = tokenizer.nextToken();
355
356     Student new_student = new Student(name, term);
357
358     if (passed.equals("yes"))
359         new_student.pass();
360
361     students.addStudent(new_student);
362 }
363 ...
364 } // StudentApplication
    
```

The following method reads one line at a time from a file.

```

365     private void readFromFile()
366     {
367         try {
368             BufferedReader input = new BufferedReader
369                 (new FileReader
370                 (new File(filename)));
371
372             while (input.ready()) {
373                 String line = input.readLine();
374                 processLine(line);
375             }
376         } catch (FileNotFoundException e) {
377             reportError("file not found: " + filename);
378             e.printStackTrace();
379             System.exit(1);
380         } catch (IOException e) {
381             e.printStackTrace();
382             System.exit(1);
383         }
384     }
    
```

List of Tables

List of Figures

1	Return statement and runtime stack	3
2	Top-Down Approach	4
3	Input-Process-Output	5

List of Acronyms

References