Blocks / Sequences

Constructing **blocks** from single instructions is one of the basic operations to structure programs!

- With ‘`{`’ and ‘`}`’ it is possible to combine multiple instructions to a block (in the sense of a **sequence**).
- A block can be used wherever an instruction can be used.
- The boundaries of a block restrict the life span of variables.
- ‘`{ }`’ represents the empty block.
- Instead of ‘`{ }`’ one can also use ‘`;`’ for an empty instruction.

```java
//correct block construction!
public class K3B02E_Block {
  public static void
       main(String[] args) {

    int i = 3;
    if (i < 0) {}
    else {
      String out;
      out = "i : " + i;
      System.out.println(out);
    }
  }
}
```

```
>javac K3B02E_Block.java

>java K3B02E_Block
i : 3
```

```java
//wrong block construction!
public class K3B03E_Block {
  public static void
       main(String[] args) {

    int i = 3;
    if (i < 0) {}
    else {
      String out;
      out = "i : " + i;
    }
      System.out.println(out);
  }
}
```

```
>javac K3B03E_Block.java
K3B03E_Block.java:11: error: cannot find symbol
        System.out.println(out);
                           ^
  symbol:   variable out
  location: class K3B03E_Block
1 error
```

Branching instructions with `if`

possible instances:

- **`if (condition) sequence1 else sequence2`**
- **`if (condition) sequence`**

```
1  public class K3B04E_Minimum {
2     public static void main(String[] args) {
3     int num1, num2, num3, min;
4     num1 = Integer.parseInt(args[0]);
5     num2 = Integer.parseInt(args[1]);
6     num3 = Integer.parseInt(args[2]);
7
8     if (num1 < num2)
9        if (num1 < num3) min = num1;
10       else min = num3;
11    else
12       if (num2 < num3) min = num2;
13       else min = num3;
14
15    System.out.println("Minimum: " + min);
16    }
17 }
```

```
1  import javax.swing.JOptionPane;
2  public class K3B05E_Branches {
3   public static void main(String[] args) {
4
5    int points = 0, grade = 0;
6         // list of declarations with initialization
7    String inWords = "wrong input";
8         // more on Strings later...
9    points = Integer.parseInt(
10            JOptionPane.showInputDialog ("Points: ") );
11
12   if (points <  40 && points >= 0)
13     { grade  = 5; inWords  = "not sufficient";  }
14   if (points >=  40 && points < 50 )
15     { grade = 4; inWords  = "sufficient";  }
16   if (points >=  50 && points < 60 )
17     { grade = 3; inWords  = "satisfactory";  }
18   if (points >=  60 && points < 70 )
19     { grade = 2; inWords  = "good";  }
20   if (points >=  70 && points <= 80)
21     { grade = 1; inWords  = "very good";  }
22
23    JOptionPane.showMessageDialog (null,
24               "grade: " + grade + "␣(" + inWords + ")");
25    }
26 }
```

```
1  import javax.swing.JOptionPane;
2  public class K3B06E_nested_Branches {
3   public static void main(String[] args) {
4
5    int points = 0, grade = 0;
6    String inWords = "wrong input";
7    points = Integer.parseInt(
8         JOptionPane.showInputDialog ("points: "));
9
10   if (points < 0  || points > 80) {}
11   else
12    if (points <  40) { grade = 5; inWords = "not sufficient"; }
13    else
14     if (points < 50 ) { grade = 4; inWords = "sufficient"; }
15     else
16      if (points < 60 ) { grade = 3; inWords = "satisfactory"; }
17      else
18       if (points < 70 ) { grade = 2; inWords = "good"; }
19       else
20         { grade = 1; inWords = "very good"; }
21
22   JOptionPane.showMessageDialog (null,
23                "grade: " + grade + "␣(" + inWords + ")");
24  }
25 }
```

Be careful with **dangling else**:

- Every **`if`** can come with an optional(!) **`else`**...
- Problem with two nested **`if`**s with only one **`else`**:

```
1 if (a == 1)
2    if (b == 1)
3       c = 42;
4 else
5    d = 42;
```

- To which **`if`** does the **`else`** belong? When are **`c`** or **`d`** changed?
- Basic rule in **`JAVA`**, **`C`**, **`C++`**:
  `else` belongs to the directly preceding **`if`**.
- In the example: If **`a`** is not **`1`**, then neither **`c`** nor **`d`** are changed.
  If **`a`**=**`1`** and **`b`**≠**`1`**, then only **`d`** is changed.
- Better: mark all alternatives with **`{}`** as blocks.
- other programming languages (**`ADA`**, **`Delphi`**, **`BASIC`**,...):
  usually use bracketing with closing **`endif`**, **`fi`** or similar

The previous example is thus equivalent to

```
1 if (a == 1) {
2     if (b == 1)
3         c = 42;
4     else
5         d = 42;
6 }
```

Better: always use brackets, e.g.:

```
1 if (a == 1) {
2     if (b == 1) {
3         c = 42;
4     } else {
5         d = 42;
6     }
7 }
```

Branching instruction with `switch`:

With a larger number of alternatives nested **`if`** blocks quickly get confusing, then it is better to use **`switch`**:

- **`switch`** with **`break`** (causes remaining cases to be skipped...)

```
1 switch (expression){
2     case value1: sequence1; break;
3     case value2: sequence2; break;
4     ...
5     case valueN: sequenceN; break;
6 [ default: defaultSequence ] // optional
7 }
```

- **`switch`** without **`break`**

```
1 switch (expression){
2     case value1: sequence1;
3     case value2: sequence2;
4     ...
5     case valueN: sequenceN;
6 [ default: defaultSequence ] // optional
7 }
```

```
1  import javax.swing.JOptionPane;
2  public class K3B07E_Switch {
3   public static void main(String[] args) {
4
5    int points = 0, case, grade = 0;
6    String inWords = "wrong input";
7    points = Integer.parseInt(
8           JOptionPane.showInputDialog ("points: "));
9
10   if (points < 0  || points > 80) {}
11   else {
12     case = points / 10;
13     switch (case) {
14       case 8:
15       case 7:  grade = 1; inWords = "very good";    break;
16       case 6:  grade = 2; inWords = "good";         break;
17       case 5:  grade = 3; inWords = "satisfactory"; break;
18       case 4:  grade = 4; inWords = "sufficient";   break;
19       default: grade = 5; inWords = "not sufficient";
20     }
21   }
22
23   JOptionPane.showMessageDialog (null,
24         "grade: " + grade + "␣(" + inWords + ")");
25  }
26 }
```

```
1  public class K3B08E_Switch_without_Break {
2   public static void main(String[] args) {
3    int in = 0, out = 0;
4    in = Integer.parseInt( args[0] );
5
6    switch (in) {
7           case 8:
8           case 7: out = out + 1 ;
9           case 6: out = out + 1 ;
10          case 5: out = out + 1 ;
11          case 4: out = out + 1 ;
12          case 3:
13          case 2:
14          case 1:
15          case 0:  out = out + 1 ;
16       }
17
18   System.out.println ( "Input: " + in);
19   System.out.println ( "Output: " + out);
20  }
21 }
```

Input:	0	1	2	3	4	5	6	7	8
Output:	1	1	1	1	2	3	4	5	5

Java essentially provides three kinds of **iterations** / **loops**: **`while`**, **`do-while`**, and **`for`**.

- Syntax of the **`while`** loop

```
1 while (condition)  sequence
```

- Semantics:
  - Evaluate the value of **`condition`**.
  - If value is **`false`**, stop execution of the loop.
  - If value is **`true`**, execute **`sequence`**.
  - repeat the loop...
- sequence of processing:
  - **`condition true`**
  - **`sequence`**
  - **`condition true`**
  - **`sequence`**
  - ...
  - **`condition false`**
- If **`condition`** is already **`false`** in the beginning, **`sequence`** is not executed at all.

Example: Given input n, compute $\sum_{i=1}^{n} i$ with **`while`** loop.

```
1  public class K3B09E_While {
2     public static void main(String[] args) {
3
4        int i = 1, n = 0, sum = 0;
5        n = Integer.parseInt( args[0]);
6
7        while (i <= n) {  // correct for n >= 0
8           sum = sum + i;
9           i++;
10       }
11
12       System.out.println( "For n = " + n
13            + "␣the sum is = " + sum);
14    }
15 }
```

- Syntax of the **`do-while`** loop:

```
1 do sequence while (condition)
```

- Semantics:
  - execute **`sequence`**.
  - evaluate the value of **`condition`**.
  - If value is **`false`**, stop execution of the loop.
  - If value is **`true`**, repeat the loop...
- sequence of processing:
  - **`sequence`**
  - **`condition`** **`true`**
  - **`sequence`**
  - ...
  - **`condition`** **`false`**
- **`sequence`** is always executed at least once!
- **`sequence`** is often called the body of the loop.
- If **`condition`** never gets false: ‘infinite loop’

```
1  public class K3B10E_DoWhile {
2     public static void main(String[] args) {
3
4        int i = 1, n = 0, sum = 0;
5        n = Integer.parseInt( args[0]);
6
7        do  {  // correct only for n > 0
8           sum = sum + i;
9           i++;
10       } while (i <= n)
11
12       System.out.println( "For n = " + n
13            + "␣the sum is = " + sum);
14    }
15 }
```

Example for complex loops: “Collatz Problem”,
“(3n+1) Conjecture”

Consider number sequences constructed using the following rules:

- Start with an arbitrary natural number $\boldsymbol{n > 0}$.
- If $\boldsymbol{n}$ is even, then continue with $\boldsymbol{n/2}$.
- If $\boldsymbol{n}$ is odd, then continue with $\boldsymbol{3 \cdot n + 1}$.

Examples:

- $\boldsymbol{12 \rightsquigarrow 6 \rightsquigarrow 3 \rightsquigarrow 10 \rightsquigarrow 5 \rightsquigarrow 16 \rightsquigarrow 8 \rightsquigarrow 4 \rightsquigarrow 2 \rightsquigarrow 1 \rightsquigarrow 4 \rightsquigarrow 2 \rightsquigarrow 1 \ldots}$
- $\boldsymbol{14 \rightsquigarrow 7 \rightsquigarrow 22 \rightsquigarrow 11 \rightsquigarrow 34 \rightsquigarrow 17 \rightsquigarrow 52 \rightsquigarrow 26 \rightsquigarrow 13 \rightsquigarrow 40 \rightsquigarrow 20 \rightsquigarrow 10 \ldots}$

Thus: we will often (or always?) eventually get the sequence
$\boldsymbol{1 \rightsquigarrow 4 \rightsquigarrow 2 \rightsquigarrow 1 \ldots}$

Implementation with nested loops:

```
1  public class K3B11E_Collatz {
2     public static void main(String[] args){
3
4        int n = 0, inner = 0, outer = 0;
5        n = Integer.parseInt( args[0] );
6
7        while (n > 1) {
8           if (n % 2 != 0) {
9              n = 3 * n + 1; outer ++;
10          }
11          while (n % 2 == 0) {
12             n = n / 2; inner ++;
13          }
14       }
15       System.out.println( "n = 1 after " + inner
16                + " steps in the inner loop and " + outer
17                + " steps in the outer loop");
18    }
19 }
```

- How many iterations inner/outer are there for an input?

Still unsolved Collatz conjecture:

- does the problem stop at all for every input?

(introduced in 1937 by Lothar Collatz)

n	inner	outer
11	**10**	**4**
101	**18**	**7**
1001	**91**	**51**
10001	**115**	**64**

- Syntax of the **`for`** loop:

```
1 for ( initialization(s); condition; assignment(s) )
2     sequence
```

- Semantics:
  - First the initializations are executed.
  - Then **`condition`** is evaluated.
  - If the condition is satisfied, **`sequence`** is executed.
  - Then the assignments are executed and the execution restarts with the condition test.
- Processed like the following while loop:

```
1 initialization(s);
2 while (condition) {
3   sequence;
4   assignment(s)
5 }
```

```
1  public class K3B12E_For {
2     public static void main(String[] args) {
3
4        int i = 1, n = 0, sum = 0;
5        n = Integer.parseInt( args[0]);
6
7        for (i = 1; i <= n; i++) {
8           sum = sum + i;
9        }
10
11       System.out.println( "For n = " + n
12            + "␣the sum is = " + sum);
13    }
14 }
```

`for` loops can be very complex:

```
1  public class K3B13E_Collatz_with_For {
2     public static void main(String[] args) {
3
4        int n = 0, inner = 0, outer = 0;
5        n = Integer.parseInt( args[0] );
6
7        for ( ;
8              n > 1 ;
9              n= (n%2!=0) ?  (3 * n + 1 + 0*(outer++))
10                         :  (n / 2     + 0*(inner++ )) )
11       {}
12
13       System.out.println( "n␣=␣1␣after " + inner
14                + "␣steps in the inner loop and " + outer
15                + "␣steps in the outer loop");
16    }
17 }
```

- Here, the iteration of the Collatz problem is done solely in the loop parameters.
- Almost unreadable program, very bad programming style!
- Please do not imitate!

Usual application of for loops: **nested counting loops**

```
1  public class K3B14E_Flag {
2     public static void main(String[] args) {
3        int i, j, flagSize;
4        flagSize = Integer.parseInt(
5              System.console().readLine("size of the flag: "));
6        for (i = 1; i <= flagSize; i++) {
7           System.out.print("|");
8           for (j = 1; j <= flagSize; j++)
9              if (i == j || (flagSize + 1 - j) == i)
10                System.out.print("xxx");
11             else
12                System.out.print("␣␣␣");
13          System.out.println("|");
14       }
15    }
16 }
```

output for **`flagSize`** 7:

```
|xxx               xxx|
|   xxx         xxx   |
|      xxx   xxx      |
|         xxx         |
|      xxx   xxx      |
|   xxx         xxx   |
|xxx               xxx|
```

Comparison of `break` and `continue`:

- If the instruction

  `continue;`

  is executed within a loop
  (usually in an if instruction...),
  then *the current iteration is stopped* and the loop continues with the next iteration.

- If the instruction

  `break;`

  is executed within a loop
  (usually in an if instruction...),
  then *the whole loop is stopped* and the execution continues with the first instruction after the loop.
  (analogously to **`break`** with **`switch`**...)

```java
public class K3B15E_Break {
 public static void
          main(String[] args) {

 System.out.println("Start");
  for (int i = 0; i <= 6; i++) {
   if (i == 3) break;
   System.out.println(i);
  }

 System.out.println("End");
 }
}
```

```
>java K3B15E_Break
Start
0
1
2
End

>
>
>
```

```java
public class K3B16E_Continue {
 public static void
          main(String[] args) {

 System.out.println("Start");
  for (int i = 0; i <= 6; i++) {
   if (i == 3) continue;
   System.out.println(i);
  }

 System.out.println("End");
 }
}
```

```
>java K3B16E_Continue
Start
0
1
2
4
5
6
End

>
```