Exam1 review
What’s a computer?

(input/output device(s) (Disks, network)

Central Processing Unit

(memory)

(input device(s) (mouse, keyboard)

(output device(s) (Terminal, printer))
Memory

- **bit** = (binary digit) a *smallest* memory device
  - A *bit* is in fact a *switch* that can remember 0 or 1
  - (The digits 0 and 1 are digits used in the *binary number system*)

- **Byte** = 8 bits
  - A *byte* is in fact one row of the RAM memory

- **KByte** = kilo byte = \(1024 (= 2^{10})\) bytes (approximately 1,000 bytes)

- **MByte** = mega byte = \(1048576 (= 2^{20})\) bytes (approximately 1,000,000 bytes)

- **GByte** = giga byte = \(1073741824 (= 2^{30})\) bytes (approximately 1,000,000,000 bytes)

- **TByte** = tera byte
Binary numbers

• The **binary number system** uses **2 digits** (0 and 1) to encode a **number**.

• An \( n \) digit cell can encode \( 2^n \) different numbers.

- A byte has **8 bits** and therefore, it can store:
  
  \[ 2^8 = 256 \text{ different patterns} \]
  (These 256 patterns are: 00000000, 00000001, 00000010, 00000011, .... 11111111)

- Each pattern can are **encoded** exactly one number:

  - 00000000 = 0
  - 00000001 = 1
  - 00000010 = 2
  - 00000011 = 3
  - ...
  - 11111111 = 255

Therefore, one byte can store one of **256 possible values**

(You can store the number 34 into a byte, but you **cannot** store the number 556, the value is **out of range**.)
- Binary and decimal numbers (non-negative)

\[
\begin{align*}
1 \times 2^0 &= 1 \\
0 \times 2^1 &= 0 \\
0 \times 2^2 &= 0 \\
1 \times 2^3 &= 8 \\
1 \times 2^4 &= 16 \\
0 \times 2^5 &= 0 \\
1 \times 2^6 &= 64 \\
1 \times 2^7 &= 128
\end{align*}
\]

\[1 + 8 + 16 + 64 + 128 = 217\]

\[
\begin{array}{c}
2) 156 \\
2) 78 \\
2) 39 \\
2) 19 \\
2) 9 \\
2) 4 \\
2) 2 \\
2) 1 \\
\hline
0 \\
0 \\
1 \\
1 \\
1 \\
0 \\
0 \\
1
\end{array}
\]

\[156_{10} = 10011100_2\]
Algorithm written in prog. language

```c
main()
{
    printf("Hello\n");
}
```

Compiler

First you write the algorithm in a prog. language with an editor (gedit)

Then you translate the algorithm into machine instructions with a compiler

Machine instructions

Finally, you execute the algorithm in machine instructions
Identifiers

• An identifier is a sequence of characters that consists of letters, digits, underscores (_), and dollar signs ($).
• An identifier **must** start with a letter, an underscore (_), or a dollar sign ($). It cannot start with a digit.
• An identifier cannot be a **reserved word/Java Keywords**.
• An identifier cannot be **true, false, or null**.
• An identifier can be of any length.

<table>
<thead>
<tr>
<th>legal</th>
<th>illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2, ComputeArea, area, radius, showMessageDialog</td>
<td>2A, d+4, ...</td>
</tr>
</tbody>
</table>
Conversion/Casting

**Safe conversions:**

```
char
byte  short  int  long  float  double
[-128 .. 127]
[-32768 .. 32767]
[-2147483648 .. 2147483647]
```

\[ [a .. b] = \text{the range of value of a given type} \]

```
[0 .. 65535]
```

\[ [\text{-}1.8 \times 10^{308} .. 1.8 \times 10^{308}] \]

\[ [-3.4 \times 10^{38} .. 3.4 \times 10^{38}] \]

\[ [-9223372036854775808 .. 9223372036854775807] \]

**Unsafe conversions:**

```
double  float  long  int  short  byte
```
# Priority and associativity of Java's operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Level</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]</td>
<td>access array element</td>
<td>1</td>
<td>left to right</td>
</tr>
<tr>
<td>.</td>
<td>access object member</td>
<td>1</td>
<td>left to right</td>
</tr>
<tr>
<td>++</td>
<td>invoke a method</td>
<td>1</td>
<td>left to right</td>
</tr>
<tr>
<td>--</td>
<td>post-increment</td>
<td>1</td>
<td>left to right</td>
</tr>
<tr>
<td>-</td>
<td>post-decrement</td>
<td>1</td>
<td>left to right</td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
<td>1</td>
<td>left to right</td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>1</td>
<td>left to right</td>
</tr>
<tr>
<td>(</td>
<td>pre-increment</td>
<td>2</td>
<td>right to left</td>
</tr>
<tr>
<td>)</td>
<td>pre-decrement</td>
<td>2</td>
<td>right to left</td>
</tr>
<tr>
<td>+</td>
<td>unary plus</td>
<td>2</td>
<td>right to left</td>
</tr>
<tr>
<td>-</td>
<td>unary minus</td>
<td>2</td>
<td>right to left</td>
</tr>
<tr>
<td>!</td>
<td>logical NOT</td>
<td>2</td>
<td>right to left</td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>2</td>
<td>right to left</td>
</tr>
<tr>
<td>()</td>
<td>cast object creation</td>
<td>3</td>
<td>left to right</td>
</tr>
<tr>
<td>/</td>
<td>multiplicative</td>
<td>4</td>
<td>left to right</td>
</tr>
<tr>
<td>%</td>
<td></td>
<td>4</td>
<td>left to right</td>
</tr>
<tr>
<td>++</td>
<td>additive</td>
<td>5</td>
<td>left to right</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift</td>
<td>6</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift</td>
<td>6</td>
<td>left to right</td>
</tr>
</tbody>
</table>

### Comparison Operators

- `<`  
- `<=`  
- `>`  
- `>=`  
- `instanceof`  
- `==`  
- `!=`  
- `&`  
- `^`  
- `|`  
- `&&`  
- `||`  
- `?:`  
- `=`  
- `+=`  
- `-=`  
- `*=`  
- `/=`  
- `%=`  
- `<<`  
- `>>`  
- `>>>`  

### Other Operators

- `relational type comparison`  
- `equality`  
- `bitwise AND`  
- `bitwise XOR`  
- `bitwise OR`  
- `conditional AND`  
- `conditional OR`  
- `conditional`  
- `assignment`  

Levels: 1 (lowest) to 6 (highest).
• **Precedence Order.** When two operators share an operand the operator with the higher *precedence* goes first.

• **Associativity.** When two operators with the same precedence the expression is evaluated according to its *associativity*. 
Assignment

- Assignment statements.

- Assignment expressions.

  var = expr;

  Result of var = expr is equal to the value of expr

- The assignment operator will (still) update the value stored inside the variable var.
- In addition to updating the variable, the assignment operator also returns a value.
char data type

- A char type variable consists of 2 bytes
- It contains the Unicode code of some character
- A char type variable is an integer type variable that contains a positive number

- Character literals:
  - A character literal is written between quotes '...'
  - Examples:
    - 'A' is the character literal A (Unicode code value 65)
    - 'a' is the character literal a (Unicode code value 97)
    - '1' is the character literal 1 (Unicode code value 49)
<table>
<thead>
<tr>
<th>ASCII code: (with letters and number highlighted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 NUL  1 SOH  2 STX  3 ETX  4 EOT  5 ENQ  6 ACK  7 BEL</td>
</tr>
<tr>
<td>8 BS   9 HT  10 NL  11 VT  12 NP  13 CR  14 SO  15 SI</td>
</tr>
<tr>
<td>16 DLE  17 DC1 18 DC2 19 DC3 20 DC4 21 NAK 22 SYN 23 ETB</td>
</tr>
<tr>
<td>24 CAN  25 EM  26 SUB 27 ESC 28 FS  29 GS  30 RS  31 US</td>
</tr>
<tr>
<td>32 SP   33 !  34 &quot;  35 #  36 $  37 %  38 &amp;  39 '</td>
</tr>
<tr>
<td>40 (    41 ) 42 *  43 +  44 ,  45 -  46 .  47 /</td>
</tr>
<tr>
<td>48 0   49 1  50 2  51 3  52 4  53 5  54 6  55 7</td>
</tr>
<tr>
<td>56 8   57 9  58 :  59 ;  60 &lt;  61 =  62 &gt;  63 ?</td>
</tr>
<tr>
<td>64 @   65 A  66 B  67 C  68 D  69 E  70 F  71 G</td>
</tr>
<tr>
<td>72 H   73 I  74 J  75 K  76 L  77 M  78 N  79 O</td>
</tr>
<tr>
<td>80 P   81 Q  82 R  83 S  84 T  85 U  86 V  87 W</td>
</tr>
<tr>
<td>88 X   89 Y  90 Z  91 [  92 \  93 ]  94 ^  95 _</td>
</tr>
<tr>
<td>96 `   97 a  98 b  99 c 100 d 101 e 102 f 103 g</td>
</tr>
<tr>
<td>104 h  105 i 106 j 107 k 108 l 109 m 110 n 111 o</td>
</tr>
<tr>
<td>112 p  113 q 114 r 115 s 116 t 117 u 118 v 119 w</td>
</tr>
<tr>
<td>120 x  121 y 122 z 123 { 124</td>
</tr>
</tbody>
</table>
The *string* data type *String*

- A *string* is a **sequence of characters** enclosed between the **double quotes** "...

  **Example:**

  "abc123"
  "Hello World"
  "Hello, what is your name ?"

- **Each character** in a *string* is of the type *char* and uses the **Unicode** as encoding method
- **Escape character:**
  - Escape character = a special character that allow Java to *change the meaning* of the *next character*
  - `\` (backslash) = the escape character for strings
    - The escape character is usually used to express *"unprintable" characters*

- **Escape sequence:**
  - Escape sequence = the escape character `\` followed by *one character*

    Example:
    
    \n
  - A escape sequence denotes *one character*

    The character denoted by an escape sequence is usually one that you *cannot type in with the keyboard*

- **Commonly used escape sequences:**

<table>
<thead>
<tr>
<th>Escape sequence</th>
<th>Denoted character</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\t</code></td>
<td>Tab</td>
<td>&quot;\t&quot; (string with a TAB character)</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>New line (NL)</td>
<td>&quot;\n&quot; (string with a NL character)</td>
</tr>
<tr>
<td><code>\</code></td>
<td>Backslash ()</td>
<td>&quot;&quot; (string with a )</td>
</tr>
<tr>
<td><code>\&quot;</code></td>
<td>Double quote (&quot;)</td>
<td>&quot;&quot;&quot; (string with a &quot;)</td>
</tr>
</tbody>
</table>
Operators on the String data type

- **Converting**: `int ↔ String`

```java
String s;
int x;
s = Integer.toString(x); // Returns the String representation for the integer value in variable x
x = Integer.parseInt(s); // Returns the binary number representation for the numeric string in variable s
```

- **Converting**: `double ↔ String`

```java
String s;
double x;
s = Double.toString(x); // Returns the String representation for the double value in variable x
x = Double.parseDouble(s); // Returns the IEEE 754 double precision number representation for the numeric string in variable s
```

- **Converting**: `float ↔ String`

```java
String a;
float x;
s = Float.toString(x); // Returns the String representation for the float value in variable x
x = Float.parseFloat(s); // Returns the IEEE 754 single precision number representation for the numeric string in variable s
```
There is **only one operator** (the + operator) defined for the **string data type**:

- `string1 + string2 = concatenate` the strings `string1` and `string2`

**Example:**

```
"abc" + "12" returns the string: "abc12"
```

- **Java** provides an **automatic conversion feature** when the + operator is used between:
  - a **number typed data** and a **string typed data**

**Java's automatic conversion rule for number ⇒ string:**

- When the + operator is used between
  - a **number** and
  - a **string**

  Then:
  - the **number** is **automatically converted** to a **string**
  - The + operator is then **applied** on **2 strings** (i.e., the + operator is a concatenation !)
• `charAt(i)`: return the character at position `i` in the string.

• `substring(i, j)`: return the substring consisting of the characters between positions `i` and `j-1` in the string.

• `length()` returns the length of the string stored in the string variable.
The Boolean (logical) data type boolean

- The boolean data type:
  - is a built-in (primitive) data type of Java
  - is used to represent the logical values
  - There are 2 logical values:
    - `true`
    - `false`

- Encoding scheme used in the boolean data type:
  - `0` represents false
  - `1` represents true

- uses 1 byte of memory (to store 0 or 1)
Compare operators

- Compare operators
  - Compare operators compare 2 numerical values and return a Boolean (logical) value.
  - A compare operator will return the value true if the test is successful.
  - A compare operator will return the value false if the test is unsuccessful.

- Compare operators in Java:

<table>
<thead>
<tr>
<th>Operator symbol</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>a &lt; b</td>
<td>Returns true if a &lt; b, otherwise returns false</td>
</tr>
<tr>
<td>&lt;=</td>
<td>a &lt;= b</td>
<td>Returns true if a ≤ b, otherwise returns false</td>
</tr>
<tr>
<td>&gt;</td>
<td>a &gt; b</td>
<td>Returns true if a &gt; b, otherwise returns false</td>
</tr>
<tr>
<td>&gt;=</td>
<td>a &gt;= b</td>
<td>Returns true if a ≥ b, otherwise returns false</td>
</tr>
<tr>
<td>!=</td>
<td>a != b</td>
<td>Returns true if a is not equal to b, otherwise returns false</td>
</tr>
</tbody>
</table>
Logical operators

- Logical operators in *Java*:

<table>
<thead>
<tr>
<th>Operator symbol</th>
<th>Meaning</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>The logical AND operator</td>
<td>Binary operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>The logical NOT operator</td>
<td>Unary operator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true &amp;&amp; true</td>
<td>true</td>
</tr>
<tr>
<td>true &amp;&amp; false</td>
<td>false</td>
</tr>
<tr>
<td>false &amp;&amp; true</td>
<td>false</td>
</tr>
<tr>
<td>false &amp;&amp; false</td>
<td>false</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>true</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
<tr>
<td>false</td>
<td></td>
</tr>
</tbody>
</table>
Operator Precedence and Associativity

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>var++ and var--</td>
<td>(Postfix)</td>
</tr>
<tr>
<td>+, -</td>
<td>(Unary plus and minus), ++var and --var (Prefix)</td>
</tr>
<tr>
<td>(type)</td>
<td>(Casting)</td>
</tr>
<tr>
<td>!</td>
<td>(Not)</td>
</tr>
<tr>
<td>*, /, %</td>
<td>(Multiplication, division, and remainder)</td>
</tr>
<tr>
<td>+, -</td>
<td>(Binary addition and subtraction)</td>
</tr>
<tr>
<td>&lt;, &lt;=, &gt;, &gt;=</td>
<td>(Comparison)</td>
</tr>
<tr>
<td>==, !=</td>
<td>(Equality)</td>
</tr>
<tr>
<td>^</td>
<td>(Exclusive OR)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>(AND)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=, +=, -=, *=, /=, %=</td>
<td>(Assignment operator)</td>
</tr>
<tr>
<td>Operator</td>
<td>Name</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>()</td>
<td>Parentheses</td>
</tr>
<tr>
<td>()</td>
<td>Function call</td>
</tr>
<tr>
<td>[]</td>
<td>Array subscript</td>
</tr>
<tr>
<td>.</td>
<td>Object member access</td>
</tr>
<tr>
<td>++</td>
<td>Postincrement</td>
</tr>
<tr>
<td>--</td>
<td>Postdecrement</td>
</tr>
<tr>
<td>++</td>
<td>Preincrement</td>
</tr>
<tr>
<td>--</td>
<td>Predecrement</td>
</tr>
<tr>
<td>+</td>
<td>Unary plus</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
</tr>
<tr>
<td>!</td>
<td>Unary logical negation</td>
</tr>
<tr>
<td>(type)</td>
<td>Unary casting</td>
</tr>
<tr>
<td>new</td>
<td>Creating object</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>Left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift with sign extension</td>
</tr>
<tr>
<td>&gt;&gt;&gt;</td>
<td>Right shift with zero extension</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>instanceof</td>
<td>Checking object type</td>
</tr>
</tbody>
</table>
Selections

Syntax of the *if*-statement:

```
if ( CONDITION )
ONE-statement
```

Explanation:

- The keyword *if* announces (to the Java compiler) that we started an *if*-statement.

- A conditional clause (*CONDITION*) follows the keyword *if*.
  - This is the **condition** of the *if*-statement.

- Following the condition clause, you can write (only) one statement.
  - This statement will *only* be executed if the condition is *true*. 


• Syntax of the if-else-statement:

```
if ( CONDITION )
    ONE-statement
else
    ONE-statement
```

Explanation:

- The keyword `if` announces (to the Java compiler) that we started an if-else-statement

- A conditional clause (CONDITION) follows the keyword `if`.

  - This is the **condition** of the if-else-statement

- Following the **then-part**, you must specify the keyword `else` followed by (only) one statement

  - This statement will **only** be executed if the **condition is true**

- Following the **then-part**, you must specify the keyword `else` followed by (only) one statement

  - This statement will **only** be executed if the **condition is false**
• **Nested conditional statement** = a conditional statement where the **then-part** and/or the **else-part** contains another conditional statement

**A $N$-way selection** construct looks like the following:

```plaintext
if ( condition_1 )
{
    S_1; (one or more statements)
}
else if ( condition_2 )
{
    S_2; (one or more statements)
}
else if ( condition_3 )
{
    S_3; (one or more statements)
}
...
else if ( condition_{N-1} )
{
    S_{N-1}; (one or more statements)
}
else
{
    S_N; (one or more statements)
}
```
Conditional Expressions

```java
boolean-expression ? expression1 : expression2;
```

```java
if (x > 0)
  y = 1;
else
  y = -1;
```

```java
y = (x > 0) ? 1 : -1;
```
switch statement

- Syntax of the `switch-statement`:

```
switch ( INT-EXPRESSION )
{
  case INT-VALUE₁: STATEMENT₁₁;
  STATEMENT₁₂;
  ...
  break; // Marks the end of case INT-VALUE₁
  case INT-VALUE₂: STATEMENT₂₁;
  STATEMENT₂₂;
  ...
  break; // Marks the end of case INT-VALUE₂
  [default: STATEMENT₃₁; // Optional clause
  STATEMENT₃₂;
  ...
  break; ]

```
The *dangling-else* ambiguity

- **Java** (and **C**, **C++**) imposes the following syntax rule:
  
  - The keyword *else* is *associated* to the *nearest* keyword if that makes a *syntactically correct* statement.
The syntax to invoke this method is

System.out.printf(format, item1, item2, ..., itemk)

where `format` is a string that may consist of substrings and format specifiers.

<table>
<thead>
<tr>
<th>Format Specifier</th>
<th>Output</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%b</td>
<td>a Boolean value</td>
<td>true or false</td>
</tr>
<tr>
<td>%c</td>
<td>a character</td>
<td>‘a’</td>
</tr>
<tr>
<td>%d</td>
<td>a decimal integer</td>
<td>200</td>
</tr>
<tr>
<td>%f</td>
<td>a floating-point number</td>
<td>45.460000</td>
</tr>
<tr>
<td>%e</td>
<td>a number in standard scientific notation</td>
<td>4.556000e+01</td>
</tr>
<tr>
<td>%s</td>
<td>a string</td>
<td>“Java is cool”</td>
</tr>
<tr>
<td>Example</td>
<td>Output</td>
<td></td>
</tr>
<tr>
<td>----------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>%5c</td>
<td>Output the character and add four spaces before the character item, because the width is 5.</td>
<td></td>
</tr>
<tr>
<td>%6b</td>
<td>Output the Boolean value and add one space before the false value and two spaces before the true value.</td>
<td></td>
</tr>
<tr>
<td>%5d</td>
<td>Output the integer item with width at least 5. If the number of digits in the item is &lt;5, add spaces before the number. If the number of digits in the item is &gt;5, the width is automatically increased.</td>
<td></td>
</tr>
<tr>
<td>%10.2f</td>
<td>Output the floating-point item with width at least 10 including a decimal point and two digits after the point. Thus there are 7 digits allocated before the decimal point. If the number of digits before the decimal point in the item is &lt;7, add spaces before the number. If the number of digits before the decimal point in the item is &gt;7, the width is automatically increased.</td>
<td></td>
</tr>
<tr>
<td>%10.2e</td>
<td>Output the floating-point item with width at least 10 including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width less than 10, add spaces before the number.</td>
<td></td>
</tr>
<tr>
<td>%12s</td>
<td>Output the string with width at least 12 characters. If the string item has fewer than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.</td>
<td></td>
</tr>
</tbody>
</table>
Programming Errors

• Syntax Errors
  – Errors that are detected by the compiler are called syntax errors or compile errors. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a corresponding closing brace. These errors are usually easy to detect, because the compiler tells you where they are and what caused them.
• Runtime Errors

  – Runtime errors are errors that cause a program to terminate abnormally. They occur while a program is running if the environment detects an operation that is impossible to carry out. Input mistakes typically cause runtime errors.
• Logic Errors
  – Logic errors are called bugs. The process of finding and correcting errors is called debugging.
The while-statement

- **Syntax** of the `while`-statement:

```java
while ( loop-continuation-condition )

Body of the while-loop

ONE-statement
```

Explanation:

- The keyword `while` announces (to the Java compiler) that we started an `while`-statement.
- A conditional clause (`LOOP-CONTINUATION-CONDITION`) follows the keyword `while`.
  - The `LOOP-CONTINUATION-CONDITION` is a Boolean expression (exactly the same as in the `condition` clause of an `if`-statement).
  - This is the condition of the `while`-statement.
- Following the `loop-continuation-condition` clause, you can write (only) one statement.
  - This is the body of the `while`-statement.
  - The body will be executed as long as the `loop-continuation-condition` is true !!!
○ Fact:
  - The body of any loop-statement will almost always contain multiple statements

○ Common practice in while-loops:
  - Use a block as body of loop-statements

○ A typical while-loop looks like this:

```java
while ( loop-continuation-condition )
{
  statement1
  statement2
  ....
}
```

Body of the while-loop
Flow chart representing a `while-statement`:

- **loop-cont-condition**
- **true** → statement1
- **true** → statement2
- **true** → ...

- **false** → Body of while-loop

*Execution repeats after body is done*
The break and continue statements

- **Introduction**
  - There are 2 special statements that can affect the execution of loop statements (such as a while-statement).
  - The special statements are: 
    - break
    - continue

We will study their meaning and how to use these special statements inside the while-statement.
• The **break statement**
  
  **Syntax:**
  ```
  break;
  ```
  
  **Effect:**
  - When the **break statement** is *executed* inside a **loop-statement**, the **loop-statement** is terminated **immediately**
  - The **execution** of the **program** will **continue** with the **statement following the loop-statement**
  
  **Schematically:**
  ```
  while ( loop-continuation-condition )
  {
    statement1
    statement2
    ....
    break;
    ....
  }
  ```
  
  **Execution proceeds to here**
  ```
  statement following the while loop
  ```
• The continue statement

  - Syntax:

    ```
    continue;
    ```

  - Schematically:

    ```
    while ( loop-continuation-condition )
    {
      statement1
      statement2
      ....
      continue;
      ....
    }
    ```

    Execution proceeds to here

    statement following the while loop
The for-statement

- The **most common form** of the for-statement is as follows:

```java
for ( var = START_VALUE ; var <= STOP_VALUE ; var = var + INCR )
{
    /* for-body (statements) */
}
```

**Meaning:**

- The **variable var** will take on the following values:
  
  - START_VALUE
  - START_VALUE + INCR
  - START_VALUE + 2×INCR
  - ...
  - Up to the **largest value** that is ≤ STOP_VALUE

- For **each of the value**, the statements in the for-body are **executed**

```java
repeat ( for var = START_VALUE,
    START_VALUE+INCR,
    START_VALUE+2×INCR,
    ...
    STOP_VALUE )
execute statements in the for-body
(one time for each value in the list);
```
Example 1:

```java
public class For01
{
    public static void main(String[] args)
    {
        int a;

        // Example "for-statement"
        for ( a = 1 ; a <= 10 ; a = a + 1 )
        {
            System.out.println(a);   // Print a
        }

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

Output:

1
2  (1+1×1)
3  (1+2×1)
4  (1+3×1)
5  (1+4×1)
6  (1+5×1)
7  (1+6×1)
8  (1+7×1)
9  (1+8×1)
10 (1+9×1)
Done

Notice that in the for-statement:

- `a = 1` specifies the starting value
- `a <= 10` specifies when the for-loop will stop
- `a = a + 1` specifies how the variable `a` will change each time through the loop
The *general syntax* of the **for-statement** is as follows:

```plaintext
for (init-expression; loop-cont-condition; incr-expression)

ONE-statement

Body of the for-loop
```

A typical **for-loop** looks like this:

```plaintext
for (init-expression; loop-cont-condition; incr-expression)
{
    statement1
    statement2
    ....
}

Body of the for-loop
```
Flow chart representing the for-statement:

```
init-expression;

loop-cont-condition

true

statement1
statement2
....

false (terminates)

Body of for-statement

incr-expression;

Execution repeats after body is done
```
Effect of the break-statement on a for-statement

The execution of the break statement illustrated with a flow chart:

- For-statement
  - init-expression ;
  - loop-cont-condition
    - true
      - Body of for-statement
      - break;
      - ****
      - ****
  - incr-expression ;

break statement will cause this jump
Effect of the *continue*-statement on a *for*-statement

- The *execution of the continue statement* illustrated with a flow chart:

```
init-expression;

loop-cont-condition
  true
  ....
  continue;
  ....

false (terminates)
```

*Body of for-statement*

*continue statement will cause this jump*

*Execution repeats*
The *do*-statement

- **Syntax**

```
do
    ONE-statement
while ( loop-continuation-condition ) ;
```

```
do
{  statement1
    statement2
    ....
}
while ( loop-continuation-condition ) ;
```
Flow chart representing a **while**-statement:

- **Body of do-loop**
- **do-statement terminates**
Break and continue statement used in a do-statement

- The break statement will cause the do-statement to terminate immediately:

```java
do
{
    ....
    break;
    ....
}
while ( loop-continuation-condition ) ;
```

The execution will continue with the statement following the do-statement.

- The continue statement will cause the do-statement to jump to the end of the body:

```java
do
{
    ....
    continue;
    ....
}
while ( loop-continuation-condition ) ;
```

If the condition is true, the do-statement is repeated, and otherwise (if the condition is false), the do-statement is terminated.
• Nested loops consist of an outer loop and one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, and started anew.
Infinite loop

• Infinite loop = a loop statement that does not end

• You must type `control-C` in the terminal window to terminate the Java program when you do run it!

• The **while-body must contain** statements that will **affect** (change) the outcome of the **loop-continuation-condition** of the **while-loop** !!!