

BERUFSAKADEMIE MANNHEIM
Information Technology International

Lecture Notes

Programming I
Part 2 The Java Programming Language

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

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

Contents

1	Introduction	2
2	Getting Started	3
3	Lexical Structure	4
3.1	Lexical Translations	4
3.2	Unicode and Unicode Escapes	4
3.3	Line Terminators	4
3.4	White space	4
3.5	Comments	4
3.6	Identifiers	5
3.7	Keywords	5
3.8	Separators	5
4	Variables	6
4.1	Types	6
4.2	Literals	6
4.2.1	Integer Literals	6
4.2.2	Floating Point Literals	7
4.2.3	Escape Sequences	7
4.2.4	Character Literals	7
4.2.5	String Literals	7
4.2.6	Boolean Literals	8
4.2.7	The Null Literal	8
4.3	Declaration and Scope	8
5	Arrays	9

6	Operators	10
6.1	Assignment Operator	10
6.2	Arithmetic Operators	10
6.3	Comparison Operators	10
6.4	Logical Operators	11
6.5	Bitwise Operators	11
6.6	Conditional Operator	11
7	Expressions	12
7.1	Evaluation Order	12
7.2	Associativity	12
7.3	Priority	12
8	Structured Programming	12
8.1	Sequence	13
8.2	Iteration	14
8.2.1	The while Statement	14
8.2.2	The do Statement	14
8.2.3	The for Statement	14
8.3	Selection	14
8.3.1	The if Statement	15
8.3.2	The switch Statement	15
	References	16

1 Introduction

Java has been presented on May, 23rd 1995 on SunWorld in San Francisco. Java is a very elegant and sufficiently clearly designed language and well suited for the introduction into the Art of Programming.

Some statements or properties about the Java programming language should be noted in advance [see Gosling et al., 2000]:

general purpose Java is a general purpose language, meaning that is has not been designed to be used in a very specific environment or for a very specific purpose.

concurrent Aspects of concurrency are directly supported by the language, making it unnecessary to use libraries that provide this functionality. Based upon the concept of monitors, parts of a Java program can be executed in parallel, and are synchronized at explicitly stated points.

class-based The main mechanism for organizing a software system implemented in the Java programming language are *classes*.

object-oriented Java directly supports the object-oriented programming paradigm, implementing concepts like *inheritance*, *polymorphism*, and *encapsulation*.

simple Java is simple enough so that many programmers can achieve fluency in the language. The amount of key concepts and language constructs is manageable, and most of the advanced features can safely be ignored for the beginning. Additionally, almost all functionality is provided through extensive class libraries instead of language constructs.

interpreted Interpreted languages are inherently more dynamic as more static information that would only be available at compile-time, is also available at run time.

distributed The usually encountered restriction that executables have to be on the machine the program is running on has been loosened. Instead, a separate component – the *classloader* – is responsible for retrieving the byte code, e.g. via HTTP (applets).

related to C and C++ Through the syntactical closeness to the very familiar and widespread languages like C and C++, entering Java is rather easy.

production language Java is clearly designed as a production language in contrast to a research language. It can safely be used in industrial strength and mission critical systems. Also, Java doesn't introduce any new concepts or untested features. It just happens that Java has "picked the best things and packaged them well".

strongly-typed The clear distinction between compile-time and run time errors makes Java a *strongly typed* language. Strongly typed languages make it possible to detect failures due to mismatch of types already at compile time, instead of later at run time [Aho et al., 1988, pp. 427].

relatively high-level Details und peculiarities of the machine a Java program is running on are not available through the language, making Java very platform independent.

garbage collection Automatic storage management by means of garbage collection is supported through the virtual machine. This omits many potential programming errors and safety problems due to the use of explicit memory allocation and deallocation (as in C's 'free' or C++'s 'delete').

robust, omits unsafe constructs Constructs that are known to be unsafe in other programming languages have either been omitted or tackled by thorough analysis through the compiler. Potential problems like array access without index checking, direct memory access or pointer arithmetic are disallowed or just simply not necessary.

2 Getting Started

Any programming language is best introduced by presenting the traditional *Hello-World program*.

Listing 1: Hello-World Application

```
1  /*
2  * This class implements a simple application that just writes "Hello,
3  * World!" to stdout and exits. */
4
5  class HelloWorldApp
6  {
7      // Execution starts here.
8      public static void main(String[] args)
9      {
10         // Display the message
11         System.out.println("Hello, World!");
12     } // main
13 } // HelloWorldApp
```

Java does not support global functions or variables. Everything has to be enclosed within a *class*. The only class here is `HelloWorldApp`. Program execution starts, by definition, at a method with the following signature:

```
public static void main(String[])
```

To compile and run that Java application, the following commands have to be issued:

```
1  $ /usr/bin/javac HelloWorld.java
2  $ /usr/bin/java HelloWorld
```

3 Lexical Structure

3.1 Lexical Translations

The following three steps of translation are performed before the compiler sees the Java source code:

1. Unicode escapes are translated into the corresponding Unicode characters. This makes it possible to write any code just using ASCII characters.
2. The result of step 1 is transformed into a stream of characters and line terminators. This makes it possible to use code independently of line endings.¹
3. From the result of step 2, all white space and comments are discarded. This makes white space (indentation) and comments invisible to the compiler.

3.2 Unicode and Unicode Escapes

Java programs are written using the Unicode character set [see Unicode]. If the corresponding Unicode character is not available, it can be encoded using $\langle UnicodeEscape \rangle$.

$\langle UnicodeEscape \rangle ::= \text{---} \backslash - u - \langle HexDigit \rangle - \langle HexDigit \rangle - \langle HexDigit \rangle - \langle HexDigit \rangle \text{---}$

$\langle HexDigit \rangle ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9$
 $\quad a | b | c | d | e | f |$
 $\quad A | B | C | D | E | F$

3.3 Line Terminators

$\langle LineTerminator \rangle ::= \langle ASCII\ LF\ character \rangle /*\ newline\ */$
 $\quad \langle ASCII\ CR\ character \rangle /*\ carriage\ return\ */$
 $\quad \langle ASCII\ LF\ character \rangle \langle ASCII\ CR\ character \rangle$

3.4 White space

$\langle WhiteSpace \rangle ::= \langle ASCII\ SP\ character \rangle /*\ space\ */$
 $\quad \langle ASCII\ HT\ character \rangle /*\ horizontal\ tab\ */$
 $\quad \langle ASCII\ FF\ character \rangle /*\ form\ feed\ */$

3.5 Comments

Java know three different types of comments (see table 1).

Table 1: Types of Comments

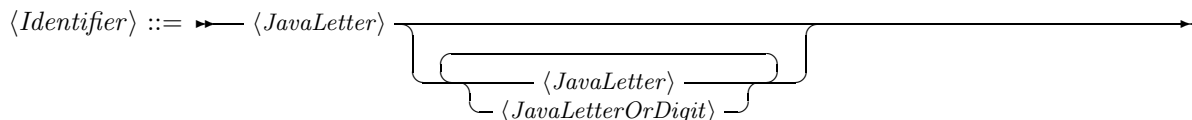
Comment	Description
<code>/* ... */</code>	A traditional comment like in C. All characters between <code>/*</code> and <code>*/</code> are ignored.
<code>//</code>	An end-of-line comment like in C++. All characters from <code>//</code> until the end of the line are ignored.
<code>/** ... */</code>	JavaDoc comments

¹ Unix, DOS and Macintosh platforms all use different encodings for line endings. Unix lines are terminated by `\0x0a` (newline), DOS lines by `\0x0d, \0x0a` (carriage return followed by newline), and Macintosh lines are terminated by `\0x0d` (carriage return).

3.6 Identifiers

Identifiers are used to name entities in a program. Identifiers have the following properties:

- Java treats identifiers case-sensitive, i.e. that ‘name’ and ‘Name’ are treated as different identifiers.
- Identifiers can have unlimited length.
- Identifiers must not start with digits, but can contain them.
- $\langle \text{JavaLetter} \rangle$ also contains – for historical reasons – the underscore ‘_’ and the dollar sign ‘\$’.
- Identifiers must not be equal to *keywords*, the boolean literal or the null literal.



3.7 Keywords

The Java keywords (see table 2) are reserved and can thus not be used as identifiers.

Table 2: Java Keywords

abstract	default	if	private	this
boolean	do	implements	protected	throw
break	double	import	public	throws
byte	else	instanceof	return	transient
case	extends	int	short	try
catch	final	interface	static	void
char	finally	long	strictfp	volatile
class	float	native	super	while
const ^a	for	new	switch	
continue	goto ^a	package	synchronized	

^a Reserved, but not used

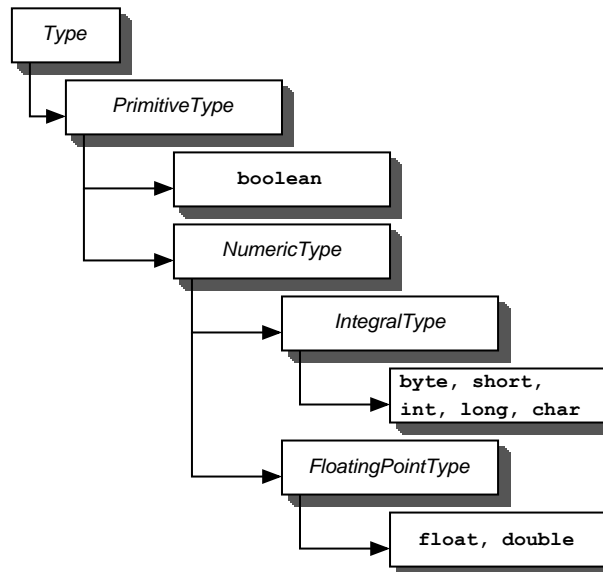
3.8 Separators

The separators (see table 3) are used for punctuation.

Table 3: Separators

Separator	Name	Used as
‘(’, ‘)’	left and right parenthesis	encloses parameter and argument list in function call
‘{’, ‘}’	left and right curly brackets	encloses blocks of statements
‘[’, ‘]’	left and right square brackets	array index operator
‘;’	semicolon	statement terminator
‘,’	komma	sequence operator
‘.’	dot	member access operator

Figure 1: Kinds of Primitive Types



4 Variables

4.1 Types

Variables are storage locations for data of some *type*. The type specifies the *values* available. *Literals* are source-code representations of values. Figure 1 gives an overview of the primitive types Java supports.

Table 4 gives an overview of the storage requirements and value ranges of those primitive types.

Table 4: Java Datatypes

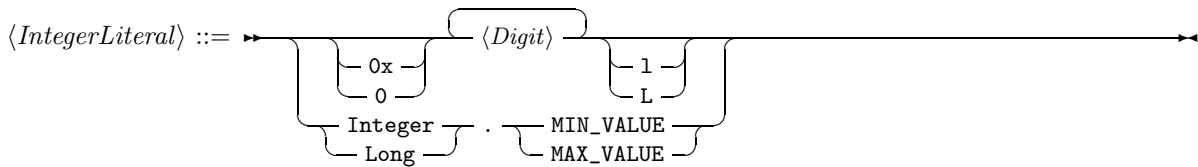
Typename	Valid Values resp. Range	Space
<code>boolean</code>	<code>true, false</code>	1 bit
<code>byte</code>	-128 to 127	8 bit
<code>short</code>	-32 768 to 32 767	16 bit
<code>int</code>	-2 147 483 648 to 2 147 483 647	32 bit
<code>long</code>	-9 223 372 036 854 775 808 to 9 223 372 036 854 775 807	64 bit
<code>char</code>	'\u0000' to '\uffff'	16 bit
<code>float</code>	+/- 1.402 398 46e - 45 to +/- 3.402 823 47e38	32 bit
<code>double</code>	+/- 4.940 656 458 412 465 44e - 324 to +/- 1.797 693 134 862 315 70e308	64 bit

4.2 Literals

A *<Literal>* is a source code representation of a value of a *primitive type*, the `'String'` type or the *null type*.

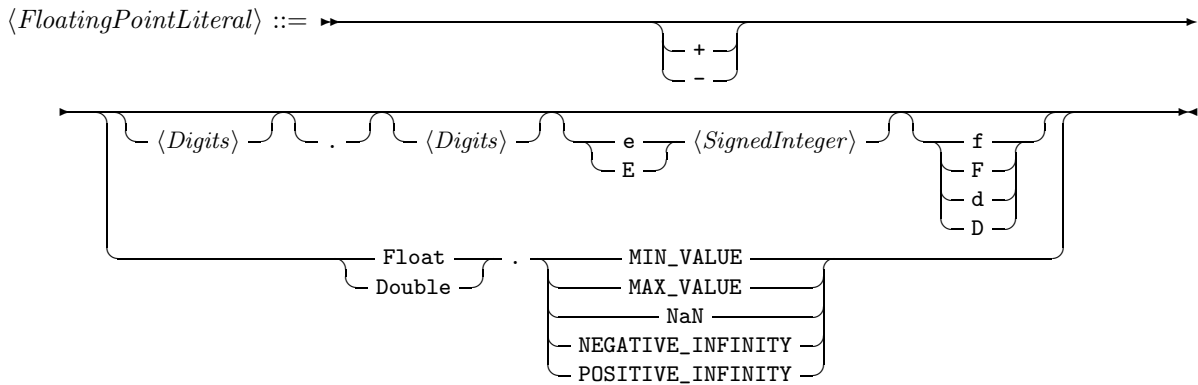
4.2.1 Integer Literals

Integer literals can be stated in base 10 (decimal), base 16 (hexadecimal, prefixed by `'0x'`) or base 8 (octal, prefixed by `'0'`). Additionally, they can be stated as long, postfixed by `'l'` or `'L'`. Examples are `'1996'`, `'0xCafeBabe'`, `'0xC0B0L'`, and `'0777L'`.



4.2.2 Floating Point Literals

Floating point literals can be stated as either 'double' (the default, prefixed by 'd' or 'D') or 'float', prefixed by 'f' or 'F'. To be recognized as floating point literal, at least the decimal point '.' or either the exponent part or the float type suffix has to be present.



4.2.3 Escape Sequences

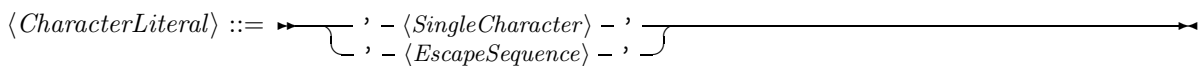
Escape sequences are used within character and string literals (see table 5).

Table 5: Escape Sequences

Escape Sequence	Description
\b	backspace BS
\t	horizontal tab HT
\n	linefeed LF
\f	form feed FF
\r	carriage return CR
\"	double quote
\'	single quote
\\	backslash

4.2.4 Character Literals

Character literals consist of one single character or escape sequence enclosed within a pair of single quotes ''.



4.2.5 String Literals

Character literals consist of any number of characters or escape sequences enclosed within a pair of double quotes "".

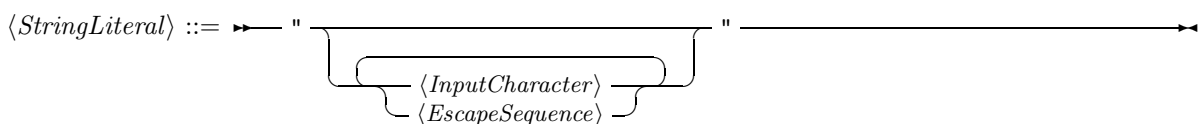
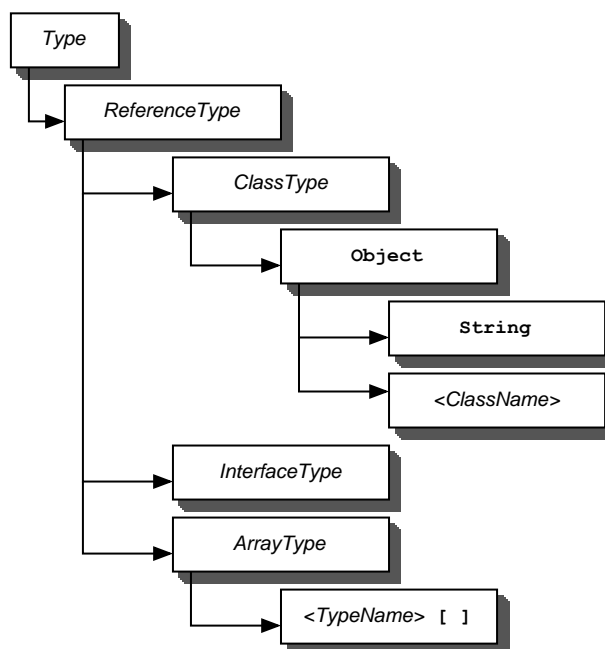


Figure 2: Kinds of Reference Types



```

30     System.out.println(largeNumber);
31 }
32 }
33 }
    
```

5 Arrays

Array objects belong to the *reference type* (see figure 2) kind of objects and contain a number of variables. Those variables may either be primitive or reference types, and may also be arrays, again.

If an array object contains no variables, it is said to be *empty*. Variables inside an array have no names and are instead addressed by nonnegative integer index values from 0 to $n - 1$, inclusive, with n being the length of the array and also the number of variables it contains.

Objects of type array have a 'public final' field 'length', containing n .

Listing 3: Arrays

```

1  class ArrayApp
2  {
3      public static void main(String[] args)
4      {
5          // declaration
6          int [] intArray;
7
8          // declaration and creation
9          byte[] byteArray = new byte[3];
10
11         // declaration, creation and initialization
12         String[] animals = { "Lion", "Tiger" };
13
14         for (int i = 0; i < animals.length; i++)
    
```

```
15     System.out.println(animals[i]);
16
17     // two-dimensional array
18     double[] [] identity_matrix = {
19         { 1.0, 0.0, 0.0 },
20         { 0.0, 1.0, 0.0 },
21         { 0.0, 0.0, 1.0 }};
22
23     for (int i = 0; i < identity_matrix.length; i++)
24     {
25         for (int j = 0; j < identity_matrix[i].length; j++)
26             System.out.print(identity_matrix[i][j] + " ");
27         System.out.println();
28     }
29
30     } // main
31 } // ArrayApp
```

6 Operators

6.1 Assignment Operator

Values are assigned using the *assignment operator* '='.

6.2 Arithmetic Operators

The arithmetic operators (see table 6) can be performed on integral or floating point values.

Strings can be concatenated using '+'.
'++' and '--' are the increment and decrement operators. For example, 'i++' is equivalent to 'i = i + 1'.

Table 6: Arithmetic Operators

Operator	Operation	Applicability
+, +=	addition	byte, short, int, long, float, double
-, -=	subtraction	byte, short, int, long, float, double
*, *=	multiplication	byte, short, int, long, float, double
/, /=	division	byte, short, int, long, float, double
%, %=	remainder	byte, short, int, long, float, double
+	unary	byte, short, int, long, float, double
-	unary negation	byte, short, int, long, float, double
++	increment	byte, short, int, long, float, double
--	decrement negation	byte, short, int, long, float, double

6.3 Comparison Operators

The comparison operators (see table 7 on the following page) return values of type 'boolean'.

Table 7: Comparison Operators

Operator	Operation	Applicability
>	greater than	byte, short, int, long, float, double
>=	greater than or equal	byte, short, int, long, float, double
<	less than	byte, short, int, long, float, double
<=	less than or equal	byte, short, int, long, float, double
==	equal	byte, short, int, long, float, double, boolean
!=	not equal	byte, short, int, long, float, double, boolean

6.4 Logical Operators

Logical operators (see table 8) are evaluated *short-circuit*, i.e. evaluation of a statement is stopped as soon as the result would not change by further evaluation:

- in 'x && y', 'y' will only be evaluated if 'x' is 'true'
- in 'x || y', 'y' will only be evaluated if 'x' is 'false'

Table 8: Logical Operators

Operator	Operation	Applicability
!	negation	boolean
&&	conditional AND	boolean
	conditional OR	boolean

6.5 Bitwise Operators

Bitwise logical operators (see table 9) are only applicable to integral and boolean values.

Table 9: Bitwise Logical Operators

Operator	Operation	Applicability
&, &=	bitwise AND	byte, short, int, long, boolean
, =	bitwise OR (inclusive)	byte, short, int, long, boolean
^, ^=	bitwise XOR (exclusive OR)	byte, short, int, long, boolean
~, ~=	bitwise complement (negation)	byte, short, int, long, boolean

Bitwise shift operators (see table 10 on the next page) are only applicable to integral values.

6.6 Conditional Operator

The *conditional operator* '?' ':' is the only *ternary operator*.

Listing 4: Conditional Operator

```

1 value = <Expression> ? <>true-Statement> : <>false-Statement>
2
3 // is equivalent to
4
```

Table 10: Shift Operators

Operator	Operation	Applicability
<<, <<=	left shift	byte, short, int, long
>>, >>=	right shift	byte, short, int, long
>>>, >>>=	unsigned right shift	byte, short, int, long

```
5 if ( <Expression> )
6     value = <true-Statement>
7 else
8     value = <false-Statement>
```

7 Expressions

7.1 Evaluation Order

Evaluation of expressions is performed from left to right.

```
1 int i = 2;
2 int j = (i = 3) * i;
3
4 // j equals 9, not 6
```

7.2 Associativity

All binary Operators except for the assignment operators are left-associative, i.e. they group left first.

```
5 a <op> b <op> c
6
7 // is equivalent to
8 (a <op> b) <op> c
```

The assignment operators are right-associative

```
9 a = b = c
10
11 // is equivalent to
12 a = (b = c)
```

7.3 Priority

First, parentheses are used to explicitly state evaluation order. Then, if no parentheses are given, priority (see table 11 on the following page) determines evaluation order. Then, in case of same priority, associativity of the operator determines evaluation order.

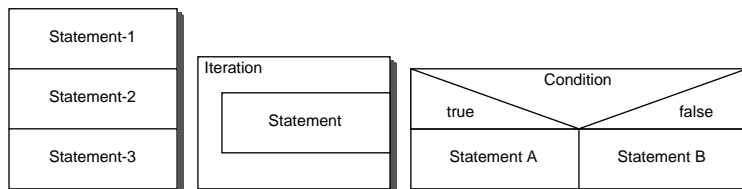
8 Structured Programming

GO TO considered harmful.
That means that the more GO TO statements there are in a program
the harder it is to follow the program's source code.
– Edsger Dijkstra

Table 11: Priority of Operators

Priority		Operators
highest	postfix	<code>[]</code> , <code>(⟨params⟩)</code> , <code>⟨expr⟩++</code> , <code>⟨expr⟩--</code>
	unary	<code>++⟨expr⟩</code>
	creation and casting	<code>new</code> , <code>(⟨type⟩)⟨expr⟩</code>
	multiplicative	<code>*</code> , <code>/</code> , <code>%</code>
	additive	<code>+</code> , <code>-</code>
	shift	<code><<</code> , <code>>></code> , <code>>>></code>
	comparison	<code><</code> , <code>></code> , <code>>=</code> , <code><=</code>
	equality	<code>==</code> , <code>!=</code>
	bitwise AND	<code>&</code>
	bitwise XOR	<code>^</code>
	bitwise OR	<code> </code>
	logical AND	<code>&&</code>
	logical OR	<code> </code>
	conditional	<code>? :</code>
lowest	assignment	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> , <code>%=</code> , <code>>>=</code> , <code><<=</code> , <code>>>>=</code> , <code>&=</code> , <code>^=</code> , <code> =</code>

Figure 3: Structured Programming



Definition 3 (Structured Programming) Structured programming is a technique for organizing and coding computer programs in which a hierarchy of *modules* is used, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. Three types of control flow are used:

- sequence,
- iteration, and
- selection.

□

It has been formally proved, that any algorithm can be expressed using those three basic building blocks. The ‘goto’-statement, therefore, is not only obsolete, but also considered *harmful*.

Figure 3 represents the building blocks as *Nassi-Schneidermann structure charts*.

8.1 Sequence

All statements of a sequence are executed one after the other.

Listing 5: Sequence

```

1  /**
2   * This class demonstrates a sequence of statements.
3   **/
4
5  class SequenceApp

```

```
6 {
7     public static void main(String[] args)
8     {
9         // statement 1
10        System.out.println("Hello, World!");
11        // statement 2
12        System.out.println("Programming in Java is not only fun,");
13        // statement 3
14        System.out.println("but also easy!");
15    } // main
16 } // SequenceApp
```

8.2 Iteration

All statements of an iteration are executed several times.

8.2.1 The while Statement

A while statement is executed by first evaluating $\langle Expression \rangle$. If its value is 'true', then execution continues with execution of $\langle Statement \rangle$ and re-evaluation of $\langle Expression \rangle$. If $\langle Expression \rangle$ evaluates to 'false', execution of the 'while'-statement completes.

Listing 6: The while Statement

```
1 while (  $\langle Expression \rangle$  )
2      $\langle Statement \rangle$ 
```

8.2.2 The do Statement

The 'do'-statement is executed by first executing $\langle Statement \rangle$. Then, $\langle Expression \rangle$ is evaluated. If its value is 'true', execution of the 'do'-statement continues by re-execution of $\langle Statement \rangle$. If the value of $\langle Expression \rangle$ is false, execution completes normally.

Listing 7: The do Statement

```
1 do
2      $\langle Statement \rangle$ 
3 while (  $\langle Expression \rangle$  ) ;
```

8.2.3 The for Statement

Listing 8: The for Statement

```
1 for (  $\langle Init-Expression \rangle$  ;  $\langle Expression \rangle$  ;  $\langle Update-Expression \rangle$  )
2      $\langle Statement \rangle$ 
```

The 'for' Statement is semantically equivalent to the following fragment:

Listing 9: Equivalence of the for Statement

```
1  $\langle Init-Expression \rangle$ 
2 while (  $\langle Expression \rangle$  )
3 {
4      $\langle Statement \rangle$ 
5      $\langle Update-Expression \rangle$ ;
6 }
```

8.3 Selection

Depending on a condition, either $\langle Statement-1 \rangle$ or $\langle Statement-2 \rangle$ are evaluated.

8.3.1 The if Statement

The ‘if’ statement allows conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both.

Listing 10: The if Statement

```
1 // if-statement
2 if ( <Expression> )
3     <Statement>
4
5 // if-else statement
6 if ( <Expression> )
7     <Statement-1>
8 else
9     <Statement-2>
10
11 // if-else-else statement
12 if ( <Expression-1> )
13     <Statement-1>
14 else if ( <Expression-2> )
15     <Statement-2>
16 else if ( <Expression-3> )
17     <Statement-3>
18 else if ( <Expression-4> )
19     <Statement-4>
20 else
21     <Statement-5>
```

8.3.2 The switch Statement

First $\langle Expression \rangle$ is evaluated. If it matches either $\langle ConstantExpression \rangle$, execution continues at the specified position. If no ‘break’-statement is present, execution continues normally until the end of the ‘switch’-statement, including all statements associated with following $\langle ConstantExpression \rangle$ s and the ‘default’-label.

If $\langle Expression \rangle$ matches neither $\langle ConstantExpression \rangle$, the execution continues at the ‘default’-label. If no ‘default’-label is present, the ‘switch’-statement completes successfully, although no statements have been executed.

Listing 11: The switch Statement

```
1 switch ( <Expression> ) {
2 case <ConstantExpression-1> :
3     <Statement-1>
4     [ break; ]
5 case <ConstantExpression-2> :
6     <Statement-2>
7     [ break; ]
8 case <ConstantExpression-3> :
9     <Statement-3>
10    [ break; ]
11 [
12 default :
13     <Statement-4>
14     [ break; // not necessary, but considered good style ]
15 ]
16 }
```

List of Tables

1	Types of Comments	4
2	Java Keywords	5
3	Separators	5
4	Java Datatypes	6
5	Escape Sequences	7
6	Arithmetic Operators	10
7	Comparison Operators	11
8	Logical Operators	11
9	Bitwise Logical Operators	11
10	Shift Operators	12
11	Priority of Operators	13

List of Figures

1	Kinds of Primitive Types	6
2	Kinds of Reference Types	9
3	Structured Programming	13

List of Acronyms

References

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilerbau*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1988. ISBN 3-89319-151-8.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 2nd edition, June 2000. ISBN 0-201-31008-2. URL <http://java.sun.com/docs/books/jls/index.html>.

Unicode. Unicode home page. URL <http://www.unicode.org/>.