4. Grundlegende Sprachelemente

ARINKO

- Whitespaces (Tab, Leerzeichen, Carriage Returns, Linefeeds etc.)
- Kommentare
- Bezeichner
- Schlüsselwörter
- Literale
- Interpunktionszeichen (inkl. Klammerungen)
- Operatoren

- Für einige dieser Elemente gibt es strikte Vorgaben (z.B. welche Wörter Schlüsselwörter sind).
- Für einige andere Elemente gibt es (mehr oder weniger zwingende) Konventionen (z.B. wie Bezeichner gebildet werden).
- Für andere wiederum sollte es in Projekten Vorgaben geben (z.B. wo und wann Whitespaces, wie sollten Kommentare beschaffen sein etc.)

- Java kennt 3 Kommentararten.
  1. Blockkommentar (aus C übernommen)
  2. Zeilenkommentar (aus C++ übernommen)
  3. Dokumentationskommentar (Java-eigen)

```
/**
  * Dokumentationskommentar
  * @param args - Parameter, die von der Kommandozeile
  * übergeben werden
  */
 public static void main(String[] args)
 {
     /*
      * Blockkommentar
      */

     int x; // Zeilenkommmentar
 }
```

- Wo immer in einem Programm selbstdefinierte Namen vorkommen, werden diese aus Bezeichnern gebildet.
  → Variablen, Klassen/Typen, Methoden, Pakete, Labels etc.
- Bezeichner bestehen aus einer beliebig langen Folge von Unicode-Buchstaben und Ziffern, müssen aber mit einem Buchstaben beginnen.
- Groß- und Kleinschrift wird unterschieden („case sensitive“)
- Typischerweise werden lateinische Buchstaben und arabische Ziffern benutzt, also **`a-z`**, **`A-Z`** und **`0-9`**.
- Weiterhin gelten der Unterstrich **`_`** und das Dollarzeichen **`$`** als Buchstaben, über deren Benutzung normalerweise Konventionen und Projektrichtlinien entscheiden.

- Konventionen sind erwünschte und (weltweit übliche) Richtlinien, um Programme, die leichter von Menschen lesbar sein sollen zu gestalten.
- Konventionen werden nicht (oder nicht restriktiv) vom Compiler überwacht, es gibt aber Tools die Konventionen einfordern können.
- Manche Konventionen dienen lediglich der Lesbarkeit, andere Konventionen ermöglichen weitergehenden Nutzen (siehe JavaBeans).
- Grundlegende wichtige Konventionen im Java-Umfeld:
  1. Namen von Typen (Klassen etc.) sind normalerweise eher substantivisch und beginnen daher mit einem Großbuchstaben:
     `Konto`
  2. Namen von Variablen sollen sich vom Namen der Klasse unterscheiden und beginnen daher mit einem Kleinbuchstaben. Man beachte:
     `konto.aendereZinssatz()` und `Konto.aendereZinssatz()` stellen andere Dinge dar!
  3. Namen von Methoden sind normalerweise an Verben angelehnt und beginnen daher mit einem Kleinbuchstaben.
  4. Bestehen Namen aus Wortfolgen, so werden diese zusammengeschrieben und an Wortgrenzen mit einem Großbuchstaben kenntlich gemacht („camel case“):
     `stelleMethodeInCamelCaseDar()`

- Schlüsselwörter in Java sind stets in Kleinbuchstaben.
- Einige Schlüsselwörter sind lediglich reserviert und dürfen in einem Programm nicht verwendet werden.
- Reservierte Schlüsselwörter dürfen nicht als Bezeichner verwendet werden!
- Die reservierten Schlüsselwörter in Java lauten:

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

Reserviert Java 1.4
Java 1.2 Java 5

- In neueren Java-Versionen existieren sog. „contextual keywords“, also Schlüsselwörter, die nicht reserviert sind und nur kontextbezogen zu solchen werden.
- Diese lauten für Java 17:

`exports`	`opens`	`requires`	`uses`
`module`	`permits`	`sealed`	`var`
`non-sealed`	`provides`	`to`	`with`
`open`	`record`	`transitive`	`yield`

- Java unterscheidet zwischen elementaren (primitiven) Datentypen und Referenztypen.
- Elementare Typen sind:
  - Integers (ganzzahlige)
  - Floating Points (Gleitkomma bzw. Fließpunktzahlen)
  - Boolean
- Referenztypen:
  - Klassen
  - Interfaces
  - Enums
  - Arrays von elementaren Typen und Referenztypen
  - (und einige Spezialfälle)

Elementarer Typ	*Speichermenge in Bytes*	*Wertebereich*
`byte`	1	-128 bis 127
`short`	2	-32768 bis 32767
`int`	4	-2^{31} bis $2^{31} - 1$
`long`	8	-2^{63} bis $2^{63} - 1$
`char`	2	0 bis 65535 `\u0000` bis `\uffff`

Elementarer Typ	*Speichermenge in Bytes*	*Wertebereich*
`float`	4	Wertebereich nach IEEE-754-Standard
`double`	8	
`boolean`		`true` `false`

- 4 verschiedene Zahlendarstellungen für ganze Zahlen:

dezimal	`31`
hexadezimal	`0x1f`
oktal	`037`
dual/binär	`0b00011111`

- Verschiedene Exponentialformate für Gleitkommazahlen (nach IEEE):

„normal“	`3.141`
$x10^{-3}$	`3141e-3`
$x10^{1}$	`.3141E1`

- Character-Literale enthalten immer ein einziges Zeichen:

```
'a' '+' 'ö'
'\''
'\\'
'\u0041'
```

- Zeichenketten bestehen aus Folgen von Chars.
- Eine Folge kann auch leer sein (Character-Literale können *nicht* leer sein).

```
"Java-Vorlesung"
""
"\"Java-Vorlesung\""
```

- Variablen sind (variable, aha!) Speicherplätze für Daten.
- Variablen werden im Programm deklariert, initialisiert und verwendet (gelesen und beschrieben).
- Variablen haben einen Typ, entweder ein elementarer oder ein Referenztyp.
- Variablen haben einen Namen (Bezeichner).

- Variablen eines elementaren Typs:

```
// Nur Deklaration
int a;
char c;

// Mit Initialisierung
int b = 1;
char x = 'x';

// Deklaration und Verwendung
long l;
l = 5; // "schreiben"
System.out.println(l); // "lesen"
```

- Objekte werden zur Laufzeit (aka dynamisch) erzeugt.
- Bei der Erzeugung erhält man eine Art Zeiger auf die Speicheradresse des Objektes.
- Da man (im Gegensatz zu C oder C++) mit diesen Zeigern nichts anderes machen kann, als auf das Element zuzugreifen (vgl. Pointer-Arithmetik in C/C++), spricht man in Java von ***Referenzen***
- Eine Referenz hat Typ und Wert:
  - Typ ist der Typ des referenzierten Objektes
  - Wert ist die (nicht sichtbare) Speicheradresse
- Eine Referenz erhält man in den meisten Fällen
  - Direkt durch den Operator new
  - Indirekt, z.B. als Rückgabewert einer Methode
- Referenzen können in Referenzvariablen abgelegt werden

```
public class KontoTest
{
    public static void ma
    {
        Konto konto1 = new Konto("DE68-0815-4711", 0.01, 0);
    }
}
```

Referenzvariable

Objekterzeugung

- Die Nullreferenz stellt dar, dass an dieser Stelle explizit ***kein*** Objekt vorhanden ist.
- Dafür existiert ein Schlüsselwort: `null`
- Die Nullreferenz ist typneutral (kann also auf alle Referenztypen angewendet werden)

```
public class KontoTest
{
    public static void main(String[] args)
    {
        Konto konto1 = new Konto("DE68-0815-4711", 0.01, 0);
        konto1 = null; // das Kontoobjekt wird nicht mehr über konto1
                       // referenziert
    }
}
```

- Java kennt nicht direkt das Konzept einer Konstante.
- Stattdessen existieren Variablen, die einmalig belegt werden können.
  (Merke: die erstmalige Verwendung ist "final", daher auch das Schlüsselwort...)

```
final double AVOGADRO = 6.02214076E23;
final String TEXT = "Ein Text. Na super.";

// Deklaration
final int a;

// erstmalige Belegung
a = 42;

// weitere Schreibzugriffe erzeugen einen Fehler
a = 0;
```

```
SymbolischeKonstanten.java: error: variable a might already
have been assigned
```

- Ab Java 10 kann die Typinformation bei lokalen Variablen ausgelassen werden.
- Empfehlenswert ist aber, dies nicht zu tun (Lesbarkeit, Nachvollziehbarkeit)
- Syntax:

```
[final] var name = initializer;
```

```
// Beispiele aus der Java Language Specification (JLS 14.4)
var a = 1; // Legal

var b = 2, c = 3.0; // Illegal: multiple declarators

var d[] = new int[4]; // Illegal: extra bracket pairs

var e; // Illegal: no initializer

var f = { 6 }; // Illegal: array initializer

var g = (g = 7); // Illegal: self reference in initializer

var h = null; // Illegal: null type
```

- Ein Ausdruck ist ein zusammengefasster typisierter Wert.
- Ausdrücke werden ausgewertet und dabei entsteht der Wert und Typ.
- Die Auswertung erfolgt in gewissen, einfachen Fällen vom Compiler, ansonsten zur Laufzeit.
- Ausdrücke bestehen
  - In einfachen Fällen aus einem Literal, dem Namen einer Variablen oder einer Konstanten, oder aus einem einfachen Methodenaufruf.
  - In komplexeren Fällen aus Kombinationen, die mit Hilfe von Operatoren gebildet werden.

- Der Typ eines Ausdrucks hängt von den Operatoren und vom Typ der Operanden ab.
- Dabei kann bei der Auswertung eine Typangleichung durchgeführt werden.

- Bei der Auswertung von Operatoren wird notfalls eine Typangleichung (implizite Typkonvertierung) vorgenommen.
- `byte`, `short` und `char` werden zu `int` konvertiert.
- Danach wird zum höherwertigen Typ konvertiert (d.h. immer in die vorgeblich verlustfreie Richtung):
  `int` → `long` → `float` → `double`
- Beispiel:

```
int o = 42;
long p = 5;
double q = o/p;
```

- Typangleichungen werden vom Compiler ohne Warnung durchgeführt, solange die Wertebereiche enthalten sind ("int passt in long").

- Soll in die verlustbehaftete Richtung konvertiert werden, muss man dem Java-Compiler mitteilen, dass man weiß, was man tut ("Trust me, I know what I'm doing").
- Diese Typkonvertierung nennt man explizite Typkonvertierung.
- Dafür kommt der Cast-Operator zum Einsatz, der syntaktisch aus einem Paar vorgestellter runder Klammern besteht:

```
double zahl = 12.34;
int zwoelf = (int) zahl; // Die Nachkommastellen sind weg
char zero = (char) (zwoelf * 4);
```

Additiv	Addition	+
	Subtraktion	-
Multiplikativ	Multiplikation	*
	Division	/
	Rest der Division	%

- Die Operanden sind immer vom Typ eines elementaren Zahlenwertes (also keine Referenzen [Ausnahme + und Strings] oder boolean).
- Der Typ des Resultats hängt vom Typ der Operanden ab (Typangleichungen).

Relational	kleiner als	<
	kleiner gleich	<=
	größer als	>
	größer gleich	>=
Gleichheit	gleich	==
	ungleich	!=

- Die Relationalen Vergleichsoperatoren können nur arithmetische Ausdrücke vergleichen.
- Die Operatoren für Gleichheit können auch auf Referenzen arbeiten.
- Das Ergebnis ist immer ein Boolescher Wert.

Logisch	AND	&
	XOR	^
	OR	|
	NOT	!
Logisch, bedingt	bedingtes AND	&&
	bedingtes OR	||

- Variante 1: beide Operanden sind boolesch, dann ist das Resultat ebenfalls boolean (hier handelt es sich um logische Operatoren)
- Variante 2: beide Operanden haben einen ganzzahligen Typ, dann ist das Ergebnis die Bit-für-Bit-Anwendung der logischen Verknüpfung und ebenfalls ein ganzzahliger Typ mit der nötigen Genauigkeit (hier handelt es sich um bitwise Operatoren)
- Die bedingten, logischen Operatoren können ausschließlich auf boolesche Operanden angewendet werden. Die jeweils rechte Seite des Ausdrucks wird nur wenn nötig ausgewertet (daher "bedingt").

- Der Konditionaloperator ? : ist dreistellig (daher gerne auch "ternärer" Operator).
- Man kann ihn lax so verstehen:
  hä? ja : nein
- Somit ist der Ausdruck vor dem ? ein boolescher Ausdruck, die beiden Werte dahinter sollten vom selben (oder verwandten) Typ sein.
- Beispiel:

```
int x = 5;

System.out.println(x > 0 ? "positiv" : "negativ");
```

wenn ja, dann nimm den Wert "positiv" an

wenn nein, dann nimm den Wert "negativ" an

hä? ist x > 0???

- Bit-Shift-Operatoren arbeiten nur mit ganzzahligen Operanden.
- Der linke Operand ist der Ausgangswert, der rechte Operand gibt an, um wieviele Bits dieser in Richtung, die das Operatorsymbol angibt, verschoben werden soll.

Bewegung nach links	<<
Bewegung nach rechts	>>
Bewegung nach rechts, vorzeichenlos	>>>

```
System.out.println(-16 << 3); // -128
System.out.println(-16 >> 3); // -2
System.out.println(-16 >>> 3); // 536870910
```

- Die Zuweisung mit Hilfe von = ist in Java (wie in C/C++/C#...) ebenfalls ein Operator.
- Somit kann der Zuweisungsoperator selbst wieder in Ausdrücken vorkommen, was wundervoll verständliche und Seiteneffekt-arme Programme erzeugt...

```
// "normale" Verwendung
int a = 12;
int b = 4;
int c = a * b;

// Seiteneffekte
int d = c > 20 ? (c - 20) : (b = 0);

boolean f = (d < 5) && (b == (a = 0)); // ach so...
```

- Arithmetische und Shift-Operatoren können mit dem Zuweisungsoperator kombiniert werden, wenn links und rechts derselbe Operand vorkommt.
- Aus `a = a + 5` wird dann `a += 5`
- Noch kürzer geht es für Addition und Subtraktion von eins (Inkrement/Dekrement)
- Aus `a = a + 1` wird `a += 1` wird `a++`
  (Hier sieht man, wie der Name C++ zustande kam)

- Für die Inkrement (++) und Dekrement-Operatoren (--) gibt es Post- und Präfix-Notationen.
- Postfix-Notation: für a++ und a-- ist der Wert des Ausdrucks der Wert des Operanden vor der Modifikation.
- Präfix-Notation: für ++a und --a ist der Wert des Ausdrucks der Wert des Operanden nach der Modifikation.

- Ein Statement (Ausdrucksanweisung, bzw. salopp eine Java-Anweisung) ist ein Ausdruck gefolgt von einem Semikolon ;
- Zulässige Ausdrücke:
  - Zuweisungen
  - Inkrements und Dekrements
  - Objekterzeugung mit new
  - Methodenaufrufe
  - das leere Statement (leere Zeile mit Semikolon, nicht sehr nützlich)

```
c = a * b; // Zuweisung

new Konto("0815", 0.01, c); // Objekterzeugung

c++; // Inkrement

System.out.println(c); // Methodenaufruf
```

- Ein Block ist eine zusammengesetzte Anweisung.
- Er wird in allen C-ähnlichen Sprachen mit den geschweiften Klammern gebildet { }.
- Ein Block
  - definiert einen Geltungsbereich für in ihm enthaltene Elemente
  - kann wiederum in einem Block stehen (beliebige Schachtelung)

- Die geschweiften Klammern für die Klassendeklaration sind syntaktisch angelehnt und haben einige Gemeinsamkeiten, sind aber nicht dasselbe wie ein Block in einer Methode.

- `if (`*condition*`) statement [ else` *statement* `];`

```
int a = 42;

if (a == 42)
{
  System.out.println("Cool.");
}

if (a >= 0)
{
   System.out.println("Schon mal positiv.");
}
else
{
  System.out.println("Das fängt ja gut an.");
}
```

- Der Switch-Ausdruck muss einen der folgenden Werte haben:
  - `char`, `byte`, `short`, `int`
  - ein enum-Typ (Java 5)
  - ein `String` (Java 7)

```
int tag = 3;

switch (tag)
{
    case 1: System.out.println("Montag"); break;
    case 2: System.out.println("Dienstag"); break;
    case 3: System.out.println("Mittwoch"); break;
    //...
    default: System.out.println("Den Tag gibts nich");
}
```

- `case`-Marken dürfen nicht mehrfach vorkommen.
- Sie dürfen aber unmittelbar aufeinander folgen.
- Abarbeitung des `switch`:
  - der `switch-Ausdruck` wird ausgewertet und mit den `case-Marken` verglichen.
  - Stimmt ein Wert überein, wird der Kontrollfluss an der entsprechenden Marke fortgesetzt und Java vergisst sozusagen das `switch`
  - Wird keine Übereinstimmung erzielt, wird das Programm an der `default-Marke` fortgeführt, falls diese existiert.
- Sobald der Sprung im Kontrollfluss ausgeführt ist, werden alle Anweisungen ab der Zielmarke ausgeführt.
- Die Anweisungen laufen entweder einfach weiter bis zum Ende des `switch-Blocks` („fall-through“) oder bis zu einem angetroffenen `break`.

- Java 13 führt `switch` als Expression ein. Hier darf entweder mit der neuen Pfeilsyntax -> oder mit dem ebenfalls neu eingeführten Schlüsselwort `yield` gearbeitet werden.
- Die Pfeilsyntax erlaubt kein „fall-through“, im Gegensatz zur Doppelpunkt-Syntax.

```
String name = "Dienstag";

int tag = switch (name)
{
  case "Montag" -> 1;
  case "Dienstag" -> 2;
  case "Mittwoch" -> 3;
  // ...
  default -> 0;
};
```

```
String name = "Dienstag";

int tag = switch (name)
{
  case "Montag": yield 1;
  case "Dienstag": yield 2;
  case "Mittwoch": yield 3;
  // ...
  default: yield 0;
};
```

- `while (`*condition*`) statement;`
- Das Statement wird wiederholt ausgeführt, solange die Bedingung zutrifft (true).

- `do statement while (`*condition*`);`
- Das Statement wird zumindest einmal ausgeführt und dann noch so oft, solange die Bedingung zutrifft.

```
int a = 5;

while (a > 0)
{
    System.out.println(a--);
}

do
{
    System.out.println(a++);
}
while (a <= 5);
```

- Die for-Schleife ist die Allzweckwaffe des Java-Programmierers.
- Syntax 1: `for (`*init*`;` *condition*`;` *update*`)` *statement*`;`
- Syntax 2: `for (`*variable* `:` *iterable*`)` *statement*`;` [später erklärt]

1. Zunächst wird der Initialisierungsabschnitt durchlaufen.
2. Dann die Bedingung überprüft. Sollte diese `true` sein, dann wird das Statement ausgeführt.
3. Abschließend wird das update-Statement (Schleifenschritt-Epilog) ausgeführt und mit 2. weitergemacht.

- Jeder Abschnitt darf dabei leer sein.
- Im krassesten Fall sieht der Schleifenkopf so aus:

```
for (;;)
```

was einer Endlosschleife entspricht.

- Durch den aus C entlehnten Kommaoperator für das "hintereinander ausführen", der in Java nur in for-Schleifen erlaubt ist, können auch komplexere Gebilde gebaut werden.
- Beispiel:

```
// Klassische Zählschleife
for (int i = 0; i < 42; i++)
{
    System.out.println(i);
}

// komplexere Schleife
for (int a = 42, b = 1; a > b; a--, b*= 2)
{
    System.out.println(a + " " + b);
}
```

Komma-Operator

- `break` bricht die jeweilige Schleife ab.
- `continue` bricht den Schleifendurchgang ab.

- Geschachtelte Schleifen können mit Labels versehen werden, dann bricht
  `break label;`
  die benannte Schleife ab und
  `continue label;`
  macht mit dem nächsten Durchgang der benannten Schleife weiter.

Label

```
gesamt:
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        if (i * j > 42)
        {
            System.out.println(i + " " + j);
            break gesamt;
        }
    }
}
```

- Arrays sind Objekte.
- Sie bestehen aus einer bestimmten (festen) Anzahl von Variablen des selben Typs.
- Arrays und Arrayelemente haben keinen Namen, man kann nur mit Hilfe der Objektreferenz auf dem Array arbeiten.
- Komponenten eines Arrays:
  - Die Anzahl ist unveränderlich.
  - Die n Komponenten eines Arrays sind von 0 bis n-1 durchnummeriert.
  - Diese Komponentennummer heißt Index.
  - Der Komponent-weise Zugriff erfolgt immer über den Index.

- Array-Variablen sind Referenzvariablen eines Array-Typs.
- Diese bilden sich aus
  - Angabe des (gemeinsamen) Inhaltstyps der Komponenten
  - gefolgt von eckigen Klammern [ ]

```
int[] zahlen;
```

- Arrays werden dynamisch erzeugt (d.h. zur Laufzeit), allerdings kann man in div. Syntaxvarianten bereits eine Vorbelegung durchführen.
- Eine Syntaxvariante kann nur bei der Deklaration verwendet werden, die beiden anderen auch bei der Wiederverwendung einer Array-Variablen.

```
// Nur bei Deklaration
int[] zahlen = { 1, 2, 4, 8, 16 };

// Deklaration und Wiederverwendung
int[] zahlen = new int[] { 1, 2, 3, 4, 5 };
zahlen = new int[] { 3, 6, 9, 12, 15 };

// Deklaration und Wiederverwendung
int[] zahlen = new int[20];
zahlen = new int[40];
```

- Der Zugriff auf einzelne Elemente des Arrays erfolgt über den Index-Operator [].
- Als Index dürfen nur `byte`, `short`, `char` oder `int`-Werte verwendet werden.

```
int[] zahlen = { 1, 2, 4, 8, 16 };

// Vertausche 1 und 16
int h = zahlen[0];
zahlen[0] = zahlen[4];
zahlen[4] = h;
```

- Die Gesamtzahl der Elemente eines Arrays nennt man die Länge des Arrays.
- Die Länge ist unveränderlich und kann mit `.length` abgefragt werden.

```
System.out.println(zahlen.length); // 5
```

- Sehr häufig möchte man in Programmen die Elemente eines Arrays ausgeben.
- Dies ist die "klassische" Variante:

```
int[] zahlen = { 1, 2, 4, 8, 16 };

for (int i = 0; i < zahlen.length; i++)
{
    System.out.println(zahlen[i]);
}
```

- Seit Java 5 kann für den **lesenden** Zugriff beim Iterieren auch die sog. "enhanced for loop" verwendet werden:

```
int[] zahlen = { 1, 2, 4, 8, 16 };

for (int zahl : zahlen)
{
    System.out.println(zahl);
}
```

- Sie wird auch als "foreach"-Loop bezeichnet, weil die Lesart folgende ist:

  `for (int zahl : zahlen)`

  for each int value called "zahl" found in [array] "zahlen" do…

1	L	`. []`	Punkt- und Index-Operator

2	R	`++ --`
		`() new`
		`!`
			Inkrement, Dekrement
			Casting und New
			Logische Negation

3	L	`* / %`	Multiplikative Operatoren
4	L	`+ -`	Additive Operatoren
5	L	`>> <<`	Bit-Shift-Operatoren
6	L	`< <= > >=`	Relationale Operatoren
7	L	`== !=`	Gleichheits-Operatoren
8	L	`&`	Logisches UND
9	L	`^`	Logisches Exclusiv-ODER
10	L	`|`	Logisches Inklusiv-ODER
11	L	`&&`	Bedingtes logisches UND
12	L	`||`	Bedingtes logisches ODER
13	R	`?:`	Konditional-Operator
14	R	`= += -= ...`	Zuweisungsoperatoren


Praxis
Theorie