

# Ausdrücke und Operatoren

Prof. Dr. Helmut G. Folz



## Ausdrücke

- Ein **Ausdruck** steht für einen Wert, der entweder bereits bei der Übersetzung des Programmtextes ermittelt werden kann oder erst zur Laufzeit des Programmes berechnet wird.
- Ein Ausdruck besteht aus
  - ⇒ Operatoren (z. B. +, -, \*, /)
  - ⇒ Operanden (z. B. Konstanten, Variablen, Funktionswerten)
  - ⇒ Interpunktionszeichen (z.B. Klammern)
- Im einfachsten Falle ist ein Ausdruck in Java eine **Konstante** oder eine **Variable**.

## Ausdrücke

- Bei der Ermittlung eines Werts eines Ausdrucks sind **Seiteneffekte** möglich, d. h. bei bestimmten Operatoren ist es möglich, dass der Wert eines Operanden verändert werden kann.
- Jeder Ausdruck hat einen Wert eines bestimmten Datentyps.
- Der Wert eines Ausdrucks wird bestimmt durch die Typen der Operanden und durch die Operatoren.

## Prioritätentabelle der Operatoren (1)

Priorität	Operatoren	Erläuterung	Assoziativität
Priorität 1	[ ]	Arrayindex	links
	( )	Methodenaufruf	
	.	Komponentenzugriff	
Priorität 2	++ --	Inkrement/Dekrement (Präfix oder Postfix)	rechts
	+ -	Vorzeichen (unär)	
	~	bitweises Komplement	
	!	logischer Negationsoperator	
	(type)	Typkonvertierung (cast)	
	new	Erzeugung	
Priorität 3	* / %	Multiplikation, Division, Modulo (Rest)	links
Priorität 4	+ -	Addition, Subtraktion	links
	+	Stringverkettung	links
Priorität 5	<< >> >>>	Shift-Operatoren	links
Priorität 6	< <=	Vergleich kleiner, kleiner gleich	links
	> >=	Vergleich größer, größer gleich	links
	instanceof	Typüberprüfung eines Objekts	links

## Prioritätentabelle der Operatoren (2)

Priorität	Operatoren	Erläuterung	Assoziativität
Priorität 7	== !=	Gleichheit, Ungleichheit	links
Priorität 8	&	bitweises UND	links
	&	logisches UND	links
Priorität 9	^	bitweises Exclusives ODER	links
	^	logisches Exclusives ODER	links
Priorität 10		bitweises ODER	links
		logisches ODER	links
Priorität 11	&&	logisches UND	links
Priorität 12		logisches ODER	links
Priorität 13	? :	bedingte Auswertung	rechts
Priorität 14	=	Wertzuweisung	rechts
	*= /= %= +=	kombinierte Zuweisungsoperatoren	rechts
	-= <<= >>=		
	>>>=		
	&= ^=  =		

## Arithmetische Operatoren

Priorität	Operator	Bedeutung	Beispiele
1	+ -	unäre Vorzeichenoperatoren	+1 -a
2	* / %	Multiplikations-, Divisions- und Modulo-Operator	a * b 2 * -3 3 / 2.0 17 % 4
3	+ -	Additions- und Subtraktionsoperator	a + b 2 - 3.14 -a + -b

## Quersumme

- Aufgabe: Berechne die Quersumme einer ganzen Zahl
- Mögliche Vorgehensweise?

## Beispiel: Quersumme

```
import java.util.Scanner;

public class Quersumme {
    private Scanner input = new Scanner(System.in);

    public long quersumme(long zahl) {
        long quersumme = 0;
        while(zahl > 0) {
            quersumme = quersumme + zahl % 10;
            zahl = zahl / 10;
        }
        return quersumme;
    }

    public void start() {
        long zahl;
        System.out.print("Positive Zahl eingeben: ");
        zahl = input.nextLong();
        System.out.println("Quersumme: " + quersumme(zahl));
    }
}
```

## Operatoren- und Ergebnistyp

- Als Operanden für die vier arithmetischen Operatoren können **Ganzzahl-** und **Gleitpunktwerte** gemischt werden.
- Wenn zwei Ganzzahl-Werte verknüpft werden, ergibt sich wieder ein Ganzzahl-Wert als Ergebnis.
- Wenn einer oder beide Operanden ein Gleitpunktwert ist, ist auch das Ergebnis ein Gleitpunktwert.

## Assoziativität

- Bei den Operatoren + und \* gilt das **Assoziativgesetz**:  
$$a + b + c = (a + b) + c = a + (b + c)$$
$$a * b * c = (a * b) * c = a * (b * c)$$
- Wie ist das bei den Operatoren -, / und % ?
- Die Operatoren +, -, \*, / und % sind **links-assoziativ**, d. h. sie werden von links nach rechts ausgewertet.  
$$a - b - c = (a - b) - c$$
$$a / b / c = (a / b) / c$$

# Priorität

- Wenn Operatoren unterschiedlicher Prioritäten nebeneinander stehen, werden sie in der Reihenfolge fallender Priorität angewendet.
  - ⇒ Punktrechnung geht vor Strichrechnung,
  - ⇒ Vorzeichenoperatoren vor den anderen
- Wenn Operatoren gleicher Priorität nebeneinander stehen, werden sie in **Richtung der Assoziativität** angewendet.
- Die von Operator-Assoziativitäten und -Prioritäten vorgegebene Auswertungsreihenfolge eines zusammengesetzten Ausdrucks lässt sich mit runden Klammern beeinflussen.
  - Teilausdrücke in runden Klammern werden immer zuerst von innen nach außen ausgerechnet.
  - Formal spielen runde Klammern damit die Rolle eines Operators höchster Priorität.

# Einschub: Mathematische Funktionen

- Viele mathematische Funktionen (Quadratwurzel, trigonometrische und logarithmische Funktionen, etc.) werden häufig gebraucht.
- Bei Java werden diese als Methoden der Standardklasse `java.lang.Math` zur Verfügung gestellt.
- Beispiel:

```
double x;  
x = Math.sqrt(2.0); // Quadratwurzel von 2
```

## Die Klasse java.lang.Math (1)

<i>Konstanten</i>	<i>Bedeutung</i>
<code>static final double E = 2.7182818284590452354</code>	Basiszahl für den natürlichen Logarithmus
<code>static final double PI = 3.14159265358979323846</code>	die Kreiszahl Pi

## Die Klasse java.lang.Math (2)

<i>Methode</i>	<i>Bedeutung</i>
<code>static int abs(int x)</code>	Absolutbetrag $ x $ gibt es auch für die Typen long, float und double
<code>static double random()</code>	Gibt eine Pseudozufallszahl zwischen 0.0 und 1.0 zurück
<code>static double acos(double x)</code>	$\arccos x$ für x zwischen 0.0 und pi.
<code>static double asin(double x)</code>	$\arcsin x$ für x zwischen -pi/2 und pi/2.
<code>static double atan(double x)</code>	$\arctan x$ für x zwischen -pi/2 und pi/2.
<code>static double sin(double x)</code>	$\sin x$
<code>static double sinh(double x)</code>	$\sinh x$
<code>static double cos(double x)</code>	$\cos x$
<code>static double cosh(double x)</code>	$\cosh x$
<code>static double tan(double x)</code>	$\tan x$
<code>static double tanh(double x)</code>	$\tanh x$

## Die Klasse java.lang.Math (3)

<i>Methode</i>	<i>Bedeutung</i>
static double exp(double x)	$e^x$
static double expm1(double x)	$e^x - 1$
static double log(double x)	$\ln x$
static double log10(double x)	$\log x$
static double pow(double x, double y)	$x^y$
static double sqrt(double x)	Quadratwurzel
static double cbrt(double x)	Dritte Wurzel
static double floor(double x)	Größte ganze Zahl $\leq x$
static double ceil (double x)	Kleinste ganze Zahl $\geq x$
static long round (double x)	Runden zur nächsten ganzen Zahl
static int round (float x)	

## Beispiel: Zufallszahlen1 (1)

```
public class Zufallszahlen1 {
    public void start() {
        int i = 0;

        while (i < 20) {
            System.out.println(Math.random() + "\t"
                               + Math.random());
            i = i + 2;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        new Zufallszahlen1().start();
    }
}
```



## Beispiel: Zufallszahlen1 (2)

```
/*
0.15006903515022918      0.947456225693527
0.28589546106460606      0.5872083739687138
0.882717163648184        0.9916706042800885
0.1626543401715087        0.32366954195912423
0.4972098025887347        0.40194394431471614
0.07171537098026193        0.17525001078729485
0.4599988655839422        0.0282168241724986
0.6566847491316734        0.2748917543242655
0.6475045823488491        0.5417758823138883
0.31363398469290116        0.9254715904723687
*/
```

## Beispiel: Zufallszahlen2 (1)

```
public class Zufallszahlen2 {
    private Scanner input = new Scanner(System.in);
    /** Bestimme eine Zufallszahl zwischen 0 und max */
    public int nextInt(int max) {
        return (int) round(max * random());
    }

    public void start() {
        int i = 0, zufall;
        System.out.print("Obere Grenze: ");
        int max = input.nextInt();

        while (i < 20) {
            zufall = nextInt(max);
            System.out.print(zufall + "\t");
            i = i + 1;
            if (i % 5 == 0)
                System.out.println();
        }
        System.out.println();
    }
}
```

## Beispiel: Zufallszahlen2 (2)

```
public static void main(String[] args) {
    new Zufallszahlen2().start();
}
/*
Obere Grenze: 1000
622    181    985    164    219
480    713    468    930    705
849    25     943    63     64
394    217    945    640    200
*/
```

## Beispiel: Zufallszahlen3 (1)

```
public class Zufallszahlen3 {
    /**
     * start: Starten der Tests
     */
    public void start() {
        java.util.Random random = new java.util.Random();
        System.out.println("double Zufallszahlen");
        for (int i = 0; i < 5; i++)
            System.out.print(random.nextDouble() + " ");

        System.out.println("\n\nLong Zufallszahlen");
        for (int i = 0; i < 5; i++)
            System.out.print(random.nextLong() + " ");

        System.out.println("\n\nint Zufallszahlen");
        for (int i = 0; i < 5; i++)
            System.out.print(random.nextInt() + " ");
    }
}
```

## Beispiel: Zufallszahlen3 (2)

```
System.out.println("\n\nint Zufallszahlen von 0 - 100");
    for (int i = 0; i < 20; i++)
        System.out.print(random.nextInt(100) + " ");

    System.out.println("\n\n");
}

public static void main (String[] args) {
    new Zufallszahlen3().start();
}
}
```

## static import

- Es ist umständlich, wenn man vor dem Aufruf des Methodennamens immer den Klassennamen angeben muss, wie dies bei Klassenmethoden leider notwendig ist.
- Seit Java 5 gibt es hier eine Möglichkeit dies zu vereinfachen:
  - ⇒ `import static java.lang.Math.sqrt;`  
Klassenmethode sqrt importieren
  - ⇒ `import static java.lang.Math.*;`  
alle Klassenmethoden der Klasse Math importieren

## Beispiel: FunktionsTest (1)

```
import static java.util.Scanner;  
import static java.lang.Math.*;  
  
public class FunktionsTest {  
    private Scanner input = new Scanner(System.in);  
    public void start() {  
        double x = -2.0;  
        System.out.println("x          sqrt(x)");  
        // Quadratwurzeln ausgeben  
        while (x <= 20.0) {  
            System.out.println(x + "\t" + sqrt(x));  
            x = x + 1.0;  
        }  
    }  
}
```

x	sqrt(x)
-2.0	NaN
-1.0	NaN
0.0	0.0
1.0	1.0
2.0	1.4142135623730951
3.0	1.7320508075688772
4.0	2.0
5.0	2.23606797749979

Prof. Dr. H. G.

-23-

## Beispiel: FunktionsTest (2)

```
input.next();  
x = 0.0;  
// Kubikwurzeln ausgeben  
System.out.println("x\tx^1/3");  
while (x <= 20.0) {  
    System.out.println(x + "\t" + pow(x, 1.0/3.0)  
                        + "\t" + cbrt(x));  
    x = x + 1.0;  
}
```

x	x^1/3	cbrt(x)
0.0	0.0	0.0
1.0	1.0	1.0
2.0	1.2599210498948732	1.2599210498948732
3.0	1.4422495703074083	1.4422495703074083
4.0	1.5874010519681994	1.5874010519681996
5.0	1.709975946676697	1.709975946676697
6.0	1.8171205928321397	1.8171205928321397
7.0	1.912931182772389	1.9129311827723892
8.0	2.0	2.0
...		

Prof. Dr. H. G. Folz

Programmierung 1: Ausdrücke und Operatoren:

-24-

## Beispiel: FunktionsTest (3)

```
input.next();  
x = 0.0;  
// Exponentialfunktion ausgeben  
System.out.println("x      exp(x)");  
while (x <= 100.0) {  
    System.out.println(x + "\t" + exp(x));  
    x = x + 1.0;  
}
```

x	exp(x)
0.0	1.0
1.0	2.7182818284590455
2.0	7.38905609893065
3.0	20.085536923187668
4.0	54.598150033144236
5.0	148.4131591025766
6.0	403.4287934927351
7.0	1096.6331584284585
8.0	2980.9579870417283
9.0	8103.083927575384
10.0	22026.465794806718

## Beispiel: Klasse MyMath (1)

```
public class MyMath {  
  
    public static double sqr(double x) {  
        return x * x;  
    }  
  
    public static double pow(double x, int n) {  
        assert n >= 0 : "n muss >= 0 sein!";  
        double wert = 1.0;  
        while (n > 0) {  
            wert = wert * x;  
            n = n - 1;  
        }  
        return wert;  
    }  
}
```

## Beispiel: Klasse MyMath (2)

```
public static boolean even(int n) {  
    return (n % 2) == 0;  
}  
  
public static boolean odd(int n) {  
    return (n % 2) != 0;  
    // alternativ: return !even(n);  
}
```

## Klasse MyMath: Testklasse

```
public class MyMathTest {  
    public static void main(String[] args) {  
        Scanner input = new Scanner(System.in);  
        double a;  
        System.out.print("Double-Wert eingeben: ");  
        a = input.nextDouble();  
        double aQuadrat = MyMath.sqr(a);  
        System.out.println(aQuadrat);  
  
        int n;  
        System.out.print("Positive ganze Zahl eingeben: ");  
        n = input.nextInt();  
        double aHochN = MyMath.pow(a, n);  
  
        System.out.println(aHochN);  
    }  
}
```

## Wertzuweisungen

- Mit einer Wertzuweisung (engl.: "assignment") wird einer Variablen ein neuer Wert zugewiesen.
- Eine Wertzuweisung besteht aus einer linken Seite (das Ziel) und einer rechten Seite (die Quelle):  
`variable = ausdruck;`
- Links muss eine Variable stehen, rechts kann ein beliebiger Ausdruck stehen.
- Eine Wertzuweisung läuft in zwei Schritten ab, die nacheinander abgewickelt werden:
  - der Wert des Ausdrucks auf der rechten Seite wird ausgerechnet;
  - dieser Wert wird an die Variable auf der linken Seite zugewiesen;

## L-Wert und R-Wert

- Auf der linken Seite einer Zuweisung muss ein sogenannter **L-Wert** (engl. *lvalue*) stehen,
  - ⇒ d.h. normalerweise eine Variable,
  - ⇒ allgemein jedoch ein Ausdruck, der einen Ort im Speicher darstellt, der einen zugewiesenen Wert aufnehmen kann.
- Rechts darf ein allgemeiner Ausdruck (im Rahmen der Typverträglichkeit) stehen, der sich zu einem Wert ausrechnen lässt.
- Ein Ausdruck, der auf der rechten Seite einer Wertzuweisung stehen darf, wird **R-Wert** (engl. *rvalue*) genannt.
- Beispiel:  

```
int a;    // a ist ein L-Wert
2 + 3;    // ein R-Wert
```

## Wertzuweisung als Ausdruck

- Der **Zuweisungsoperator** = ist ein Operator niedriger Priorität. Eine Wertzuweisung ist daher ebenfalls ein Ausdruck.
- Der "Wert" einer Wertzuweisung ist dabei der zugewiesene Wert (ein R-Wert). Man kann daher eine Wertzuweisung auch folgendes schreiben:

**a = b = 2;**

- Derartige mehrfache Zuweisungen werden als Kettenzuweisungen bezeichnet. Der Zuweisungsoperator bindet anders als die arithmetischen Operatoren von rechts nach links, ist also **rechts-assoziativ**.

## Wertzuweisung mit Operatoren

- Eine Wertzuweisung wird oft eingesetzt, um den Wert einer einzelnen Variablen gegenüber dem vorhergehenden Wert zu modifizieren, wie z.B. in:

```
a = a + 2;    // Wert von a um 2 hochzaehlen
a = a - 1;    // Wert von a um 1 vermindern
a = a / 2;    // Wert von a halbieren
a = a * 10;   // Wert von a verzehnfachen
a = a % 10;   // Wert von a auf a modulo 10 setzen
```

- Abkürzung: Statt

**variable = variable operator ausdrück**

schreibt man kürzer

**variable operator= ausdrück**



## Wertzuweisung mit Operatoren

- Die Beispiele lassen sich kürzer schreiben als:  

```
a += 2;      // Wert von a um 2 hochzaehlen  
a -= 1;      // Wert von a um 1 vermindern  
a /= 2;      // Wert von a halbieren  
a *= 10;     // Wert von a verzehnfachen  
a %= 10;     // Wert von a auf a modulo 10 setzen
```
- Diese Operatorzuweisungsoperatoren gibt es für alle arithmetischen Operatoren, sowie auch die noch nicht besprochenen Bit-Operatoren und Bitshift-Operatoren.
- Sie haben die gleiche niedrige Priorität wie der normale Zuweisungsoperator.

## Inkrement und Dekrement

	Ausdruck	Bedeutung	Wert des Ausdrucks
Präfix-Schreibweise	++a	a um eins erhöhen	a nach der Erhöhung
	--a	a um eins vermindern	a nach der Verminderung
Postfix-Schreibweise	a++	a um eins erhöhen	a vor der Erhöhung
	a--	a um eins vermindern	a vor der Verminderung

## Inkrement und Dekrement

```
public class InkrementDekrement {
    public void start() {
        int a = 1;
        int b = a++; // b == 1, a == 2
        System.out.println("a = " + a + ", b = " + b);

        int c = 1;
        int d = --c; // d == 0, c == 0
        System.out.println("c = " + c + ", d = " + d);

        double x = 2.5;
        double y = x--; // x = 1.5, y = 2.5
        System.out.println("x = " + x + ", y = " + y);

        a = b = 1;
        c = ++a + ++b; // c = 3
        System.out.println("c = " + c);
    }
}
```

Prof. Dr. H. G. Folz

Programmierung 1: Ausdrücke und Operatoren:

-35-

## Inkrement und Dekrement

```
public class DekrementTest {
    public void start() {
        int i = 5;
        while (i-- > 0) —→ 4 3 2 1 0
            System.out.print(i + " ");

        System.out.println();

        i = 5;
        while (--i > 0) —→ 4 3 2 1
            System.out.print(i + " ");

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

Prof. Dr. H. G. Folz

Programmierung 1: Ausdrücke und Operatoren:

-36-

## Logische Operatoren

Java kennt die folgenden logischen Operatoren

- Logische UND-Operatoren :      **&&**    **&**
- Logische ODER-Operatoren:      **||**    **|**
- Logischer Negations-Operator:    **!**
- Logischer XOR-Operator:          **^**
- Eigenschaften
  - ⇒ Die logischen UND/ODER-Operatoren sind zweistellig, der logische Negationsoperator ist einstellig.
  - ⇒ Die logischen Operatoren können nur auf Operanden vom Typ boolean angewandt werden. Der Ergebnistyp ist boolean.
  - ⇒ Die Operatoren &, | und ^ sind in gegenüber C/C++ neu bei Java.

## Logische UND-Operatoren: A && B und A & B

A	B	A && B
false	false	false
false	true	false
true	false	false
true	true	true

- Wird der Operator **&** zwischen zwei Operanden verwendet, so wird der rechte Operand immer ausgewertet, egal ob der linke Operand true oder false ist.
- Wird dagegen der Operator **&&** angewendet, so wird der rechte Ausdruck nur dann ausgewertet, wenn der linke Ausdruck true ist

## Logische ODER-Operatoren: $A \parallel B$ , $A \mid B$

A	B	$A \parallel B$
false	false	false
false	true	true
true	false	true
true	true	true

- Wird der Operator  $\mid$  zwischen zwei Operanden verwendet, so wird der rechte Operand immer ausgewertet, egal ob der linke Operand true oder false ist.
- Wird dagegen der Operator  $\parallel$  angewendet, so wird der rechte Ausdruck nur dann ausgewertet, wenn der linke Ausdruck false ist

## Logische ODER-Operatoren: $A \wedge B$

A	B	$A \wedge B$
false	false	false
false	true	true
true	false	true
true	true	false

- Der Operator  $\wedge$  für exklusives Oder ist bei Java neu gegenüber C/C++.
- Hier werden generell beide Ausdrücke ausgewertet (müssen sie auch!)

## Logische Operatoren: Unterschiede

```
public class Operatortest {  
    public void start() {  
        int a = 1, b = 1;  
  
        if (a++ < 0 && b++ > 0)  
            System.out.println("Kann nicht sein!");  
  
        System.out.println("a = " + a + ", b = " + b);  
        → a = 2, b = 1  
  
        a = b = 1;  
        if (a++ < 0 & b++ > 0)  
            System.out.println("Kann nicht sein!");  
  
        System.out.println("a = " + a + ", b = " + b);  
        → a = 2, b = 2  
    }  
}
```

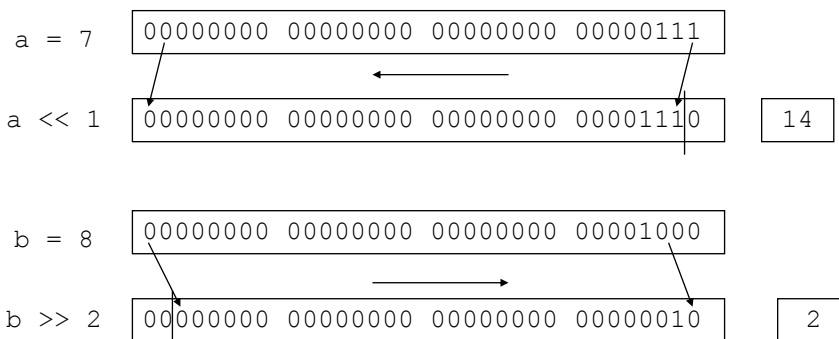
## Logische Operatoren: Unterschiede

```
a = b = 1;  
if (a++ > 0 || b++ > 0)  
    System.out.println("a++ > 0 || b++ > 0");  
  
System.out.println("a = " + a + ", b = " + b);  
    → a++ > 0 || b++ > 0  
      a = 2, b = 1  
  
a = b = 1;  
if (a++ > 0 | b++ > 0)  
    System.out.println("a++ > 0 | b++ > 0");  
  
System.out.println("a = " + a + ", b = " + b);  
    → a++ > 0 | b++ > 0  
      a = 2, b = 2  
}
```

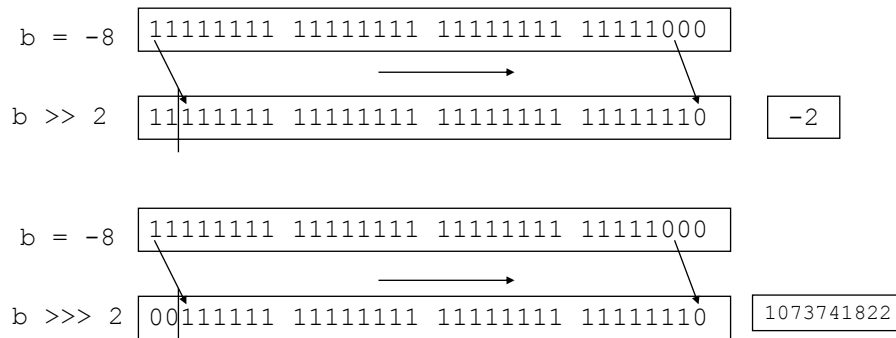
## Bit-Operatoren

Operator	Beispiel	Bedeutung
<<	i << 2	Linksschieben (Multiplikation mit Zweier-Potenzen)
>>	i >> 2	Vorzeichenbehaftetes Rechtsschieben (Division durch Zweier-Potenzen)
>>>	i >>> 2	Vorzeichenloses Rechtsschieben (Division durch Zweier-Potenzen)
&	i & 7	bitweises UND
^	i ^ 7	bitweises XOR (Exklusives ODER)
	i   7	bitweises ODER
~	~i	bitweise Negation
<<=	i <<= 3	i = i << 3
>>=	i >>= 3	i = i >> 3
>>>=	i >>>= 3	i = i >>> 3
&=	i &= 3	i = i & 3
^=	i ^= 3	i = i ^ 3
=	i  = 3	i = i   3

## Shift-Operatoren (1)



## Shift-Operatoren (2)



## Die logischen Bit-Operatoren

- Die Operatoren **&**, **|** und **^** verknüpfen jeweils zwei ganzzahlige Ausdrücke bitweise.
- Dabei werden jeweils die Bits der entsprechenden Positionen der beiden beteiligten Ausdrücke miteinander verknüpft.
- Beispiele

```
int a = 5, b = 12, c;  
c = a & b;  // c == 4 (0101 & 1100 = 0100)  
c = a | b;  // c == 13 (0101 | 1100 = 1101)  
c = a ^ b;  // c == 9 (0101 ^ 1100 = 1001)
```

## Die logischen Bit-Operatoren

- Die bitweise Negation ist ein unärer Operator, der bei einem ganzzahligen Ausdruck alle Bits umkehrt.

- Beispiel:

```
int a = 5, b;
b = ~a;
// b == -6
```

00000000	00000000	00000000	00000101
11111111	11111111	11111111	11111010

## Bestimmtes Bit setzen

```
/**
 * Setzen eines bestimmten Bits in einer int-Zahl
 *
 * @param zahl umzuwandelnde Zahl
 * @param n    zu setzendes Bit (0 .. 31)
 * @return zahl | (1 << n)
 */
public static int setzeBit(int zahl, int n) {
    int maske = 1 << n;
    return (zahl | maske);
}
```



## Bestimmtes Bit setzen

- zahl = 13, n = 1

	7	6	5	4	3	2	1	0
zahl = 13	0	0	0	0	1	1	0	1
maske = 1 << 1	0	0	0	0	0	0	1	0
zahl   maske	0	0	0	0	1	1	1	1



- zahl = 13, n = 3

	7	6	5	4	3	2	1	0
zahl = 13	0	0	0	0	1	1	0	1
maske = 1 << 3	0	0	0	0	1	0	0	0
zahl   maske	0	0	0	0	1	1	0	1



## Bestimmtes Bit prüfen

```
/**
 * Ueberpruefen eines bestimmten Bits in einem int
 *
 * @param zahl zu pruefende Zahl
 * @param n    zu pruefendes Bit (0 .. 31)
 * @return (zahl & (1 << n)) != 0
 */
public static boolean istBitGesetzt(int zahl, int n) {
    int maske = 1 << n;
    return (zahl & maske) != 0;
}
```

## Bestimmtes Bit prüfen

- zahl = 13, n = 1

	7	6	5	4	3	2	1	0
zahl = 13	0	0	0	0	1	1	0	1
maske = 1 << 1	0	0	0	0	0	0	1	0
zahl & maske	0	0	0	0	0	0	0	0



- zahl = 13, n = 3

	7	6	5	4	3	2	1	0
zahl = 13	0	0	0	0	1	1	0	1
maske = 1 << 3	0	0	0	0	1	0	0	0
zahl & maske	0	0	0	0	1	0	0	0



## Der Bedingungs-Operator

- Der Bedingungsoperator ist der einzige ternäre (dreistellige) Operator bei Java.
- Syntax:  
(logischer Ausdruck) ? Ausdruck1 : Ausdruck2
- Wert des Ausdrucks:
  - ⇒ Wert von *Ausdruck1*, falls der logische Ausdruck true ist
  - ⇒ Wert von *Ausdruck2* andernfalls
- Beispiel:

```
// Bestimme das Minimum zweier Werte
public double min (double a, double b) {
    return (a < b) ? a : b;
}
```