



PM2- Java – Basisdatentypen und Operatoren



Fahrplan

- Basisdatentypen
- Wrappertypen
- Exkurs: Benutzereingaben von der Konsole lesen
- Operatoren
 - Operatortypen
 - Operatorzuweisung
 - Implizite Typkonvertierung und Polymorphie
 - Explizite Typkonvertierung mit dem Castoperator
 - Operator Präzedenz und Assoziativität
- Operatoren für Wrappertypen



BASISDATENTYPEN

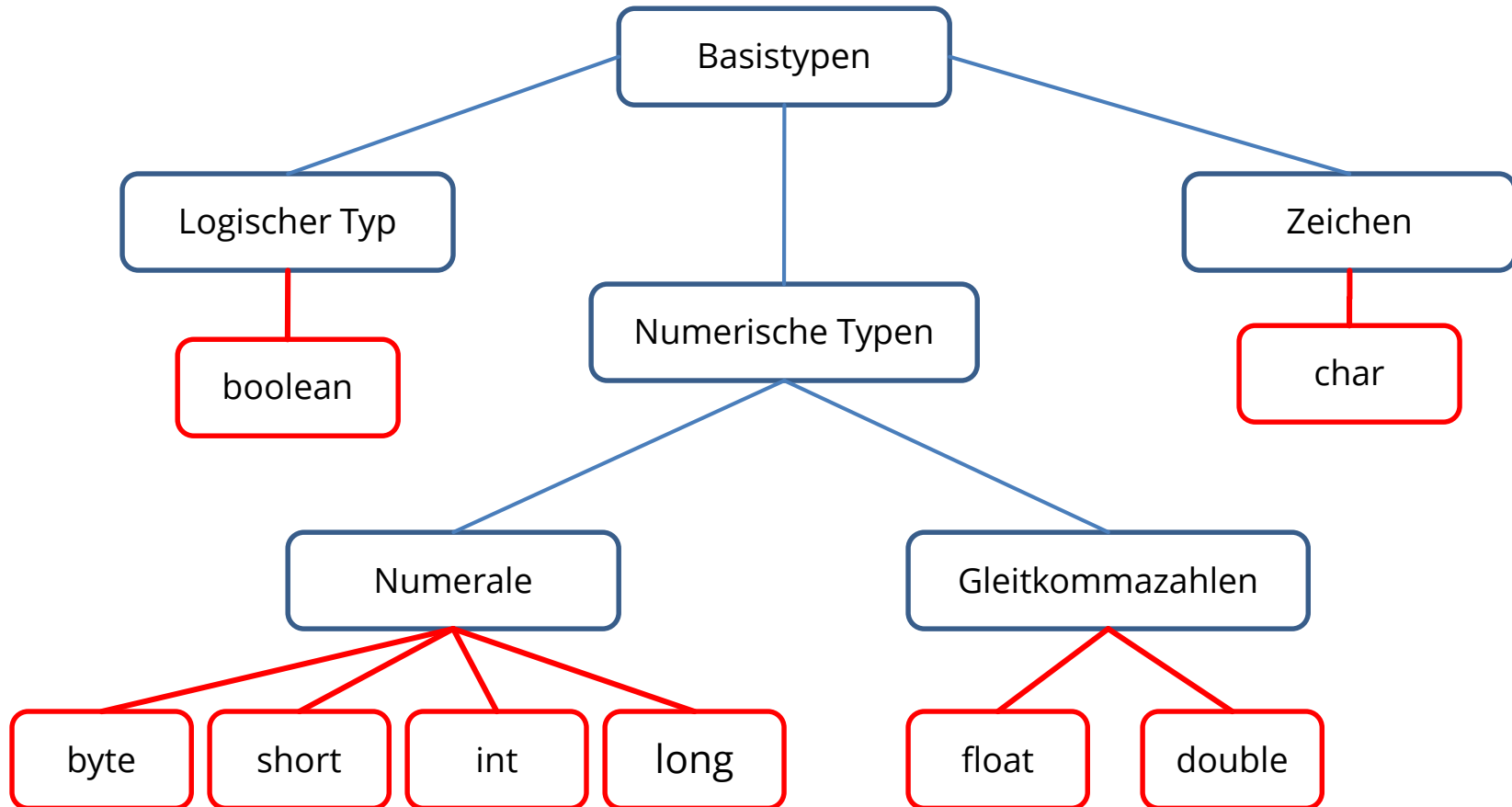


Basisdatentypen in Java

- eine Menge fest definierter Typen mit fest definierten Wertebereichen
- werden immer klein geschrieben
- haben **keine** Methoden und können nur mittels Operatoren manipuliert werden
- können nicht verändert werden, von ihnen kann nicht abgeleitet werden
- „*Instanzen*“ sind **Werte**, die typischerweise in einem Vielfachen von einem Speicherwort (**byte**) repräsentiert werden.
- Werte sind **keine** Objekte.
- Werte sind **immutable** → Operationen auf Werten erzeugen neue Werte



Basisdatentypen in Java





Wertebereich der Basisdatentypen in Java

Typ	Größe / Speicherbedarf	Wertebereich Minimum	Wertebereich Maximum
boolean	1 bit	<i>false</i>	<i>true</i>
char	16 bit	'\u0000' 0 <i>Character.MIN_VALUE</i>	'\uffff' $2^{16}-1$ <i>Character.MAX_VALUE</i>
byte	8 bit	-128: <i>Byte.MIN_VALUE</i>	+127: <i>Byte.MAX_VALUE</i>
short	16 bit	-2^{15} : -32,768 <i>Short.MIN_VALUE</i>	$+2^{15}-1$: +32,767 <i>Short.MAX_VALUE</i>
int	32 bit	-2^{31} : -2,147,483,648 <i>Integer.MIN_VALUE</i>	$+2^{31}-1$: +2,147,483,647 <i>Integer.MAX_VALUE</i>
long	64 bit	-2^{63} : -9,223,372,036,854,775,808 <i>Long.MIN_VALUE</i>	$+2^{63}-1$: +9,223,372,036,854,775,807 <i>Long.MAX_VALUE</i>
float	32 bit Exponent 8 Mantisse 23	$\pm 1.40129846432481707e-45$ <i>Float.MIN_VALUE</i> kleinste darstellbare positive Zahl	$\pm 3.40282346638528860e+38$ <i>Float.MAX_VALUE</i>
double	64 bit Exponent 11 Mantisse 52	$\pm 4.94065645841246544e-324$ <i>Double.MIN_VALUE</i> kleinste darstellbare positive Zahl	$\pm 1.79769313486231570e+308$ <i>Double.MAX_VALUE</i>



Literale in Java

- Zeichenfolgen zur Darstellung der Werte von Basistypen
- Anhand des Literals erkennt der Compiler den Typ des Wertes.
- Manchmal sind Typen nicht eindeutig: Dann müssen dem Literal Zeichen hinzugefügt werden, die den Typ des Literals eindeutig machen.
- In Java gibt es keine Literale für binäre Zahlen, aber die Möglichkeit eine Binärdarstellung in der Ausgabe zu erzeugen (`Integer.toBinaryString(789);`). Dabei werden die führenden Nullen nicht angezeigt.



Literale für *char*

- *char* hat drei mögliche Schreibweisen
 - Darstellung als *int*. Bei der Ausgabe wird der *int* Wert in ein darstellbares Zeichen anhand der Zeichentabellen umgewandelt. Soll der *int* Wert dargestellt werden, dann muss *char* explizit in ein *int* umgewandelt werden *((int)c1)*.
 - direkte Darstellung als Zeichen: Bietet sich für alle darstellbaren Zeichen an.
 - Darstellung in Unicode.



Literale für *char*

```
package basedatatypes;
public class CharacterLiterals {
    public static void main(String[] args) {
        char c1,c2, c3;
        c1 = 226;           // int Darstellung
        c2 = '\u00E2';      // Unicode Darstellung (Hexadezimal 4 Stellen)
        c3 = 'â';           // Zeichendarstellung
        print("c1 : " + c1);
        print("c2 : " + c2);
        print("c3 : " + c3);
        print("c1 int value : " + (int)c1); // Umwandeln in int
        print("c2 int value : " + (int)c2); // Umwandeln in int
        print("c3 int value : " + (int)c3); // Umwandeln in int
    }

    private static void print(String string) {
        System.out.println(string);
    }
}
```



Zeichendarstellung

- ASCII (American Standard Code for Information Interchange):
 - 7 Bit für die Darstellung von Zeichen (insgesamt 128 Zeichen)
 - Codierung durch Position in einer Tabelle (Codepoint)
 - Pos. 0-31 Kontrollzeichen, Pos. 32 Leerzeichen, Pos. 127 Kontrollzeichen
- ISO-8859-1:
 - 8 Bit für die Darstellung von Zeichen (insgesamt 256 Zeichen)
 - Pos. 0-127 ASCII Zeichen
 - Pos. 128-159 Kontrollzeichen
 - Pos. 160-255 diakritische Zeichen (z.B. Umlaute)
- Windows 1252:
 - nutzt Pos. 128-159 für z.B. Interpunktionszeichen
- Unicode
 - mehrere Bytes (1,2,3,4) zur Darstellung der Zeichen aller Sprachen
 - enthält alle Zeichen von ISO-8859-1 mit den gleichen Positionen
 - Darstellung der Bytes als Hexadezimalzahlen mit dem Präfix U+ (Unicode für „A“: U+0041, für „Ä“ U+00C4)



Unicode Standards und kompakte Kodierung

- Unicode-Standard 3.0 (Java 1.1 bis 1.4)
 - 16 Bit / 2 Byte pro Zeichen
- Unicode 4.0-Standard (ab Java 5)
 - 32 Bit / 4 Byte pro Zeichen
- Unicode- Standard 6.0 (ab Java 7)
 - Erweiterung auf Emojii's und Symbole weiterer Sprachen
- 4 Byte pro Zeichen = Platzverschwendung / daher kompakte Kodierungen
 - UTF-8: 1,2,3 oder 4 Byte pro Zeichen
 - UTF-16: 2 oder 4 Byte pro Zeichen
 - UTF-32: 4 Byte pro Zeichen



Unicode und UTF Kodierung

			chinesisch Osten	Zeichen aus dem Deseret (phonetisches Alphabet)
Zeichen	A	ß	睽	ð
Unicode- Codepoint	U+0041	U+00DF	U+6771	U+10400
UTF-32	00 00 00 41	00 00 00 DF	00 00 67 61	00 01 04 00
UTF-16	00 41	00 DF	66 71	D801 DC00
UTF-8	41	C3 9F	E6 9D B1	F0 90 90 80
Zeichen-Literal in Java	'\u0041'	'\u00DF'	'\u6771'	'\uD801\uDC00'



Darstellung von Zahlen als Binärzahl

```
package basedatatypes;

public class NumeralsBinaryRepresentation {

    public static void main(String[] args) {
        int i1 = 0x2f; // hexadecimal
        print("i1: " + Integer.toBinaryString(i1));
        // analog für i2 und i3
        char c = 0xffff; // max char in hex
        print("c: " + Integer.toBinaryString(c));
        byte b = 0x7f; // max byte in hex
        print("b: " + Long.toBinaryString(b));
        short s = 0x7fff; // max short in hex
        print("s: " + Integer.toBinaryString(s));
        long n1 = 200L; // long suffix
        print("n1: " + Long.toBinaryString(s));
    }

    private static void print(String string) {
        System.out.println(string);
    }
}
```



```
i1: 101111
c: 111111111111111111
b: 1111111
s: 1111111111111111
n1: 11001000
```



Zahlenliterale

- Numerale

14 (**byte ... long**)

130 (kein **byte** aber **short ... long**)

130l, 130L (nur **long**)

- Gleitkommazahlen

Dezimalschreibweise

3.14

-123.04

Exponentialschreibweise

1E23 **10²³**

1e-34 **10⁻³⁴**

6.670E-11 **6.670*10⁻¹¹**

-4.17e-4

double-Suffix float-Suffix (*kein suffix = double*)

1D **1F**

1e-34d **1e-10f**

1.4 (**double**)

- ein **L / l** am Ende einer ganzen Zahl steht für den Typ **long**.
- Ein **F / f** am Ende einer Zahl steht für den Typ **float**.
- Ein **D/d** am Ende einer Zahl steht für den Typ **double**.
- Ein **0x, 0X**, gefolgt von **(0-9)** oder **(a-f)** steht für eine *hexadezimale* Zahl (Basis 16)
- Eine **0** gefolgt von **(0-7)** steht für eine *oktale* Zahl (Basis 8).



Zahlenliterale (Beispiele)

Compilerfehler
1.4 double nicht
kompatibel zu
float.

```
package basedatatypes;

public class NumericLiterals {
    public static void main(String[] args) {
        int i1 = 0x2f; // hex (47)
        i1 = 47;
        int i2 = 0X2F; // hex (47)
        int i3 = 0177; // octal (leading 0) (127)
        char c = 0xffff; // max char in hex (2**16-1)
        byte b = 0x7f; // max byte in hex (2**7-1)
        short s = 0x7fff; // max short in hex (2**15 -1)
        long n1 = 200L; // long suffix
        long n2 = 200l; // long suffix
        long n3 = 200; //
        float f1 = 1;
        f1 = 1.4; // 1.4 wird als double interpretiert Fehler
        float f2 = 1F; // float suffix
        float f3 = 1f; // float suffix
        double d1 = 1;
        d1 = 1.4; //ok
        double d2 = 1d; // double suffix
        double d3 = 1D; // double suffix
    }
}
```



Typkompatibilität

- Basisdatentyp **A** ist *typkompatibel* zu Basisdatentyp **B**, wenn $A \leq B$.
- $A \leq B$, wenn **A** weniger oder gleich viel Speicher als **B** benötigt und der *Wertebereich(A) \subseteq Wertebereich(B)*
- **char** ist **nicht** typkompatibel zu **short** und **byte**, da die Wertebereiche eine echte Schnittmenge haben.
- **Ausnahme:** **boolean** ist weder typkompatibel zu **char** noch zu den *numerischen Typen*.
- Werte passen immer in kompatible Typen. Bei einer Zuweisung geht keine Information verloren.
- Wir dürfen z.B. einer Variable vom Typ **double** einen Wert vom Typ **byte** zuweisen.
- Die Umkehrung gilt nicht, da Information verloren geht, wenn der Wert nicht im Wertebereich des Zieltypen liegt.



Beispiele zur Typkompatibilität

```
package basedatatypes;
public class TypeCompDemo {

    public static void main(String[] args) {

        boolean bool = true;
        char ch = 'a';
        byte b = 127;
        short sh = 354;
        int i = 1089;
        long l = 566666666;
        float f = 1.4f;
        double d = 454999995.7878;

        ch=bool; b = bool; sh= bool; i = bool; l = bool; f=bool; d = bool;
        ch = b; ch = sh;
        sh = ch; sh = b;
        i = ch; i = b; i = sh;
        l = ch; l = b; l = sh; l = i;
        f = ch; f = b; f = sh; f = i; f = l;
        d =ch; d = b; d = sh; d = i; d = l; d = f;

    }
}
```

Compilerfehler
Typen nicht
kompatibel



Typkompatibilitätstabelle

passt in	boolean	byte	char	short	int	long	float	double
boolean	x							
byte		x						
char			x					
short		x		x				
int		x	x	x	x			
long		x	x	x	x	x		
float		x	x	x	x	x	x	
double		x	x	x	x	x	x	x



WRAPPERTYPEN



Wrappertypen für Basisdatentypen

Erzeugen eines Wrappertypen

`Integer anInteger = new Integer(15);` oder
`Integer anInteger = 15;` (Autoboxing)

Typ	Wrapper Typ	Zugriff Basistyp
boolean	Boolean	aBoolean.booleanValue
char	Character	aCharacter.charValue
byte	Byte	aByte.byteValue
short	Short	aShort.shortValue
int	Integer	anInteger.intValue
long	Long	aLong.longValue
float	Float	aFloat.floatValue
double	Double	aDouble.doubleValue

- Jeder Basisdatentyp in Java hat einen korrespondierenden Wrappertyp.(→ Tabelle).
- Werte werden Wrapperobjekten bei der Erzeugung im Konstruktor übergeben. (`new Integer(15)`)
- Wrapperobjekte verwalten den Wert in einer Instanzvariable, der sich über Name des Basistypen plus **Value** abfragen lässt. (z.B. `aCharacter.charValue`)
- Wrappertypen sind notwendig,
 - da Java Generics keine Basisdatentypen zulassen.
 - Typinformation, wie z.B. die Größe Typs oder der Wertebereich eines Typs sonst nicht verfügbar sind.
 - Spezielle Zahlen, z.B. NaN als Ergebnis einer Operation nicht darstellbar sind.



gemeinsame Konstanten / Klassen-Methoden der Wrappertypen

Konstante/Methode	Beschreibung
<i>static</i> <zugeordneter primitiver Typ> <i>MAX_VALUE</i>	Maximaler Wert
<i>static</i> <zugeordneter primitiver Typ> <i>MIN_VALUE</i>	Minimaler Wert
<i>static int SIZE</i>	Speicherbedarf für einen Wert des Basisdatentyps
<i>static</i> <zugeordneter primitiver Typ> <i>valueOf(String s)</i>	Liest einen <i>String</i> und wandelt den Inhalt in einen Wert des primitiven Typs um. Die Zahlenbasis für Numerae ist 10. Erzeugt eine <i>NumberFormatException</i> , wenn der String nicht den zugeordneten primitiven Typ darstellt. Nicht für Character.
<i>static</i> <zugeordneter primitiver Typ> <i>valueOf(String s, int radix)</i>	Liest einen <i>String</i> und wandelt den Inhalt in einen Wert des primitiven Typs um. Die Zahlenbasis wird als <i>int</i> übergeben. z.B. 8 für Oktal.
<i>static</i> <zugeordneter primitiver Typ> <i>parse</i> <Zugeordneter primitiver Typ>(<i>String s</i>)	Bsp.: <i>Integer.parseInt</i> , <i>Short.parseShort</i> , etc... Wie <i>valueOf</i>
<i>static</i> <zugeordneter primitiver Typ> <i>parse</i> <Zugeordneter primitiver Typ>(<i>String s, int radix</i>)	nur für Numerae, wie <i>valueOf</i> . Nicht für Character



Objekt-Methoden von Wrappertypen

Methode	Beschreibung
<i>byteValue, shortValue, intValue, longValue, floatValue, doubleValue</i>	Gibt den Inhalt des Wrappertypen als <i>byte</i> ... <i>double</i> zurück. Dabei wird der tatsächliche Wert auf den angefragten Typ gecastet. (<i>Zu casten später mehr</i>)
<i>equals, hash</i>	Gleichheit und hash Methode.
<i>public static String toString()</i>	Umwandlung in einen String
<i>public Byte(byte b), public Short(short s), public Char(char c), public Integer(int i), public Long(long l), public Float(float f), public Double(double d)</i>	Konstruktoren erzeugen eines Wrappertyps
<i>public int compareTo(<gleicherWrapperTyp> obf)</i>	Vergleicht die Inhalte zweier Wrappertypen. Immer zu verwenden beim Vergleich von Gleitkommazahlen!!!!



Methoden von Wrappertypen

```
Integer in = new Integer(0x7fffffff);  
// Groesse des Typs  
System.out.println("size: " + Integer.SIZE);  
// Lesen des Basisdatentypen  
System.out.println("int:" + in.intValue());
```



```
size: 32  
int:2147483647
```



Typumwandlungen mit Wrappertypen

```
Integer in = new Integer(127);
System.out.println("as binary integer: "
    + Integer.toBinaryString(in.intValue()));
// hier wird der Typ verkleinert Information geht verloren
System.out.println("byte: " + in.byteValue());
System.out.println("as binary integer: "
    + Integer.toBinaryString(in.byteValue()));
// hier wird der Typ vergrößert. Information bleibt erhalten
System.out.println("long: " + in.longValue());
System.out.println("float: " + in.floatValue());
System.out.println("double: " + in.doubleValue());
```

Hier konnte ein *int*
ohne Verluste in ein
byte umgewandelt
werden
long...double sind
unkritisch, da *int*
immer passt.



```
size: 32
int:127
as binary integer: 1111111
byte: 127
as binary integer: 1111111
long: 127
float: 127.0
double: 127.0
```




Typumwandlungen mit Wrappertypen

```
Integer in = new Integer(0x7fffffff); // Wert 2147483647
System.out.println("as binary integer: "
    + Integer.toBinaryString(in.intValue()));
// hier wird der Typ verkleinert Information geht verloren
System.out.println("byte: " + in.byteValue());
System.out.println("as binary integer: "
    + Integer.toBinaryString(in.byteValue()));
// hier wird der Typ vergrößert. Information bleibt erhalten
System.out.println("long: " + in.longValue());
System.out.println("float: " + in.floatValue());
System.out.println("double: " + in.doubleValue());
```

Hier kommt es bei der
Umwandlung von *int*
in *byte* zu einem
Informationsverlust.
Bei *float* tritt ein
Rundungsfehler auf.



```
as binary integer: 11111111111111111111111111111111
byte: -1
as binary integer: 11111111111111111111111111111111
long: 2147483647
float: 2.14748365E9
double: 2.147483647E9
```



Typumwandlungen mit Wrappertypen

```
Integer in = new Integer(0x7fffffff);  
System.out.println("as binary integer: "  
    + Integer.toBinaryString(in.intValue()));  
// hier wird der Typ verkleinert es geht Information  
// verloren. Gleiches Phänomen bei in.shortValue();  
System.out.println("byte: " + in.byteValue());  
System.out.println("as binary integer: "  
    + Integer.toBinaryString(in.byteValue()));  
// hier wird der Typ vergrößert. Information bleibt erhalten  
System.out.println("long: " + in.longValue());  
System.out.println("float: " + in.floatValue());  
System.out.println("double: " + in.doubleValue());
```

Durch Umwandlung in *byte* werden die ersten 24 *bit* des *int* abgeschnitten.
Das führende *bit* ist 1 → Zahl ist negativ. Alle weiteren *bits* sind 1 → Es
handelt sich um -1.

Anschließendes Umwandeln von -1 als *byte* in einen Integer erzeugt
wiederum eine -1 als *int*. Binär stellt sich das als Sequenz von 32 1'ern dar.



- | | Literal | Binärdarstellung |
|---------------|-------------|--|
| int (32 bit) | 126 | 1111110 |
| | -126 | 1111111111111111111111110000010 |
| long (64 bit) | 2147483647 | 11111111111111111111111111111111 |
| | -2147483647 | 1111111111111111111111111111111110000000000000000000000000000001 |

Wrappertypen sind untereinander nicht typkompatibel



```
// Wrappertypen sind untereinander nicht typkompatibel
```

```
Double dD = 4.5;
```

```
Float fF = 4.5f;
```

```
Integer iI = 999;
```

```
dD = iI;
```

```
dD = fF;
```

```
fF = iI;
```

Compilerfehler: Can't convert
from Integer to Double, ...

```
// Basisdatentypen hingegen schon
```

```
double d = 4.5;
```

```
float f = 4.5f;
```

```
int i = 900;
```

```
d = f; d = i;
```

```
f = i;
```

Besonderheiten der Gleitkommazahlen Wrapper



- *Float* und *Double* kennen die 3 Konstanten *Infinity* und *-Infinity* und *NaN*.
- *Infinity* steht für die positiv unendliche Zahl, die sich z.B. durch Division einer positiven Zahl durch 0.0 ergibt.
- *-Infinity* steht für die negativ unendliche Zahl, die sich z.B. durch Division einer negativen Zahl durch 0.0 ergibt.
- *NaN* steht für eine undefinierte Rechenoperation für reelle Zahlen. Z.b. *Math.sqrt(-4)*

Besonderheiten der Gleitkommazahlen Wrapper



```
package basedatatypes;
public class NaNDemo {
    public static void main(String[] args) {
        Double d1 = 167.0;
        Double d2 = 0.0;
        Double d3 = d1 / d2;
        Double d4 = -d1 / d2;
        double d5 = d1 / 0.0;
        System.out.println(d3);
        System.out.println(d4);
        System.out.println(d5);
        System.out.println(0.0/0.0);
        System.out.println(Math.sqrt(-12));
    }
}
```



Infinity
Infinity
-Infinity
Infinity
NaN



Spezielle Methoden von *Character*

Methode	Beschreibung
<i>static boolean isDigit(char ch)</i>	Ist das Zeichen eine Ziffer?
<i>public static boolean isISOControl(char ch)</i>	Ist das Zeichen ein Kontrollzeichen?
<i>public static boolean isLetter(char ch),</i>	Ist das Zeichen ein Buchstabe?
<i>public static boolean isWhitespace(char ch)</i>	Ist das Zeichen ein Leerzeichen?
<i>public static boolean isLowerCase(char ch)</i>	Ist das Zeichen ein Kleinbuchstabe?
<i>public static boolean isUpperCase(char ch)</i>	Ist das Zeichen ein Grossbuchstabe?
<i>public static char toLowerCase(char ch)</i>	wandelt ein Zeichen in Grossbuchstaben, falls es eine Entsprechung gibt, gibt sonst das Zeichen selbst zurück.
<i>public static char toUpperCase(char ch)</i>	wandelt ein Zeichen in Grossbuchstaben, falls es eine Entsprechung gibt, gibt sonst das Zeichen selbst zurück.



Exkurs

BENUTZEREINGABEN VON CONSOLE LESEN



Lesen von Konsole (old school)

Zeilen lesen

Öffnen eines gepufferten
Lesestroms für die
Eingabestrom der Konsole
(System.in).

```
private static void readFromConsoleTraditional() {  
    try (BufferedReader br = new BufferedReader(new InputStreamReader(System.in)))  
    {  
  
        String userInput = "";  
        while ((userInput = br.readLine()) != null) {  
            System.out.println(userInput);  
        }  
  
    } catch (IOException ioe) {  
        System.out.println("Something went wrong while reading from console. Please  
            restart programm");  
    }  
    System.out.println("bye");  
}
```

typische Leseschleife hier
ohne
Abbruchmöglichkeiten
durch den Nutzer.

Beim Öffnen eines
Eingabe-/Ausgabekanals
kann ein Fehler auftreten,
der hier behandelt wird.



Lesen von Konsole mit Scanner

Zeilen lesen

```
private static void readFromConsoleWithScanner() {  
    Scanner sc = new Scanner(System.in);  
  
    while (sc.hasNextLine()) {  
        System.out.println(sc.nextLine());  
    }  
  
    sc.close();  
  
    System.out.println("bye");  
}
```

Quelle für den Scanner ist der Eingabestrom der Konsole `System.in`

typische Leseschleife hier ohne Abbruchmöglichkeiten durch den Nutzer.

Nach der Verarbeitung muss der Scanner geschlossen werden, um Ressourcen (hier `System.in`) wieder frei zu geben.



Lesen und Prüfen von Eingaben

```
private static void readAndTestTypeWithScanner() {
    Scanner sc = new Scanner(System.in);
    while (true) {
        if (sc.hasNextLong()) {
            long l = sc.nextLong();
            System.out.println("Long " + l);
        }
        if (sc.hasNext()) {
            String s = sc.next();
            if ("exit".equals(s))
                break;
        }
    }
    sc.close();
    System.out.println("bis bald");
}
```

Prüft, ob die Eingabe in einen long Wert umgewandelt werden kann.

Liest und wandelt die Eingabe in einen long Wert um.

Prüft, ob eine Eingabe vorliegt.

Liest die Eingabe als Zeichenkette



Übungen

In der Klasse *BaseTypeExercise* und deren statischen Methoden soll das Gelernte zu Basis- und Wrapper-Typen vertieft werden. Ebenso sollen das Einlesen von Benutzereingaben geübt werden.

- **ue-2-1:** Schreiben Sie statische Methoden (*public static*), die prüfen ob ein übergebener Wert im Wertebereich eines Basisdatentyps liegt. Die Methoden heißen *isByte*, *isShort*, *isInt*, *isLong*. Der Parameter der Methoden ist immer vom Typ *long*. Der Ergebnistyp aller Methoden ist *boolean*.
- **ue-2-2:** Schreiben Sie statische Methoden, die prüfen, ob bei einer Addition der Ergebniswert im Wertebereich der Argumenttypen bleibt. Die Methoden für die Überprüfung mit *byte*-Argumenten heißt z.B. *numberOverflowPlus(byte b1, byte b2)*. Der Ergebnistyp aller Methoden ist *boolean*.
- **ue-2-3:** Ergänzen Sie die main-Methode der Klasse *BaseTypeExercise*. In jedem Schleifendurchlauf sollen 2 Eingaben gelesen und in *long*-Werte umgewandelt werden, es sei denn der erste Wert ist das Schlüsselwort „*exit*“ für den Abbruch des Programms. Dann soll der minimale Typ der beiden Werte bestimmt werden (→ **ue-2-1**). Wenn die Addition der beiden Werte korrekt ohne Überlauf erfolgt (→ **ue-2-2**), soll das Ergebnis ausgegeben werden, sonst soll für die Addition zweier Bytes die Fehlermeldung „*Addition von zwei Byte Werten erzeugt einen Überlauf im Wertebereich*“ gedruckt werden (analog für die anderen Typen).
- **ue-2-4:** Testen Sie die Lösungen **ue-2-1** bis **ue-2-3** durch Ausführen des Programms.



Übungen

- **ue-2-5:** Schreiben Sie ein Programm, dem beim Start eine Zeichenkette übergeben wird und das die Anzahl der Zeichen für die Zeichenklassen: *Ziffer*, *Buchstabe*, *Kleinbuchstabe*, *Großbuchstabe* und *Leerzeichen* zählt und die Anzahl auf der Konsole ausgibt.
- **Hilfestellung zu ue-2-5:**
 - Einlesen von Parametern beim Programmaufruf und Starten von Programmen von der Konsole war u.a. Inhalt der ersten Vorlesung.
 - Das folgende Codeschnipsel liest nacheinander die Zeichen eines Strings in die Variable `c`:

```
for (char c: string.toCharArray()) { ... }
```
- **ue-2-6:** Schreiben Sie die Methoden aus **ue-2-1** mit Hilfe von Methoden der Klasse *Scanner*. Den Methoden wird dann ein zweiter Parameter für das *Scanner*-Objekt übergeben.
- **ue-2-7:** Schreiben Sie das Programm **ue-2-5** mit einem Scanner. Mit der Methode *setDelimiter("")* weisen Sie den Scanner an zeichenweise zu lesen. Das Ergebnis ist ein String der Länge 1. Zeichen aus einem String erhalten Sie mit *charAt(int index)*.



Operatortypen

OPERATOREN



Operatoren

- Funktionen auf Basis-Datentypen.
- Abbildungen von Werten der Operanden in einen Ergebniswert. Der Ergebnistyp bestimmt sich nach dem größten Typ der beteiligten Operanden. (**Es gibt eine Ausnahme!**)
- Man unterscheidet **unäre, binäre und ternäre** Operatoren. Unäre Operatoren haben einen, binäre zwei und ternäre drei Operanden.
- Es können in Java keine eigenen Operatoren definiert werden.
- Kategorien:
 1. arithmetische Operatoren
 2. Vergleichsoperatoren
 3. logische Operatoren
 4. Bitoperatoren
 5. Typvergleichsoperator
 6. Zuweisungsoperator
 7. Castoperator
- 1.) 2.) 3.) 4.) sind nur auf Basisdatentypen und Wrappertypen definiert.
- Unter 1.) stellt + eine Besonderheit dar, das dieser Operator auch für String definiert ist.
- 5.) ist nur für Objekttypen definiert
- 6.) und 7.) sind für alle Datentypen definiert.



Operatoren

- arithmetische Operatoren: (Zahlen und **char**)
 - binäre: **+**, **-**, *****, **/**, **%**
 - unäre: **+**, **-**, **++** (Inkrement), **--** (Dekrement)
- Vergleichsoperatoren: (Zahlen und **boolean**) **<**, **>**, **<=**, **>=**, **==**
- logische Operatoren (**boolean**)
 - binär: **&&**, **||**,
 - ternärer Operator: **?:** (siehe 1'tes Semester)
 - unär: **!**
- Bitoperatoren: (Zahlen, **char**, **boolean**) **&**, **|**, **^**
- Typvergleichsoperator (nur für Referenztypen) **instanceof**
- Zuweisungen (alle) **:** **=**
- Castoperator (alle): (**<Typ>**)
 - **(short)15261526**
 - **(Student) einPerson**

Arithmetische Operatoren mit integralen Werten (ganze Zahlen und char)



```
public class DemoInt {
    public static void main(String[] args) {
        print("Int Demo");
        // funktioniert so nicht fuer char, byte und short,
        // funktioniert analog fuer long
        int j, k, i;
        j = 135;
        print("j : " + j);
        k = 300;
        print("k : " + k);
        print("j + k : " + (i = j + k));
        print("j - k : " + (i = j - k));
        print("k / j : " + (i = k / j));
        print("k * j : " + (i = k * j));
        print("k % j : " + (i = k % j));
    }
    ...
}
```



```
Int Demo
j : 135
k : 300
j + k : 435
j - k : -165
k / j : 2
k * j : 40500
k % j : 30
```



Arithmetische Operatoren auf integralen Werten (ganze Zahlen und char)

```
private static void demoShort() {  
    print("Short Demo");  
    short j,k,i;  
    // Werte zwischen 1 und 100  
    j = 135;  
    print("j : " + j);  
    k = 70;  
    print("k : " + k);  
    print("j + k : " + (i=j+k));  
    print("j - k : " + (i=j-k));  
    print("k / j : " + (i=k/j));  
    print("k * j : " + (i=k*j));  
    print("k % j : " + (i=k%j));  
}
```

FEHLER!

warum?

Arithmetische Operatoren auf ganzen Zahlen und char
konvertieren die Argumenten vor der Anwendung
des Operators in `int`.



Arithmetische Operatoren auf integralen Werten

```
package basedatatypes;
public class ShortDemo {
    public static void main(String[] args) {
        print("Short Demo");
        short j,k,i;
        // Werte zwischen 1 und 100
        j = 135;
        print("j : " + j);
        k = 70;
        print("k : " + k);
        print("j + k : " + (i=j+k));
        print("j - k : " + (i=j-k));
        print("k / j : " + (i=k/j));
        print("k * j : " + (i=k*j));
        print("k % j : " + (i=k%j));
    }
}
```

Compilerfehler

Arithmetische Operatoren
auf ganzen Zahlen konvertie-
ren die Argumente
vor der Anwendung
des Operators in *int*.

Hier wird versucht einem
short einen *int* Wert
zuzuweisen.



Arithmetische Operatoren mit *Character*

```
public class CharacterLiterals {  
    public static void main(String[] args) {  
        char c1, c2, c3;  
        c1 = 226;  
        c2 = '\u00E2';  
        c3 = 'â';  
        print("c1 : " + c1);  
        print("c2 : " + c2);  
        print("c3 : " + c3);  
        // arithmetische Ops  
        print("c1 + 1: " + (c1 + 1));  
        print("c1 / 3: " + (c1 / 3));  
        print("-c1: " + (-c1));  
    }  
    ...  
}
```

Arithmetische Operatoren
auf Zeichen konvertie-
ren die Argumente
vor der Anwendung
des Operators in *int*.



```
c1 : â  
c2 : â  
c3 : â  
c1 + 1: 227  
c1 / 3: 75  
-c1: -226
```



Bibliotheksmethoden für Zahlen: Klasse *Math*

- In Java gibt es nur eine kleine Menge von Operatoren für arithmetischen Grundoperationen. (+, -, *, /, %)
- Die Potenzfunktion fehlt als Operator.
- Arithmetische Funktionen, die die Grundoperationen ergänzen, finden sich als statische Klassenmethoden der Klasse *Math*.

Quadratwurzel	<i>Math.sqrt</i>
Natürlicher Logarithmus	<i>Math.log</i>
Logarithmus zur Basis 10	<i>Math.log10</i>
Potenzfunktion	<i>Math.pow</i>
e-Funktion	<i>Math.exp</i>
Sinus und andere trigonometrische Funktionen	<i>Math.sin</i>
Konstanten	<i>Math.PI</i>



Bibliotheksmethoden für Zahlen: Klasse *Math*

```
package operators;
import static util.Printer.print;
import static java.lang.Math.*;

public class MathLibDemo {
    public static void main(String[] args)
    {
        print(abs(-45));
        print(abs(-67.98));
        print(sin(PI));
        print(log10(100));
        print(exp(2));
        print(cbrt(27));
        print(ceil(7878.99));
        float f = 56.8f;
        print(pow(f, f));
    }
}
```

Hier wird die statische Methode `print` aus der Klasse `util.Printer` statisch importiert. Kann dann in der Klasse `MathLibDemo` direkt verwendet werden.



45
67.98
1.2246467991473532E-16
2.0
7.38905609893065
3.0
7879.0
4.435920953579171E99



Autoinkrement und -dekrement

- **Inkrement -Dekrement als Anweisung**
 - folgende drei Anweisungen erzeugen die gleichen Werte
 - Inkrementieren
`variable = variable + 1;`
`variable += 1;`
`variable++;` (Autoinkrement)
 - Dekrementieren
`variable = variable - 1;`
`variable -= 1;`
`variable--;` (Autodekrement)
 - **Autoinkrement und -dekrement**
 - **Präfixoperatoren:** `++variable`, `--variable`
 - Es wird zuerst inkrementiert (dekrementiert) und dann der Wert zugewiesen
 - **Postfixoperatoren:** `variable++`, `variable--`
 - Es wird zuerst der Wert zugewiesen und dann inkrementiert (dekrementiert)
 - wird `variable++` in Zuweisungen verwendet, liefert `variable++` den Wert der `variable` **vor** dem Inkrementieren (`b` hat den Wert 1, `a` den Wert 2)
- ```
int a = 1;
int b = a++;
```



# Inkrement und Dekrement

```
package operators;
import static util.Printer.print;
public class DemoAutoInkrement {
 public static void main(String[] args) {
 print("Autoinkrement");
 int i = 1, j;
 j = ++i; // Prä-Inkrement
 print("j = ++i : j ist " + j + " i ist " + i);
 j = i++; // Post-Inkrement
 print("j = i++ : j ist " + j + " i ist " + i);
 j = --i; // Prä-Dekrement
 print("j = --i : j ist " + j + " i ist " + i);
 j = i--; // Post-Dekrement
 print("j = i-- : j ist " + j + " i ist " + i);
 // siehe auch
 print("++i : " + ++i);
 print("i++ : " + i++);
 print("i : " + i);
 print("--i : " + --i);
 print("i-- : " + i--);
 print("i : " + i);
 }
}
```



## Autoin(de)krement

```
j = ++i : j ist 2 i ist 2
j = i++ : j ist 2 i ist 3
j = --i : j ist 2 i ist 2
j = i-- : j ist 2 i ist 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```





# Logische &&, || Operatoren – short-circuit Auswertung

- teilweise und vollständige Auswertung bei **&&** (immer vollständige Auswertung bei **&**)

## Vollständige Auswertung

```
true && true; -> true
true && false; -> false
```

## Teilweise Auswertung

```
false && true; -> false
false && false; -> false
```

- teilweise und vollständige Auswertung bei **||** (immer vollständige Auswertung bei **|**)

## Teilweise Auswertung

```
true || true; -> true
true || false; -> true
```

## Vollständige Auswertung

```
false || true; -> true
false || false; -> false
```

- Beispiel für short-circuit:** linke Anweisung kontrolliert, ob Division zulässig ist

```
(d != 0) && (x/d) > 0
```



# Short-Circuit vermeidet Fehler und überflüssige Berechnungen

```
private static void demoShortCircuit() {
 // short-circuit Operatoren vermeiden Fehler
 // der zweite logische Ausdruck wird nur ausgewertet,
 // wenn iObj != null ist. Da null kein Typ ist, liefert der Versuch
 // eine Methode auf null anzuwenden immer einen Laufzeitfehler
 Integer iObj = null;
 if ((iObj != null) && (iObj.intValue() > 0)) {
 // do something
 }
 // der zweite Ausdruck wird immer ausgewertet
 if ((iObj != null) & (iObj.intValue() > 0)) {
 do something
 }
 // short-circuit Operatoren vermeiden ueberfluessige Auswertungen
 // Ist b <= 1, dann macht es keinen Sinn die zweite Bdg.
 // auszuwerten.
 int b = 0;
 if ((1 < b) && (b < 10)) print(b);
}
```



# Logische Operatoren sind nur für boolesche Werte definiert

```
private static void demoLogicalOps() {
 int i = 50;
 int j = 230;
 print("i = " + i);
 print("j = " + j);
 print("i > j " + (i > j));
 print("i < j " + (i < j));
 print("i >= j " + (i >= j));
 print("i <= j " + (i <= j));
 print("i == j " + (i == j));
 print("i != j " + (i != j));
 // int darf nicht als boolean interpretiert werden
 // ! print("i && j " + (i && j));
 // ! print("i || j " + (i || j));
 // ! print("!i" + (!i));
 // nur boolesche Werte dürfen mit booleschen Ops verknüpft werden
 print("(i < 10) && (j < 10) " + ((i < 10) && (j < 10)));
 print("(i < 10) & (j < 10) " + ((i < 10) & (j < 10)));
 print("(i < 10) || (j < 10) " + ((i < 10) || (j < 10)));
 print("(i < 10) | (j < 10) " + ((i < 10) | (j < 10)));
}
```

# Bitoperatoren



## Bit Und

|              |   |   |
|--------------|---|---|
| <b>&amp;</b> | 0 | 1 |
| 0            | 0 | 0 |
| 1            | 0 | 1 |

## Bit Oder

|          |   |   |
|----------|---|---|
| <b> </b> | 0 | 1 |
| 0        | 0 | 1 |
| 1        | 1 | 1 |

## Bit XOR

|          |   |   |
|----------|---|---|
| <b>^</b> | 0 | 1 |
| 0        | 0 | 1 |
| 1        | 1 | 0 |

## Komplement

|          |   |   |
|----------|---|---|
| <b>~</b> | 0 | 1 |
|          | 1 | 0 |

- Bitoperatoren manipulieren einzelne Bits eines integralen Werts.
- Es wird die boolesche Algebra auf Bits angewendet.
- Definiert sind
  - **&** : Bit Und
  - **|** : Bit Oder
  - **^** : XOR, Bit Entweder ... Oder
  - **~** : Bit Negation, Komplement
- Bitoperatoren gibt es auch als Operatorzuweisung für (**&=**, **|=**, **^=**)
- Bitoperatoren (**&**, **|**, **^**) sind auch für boolesche Werte definiert. Außer (**~**)

# Bitshift-Operationen



- Bitshift-Operationen schieben die Bits einer Zahl um je 1 Bit nach rechts (`>>`, `>>>`) oder links (`<<`).
- Der Unterschied zwischen `>>` und `>>>` ist, dass bei `>>` das Vorzeichen erhalten bleibt, bei `>>>` nicht.

```
int pos = 255001;
int neg = -255001;
print(pos);
print(Integer.toBinaryString(pos));
print(neg);
print(Integer.toBinaryString(neg));
int pos1 = pos>>3;
int neg1 = neg>>3;
int noLongerNeg = neg >>>3;
print(pos1);
print(Integer.toBinaryString(pos1));
print(neg1);
print(Integer.toBinaryString(neg1));
print(noLongerNeg);
print(Integer.toBinaryString(noLongerNeg));
```

```
255001
111110010000011001
-255001
111111111111111000001101111100111

31875
111110010000011
-31876
11111111111111111000001101111100
536839036
111111111111111000001101111100
```



Operatorzuweisung

# OPERATOREN



# Operatorzuweisungen

- Der Ausdruck: *variable = variable operator expression*
- lässt sich verkürzen zu: *variable operator= expression*
- Beispiel: *n = n / 2;* wird zu *n /= 2;*
- Was ist mit *n = 3\*n+1*?
- Operatorzuweisungen konvertieren ganzzahlige Argumente nicht in *int*.



# Operatorzuweisungen

```
private static void demoOperatorZuweisung() {
 // analog fuer char, byte, short, int, long, double
 print("Operator Zuweisung Demo");
 float u,v;
 u = 16.890f;
 v = 156.e14f;
 print("u : " + u);
 print("v : " + v);
 print("u : " + u);
 print("u += v : " + (u += v));
 print("u -= v : " + (u -= v));
 print("u *= v : " + (u *= v));
 print("u /= v : " + (u /= v));
}
```





# Operatorzuweisungen konvertieren ganzzahlige Argumente nicht in **int**.

```
private static void demoOperatorZuweisungShort() {
 // analog fuer char, byte, int, long
 print("Operator Zuweisung Demo Short");
 short u,v;
 u = 135;
 v = 70;
 print("u : " + u);
 print("v : " + v);
 print("u : " + u);
 print("u += v : " + (u += v));
 short w = u;
 print(w);
 print("u -= v : " + (u -= v));
 print("u *= v : " + (u *= v));
 print("u /= v : " + (u /= v));
}
```

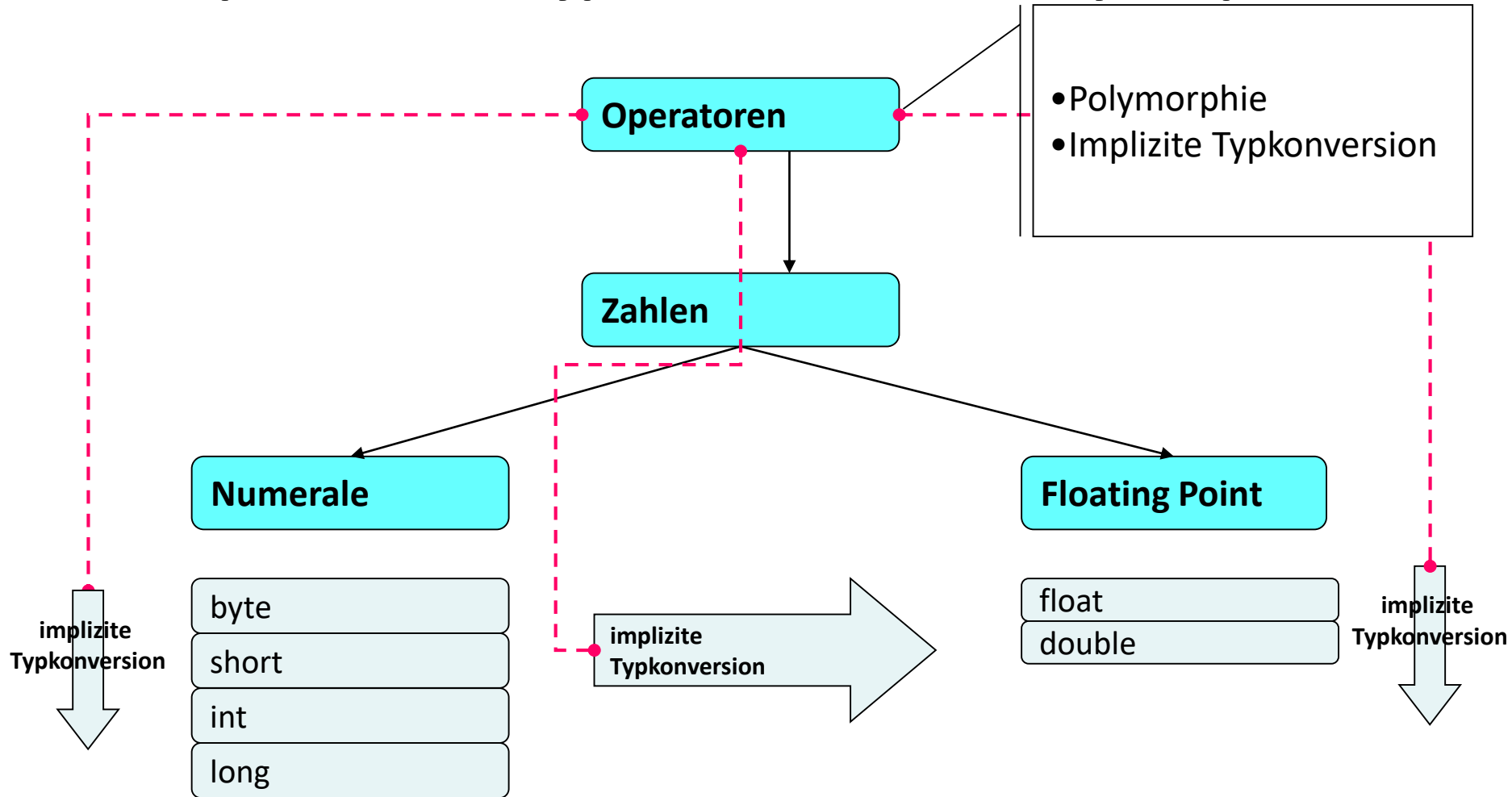


Implizite Typkonvertierung und Polymorphie

# **OPERATOREN**



# Operatoren – Typkonversion und Polymorphie





# Operatoren und implizite Typkonversion

- **implizite Typkonversion (Coercion):**
  - Operatoren führen auf ihren Operanden zunächst eine Typkonversion in den "größten" beteiligten Typen durch.
  - Den Ergebnistyp bestimmt der Operand mit dem "größten" Typ.
  - Implizite Konvertierung in "kleinere" Typen ist nicht erlaubt.
- Bei Zuweisungen (**=**) muss der Typ des Ergebnisses auf der rechten Seite immer in den Typ der Variablen auf der linken Seite passen.
- Der Compiler prüft die Kompatibilität der Typen.
- Das Verletzen der Regeln erzeugt einen Compilerfehler.
- Beispiele für **Coercion**
  - mathematische Operationen wandeln alle Operanden kleiner **int** zunächst in **int** um
  - bei gemischten Operanden-Typen wird der kleinere Typ in den größeren Typ umgewandelt.

**1 + 2 (int)**  
**1.0 + 2 (double)**  
**1 + 2.0f (float)**  
**1.0 + 2.0f (double)**



# Implizite Typkonversion-Überprüfung

| Deklaration/<br>Definition    | Zuweisungen                                                                                                       | Fehler? | Begründung |
|-------------------------------|-------------------------------------------------------------------------------------------------------------------|---------|------------|
| <code>short s1=4; s2=6</code> | <code>s2 = c2;</code><br><code>s1 = b;</code>                                                                     |         |            |
| <code>char c1;</code>         | <code>c1 = s2;</code>                                                                                             |         |            |
| <code>char c2=134;</code>     |                                                                                                                   |         |            |
| <code>byte b = 14;</code>     | <code>b = s2;</code>                                                                                              |         |            |
| <code>int i;</code>           | <code>i = c2;</code><br><code>i = s2;</code><br><code>i = b;</code>                                               |         |            |
| <code>float f;</code>         | <code>f = s2;</code><br><code>f = 1;</code>                                                                       |         |            |
| <code>long l = 1678;</code>   |                                                                                                                   |         |            |
| <code>double d;</code>        | <code>d = c1;</code><br><code>d = b;</code><br><code>d = s1;</code><br><code>d = l;</code><br><code>d = f;</code> |         |            |



# Implizite Typkonversion-Überprüfung

| Deklaration/<br>Definition      | Zuweisungen                                                                       | Fehler? | Begründung |
|---------------------------------|-----------------------------------------------------------------------------------|---------|------------|
| <code>short s1 = 4;</code>      | <code>s1 = ++s1;</code><br><code>s1 %= 4;</code><br><code>s1 = s1 % 4;</code>     |         |            |
| <code>byte b1 = 13;</code>      | <code>b1 = --b1;</code><br><code>b1 -= 5;</code><br><code>b1 = b1 - 5;</code>     |         |            |
| <code>char c1 = 'a';</code>     | <code>c1 = ++c1;</code><br><code>c1 /= 'a';</code><br><code>c1 = c1 / 'a';</code> |         |            |
| <code>int i1 = 56788;</code>    | <code>i1 = i1++;</code><br><code>i1 -= 17;</code><br><code>i1 = i1 - 17</code>    |         |            |
| <code>long l1 = 5435098;</code> | <code>l1 = l1++;</code><br><code>l1 -= 17;</code><br><code>l1 = l1 - 17</code>    |         |            |



# Implizite Typkonversion-Überprüfung

| Deklaration/<br>Definition                     | Ausdruck                                                                                                   | Fehler? | Begründung |
|------------------------------------------------|------------------------------------------------------------------------------------------------------------|---------|------------|
| <code>short s1=4;<br/>short s2=6;</code>       | <code>s2 = s2 + s1;</code>                                                                                 |         |            |
| <code>char c1 = 'a';<br/>char c2 = 'b';</code> | <code>c1 = c1 - c2;</code>                                                                                 |         |            |
| <code>byte b1 = 3;<br/>byte b2 = 15;</code>    | <code>b1 = b1 / b2;</code>                                                                                 |         |            |
| <code>int i;</code>                            | <code>i = s2 + s1;<br/>i = c1 - c2;<br/>i = b1 / b2;</code>                                                |         |            |
| <code>float f = 14.7f;</code>                  | <code>Math.pow(f, f);<br/>f = i + s1 + b1;<br/>f = c1 / s2;<br/>f = Math.pow(f, f);<br/>f = d / c1;</code> |         |            |
| <code>double d;</code>                         | <code>d = f - c2;<br/>d = Math.pow(f, f);;</code>                                                          |         |            |



# Operator-Polymorphie

- Operator Polymorphie besagt,
  - dass gleiche Operatoren für unterschiedliche Datentypen definiert sind.
  - dass sich Operatoren in Abhängigkeit vom Typ des Operanden unterschiedlich verhalten:
    - Arithmetische Operatoren verarbeiten integrale Werte und Gleitkommazahlen.
    - Der Typ des Ergebnisses hängt von Typ der Operanden ab.
    - */* verhält sich für Numerale anders als für Gleitkommazahlen.
    - *+* ist für Strings und Zahlen definiert.
    - *+* verhält sich für Strings anders als für Zahlen.

```
package operators;
import static util.Print.*;

public class PolymorphDemo {

 public static void main(String[] args) {
 print("1 / 2= " + 1/2);
 print("1.0 /2.0= " + 1.0 / 2.0);
 print("3" + "9");
 print("3" + 9);
 print(3 + 9);
 }
}
```





# Übungen

- **ue-2-8:** Erzeugen Sie den maximalen *double*-Wert und verdoppeln Sie den Wert. Geben Sie den maximalen Wert und das Ergebnis der Verdoppelung aus. Erzeugen Sie dann den maximalen *int*-Wert und verdoppeln den Wert. Geben Sie den maximalen Wert und das Ergebnis der Verdoppelung aus. Was stellen Sie fest?
- **ue-2-9:** Erzeugen Sie den maximalen *byte*-Wert und verdoppeln den Wert. Welches Ergebnis erwarten Sie bei der Verdoppelung? Geben Sie dann den maximalen Wert und das Ergebnis der Verdoppelung aus. Erklären Sie die Beobachtung. Für welche Basisdatentypen erwarten Sie gleiches Verhalten?



# Operatoren "+" und "+=" für Strings

```
package operators;

import static util.Print.*;

public class StringOperator {
 public static void main(String[]
args) {
 int x = 0, y = 1, z = 2;
 String s = "x, y , z ";
 print(s + x +y +z);
 print(x + " " + s);
 s += "(summed) = ";
 print(s + (x + y + z));
 }
}
```

- Wenn ein Ausdruck mit dem Operator "+" oder "+=" einen String enthält, dann ist der Ergebnistyp **String**.
- Alle Operanden, die keine Strings sind, werden vor Anwendung des Operators in einen String gewandelt.
- Soll diese Typkonversion vermieden werden, dann muss immer geklammert werden.



Explizite Typkonvertierung mit dem Castoperator

# OPERATOREN



# Explizite Typkonvertierung mit dem Cast-Operator

```
int i = 200;
long lo1 = (long) i; // widening
lo1 = i; // cast nicht notwendig
long lo2 = (long) 200; // widening
lo2 = 200;
// narrowing verlangt cast
// i = lo2; Fehler
i = (int) lo2;
```

- Coercion wandelt Operanden-Werte implizit in einen größeren Typ um.
- Der Cast-Operator (`<Datentyp>`) macht die Typumwandlung explizit.
- Man unterscheidet
  - **widening / upcast**: Wert eines kleineren Typs wird in den Wert eines größeren Typs gewandelt - unkritisch
  - **narrowing / downcast**: Wert eines größeren Typs wird in den Wert eines kleineren gewandelt – **kritisch**: es kann Information verloren gehen.
- Alle primitiven Typen **außer boolean** können untereinander gecastet werden.
- Upcast und Downcast ist nur entlang der Vererbungskette / Kompatibilitätskette möglich



# Informationsverlust durch Narrowing

```
int i = Integer.MAX_VALUE;
print("i " + i);
short s = (short)i; // narrow
int j = s;
print("j " + j); // j ungleich i
long l = Long.MIN_VALUE;
print("l " + l);
i = (int) l; // narrow
print("i " + i); // i ungleich l
l = Integer.MIN_VALUE;
print("l " + l);
i = (int) l; // narrow
print("i " + i); // i gleich l
```

- **Narrowing** kann, muss aber nicht immer zu Informationsverlust führen.
- In der letzten Zeile passt der Wert der **long** Variable in die **int** Variable, daher ist hier **i=l**.



# Informationsverlust durch Narrowing

```
int i = Integer.MAX_VALUE;
print("i " + i);

short s = (short)i; // narrow
int j = s;
print("j " + j); // j ungleich i


long l = Long.MIN_VALUE;
print("l " + l);

i = (int) l; // narrow
print("i " + i); // i ungleich l

l = Integer.MIN_VALUE;
print("l " + l);

i = (int) l; // narrow
print("i " + i); // i gleich l
```

## Ausgabe



```
i 2147483647
j -1
l -9223372036854775808
i 0
l -2147483648
i -2147483648
```

# Cast-Abschneiden (Truncate) - Runden (Round)



```
double above= 0.7, below=0.4;
float fabove= 0.7f, fbelow=0.4f;
print("(int)above " + (int)above);
print("(int)below " + (int)below);
print("(int)fabove " + (int)fabove);
print("(int)fbelow " + (int)fbelow);
```



```
(int)above 0
(int)below 0
(int)fabove 0
(int)fbelow 0
```

- Was passiert, wenn ein **float** / **double** in eine ganze Zahl gewandelt wird?
- **Narrowing** (downcasting) schneidet Werte ab (**truncate**), es rundet die Werte nicht.
- Um Werte zu runden (round), muss die Methode **Math.round(...)** verwendet werden.

siehe: **operators.CastingNumbers**



# Cast-Abschneiden (Truncate) - Runden (Round)

```
double above= 0.7, below=0.4;
float fabove= 0.7f, fbelow=0.4f;

print("Math.round(above) "
 + Math.round(above));

print("Math.round(below) "
 + Math.round(below));

print("Math.round(fabove) "
 + Math.round(fabove));

print("Math.round(fbelow) "
 + Math.round(fbelow));
```



```
Math.round(above) 1
Math.round(below) 0
Math.round(fabove) 1
Math.round(fbelow) 0
```

- Was passiert, wenn ein **float / double** in ein Numeral gewandelt wird?
- **Narrowing** (downcasting) schneidet Werte ab (**truncate**), es rundet die Werte nicht.
- Um Werte zu runden (round), muss die Methode **Math.round(...)** verwendet werden.

siehe: **operators.CastingNumbers**





Operator Präzedenz und Assoziativität

# OPERATOREN



# Operatorpräzedenz und Assoziativität

- Operatorpräzedenz regelt die Auswertungsreihenfolge der Operatoren. Je höher die Präzedenz, um so eher die Auswertung des Operators. (Beispiel: *Punkt vor Strich* Regel)
- Darüber hinaus gibt es in Java eine Menge von Regeln (siehe <http://mindprod.com/jgloss/precedence.html> ).
- Vorteil von Operatorpräzedenz ist das Einsparen von Klammern, der zu zahlende Preis umfangreiche Regeln kennen zu müssen.
- Daher im Zweifelsfall klammern.
- Die Auswertungsreihenfolge hängt bei gleicher Präzedenz von der Assoziativität des Operators ab.
- Rechts assoziative Operatoren werden von rechts nach links ausgewertet
- Links assoziative von links nach rechts.

Beispiel:

```
// ? ist rechts assoziativ
a = b ? c : d ? e : f;
// bedeutet daher
a = b ? c : (d ? e : f);
```



# Operatorpräzedenz und Assoziativität: ein Beispiel

- **+, -, \*** sind linksassoziativ.
- **+** mit String Objekten wandelt alle beteiligten Operanden in einen String.
- **\*** hat höhere Präzedenz als **+** und daher wird zunächst **(2\*3)** multipliziert und das Ergebnis an den String **"show "** angehängt.
- **-** hat gleiche Präzedenz wie **+**. Es wird zunächst die **2** an **"show"** angehängt und dann versucht **3** von einem String zu subtrahieren. **(Fehler!)**
- Daher muss in diesem Fall **(2-3)** geklammert werden.

```
private static void
 main(String[] args) {
 print("show " + 2 + 3);
 print("show " + 2 * 3);
 print("show " + 2 - 3);
 print("show " + (2 - 3));
 }
```



# Tabelle Operatoren

| Operator           | Name                       | Operanden | Operanden Typ | Priorität | Assoziativität |
|--------------------|----------------------------|-----------|---------------|-----------|----------------|
| ++, --             | Inkrement, Dekrement       | 1         | N             | 1         | rechts         |
| +, -               | Vorzeichen                 | 1         | N             | 1         | rechts         |
| !                  | Negation                   | 1         | B             | 1         | rechts         |
| (Typ)              | Cast                       | 1         | A             | 1         | rechts         |
| *, /, %            | Mult., Div., Modulo        | 2         | N, N          | 2         | links          |
| +, -               | Add., Subt.                | 2         | N, N          | 3         | links          |
| +                  | Add. mit Strings           | 2         | S, N          | 3         | links          |
| <, <=, >, >=       | Vergleich                  | 2         | B, B          | 5         | links          |
| instanceof         | ist das Objekt vom Typ     | 2         | O, OT         | 5         | links          |
| ==, !=             | Identität, nicht identisch | 2         | O, O und P, P | 6         | links          |
| &&,                | Und, Oder                  | 2         | B, B          | 10, 11    | links          |
| ?:                 | ternärer Operator          | 3         | B, A, A       | 12        | rechts         |
| =                  | Zuweisung                  | 2         | V, A          | 13        | rechts         |
| +=, -=, *=, /=, %= | Operatorzuweisung          | 2         | V, N          | 13        | rechts         |



# OPERATOREN UND WRAPPERTYPEN



# Operatoren und Wrappertypen

- Arithmetische Operatoren sind nur für Basisdatentypen definiert → die Wrappertypen sind keine Basisdatentypen
- Zuweisungen sind nur zwischen kompatiblen Typen zulässig → Wrappertypen sind Objekttypen. Objekttypen und Basisdatentypen sind nicht kompatibel.
- Dennoch kann mit Wrappertypen gerechnet werden.
- Dennoch sind Zuweisungen zwischen Wrappertypen und Basisdatentypen möglich.
- **WARUM?**
  - Die Lösung nennt sich ***Auto(un)boxing***. Dies ist ein **Compilertrick**, der für die Übersetzung zwischen dem Wrapper- und Basisdatentyp verantwortlich ist.
  - Bis Java 1.4 gab es diese Technik nicht, so dass die Übersetzung explizit in der Programmierung erfolgen musste.

# (Un)Boxing von Wrappern und Werten von Basisdatentypen



- **Autoboxing:**
  - An allen Stellen im Programm, an denen ein Wrappertyp erwartet wird, wird ein der Wert eines Basisdatentyps in das zugehörige Wrapperobjekt umgewandelt.
- **Autounboxing**
  - An allen Stellen im Programm, an denen ein Basisdatentyp erwartet wird, wird ein Wrapperobjekt in den zugehörigen Wert des Basisdatentyps umgewandelt.
- **Grenzen:**
  - Wrappertypen werden nur in ihre zugeordneten Basisdatentypen umgewandelt.
  - Coercion und anschließendes Boxing ist nicht möglich.
  - Generische Typen akzeptieren keine Basisdatentypen. (*später*)
  - bei überladenen Methoden wird Autoboxing nur dann angewendet, wenn keine andere Typkonversion zum Ziel führt.



# Beispiele für Auto(un)boxing

```
// Autoboxing 4 → Integer, dann Zuweisung
```

```
Integer integer;
integer = 4;
```

```
// Unboxing integer.intValue() -> int dann Zuweisung
```

```
int z = integer;
```

```
// Generische Typen
```

```
List<Integer> v = new ArrayList<Integer>(); // OK
```

```
// umständliches Schreiben bis Java 1.4
```

```
v.add(new Integer(4));
```

```
// mit Autoboxing
```

```
v.add(3);
```

```
// umständliches Lesen bis Java 1.4
```

```
int x = v.get(0).intValue();
```

```
// mit Autounboxing
```

```
int y = v.get(0);
```





# Regeln und Grenzen für Autoboxing

| keine Methoden für primitive Typen                                | < Java 1.4<br>keine Operationen für Wrapper | > Java 1.4 (5, 6 ...)<br>Operationen für Wrapper (Box/Unbox) | Einschränkungen für Box / Unbox<br>Sequenz: Unbox -> Typkonversion möglich<br>Sequenz: Typkonversion -> Box nicht möglich                                                                                                                                               |
|-------------------------------------------------------------------|---------------------------------------------|--------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| f.equals(f);<br>// error<br>c.equals(c)<br>// error<br><br>usw... | bool && boolW.booleanValue();               | bool && boolW;<br>boolW && boolW;                            | double d = new Integer(3); // ok<br><br>Double dW = 3; // Fehler<br>Double dW = new Integer(3);<br>// Fehler Integer nicht kompatibel zu<br>// Double<br><br>// Generische Typen verlangen<br>// Referenztypen<br>ArrayList<int> v =<br>new ArrayList<int>(); // Fehler |
|                                                                   | c + cW.charValue();                         | c + cW; cW + cW;                                             |                                                                                                                                                                                                                                                                         |
|                                                                   | b + bW.byteValue();                         | b + bW; bW + bW                                              |                                                                                                                                                                                                                                                                         |
|                                                                   | s / sW.shortValue();                        | s / sW; sW + sW                                              |                                                                                                                                                                                                                                                                         |
|                                                                   | i + iW.intValue();                          | i + iW; iW + iW                                              |                                                                                                                                                                                                                                                                         |
|                                                                   | l / lW.longValue();                         | l + lW; lW / lW                                              |                                                                                                                                                                                                                                                                         |
|                                                                   | f + fW.floatValue();                        | f + fW; fW + fW;                                             |                                                                                                                                                                                                                                                                         |
|                                                                   | d + dW.doubleValue();                       | d + dW; dW / dW;                                             |                                                                                                                                                                                                                                                                         |



# Operationen mit Wrappertypen-Überprüfung

| Ausdruck                                                | Fehler? | Begründung |
|---------------------------------------------------------|---------|------------|
| <code>Double d = new Double(4) + new Double(17);</code> |         |            |
| <code>d = new Double(4) + new Byte( (byte)17 );</code>  |         |            |
| <code>d = new Double(4) + 17;</code>                    |         |            |
| <code>d = 17;</code>                                    |         |            |
| <code>Double d = new Integer(17);</code>                |         |            |



Exkurs

# **VARARGS**



# Exkurs: Varargs

- Mit **Varargs** (*variable length argument list*) lassen sich Methoden mit einer beliebigen Anzahl von Argumenten definieren:
  - syntaktisch wird ein Vararg-Parameter mit drei Punkten nach der Typangabe markiert
  - bei jedem Aufruf einer Methode mit Varargs wird ein Array erzeugt, das die übergebenen Argumente enthält.
  - aus Sicht des Compilers werden Aufrufe von Vararg-Methoden behandelt wie ein Aufruf mit einem Arrayliteral.
  - In längeren Argumentlisten darf nur ein Vararg-Parameter auftreten.

```
package varargs;
import static util.Printer.*;

class NumericDemo {
 public static int sum(int... args) {
 int sum = 0;
 for (int i : args)
 sum += i;
 return sum;
 }

 public static void main(String[] args) {

 // Varargs
 print(NumericDemo.sum(1, 2, 3, 4, 5));
 // äquivalent zu
 print(NumericDemo.sum(new int[] { 1, 2,
 3, 4, 5 }));
 }
}
```



# Grenzen des Autoboxing bei überladenen Methoden

- **Autoboxing** wird nur angewendet, wenn keine andere Typkonversion zum Ziel führt
- mit dem Aufruf `foo(1)` ;
  - 1'tes Beispiel:
    - erste Methode wird gewählt (int -> double)
  - 2'tes Beispiel:
    - Varargs und Autoboxing haben gleichen Rang
    - erste wird gewählt, da sie genauer passt
  - 3'tes Beispiel:
    - Aufruf ist mehrdeutig
    - erste Methode akzeptiert wegen Autoboxing alle Aufrufe der zweiten Methode
    - zweite Methode akzeptiert wegen Autounboxing alle Aufrufe der ersten Methode

```
foo(1);
static void foo(double d){ print(d);};
static void foo(Integer i){ print(i);};
```

```
foo(1);
static void foo(Integer i){
 print("foo(Integer i)");print(i);};
static void foo(int... i) {
 print("foo(int...)");print(i);};
}
```

```
foo(1); // mehrdeutig Compilerfehler
static void foo(int... i) {};
static void foo(Integer... i){};
```



# Zusammenfassung

- **Basisdatentypen:**
  - Java hat neben Referenztypen (Klassen) das Typsystem der **Basisdatentypen**.
  - Basisdatentypen sind **keine** Klassen.
  - Basisdatentypen legen **Wertebereiche** für gültige Werte fest.
  - Werte sind **keine** Objekte.
  - Basisdatentypen haben **keine** Methoden.
  - Das Rechnen mit Werten ist nur mittels **Operatoren** möglich.
- **Wrappertypen:**
  - Zu jedem Basisdatentypen gibt es einen **Wrappertyp**. Wrappertypen sind Referenztypen.
  - Wrappertypen geben z.B. Auskunft über minimale/maximale Werte des zugehörigen Basisdatentyps.
  - Mit Wrapper-Objekten kann wie mit Werten gerechnet werden. Die Technik dahinter nennt sich **Auto(un)boxing**.
  - Zeichenketten können mit Hilfe von **Konvertierungsmethoden** der Wrappertypen in Werte der Basisdatentypen überführt werden.
  - Wrappertypen sind **nicht** untereinander kompatibel! Alle Wrappertypen sind kompatibel zu **Number**.



# Zusammenfassung

- **Operatoren:**
  - Sind Schlüsselwörter oder Zeichen mit festgelegter Semantik.
  - Es gibt **arithmetische, logische, Bit-, Zuweisungs-, Typprüfungs- und Typumwandlungs-** Operatoren. Einige der Operatoren sind auch für Referenztypen definiert.
  - Die **arithmetischen Grundrechenarten** sind als bekannte Zeichen vorhanden. Weitere arithmetische Operatoren sind als statische Methoden der Klasse **Math** verfügbar.
  - **Coercion** bezeichnet die implizite Typumwandlung von Operanden auf den größten Operandentyp. **Coercion** ist nur für Basisdatentypen **aber nicht** für die zugehörigen Wrappertypen definiert.
  - **Casten** „gaukelt“ dem Compiler vor, dass der Wert von einem anderen Typ als der inferierte, angenommene Typ ist. **Downcasten** führt bei Werten von Basisdatentypen zu Informationsverlust, wenn der Wert nicht in den angegebenen Typ passt. (Bei Referenztypen erzeugt ein falscher Downcast einen Laufzeitfehler **später**).
  - **Operatorzuweisungen** sind Kurzformen für spezielle Operatorausdrücke.
  - **Präfix- und Postfix-Operatoren** sind Auto-in/-de-krement Operatoren.
  - Operator-**Präzedenz** und –**Assoziativität** regelt die Auswertung von zusammen gesetzten Ausdrücken und reduziert die Klammerung.
- Die Klasse **Scanner** ist (u.a.) ein komfortables Tool für das Lesen und **Verarbeiten von Eingaben auf der Konsole**.