Namen, Datentypen und Operatoren in Java

- Namen – formal Bezeichner genannt – haben in Java denselben Zweck wie in natürlichen Sprachen und in C/C++
  - sie werden Dingen zugeordnet, damit man diese auseinander halten und einfach benennen kann
- u. a. folgende Dinge werden in Java im Rahmen ihrer Deklaration mit Bezeichnern versehen:
  - Packages
  - Klassen, Interfaces, Annotations, Typvariablen, enum-Konstanten
  - Attribute, Konstruktoren, Methoden
  - Parameter, Variablen

- wie in C/C++ wird auch in Java zwischen Groß- und Kleinschreibung unterschieden
- welche Gestalt müssen Bezeichner in Java haben?
  - sie müssen immer mit einem Buchstaben beginnen
  - danach können Buchstaben oder Ziffern folgen
- zulässige Bezeichner sind also zum Beispiel:

```
ErstesProgrammMain
nummer17a
x
```


- nicht zulässig sind:

```
Mein-Datum
4you
```

- ist `zähler` ein zulässiger Bezeichner in Java?
  - das hängt davon ab, ob `ä` aus Sicht von Java ein Buchstabe ist
- was also genau ist ein Buchstabe in Java?
  - die Antwort auf diese Frage ist überraschend kompliziert

- Java-Programme können im jeweils aktuellen Unicode-Zeichensatz verfasst werden
  - dabei handelt es sich um einen vom Unicode-Konsortium entwickelten Zeichensatz, dessen Ziel es ist, jedem in jeder Schriftsprache vorkommenden Zeichen ein eindeutiges Bitmuster zuzuordnen
  - zur Zeit sind mehr als 100.000 Zeichen definiert, die mehr als 220 Sprachen abdecken, darunter alle europäischen Sprachen, chinesische Dialekte, japanische Dialekte, Arabisch, indische Sprachen, aber auch die Sprache der Cherokee-Indianer

  siehe: **http://www.unicode.org/**

- aus Sicht von Java ist ein Buchstabe, was in irgendeiner dieser Sprachen ein Buchstabe ist
  - **genauer:** ein Zeichen ist genau dann ein Buchstabe, wenn die Methode `isJavaIdentifierStart` der Klasse `java.lang.Character` mit diesem Zeichen als Input `true` liefert
    - was z.B für `ä`, `ö`, `ü`, `Ä`, `Ö`, `Ü` und `ß` der Fall ist
  - zusätzlich sind auch die Zeichen `_` und `$` Buchstaben

  Achtung: Bezeichner, die mit einem dieser beiden Zeichen beginnen, sollen vermieden werden
    - sie werden für interne Zwecke oder von Programmgeneratoren benutzt

- für Ziffern gilt Analoges
  - in Java ist eine Ziffer genau das, was die Methode `isDigit` der Klasse `java.lang.Character` als Ziffer ansieht

- **Achtung:** obwohl für Java-Programme der gesamte Unicode-Zeichensatz zur Verfügung steht, kann meist nur ein kleiner Teil davon genutzt werden
  - da der auf Rechnern tatsächlich vorhandene Zeichensatz nur eine kleine Teilmenge davon ist
  - in unserem Kulturkreis wird oft der Zeichensatz Latin-1 benutzt, der den ASCII-Zeichensatz umfasst

- wir empfehlen, in allen Bezeichnern – wie in C/C++ – immer nur ASCII-Zeichen zu benutzen

- Bezeichner dürfen in Java beliebig lang sein
  - es ist üblich, lange Bezeichner mit Hilfe von Binnenmajuskeln zu strukturieren (im Englischen CamelCase genannt):

```
diesIstAberEinSehrLangerName
```

- die in C/C++ weit verbreitete Schreibweise

  `dies_ist_aber_ein_sehr_langer_name`

  ist in Java zwar möglich, aber unüblich

- generell sollte man sich beim Programmieren mit Java weitgehend an die von der Firma SUN aufgestellten Java Code Conventions halten

  siehe: http://www.oracle.com/technetwork/java/codeconv-138413.html

- Bezeichner sollen immer so gewählt werden, dass sie selbsterklärend (d.h. mnemonisch) sind
  - also

    `tag` statt nur `t`

    `monat` statt nur `m`

    `jahr` statt nur `j`
- im Zweifelsfall sind lange Bezeichner, deren Bedeutung man sofort versteht, besser als kurze, aber unverständliche Namen
  - so wird auch in der Java-Klassenbibliothek vorgegangen, z.B. `ArrayIndexOutOfBoundsException`

- alle selbst gewählten Bezeichner müssen verschieden sein von den Literalen `true`, `false` und `null` und den 50 Schlüsselwörtern der Sprache Java:

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


- wie in C/C++ haben alle Werte in Java einen Datentyp
  - explizit anzugeben ist er
    - bei der Deklaration von Attributen
    - bei der Deklaration von Parametern von Konstruktoren und Methoden
    - bei der Deklaration lokaler Variablen
    - für den Rückgabewert von Methoden
    - und noch in weiteren Situationen, die wir zur gegebener Zeit kennen lernen werden
  - bei Literalen ist er durch deren Gestalt festgelegt
  - bei der Auswertung von Ausdrücken wird er an Hand der involvierten Operatoren ermittelt

- es gibt zwei Arten von Datentypen, die in Deklarationen angegeben werden können:
  - acht so genannte primitive Datentypen

    `byte` `short` `int` `long` `char`
    `float` `double`
    `boolean`

    und drei so genannte Referenzdatentypen

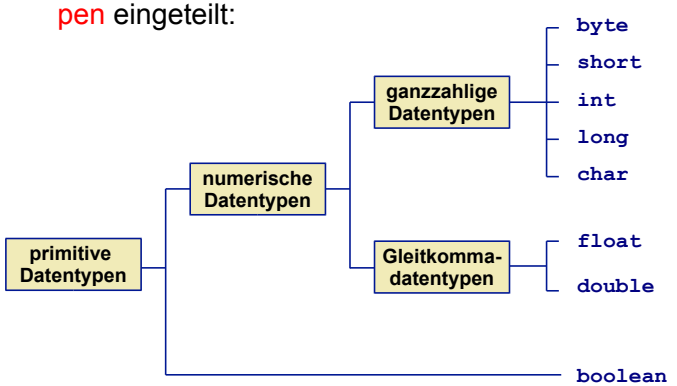
    `Arrays`
    `Klassen` (inklusive `enums`)
    `Interfaces`
- darüber hinaus gibt es einen weiteren Datentyp, der keinen Namen hat

  - und deswegen in Deklarationen nicht explizit angegeben werden kann
  - es gibt nur einen Wert dieses Datentyps:
    - das vordefinierte Literal `null`
- die acht primitiven Datentypen sind durch die Sprache vorgegeben
  - für sie ist genau festgelegt,
    - wie viel Speicherplatz ihre Werte einnehmen
    - welche Werte es gibt
    - welche Operationen mit den Werten zulässig sind

- Referenzdatentypen können dagegen selbst gestaltet werden
  - es gibt daher im Prinzip unendlich viele Referenzdatentypen
  - ihre Werte sind immer Objekte, auf die nur über Zeiger zugegriffen werden kann
    - die in Java Referenzen heißen
  - wie viel Speicherplatz für die Objekte benötigt wird, ist von Typ zu Typ verschieden
  - welche Werte die Objekte annehmen können und welche Operationen zulässig sind, ist ebenfalls von Typ zu Typ verschieden

- **Hinweis:** wegen der zwei Arten von Datentypen ist Java (wie auch C++) keine rein objektorientierte Programmiersprache (wie z.B. Smalltalk oder Eiffel)
  - dies so einzurichten war eine bewusste Entscheidung von James Gosling
  - denn die Verwaltung von Referenzdatentypen ist zeitaufwändiger als die Verwaltung primitiver Datentypen

- die primitiven Datentypen werden in drei Gruppen eingeteilt:



- die aus C/C++ bekannten Datentypen `long long` und `long double` gibt es nicht
- anders als in C/C++ ist für alle primitiven Datentypen genau festgelegt, wie die Werte zu speichern sind
  - ganze Zahlen werden stets im Zweierkomplementverfahren mit Vorzeichen gespeichert
  - die benutzte Anzahl Bits ist überall gleich

Datentyp		Wertebereich		
`byte`	8	-128	bis	127
`short`	16	-32768	bis	32767
`int`	32	-2147483648	bis	2147483647
`long`	64	-9223372036854775808	bis	9223372036854775807

- der ganzzahlige Datentyp `char` dient zum Speichern von Zeichen des Unicode-Zeichensatzes
  - es stehen 16 Bits zur Verfügung, so dass 65536 verschiedene Zeichen gespeichert werden können
  - für die so nicht speicherbaren Zeichen des Unicode-Zeichensatzes gibt es Sonderregelungen, auf die wir aber nicht eingehen
- wie in C/C++ gibt es auch in Java keinen primitiven Datentyp für Zeichenketten
  - Zeichenketten sind in Java Objekte der Klasse `java.lang.String` (und damit – anders als in C – auch nicht spezielle `char`-Arrays!)

- Gleitkommazahlen vom Datentyp `float` oder `double` werden nach dem IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985 (IEEE, New York) gespeichert
- es gibt nur zwei Werte des Datentyps `boolean`
  - die vordefinierten Literale `true` und `false`

  Achtung: anders als in C/C++ sind `boolean` und die numerischen Datentypen nicht zuweisungskompatibel

- bei den primitiven Datentypen sind nämlich nur folgende Zuweisungen zulässig:

Wert des Datentyps	kann zugewiesen werden zu
byte	byte, short, int, long, float, double
short	short, int, long, float, double
char	char, int, long, float, double
int	int, long, float, double
long	long, float, double
float	float, double
double	double

- darüber hinaus gibt es noch diese Sonderfälle:

- wenn der Wert konstant ist (z.B. ein Literal oder das Ergebnis der Auswertung eines konstanten Ausdrucks) und zum Wertebereich des rechts angegebenen Datentyps gehört, liegt auch noch in folgenden Situationen Zuweisungskompatibilität vor:

Wert des Datentyps	kann zugewiesen werden zu
short	byte
char	byte, short
int	byte, short

Hinweis: mit Hilfe des cast-Operators können noch andere Zuweisungen erzwungen werden, bei denen es aber zu einem Informationsverlust kommen kann

- wie in C/C++ können ganzzahlige Literale im Dezimalsystem, im Oktalsystem und im Hexadezimalsystem angegeben werden
  - sie haben in Java dieselbe Gestalt wie in C/C++
  - ihr Datentyp ist `int`
  - durch Anfügen des Suffixes `l` oder `L` kann er zu `long` geändert werden
- bei Zeichenliteralen gibt es zwischen Java und C/C++ Unterschiede
  - ihr Datentyp ist `char` (und nicht – wie in C/C++ – `int`)
  - Literale für druckbare ASCII-Zeichen sind in C/C++ und Java identisch

- gleich sind auch die oktalen, hexadezimalen und die vordefinierten Escape-Sequenzen
- den aus C bekannten Escape-Sequenzen vom Typ universal character name entsprechen in Java Literale für Unicode-Zeichen, die nach folgendem Schema anzugeben sind:

```
'\uxxxx'
```

  - dabei steht jedes `x` für eine Hexadezimalziffer
  - zulässig sind alle Unicode-Zeichen außer

`\u000A`	line feed
`\u000D`	carriage return
`\u0027`	'
`\u005D`	\

- mit ihnen können Sonderzeichen dargestellt werden

- aber natürlich nur dann, wenn sie im Zeichensatz des Rechners vorhanden sind

Beispiele:

`\u00A9`	©
`\u00A3`	£
`\u20AC`	€
`\u00A5`	¥
`\u2211`	∑
`\u25A0`	■
`\u2663`	♣
`\u0429`	Щ

- Gleitkommaliterale sehen in Java genau so aus wie Gleitkommaliterale in C/C++
  - ihr Datentyp ist **`double`**

- durch Anfügen des Suffixes **`f`** oder **`F`** kann er zu **`float`** geändert werden

- Zeichenkettenliterale haben in Java immer folgende Gestalt

```
"cccccc...cccccc"
```

  - dabei steht jedes **`c`** für ein Zeichenliteral
  - zwischen den doppelten Hochkommata dürfen kein, ein oder mehrere Zeichenliterale stehen
    - deren Anzahl ist die Länge der Zeichenkette
    - die Zeichenkette **`""`** der Länge **`0`** wird leere Zeichenkette genannt

- zulässig sind alle Zeichenliterale außer

`\u000A`	line feed
`\u000D`	carriage return
`"`	
`\`	

- ihr Datentyp ist **`java.lang.String`**
- anders als in C können Zeichenketten in Java nicht direkt modifiziert werden
  - die Klasse **`java.lang.String`** stellt aber zahlreiche Methoden für den Umgang mit Zeichenketten zur Verfügung
  - sie sind ähnlich zu den Methoden der C++-Klasse **`string`**, aber nicht identisch

- Java verfügt mit **`+`** über einen äußerst nützlichen binären Konkatenierungsoperator

```
zeichenkette + a
a + zeichenkette
```

  - wenn einer der Operanden von **`+`** eine Zeichenkette ist, kann der zweite Operand einen beliebigen Datentyp haben
  - der Operator wandelt (falls nötig) den zweiten Operanden in eine Zeichenkette um und liefert als Ergebnis die durch Zusammenfügen der beiden Zeichenketten entstandene Zeichenkette

Beispiele:

`"" + 1.234` hat als Ergebnis die Zeichenkette `"1.234"`

`null + ""` hat als Ergebnis die Zeichenkette `"null"`

- mit dem Konkatenierungsoperator **+** können mehrere Zeichenketten sehr einfach zu einer zusammengefasst werden:

```
"First they ignore you,\n"
"then they laugh at you,\n"
"then they fight you,\n"
"and then you win!"
"\n\n\u0020Mahatma Gandhi\n"
```

sind fünf verschiedene Zeichenketten

- durch den Konkatenierungsoperator **+** werden sie zu einer Zeichenkette der Länge 103 zusammengefügt:

```
"First they ignore you,\n" +
"then they laugh at you,\n" +
"then they fight you,\n" +
"and then you win!" +
"\n\n\u0020Mahatma Gandhi\n"
```

- **Hinweis:** auf diese Weise können Zeichenkettenliterale, die für eine Zeile zu lang sind, auf mehrere Zeilen verteilt werden

- **Achtung:** wenn der Datentyp des zweiten Operanden ein Referenzdatentyp ist – z. B. ein Arraydatentyp oder eine von `java.lang.String` verschiedene Klasse – muss selbst dafür gesorgt werden, dass dessen Werte sinnvoll in Zeichenketten umgewandelt werden können
  - wie das geht, werden wir bald lernen

- für die ganzzahligen Datentypen `byte`, `short`, `int`, `long` und `char` gibt es in Java im wesentlichen dieselben Operatoren wie in C:
  - die unären Vorzeichenoperatoren **+** und **-**
    - wenn der Operand den Datentyp `long` hat, hat auch das Ergebnis den Datentyp `long`
    - ansonsten hat es immer den Datentyp `int`

      Beispiel: `-'a'` hat den Datentyp `int` und den Wert `-97`
  - die binären additiven Operatoren **+** und **-**
    - wenn einer der Operanden den Datentyp `long` hat, hat auch das Ergebnis den Datentyp `long`
    - ansonsten hat es immer den Datentyp `int`

- die binären multiplikativen Operatoren `*` , `/` und `%`
  - wenn einer der Operanden den Datentyp `long` hat, hat auch das Ergebnis den Datentyp `long`
  - ansonsten hat es immer den Datentyp `int`
  - wie in C/C++ handelt es sich bei `/` um die ganzzahlige Division
  - **Achtung:** hat bei `/` oder `%` der rechte Operand den Wert `0`, erzeugt die JVM eine `ArithmeticException`

    wird sie nicht adäquat behandelt, bricht die Ausführung des Programms sofort ab
- den unären Präfix-Inkrementoperator `++`
- den unären Postfix-Inkrementoperator `++`
- den unären Präfix-Dekrementoperator `--`

- den unären Postfix-Dekrementoperator `--`
  - bei den Inkrement- und Dekrementoperatoren ist der Datentyp des Ergebnisses stets identisch mit dem Datentyp des Operanden
  - **Achtung:** der Operand muss eine über einen Namen ansprechbare Speicherstelle sein (z.B. Attribut, Parameter, lokale Variable, Array-Komponente)

    er darf nicht als `final` deklariert worden sein

- wie in C/C++ gibt es bitweise Operatoren, deren Operanden ganzzahlig sein müssen
  - den unären bitweisen Komplement-Operator `~`
  - den binären bitweisen Und-Operator `&`

- den binären bitweisen Oder-Operator `|`
- den binären bitweisen Entweder-Oder-Operator `^`
- den binären bitweisen Schiebe-Operator Shift-Left `<<`
- den binären bitweisen Schiebe-Operator Shift-Right `>>`
- den binären bitweisen Schiebe-Operator Shift-Right `>>>`
  - **Achtung:** `>>>` gibt es in C/C++ nicht

    das Bitmuster des Ergebnisses von `n >>> m` entsteht aus dem Bitmuster von `n`, indem dieses um den Wert von `(m & 31)` nach rechts verschoben wird, wobei links frei werdende Bits den Wert `0` erhalten (auch das Vorzeichenbit)
  - hat bei den bitweisen Operatoren einer der Operanden den Datentyp `long`, hat auch das Ergebnis den Datentyp `long`
  - ansonsten hat es immer den Datentyp `int`

- auch für die Gleitkommadatentypen `float` und `double` gibt es in Java im wesentlichen dieselben Operatoren wie in C/C++
  - die unären Vorzeichenoperatoren `+` und `-`
    - der Datentyp des Ergebnisses ist identisch mit dem Datentyp des Operanden
  - die binären additiven Operatoren `+` und `-`
  - die binären multiplikativen Operatoren `*` , `/` und `%`
    - hat bei den additiven oder multiplikativen Operatoren einer der Operanden den Datentyp `double`, hat auch das Ergebnis den Datentyp `double`
    - ansonsten hat es den Datentyp `float`

- **Hinweis:** der Operator `%` für Gleitkommazahlen ist ungewöhnlich (in C/C++ gibt es ihn nicht)
  - das Ergebnis der Operation `x % y` wird wie folgt ermittelt:

    1. Fall: `x / y` ist positiv

    dann gibt es eine größte positive ganze Zahl `n`, so dass

    `n <= x / y`

    ist; das Ergebnis von `x % y` ist in diesem Fall

    `x - n * y`

    2. Fall: `x / y` ist negativ

    dann gibt es eine kleinste negative ganze Zahl `n`, so dass

    `| n | <= | x / y |`

    ist; das Ergebnis von `x % y` ist in diesem Fall

    `x - n * y`

Beispiel: `-24.78 % -5.23` (1. Fall)

es ist `n = 4` die größte positive ganze Zahl, für die

`n <= (-24.78 / -5.23) = 4.7380497`

ist; `-24.78 % -5.23` hat daher den Wert

```
-24.78 - (4 * -5.23) = -24.78 + 20.92
                     = -3.86
```

Beispiel: `24.78 % -5.23` (2. Fall)

es ist `n = -4` die kleinste ganze Zahl, für die

`| n | <= | 24.78 / -5.23 | = 4.7380497`

ist; `-24.78 % 5.23` hat daher den Wert

```
24.78 - (-4 * -5.23) = 24.78 - 20.92
                     = 3.86
```


- wie in C/C++ ist es bei den binären additiven und multiplikativen Operatoren für die Gleitkommadatentypen zulässig, dass einer der Operanden einen ganzzahligen Datentyp und der andere einen Gleitkommadatentyp hat
  - das Ergebnis hat dann stets den Gleitkommadatentyp
- den unären Präfix-Inkrementoperator `++`
- den unären Postfix-Inkrementoperator `++`
- den unären Präfix-Dekrementoperator `--`
- den unären Postfix-Dekrementoperator `--`
  - sie verhalten sich bei Operanden, die den Datentyp `float` oder `double` haben, genau so, wie bei ganzzahligen Operanden

- **Achtung:** bei den Gleitkommadatentypen `float` und `double` gibt es spezielle Werte, die als Ergebnis von Bereichsüberschreitungen oder nicht zulässigen Operationen auftreten können

  `+∞`

  `-∞`

  `NaN` (Not a Number)

  - außerdem gibt es eine positive und eine negative Null

    `+0.0` `-0.0`

  - für diese Spezialwerte gelten besondere Regeln, auf die wir aber nicht weiter eingehen (`+∞ + -∞` hat z.B. `NaN` als Ergebnis)

- in Java gibt es dieselben Vergleichsoperatoren wie in C/C++:

  `==` `!=` `<` `<=` `>` `>=`

  - das Ergebnis der Operationen hat in Java aber immer den Datentyp **`boolean`**
  - wie in C/C++ können
    - beide Operanden einen ganzzahligen Datentyp haben
    - beide Operanden einen Gleitkommadatentyp haben
    - ein Operand einen ganzzahligen Datentyp und der andere einen Gleitkommadatentyp haben
  - bei den Operatoren `==` und `!=` dürfen ferner beide Operanden den Datentyp **`boolean`** haben

- für den Datentyp **`boolean`** gibt es in Java außer den Vergleichsoperatoren `==` und `!=` noch folgende Operatoren:
  - den unären Operator Nicht `!`
  - den binären Und-Operator `&&`
  - den binären Oder-Operator `||`
    - sie verhalten sich wie die entsprechenden Operatoren in C/C++
  - den binären Und-Operator `&` (nicht in C/C++)
    - er unterscheidet sich von `&&` dadurch, dass immer beide Operanden ausgewertet werden

- den binären Oder-Operator `|` (nicht in C/C++)
  - er unterscheidet sich von `||` dadurch, dass immer beide Operanden ausgewertet werden
- den binären Entweder-Oder-Operator `^` (nicht in C/C++)
  - er liefert **`true`**, wenn die Operanden unterschiedliche Werte haben, ansonsten **`false`**

  Hinweis: für Operanden vom Datentyp **`boolean`** liefern `!=` und `^` immer dasselbe Ergebnis
- die Operanden dieser Operatoren müssen immer den Datentyp **`boolean`** haben
- das Ergebnis hat stets den Datentyp **`boolean`**

- wie in C/C++ sind Wertzuweisungen auch in Java Operationen
  - neben dem einfachen Wertzuweisungsoperator `=` gibt es 11 zusammengesetzte Wertzuweisungsoperatoren:

    `+=` `-=` `*=` `/=` `%=`

    `&=` `|=` `^=`

    `<<=` `>>=` `>>>=`
  - sie bewirken dasselbe, wie die entsprechenden Operatoren in C/C++ (zu `>>>=` gibt es allerdings kein Pendant)

- zum Konvertieren von Datentypen zur Laufzeit gibt es – wie in C/C++ – auch in Java den unären cast-Operator
  - für die primitiven Datentypen arbeitet er nach folgendem Schema:

```
(ZielDatentyp) Ausdruck
```

  - konkret sieht das zum Beispiel so aus:

```
int basisWert = 18;
byte endWert = (byte) (7 * basisWert);
```

- **Achtung:** durch den Einsatz des cast-Operators wird die strenge Typüberprüfung des Compilers umgangen

  - dadurch kann es sehr leicht zu Fehlern kommen:
    - bei

```
int basisWert = 18;
byte endWert = (byte) (7 * basisWert);
System.out.println(endWert);                    //  126
```

      ist alles gut gegangen und das Ergebnis ist korrekt
    - in diesem Fall geht es aber schief:

```
int basisWert = 18;
byte endWert = (byte) (8 * basisWert);
System.out.println(endWert);                    //  -112
```

  - weil die Typüberprüfung umgangen wird, können derartige Fehler nur durch Testen gefunden werden
    - manchmal werden sie aber auch übersehen:

Am 4. Juni 1996 wurde der Prototyp der Ariane-5-Rakete der Europäischen Raumfahrtbehörde eine Minute nach dem Start in vier Kilometern Höhe gesprengt, weil der Programmcode, der von der Ariane 4 übernommen worden war und nur für einen von der Ariane 4 nicht überschreitbaren Bereich (Beschleunigungswert) funktionierte, die Steuersysteme zum Erliegen brachte, als eben dieser Bereich von der Ariane 5, die stärker als die Ariane 4 beschleunigt, überschritten wurde.

Dabei war es zu einem Fehler bei einer Typumwandlung gekommen, dessen Auftreten durch die verwendete Programmiersprache Ada eigentlich hätte entdeckt und behandelt werden können. Diese Sicherheitsfunktionalität ließen die Verantwortlichen jedoch abschalten. Der Schaden betrug etwa 370 Millionen US-Dollar.

Quelle: http://de.wikipedia.org/wiki/Programmfehler

- wir raten, den cast-Operator bei primitiven Datentypen nur dann zu benutzen, wenn erwiesen ist, dass das Problem nicht auf andere Weise gelöst werden kann
  - er wird später bei Referenzdatentypen – genauer: bei der Vererbung von Klassen – häufiger eingesetzt werden
- manche Konvertierungen lässt der Compiler selbst beim Einsatz des cast-Operators nicht zu
  - das folgende Beispiel zeigt, dass der Versuch, Werte vom Datentyp `boolean` in den (bei C/C++ üblichen) Datentyp `int` zu konvertieren, scheitert

```
int eingabeOk = (int) ((1 <= eingabe) && (eingabe <= 100));
```

```
inconvertible types
found   : boolean
required: int
   int eingabeOk = (int) ((1 <= eingabe) && (eingabe <= 100));
```

- wie in C/C++ kann auch in Java gelegentlich statt einer **`if`**-Anweisung der ternäre Bedingungsoperator eingesetzt werden

```
Ausdruck1 ? Ausdruck2 : Ausdruck3
```

  - der Datentyp von **`Ausdruck1`** muss **`boolean`** oder **`Boolean`** sein

  - die Werte von **`Ausdruck2`** und **`Ausdruck3`** können unterschiedliche Datentypen haben, wobei jeder Datentyp zulässig ist
  - der Datentyp des Ergebnisses hängt von den Datentypen von **`Ausdruck2`** und **`Ausdruck3`** ab
    - wenn **`Ausdruck2`** und **`Ausdruck3`** denselben Datentyp haben, hat auch das Ergebnis der Operation diesen Datentyp
    - falls sie unterschiedliche Datentypen haben, wird der Datentyp des Ergebnisses an Hand (relativ) komplexer Regeln ermittelt

      Hinweis: wegen der Komplexität dieser Regeln empfehlen wir, den Bedingungsoperator nur dann einzusetzen, wenn **`Ausdruck2`** und **`Ausdruck3`** denselben Datentyp haben

- wie in C/C++ haben auch in Java alle Operatoren eine Stärke und eine Assoziativität
  - die Stärke wird vom Compiler bei der Auswertung von Ausdrücken benutzt, um die Reihenfolge festzulegen, in der die Operatoren angewendet werden
  - die Assoziativität kommt zum Tragen, wenn Operatoren gleicher Stärke aufeinander treffen
- die Stärke und die Assoziativität der bisher betrachteten Operatoren kann der folgenden Tabelle entnommen werden:

Stärke	Assoziativität
unäre Postfix-Inkrement- und -Dekrementoperatoren `++` und `--`	rechts nach links
unäre Vorzeichenoperatoren `+` und `-` unäre Präfix-Inkrement- und -Dekrementoperatoren `++` und `--` unäres bitweises Komplement `~` unäres logisches Nicht `!`	rechts nach links
cast-Operator new-Operator	rechts nach links
binäre multiplikative Operatoren `*` , `/` und `%`	links nach rechts
binäre additive Operatoren `+` und `-`	links nach rechts
binäre Verschiebe-Operatoren `<<` , `>>` und `>>>`	links nach rechts
binäre Vergleichsoperatoren `<` , `<=` , `>` und `>=`	links nach rechts
binäre Vergleichsoperatoren `==` und `!=`	links nach rechts
binärer bitweiser oder logischer Und-Operator `&`	links nach rechts
binärer bitweiser oder logischer Entweder-Oder-Operator `^`	links nach rechts
binärer bitweiser oder logischer Oder-Operator `|`	links nach rechts
binärer logischer Und-Operator `&&`	links nach rechts
binärer logischer Oder-Operator `||`	links nach rechts
ternärer Bedingungsoperator `? :`	rechts nach links
binäre Zuweisungsoperatoren `=` , `+=` , `-=` , `*=` , `/=` , `%=` , `<<=` , `>>=` , `>>>=` , `&=` , `^=` und `|=`	rechts nach links