universität innsbruck

# Datentypen & Operatoren

Programmiermethodik

Lukas Kaltenbrunner, Simon Priller
Universität Innsbruck

# Datentypen

- Wiederholung
  - Alle Daten haben einen Typ (Datentyp)
  - Der Typ schränkt die Benutzung der Daten ein
  - Datentyp bestimmt:
    - Wertebereich
    - Operationen, die auf den Datentyp angewandt werden dürfen
- In Java existieren zwei Arten von Datentypen:
  - Primitive Datentypen
    - Nicht veränderbar oder ergänzbar
  - Referenztypen (Datentypdefinition durch Klassen)
    - Vorgegebene Klassen
    - Benutzerdefinierte Klassen
- Java ist streng typisiert → Datentyp und entsprechende Operationen sind während Programmlaufzeit unveränderlich

# Datentypen und Deklarationen

- Primitive Datentypen in Java
  - Ganze Zahlen (`byte`, `short`, `char`, `int`, `long`)
  - Gleitkommazahlen (`float`, `double`)
  - Logische Werte (`boolean`)
- Referenztypen
  - Zeichenketten (`String`)
  - Arrays
  - …
- Für Variablen müssen Bezeichner und Datentyp angegeben werden
  - Variable mit Bezeichner `counter`, die ganze Zahlen aufnehmen kann

```
int counter;
```

# Ganze Zahlen (1)

- Wertebereich ist endlich
  - `byte` (n=1; 8 bit), `short` (n=2; 16 bit), `int` (n=4; 32 bit), `long` (n=8; 64 bit)
    - $[-2^{8n-1} .. 2^{8n-1} - 1]$ –> Zweierkomplement-Darstellung
    - Wertebereich ist asymmetrisch
  - `char` (16 bit; `'\u0000'` bis `'\uffff'` das entspricht 0 bis 65535)
- Operationen
  - Beliebige Typen: `(Typ)`,`?:`, `==`, `!=`
  - Arithmetische Typen: `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`, `++`, `--`
  - Ganzzahl Typen: `~`, `<<`, `>>`, `>>>`, `&`, `^`, `|`

# Ganze Zahlen (2)

- Shift (<<, >>, >>>)
  - `n << s`: Linksverschiebung der Bits von `n` um `s` Positionen
  - `n >> s`: Rechtsverschiebung der Bits von `n` um `s` Positionen mit Vorzeichen
  - `n >>> s`: Rechtsverschiebung der Bits von `n` um `s` Positionen ohne Vorzeichen
- Bitweises Exklusiv-ODER (^)
  - Korrespondierende Bits werden miteinander Exklusiv-ODER-verknüpft
- Division (/)
  - Division durch 0 nicht erlaubt
  - Ganzzahlige Division

```
int x = 5;              // x = 5
int y = (x / 3) * 3;    // y = 3
```

# Ganze Zahlen – Überläufe

- Java zeigt Überläufe nicht an.
- Bei einem Überlauf wird mit fehlerhaften Werten weitergerechnet.
  - Bei additiven Operationen (+, -) kann über die Grenzen hinaus und wieder zurück gerechnet werden.
  - Bei den multiplikativen Operationen * bzw. / und bei den Shift-Operationen (<<, >>, >>>) ist Vorsicht geboten!

Konstante MAX_VALUE der Klasse Integer

```
int maxInt;
int overflow;
maxInt = Integer.MAX_VALUE;   // maxInt = 2147483647

overflow = maxInt + 1;        // overflow = -2147483648 = Integer.MIN_VALUE
overflow = overflow - 1;      // overflow = 2147483647 = Integer.MAX_VALUE

overflow = maxInt * 2;        // overflow = –2
overflow = overflow / 2;      // overflow = –1
```

# Gleitkommazahlen (1)

- Gleitkommadarstellung
  - x = Mantisse * Basis$^{Exponent}$
  - Die Größe und die Genauigkeit sind endlich!
    - Nicht alle Zahlen können dargestellt werden (z.B. 0.1 → binär periodisch).
    - Vergleiche sind möglich – aber nicht immer unproblematisch!
- IEEE 754 Format:
  - `float`: 32 bit (1 bit Vorzeichen, 8 bit biased Exponent, 23 bit Mantisse)
  - `double`: 64 bit (1 bit Vorzeichen, 11 bit biased Exponent, 52 bit Mantisse)
- Operationen
  - Beliebige Typen: `(Typ)`, `?:`, `==`, `!=`
  - Arithmetische Typen: `+`, `-`, `*`, `/`, `%`, `<`, `>`, `<=`, `>=`, `++`, `--`

# Gleitkommazahlen (2)

- Gleitkommazahlen können nicht nur positive und negative Zahlen darstellen.
  - Positive und negative Null
  - Positives und negatives Unendlich
  - Not-a-Number (NaN)
- Überläufe bzw. Unterläufe führen zu einem positiven bzw. negativen Unendlich.
- NaN-Werte werden verwendet, um Ergebnisse von ungültigen Operationen zu kennzeichnen.
- Abgesehen von NaN-Werten weisen Gleitkommazahlen eine totale Ordnung auf.

# Logische Werte

- Der Wertebereich von `boolean` umfasst 2 Werte
  - `true` und `false`
- Operationen
  - Beliebige Typen: `(Typ)`, `?:`, `==`, `!=`
  - Logische Typen: `!`, `&`, `^`, `|`, `&&`, `||`
- Logische Ausdrücke bestimmen den Kontrollfluss von:
  - `if`-Anweisungen
  - `while`-Schleifen
  - `do-while`-Schleifen
  - `for`-Schleifen

# Logische Ausdrücke – Auswertung

- „*Short circuit evaluation*“ bei logischen Operatoren:
  - `x && y`: ist `x` `false`, ist das Ergebnis `false` und `y` wird nicht ausgewertet
  - `x || y`: ist `x` `true`, ist das Ergebnis `true` und `y` wird nicht ausgewertet

- Erhöht die Robustheit
  - Der Ausdruck y darf Code enthalten, der ungültig ist im Fall
    - (`x && y`) bei `x == false`
    - (`x || y`) bei `x == true`
  - Beispiel
    - `(x != 0 && y/x > 2)` – keine Division durch 0 möglich!

# Zeichenketten (String)

- Strings werden zur Darstellung von Zeichenketten verwendet
- String ist kein primitiver Datentyp!
  - Ist eine Klasse
  - Wird noch ausführlich in der Vorlesung über Objektorientierung besprochen
- Deklaration (eine Möglichkeit)

```
String stringTest = "Text";
```

- Operationen
  - Beliebige Typen: `(Typ)`, `?:`, `==`, `!=`
  - String Typen: `+`

```
String s1 = "Hello" + " " + "World!";  // "Hello World!"
String s2 = "int: " + 7;               // "int: 7"
String s3 = "double: " + 3.14;         // "double: 3.14"
```

# Ganzzahlenliterale

- Ganzzahlige Literale können im Dezimal-, Binären-, Oktal- oder Hexadezimalsystem angegeben werden.

| Stellenwertsystem | Präfix | Ziffernmenge |
|---|---|---|
| Binärsystem | `0b`, `0B` | `0`, `1` |
| Oktalsystem | `0` | `0` - `7` |
| Dezimalsystem | | `0` - `9` |
| Hexadezimalsystem | `0x`, `0X` | `0` - `9`, `a` - `f`, `A` - `F` |

- Ziffernfolgen können Unterstriche enthalten
- Ein Ganzzahlenliteral ist vom Typ `long`, sofern das Suffix `L` oder `l` verwendet wird, ansonsten ist der Typ `int`
- Beispiele:
  - `int` Literale: `0`, `12`, `0327`, `0b101011`, `0xCafe`, `1669`
  - `long` Literale: `-5L`, `2_147_483_648L`, `1669L`

# Gleitkommaliterale

- Gleitkommaliterale können im Dezimal- oder Hexadezimalsystem angegeben werden.
  - Sie bestehen aus einem Vorkommateil, einem Punkt, einem Nachkommateil, einem Exponenten und einem Suffix.
  - Suffix `f` oder `F` für float, `d` oder `D` für double (default).
  - Um ein Gleitkommaliteral von einem integralen Literal unterscheiden zu können, muss mindestens der Dezimalpunkt, der Exponent oder das Suffix vorhanden sein.
- Ziffernfolgen können Unterstriche enthalten
- Beispiele:
  - `float` Literale: `2.5f`, `2.f`, `.5f`, `0f`, `3e1f`, `5.2546e+13f`
  - `double` Literale: `2.5`, `2.`, `.5`, `0.0`, `0x278.b5p+4`, `10_123.312_5`, `1e55`, `4d`

# Zeichenliterale

- Ein Zeichenliteral umschließt mit einfachen Anführungszeichen ein Zeichen oder eine Escapesequenz.
- Ist immer vom Typ `char`.
- Ist UTF-16 kodiert.
- Beispiele:
  - `'A'`, `'a'`, `'%'`, `'!'`, `'5'`, `'ä'`
  - `'\n'`, `'\t'`, `'\\'`, `'\u0041'`, `'\u0061'`, `'\101'`, `'\u00e4'`

# String-Literale

- Ein String-Literal umschließt mit doppelten Anführungszeichen eine Folge von Zeichen und Escapesequenzen.
- Ist immer vom Typ `String`.
- Kann sich nicht über mehr als eine Code-Zeilen erstrecken.
- Ein langes String-Literal kann mithilfe der Stringkonkatenation (+) in kürzere String-Literale aufgeteilt werden.
- Beispiele:
  - `""`, `"Hello World!"`, `"Universit\u00e4t"`

# Textblöcke

- Ein Textblock umschließt innerhalb von Begrenzern eine Folge von Zeichen und Escapesequenzen.
  - Start (opening delimiter): Drei Anführungszeichen, gefolgt von beliebig vielen Whitespace-Zeichen und einem Zeilenumbruch
  - Ende (closing delimiter): Drei doppelte Anführungszeichen
- Ist immer vom Typ `String`.
- Erstreckt sich mindestens über zwei Code-Zeilen.
- Der Inhalt eines Textblocks muss nicht ganz links beginnen. Gemeinsame Einrückungen (incidental whitespaces) werden im resultierenden String nicht berücksichtigt.

```
String string_literal = "This is the first line of a string.\n" +
    "This is the second line of a string.";

String text_block = """
                    This is the first line of a string.
                    This is the second line of a string.""";
```

# null-Literal

- Das `null`-Literal repräsentiert die `null`-Referenz.
- Die `null`-Referenz kann jedem Referenztyp zugewiesen werden.
- Die `null`-Referenz kann nicht in einen primitiven Typ umgewandelt werden.
- Wird versucht über `null` auf eine Variable oder Methode zuzugreifen, tritt ein Laufzeitfehler (`NullPointerException`) auf.

# Unveränderliche Werte

- Variablen können als `final` deklariert werden. Diesen Variablen kann nur einmal ein Wert zugewiesen werden.
- Finale lokale Variablen
  - Deklaration (danach keine Zuweisung möglich!)

```
final int x = 10;
```

  - Aufgeschobene Initialisierung

```
final double y;
…
y = 20.2;
```

- Konstanten
  - Ein statisches, finales Feld, welches einen primitiven Datentyp oder einen unveränderlichen Referenztyp hat.

```
static final int MINIMUM_LENGTH = 30;
```

# Bezeichner (1)

- Für Variablen (und Konstanten), Methoden, Klassen etc. werden Bezeichner vergeben.
- Ein Bezeichner ist eine Folge von Buchstaben, Ziffern und den Symbolen _ (Unterstrich) und $ (Dollar).
- Die Buchstaben dürfen aus dem Unicode-Zeichensatz entstammen.
- Das erste Zeichen des Bezeichners darf keine Ziffer sein.
- Der Bezeichner darf kein reserviertes Schlüsselwort, Boolean-Literal oder das null-Literal sein.
- Empfehlung: *englische Bezeichner*

# Bezeichner (2)

- In Java gibt es viele etablierte Namenskonventionen.
- Diese Namenskonventionen sollten nur im Ausnahmefall mit einer entsprechenden Begründung nicht eingehalten werden.
- Felder
  - Bestehen aus einem Wort oder mehreren Wörtern.
  - Keine ungeläufigen Abkürzungen.
  - Bei Konstanten werden alle Buchstaben als Großbuchstaben geschrieben, wobei Wörter durch Unterstriche getrennt werden (SCREAMING_SNAKE_CASE).
  - Felder, die keine Konstanten sind, werden klein geschrieben, wobei der erste Buchstabe jedes Folgewort groß geschrieben wird (camelCase).
- Lokale Variablen
  - Werden in der camelCase Schreibweise geschrieben.
  - Dürfen Abkürzung enthalten und aus einzelnen Buchstaben bestehen.

# Bezeichner (3)

```
public static List<Integer> getFlaggedCells(List<Integer> l) {
    List<Integer> r = new ArrayList<Integer>();
    for (Integer x : l) {
        if (x == 4) {
            r.add(x);
        }
    }
    return r;
}
```

# Avoid Single-Letter Names

```
public static List<Integer> getFlaggedCells(List<Integer> gameBoard) {
    List<Integer> flaggedCells = new ArrayList<Integer>();
    for (Integer cell : gameBoard) {
        if (cell == 4) {
            flaggedCells.add(cell);
        }
    }
    return flaggedCells;
}
```

- Vorher:
  - Leser muss Bedeutung (Semantik) der Variablen selbst herausfinden.
  - Variablen geben keinerlei Auskunft über Inhalt.
- Nachher: lesbarer Code

# Bezeichner (4)

```
public class Bankaccount {
    private double blnc;

    public void modifycurrentBlnc(double Amnt) {
        blnc += Amnt;
    }

    public double getCurrentBlnc() {
        return blnc;
    }
}
```

# Use Java Naming Conventions

```java
public class BankAccount {
    private double balance;

    public void modifyCurrentBalance(double amnt) {
        balance += amnt;
    }

    public double getCurrentBalance() {
        return balance;
    }
}
```

- Vorher:
  - Bezeichner-Schreibweise folgt keinem gemeinsamen Schema (Groß-Kleinschreibung, etc.)
- Nachher:
  - Format der Bezeichner für Methodennamen, Variablen, etc. vereinheitlicht
  - Folgt den Java Naming Convention

# Avoid Abbreviations

```java
public class BankAccount {
    private double balance;

    public void modifyCurrentBalance(double amount) {
        balance += amount;
    }

    public double getCurrentBalance() {
        return balance;
    }
}
```

- Vorher:
  - Leser muss Bedeutung (Semantik) der Variablen selbst herausfinden.
  - Lesen dauert länger, Code schwerer verständlich.
- Nachher: lesbarer Code

# Avoid Meaningless Terms

```java
public class BankAccount {
    private double balance;

    public void modifyBalance(double amount) {
        balance += amount;
    }

    public double getBalance() {
        return balance;
    }
}
```

- Vorher:
  - Bezeichner beinhalten nutzlose Information
  - Bezeichner sind dadurch länger, aber nicht aussagekräftiger
- Nachher: lesbarer Code
  - Kürzere, aber trotzdem aussagekräftige Bezeichner

# Reservierte Schlüsselwörter

- Reservierte Schlüsselwörter dürfen nicht als Bezeichner verwendet werden:

| | | | | |
|---|---|---|---|---|
| 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 |
| _ (*) | | | | |

(*) reserviert, aber nicht benutzt

# Operatoren

- Operatoren werden mit Operanden zu Ausdrücken verknüpft.
- Komplexe nicht geklammerte Ausdrücke werden in Teilausdrücke aufgelöst durch:
  - Präzedenz oder Bindungsstärke
    - Regelt die implizite Klammerung bei Operatoren auf verschiedenen Stufen.
    - Beispiel: $1 + 2 \cdot 3 = 1 + (2 \cdot 3)$
  - Links-/Rechtsassoziativität
    - Regelt die implizite Klammerung bei Operatoren auf gleicher Stufe.
    - Linksassoziativ: $a \cdot b \cdot c = (a \cdot b) \cdot c$
    - Rechtsassoziativ: $a \cdot b \cdot c = a \cdot (b \cdot c)$

# Operatorpräzedenz

| Stärke | Präzedenzgruppen | Assoziativität |
|---|---|---|
| 14 | ++ (postfix), -- (postfix) | links |
| 13 | ++ (präfix), -- (präfix), +(unär), -(unär), ~, !, (type) | rechts |
| 12 | *, /, % | links |
| 11 | +, -, +(String-Konkatenation) | links |
| 10 | <<, >>, >>> | links |
| 9 | <, <=, >, >=, instanceof | links |
| 8 | ==, !=, == (Referenz), != (Referenz) | links |
| 7 | & (bitweises UND), & (logisches UND) | links |
| 6 | ^ (bitweises XOR), ^ (logisches XOR) | links |
| 5 | \| (bitweises ODER), \| (logisches ODER) | links |
| 4 | && (logisches UND mit Short-Circuit-Evaluation) | links |
| 3 | \|\| (logisches ODER mit Short-Circuit-Evaluation) | links |
| 2 | ? : (bedingte Auswertung) | rechts |
| 1 | =, +=, -=, *=, /=, %=, &=, \|=, ^=, <<=, >>=, >>>=, -> | rechts |

# Typkompatibilität

- Ein Ausdruck, welcher ein Ergebnis erzeugt, hat einen Typ, der zur Compile-Zeit ermittelt werden kann.
- Der Typ eines Ausdrucks muss in Umwandlungskontexten kompatibel mit dem erwarteten Typ (Zieltyp) sein.
- In Umwandlungskontexten kann durch den Compiler eine implizite Typanpassung erfolgen.
- Falls in einem Umwandlungskontext kein passender Typ ermittelt werden kann, kommt es zu einem Kompilierfehler.
- Beispiel

```
int x = 2.5;     // Fehler! Die Datentypen sind nicht kompatibel
double y = 10;   // Implizite Typumwandlung von int nach double
byte z = 3;      // Implizite Typumwandlung von int nach byte
```

# Umwandlungskontexte

- Zuweisungskontext (assignment context)
  - Zuweisung des Werts eines Ausdrucks an eine Variable
- Parameterkontext (invocation context)
  - Zuweisung des Werts eines aktuellen Parameters an den formalen Parameter
- String-Kontext (string context)
  - Stringkonkatenation, wenn ein Operand nicht den Typ `String` aufweist
- Cast-Kontext (casting context)
  - Cast-Ausdrücke
- Numerischer Kontext (numeric context)
  - Operanden von arithmetische Operatoren
  - Bedingungsoperator
  - switch-Ausdrücke
  - Arrayerzeugung und Arrayzugriff

# Typkonvertierungen (primitive Datentypen)

- Identitätskonvertierung (≈)
- Erweiternde Konvertierung (ω)
- Einschränkende Konvertierung (η)
- Erweiternde und einschränkende Konvertierung (ωη)

| von↓/nach→ | byte | short | char | int | long | float | double | boolean |
|---|---|---|---|---|---|---|---|---|
| byte | ≈ | ω | ωη | ω | ω | ω | ω | - |
| short | η | ≈ | η | ω | ω | ω | ω | - |
| char | η | η | ≈ | ω | ω | ω | ω | - |
| int | η | η | η | ≈ | ω | ω | ω | - |
| long | η | η | η | η | ≈ | ω | ω | - |
| float | η | η | η | η | η | ≈ | ω | - |
| double | η | η | η | η | η | η | ≈ | - |
| boolean | - | - | - | - | - | - | - | ≈ |

# Implizite Typanpassung (primitive Datentypen)

- Zuweisungskontext
  - Erweiternde Konvertierung
  - Einschränkende Konvertierung für konstante Ausdrücke vom Typ `byte`, `short`, `char` oder `int`
- Parameterkontext
  - Erweiternde Konvertierung
- String-Kontext
  - String-Konvertierung
- Numerischer Kontext
  - Einschränkende Konvertierung für konstante Ausdrücke beim Bedingungsoperator und bei `switch` Ausdrücken
  - Erweiternde Konvertierung

# Explizite Typanpassung (primitive Datentypen)

- Bei der expliziten Typanpassung wird mit einem Cast der Typ eines Ausdrucks explizit verändert.
- Form

```
(Zieltyp) Ausdruck
```

- Beispiel

```
int x = (int) 2.5;
```

- Mögliche Konvertierungen
  - Identitätskonvertierung
  - Erweiternde Konvertierung
  - Einschränkende Konvertierung
  - Erweiternde und einschränkende Konvertierung
- Achtung: kann zu Verlusten der Größenordnung und Präzision führen