Java Intro

PART 6: Ergänzendes Wissen

Michael Inden

Freiberuflicher Consultant und Trainer

https://github.com/Michaeli71/JAVA_INTRO

Speaker Intro

- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg

- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael.inden@hotmail.ch
Blog: https://jaxenter.de/author/minden

Kurse: Bitte sprecht mich an!

https://github.com/Michaeli71/JAVA_INTRO

Agenda

Workshop Contents

- **PART 1: Schnelleinstieg Java**
  - Erste Schritte in der JShell
  - Schnelleinstieg
    - Variablen
    - Operatoren
    - Fallunterscheidungen
    - Schleifen
    - Methoden
    - Rekursion

https://github.com/Michaeli71/JAVA_INTRO

Workshop Contents

- **PART 2: Strings**
  - Gebräuchliche String-Aktionen
  - Suchen und Ersetzen
  - Formatierte Ausgaben
  - Mehrzeilige Strings
  - Einstieg Reguläre Ausdrücke

- **PART 3: Arrays**
  - Gebräuchliche Array-Aktionen
  - Mehrdimensionale Arrays
  - Beispiel: Flood Fill

Workshop Contents

- **PART 4: Klassen & Objektorientierung**
  - Basics
  - Textuelle Ausgaben
  - Gleichheit == / `equals()`
  - Klassen ausführbar machen
  - Imports & Packages
  - Information Hiding
  - Vererbung und Overloading und Overriding
  - Die Basisklasse `Object`
  - Interfaces & Implementierungen

Workshop Contents

- **PART 5: Collections**
  - Schnelleinstieg Listen, Sets und Maps
  - Iteratoren
  - Generics
  - Basisinterfaces für Container
  - Praxisbeispiel Stack und Queue selbst gebaut
  - Sortierung – `sort()` + Comparator

Workshop Contents

- **PART 6: Ergänzendes Wissen**
  - Sichtbarkeits- und Gültigkeitsbereiche
  - Primitive Typen und Wrapper
  - `?`-Operator
  - Enums
  - `switch`
  - Vererbung und Polymorphie
  - Varianten innerer Klassen
  - Records

Workshop Contents

- **PART 7: Exception-Handling**
  - Schnelleinstieg
  - Exceptions selbst auslösen
  - Eigene Exception-Typen definieren
  - Propagation von Exceptions
  - Automatic Resource Management
  - Checked / Unchecked Exceptions

- **PART 8: Dateiverarbeitung**
  - Verzeichnisse und Dateien verwalten
  - Daten schreiben / lesen
  - CSV-Dateien einlesen

Workshop Contents

- **PART 9: Einstieg in Lambdas und Streams**
  - Syntax von Lambdas
  - Lambdas im Einsatz mit `filter()`, `map()` und `reduce()`
  - Lambdas im Einsatz mit `Collectors.groupingBy()`
  - `takeWhile()` / `dropWhile()`

- **PART 10: Datumsverarbeitung**
  - Einführung Datumsverarbeitung
  - Zeitpunkte und die Klasse `LocalDateTime`
  - Datumswerte und die Klasse `LocalDate`
  - Zeit und die Klasse `LocalTime`

PART 6:
Ergänzendes Wissen

👀 Sichtbarkeit 👀

Sichtbarkeitsbereiche – Block scope

```
jshell> int first = 1;
   ...> {
   ...>     // NOCH NICHT SICHTBAR
   ...>     // System.out.println(second);
   ...>     int second = 22;
   ...>
   ...>     System.out.println(first);
   ...>     System.out.println(second);
   ...>     {
   ...>         int third = 333;
   ...>         System.out.println("Nur im Block sichtbar: " + third);
   ...>     }
   ...>     // NICHT MEHR SICHTBAR: third cannot be resolved to a variable
   ...>     // System.out.println(third);
   ...> }
   ...> // NICHT MEHR SICHTBAR: second cannot be resolved to a variable
   ...> // System.out.println(second);
```

Sichtbarkeitsbereiche – Block scope

```
jshell> int first = 1;
   ...> {
   ...>     // NOCH NICHT SICHTBAR
   ...>     // System.out.println(second);
   ...>     int second = 22;
   ...>
   ...>     System.out.println(first);
   ...>     System.out.println(second);
   ...>     {
   ...>         int third = 333;
   ...>         System.out.println("Nur im Block sichtbar: " + third);
   ...>     }
   ...>     // NICHT MEHR SICHTBAR: third cannot be resolved to a variable
   ...>     // System.out.println(third);
   ...> }
   ...> // NICHT MEHR SICHTBAR: second cannot be resolved to a variable
   ...> // System.out.println(second);

first ==> 1
1
22
```

```
Nur im Block sichtbar: 333
```

Sichtbarkeitsbereiche – Method scope

```
jshell> int add(int value1, int value2)
   ...> {
   ...>     // Parameter nur in der Methode sichtbar
   ...>
   ...>     // lokale Variable nur in der Methode sichtbar
   ...>     int result = value1 + value2;
   ...>     return result;
   ...> }
|  created method add(int,int)
```

Primitive Typen

Primitive Typen

- Wir haben bisher einige Variablen mitsamt verschiedener Typen definiert und genutzt.
- In Java unterscheidet man zwischen den sogenannten **primitiven Datentypen** und den **Referenztypen**, die sich auf **Objekte** beziehen.
- Ein **primitiver Datentyp** bietet dagegen lediglich Zugriff auf **Werte** und besitzt **keine zusätzlichen Methoden.**
- Bekanntermaßen sind das in Java die Typen **byte**, **short**, **int**, **long**, **float** und **double** für Zahlen, **boolean** für boolesche Werte sowie der Typ **char** für einzelne Zeichen.
- Zu den **primitiven Datentypen** gibt es **korrespondierende Wrapper-Klassen**, die die primitiven Datentypen als unveränderliches Objekt darstellen und mit etwas Funktionalität »umhüllen«, z. B. das Parsen eines Werts aus einem String.

Primitive Typen

Tabelle 7-1 *Primitive Datentypen und korrespondierende Wrapper-Klassen*

Primitiver Typ	Wrapper	Bits	Wertebereich und Kommentar
`byte`	`Byte`	8	$-2^7 .. 2^7-1$ (-128 .. 127)
`short`	`Short`	16	$-2^{15} .. 2^{15}-1$ (-32.768 .. 32.767)
`int`	`Integer`	32	$-2^{31} .. 2^{31}-1$ (-2.147.483.648 .. 2.147.483.647)
`long`	`Long`	64	$-2^{63} .. 2^{63}-1$ (-9.223.372.036.854.775.808 .. 9.223.372.036.854.775.807)
`float`	`Float`	32	$\pm 1.4e-45 .. \pm 3.4e+38$
`double`	`Double`	64	$\pm 4.9e-324 .. \pm 1.8e+308$
`boolean`	`Boolean`	-	`false` und `true`
`char`	`Character`	16	0 .. $2^{16}-1$ (0 (`\u0000`) .. 65535 (`\uffff`)) Einzelne Zeichen, wie 'A' oder 'c'

Primitive Typen

```
jshell> int nr = 4711;
   ...> int second = 2;
   ...>
   ...> // Achtung: l ist sehr leicht mit der Ziffer 1 zu verwechseln
   ...> long longNumber1 = 4711l;
   ...> long longNumber2 = 4711L;
nr ==> 4711
second ==> 2
longNumber1 ==> 4711
longNumber2 ==> 4711


jshell> double pi = 3.14159;
   ...> float one_quarter = 1/4F;
   ...> double pi_2 = 3.14159D;
pi ==> 3.14159
one_quarter ==> 0.25
pi_2 ==> 3.14159
```

Besonderheiten Gleitkommazahlen

```
float sum = 0.0f;
for (int i = 0; i < 10; i++)
{
   sum += 0.1;
}

System.out.println("sum:  1 != " + sum);
System.out.println("3 * add  = " + (0.1 + 0.1 + 0.1));
System.out.println("3 * 0.1  = " + 3 * 0.1);
System.out.println("7 * 0.1  = " + 7 * 0.1);
System.out.println("compare  = " + (0.3f == 0.3d)); // Float ?= Double
```

Besonderheiten Gleitkommazahlen

```
jshell> float sum = 0.0f;
   ...> for (int i = 0; i < 10; i++)
   ...> {
   ...>    sum += 0.1;
   ...> }
   ...>
   ...> System.out.println("sum:  1 != " + sum);
   ...> System.out.println("3 * add  = " + (0.1 + 0.1 + 0.1));
   ...> System.out.println("3 * 0.1  = " + 3 * 0.1);
   ...> System.out.println("7 * 0.1  = " + 7 * 0.1);
   ...> System.out.println("compare  = " + (0.3f == 0.3d)); // Float ?= Double
sum ==> 0.0
sum:  1 != 1.0000001
3 * add  = 0.30000000000000004
3 * 0.1  = 0.30000000000000004
7 * 0.1  = 0.7000000000000001
compare  = false
```

Besonderheiten Wrapper-Klassen

```
jshell> var result = Integer.valueOf(7271)
result ==> 7271

jshell> var result2 = Integer.valueOf(7271)
result2 ==> 7271

jshell> result.getClass()
$72 ==> class java.lang.Integer

jshell> result2.getClass()
$73 ==> class java.lang.Integer

jshell> result == result2
$74 ==> false

jshell> result.equals(result2)
$75 ==> true
```

```
jshell> var seven = Integer.valueOf(7)
seven ==> 7

jshell> seven == Integer.valueOf(7)
$77 ==> true

jshell> seven.equals(Integer.valueOf(7))
$78 ==> true
```

Besonderheiten Wrapper-Klassen

```
jshell> int number = Integer.parseInt("7271")
number ==> 7271

jshell> var number2 = Integer.valueOf("7271")
number2 ==> 7271

jshell> number2.getClass()
$84 ==> class java.lang.Integer
```

Besonderheiten Wrapper-Klassen

```
jshell> Integer oldDeprecatedStyle = new Integer(7221)
|  Warning:
|  Integer(int) in java.lang.Integer has been deprecated and marked for removal
|  Integer oldDeprecatedStyle = new Integer(7221);
|                               ^---------------^
oldDeprecatedStyle ==> 7221

jshell> Integer oldSeven = new Integer(7)
|  Warning:
|  Integer(int) in java.lang.Integer has been deprecated and marked for removal
|  Integer oldSeven = new Integer(7);
|                     ^-------------^
oldSeven ==> 7
```

Wrapper-Klassen spezielle Methoden

```
jshell> Integer.toHexString(1234)
$96 ==> "4d2"

jshell> Integer.toHexString(255)
$97 ==> "ff"

jshell> Integer.toOctalString(7271)
$98 ==> "16147"

jshell> Integer.toOctalString(8)
$99 ==> "10"

jshell> Integer.toBinaryString(123)
$100 ==> "1111011"

jshell> Integer.toBinaryString(7271)
$101 ==> "1110001100111"
```

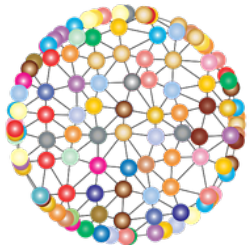
Wrapper-Klassen spezielle Methoden

```
jshell> Integer.sum(1, 2)
$102 ==> 3

jshell> Integer.sum(1, 2, 3, 4)
|  Error:
|  method sum in class java.lang.Integer cannot be applied to given types;
|    required: int,int
|    found:    int,int,int,int
|    reason: actual and formal argument lists differ in length
|  Integer.sum(1, 2, 3, 4)
|  ^---------^

jshell> Integer.compare(2, 7)
$103 ==> -1

jshell> Integer.compare(7, 2)
$104 ==> 1

jshell> Integer.compare(7, 7)
$105 ==> 0
```

Besonderheiten Auto-(Un)-Boxing

- Automatische Umwandlung von `Integer -> int -> Integer`, je nach Bedarf

```
jshell> Integer autoBoxed = 7271
autoBoxed ==> 7271

jshell> autoBoxed.getClass()
$89 ==> class java.lang.Integer

jshell> int autoUnboxed = Integer.valueOf(7271)
autoUnboxed ==> 7271

jshell> Integer.valueOf(7271).getClass()
$91 ==> class java.lang.Integer
```



Kann man auch zwischen den Typen konvertieren?

Casting (Typumwandlung)

- Ja, das nennt sich Casting
- Variablen können unterschiedliche Typen besitzen und müssen teilweise aufeinander abgebildet werden. Dabei sind verschiedene Dinge zu beachten.
- Widening: Immer problemlos möglich

```
byte ⇒ short ⇒ int ⇒ long ⇒ float ⇒ double
```

- Narrowing: Informartionsverlust möglich

```
jshell> short truncated = (short)1_000_000;
truncated ==> 16960
```

Casting (Typumwandlung)

- **Casting mit inkompatiblen Typen**

```
jshell> (int) "ABC"
|  Error:
|  incompatible types: java.lang.String cannot be converted to int
|  (int) "ABC"
|        ^---^
```

- **Explizite Typprüfung mit `instanceof`**

```
jshell> var obj = "Text-Msg"
obj ==> "Text-Msg"

jshell> if (obj instanceof String)
   ...> {
   ...>     String strMsg = (String) obj;
   ...>     System.out.println("Length: " + strMsg.length());
   ...> }
Length: 8
```

Pattern Matching bei instanceof (Java 16/17)

Pattern Matching bei instanceof

- **ALT**

```
final Object obj = new Person("Michael", "Inden");
if (obj instanceof Person)
{
    final Person person = (Person) obj;
    // ... Zugriff auf person...
}
```

Immer diese Casts…
Geht es nicht einfacher?

Pattern Matching bei instanceof

- **ALT**

```
final Object obj = new Person("Michael", "Inden");
if (obj instanceof Person)
{
    final Person person = (Person) obj;
    // ... Zugriff auf person...
}
```

- **NEU**

```
if (obj instanceof Person person)
{
    // Hier kann man auf die Variable person direkt zugreifen
}
```

Pattern Matching bei instanceof

```
Object obj2 = "Hallo Java 14";


if (obj2 instanceof String str)
{
    // Hier kann man str nutzen
    System.out.println("Länge: " + str.length());
}
else
{
    // Hier kein Zugriff auf str
    System.out.println(obj.getClass());
}
```

Pattern Matching bei instanceof

```
if (obj2 instanceof String str2 && str2.length() > 5)
{
    System.out.println("Länge: " + str2.length());
}
```

Ternary-Operator

Ternary-Operator (?-Operator)

- **Manchmal ist eine Kombination aus `if` und `else` etwas umständlich und lang.**
- **Als Abhilfe gibt es eine Kurzform, die als ternärer Operator bekannt ist. Ternär, weil sie aus drei Teilen besteht:**

```
variable = (condition) ? expressionTrue : expressionFalse;
```

```
jshell> int age = 49
age ==> 49

jshell> result = age >= 18 ? "old enough": "too young"
result ==> "old enough"
```

Beispiel Ternary-Operator (?-Operator)

```
jshell> int time = 20;
   ...> if (time < 18)
   ...> {
   ...>      System.out.println("Good day.");
   ...> }
   ...> else
   ...> {
   ...>      System.out.println("Good evening.");
   ...> }
time ==> 20
Good evening.


jshell> int time = 20;
   ...> String result = (time < 18) ? "Good day." : "Good evening.";
   ...> System.out.println(result);
time ==> 20
result ==> "Good evening."
Good evening.
```

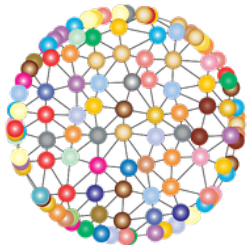
Aufzählungen und Enums

Aufzählungen

- Für diverse Anwendungsfälle gibt es einige vordefinierte, zusammengehörende Werte, etwa für Jahreszeiten (Frühling, Sommer, Herbst, Winter) oder für T-Shirt-Größen (XS, S, M, L, XL).

- Überlegen wir kurz: Wie würden wir Derartiges mit unserem bisherigen Wissen implementieren? Eine ziemlich naheliegende Idee wäre es, dafür Arrays mit fixen Werten wie folgt zu definieren:

```
jshell> String[] jahreszeiten = { "Frühling", "Sommer",
   ...>                           "Herbst", "Winter" }
jahreszeiten ==> String[4] { "Frühling", "Sommer", "Herbst", "Winter" }

jshell> String[] tshirtSizes = { "XS", "S", "M", "L", "XL" }
tshirtSizes ==> String[5] { "XS", "S", "M", "L", "XL" }
```



Was ist daran unschön?

Aufzählungen

- Es wirkt noch etwas unhandlich
- aber es birgt auch richtige Fallstricke – man kann nämlich unerwartet in dem Array ändern:

```
jshell> jahreszeiten[1] = "Zürich im Sommer"
$64 ==> "Zürich im Sommer"

jshell> jahreszeiten
jahreszeiten ==> String[4] { "Frühling", "Zürich im Sommer", "Herbst", "Winter" }


jshell> tshirtSizes[2] = "PETER"
$65 ==> "PETER"

jshell> tshirtSizes
tshirtSizes ==> String[5] { "XS", "S", "PETER", "L", "XL" }
```

Also, wie machen wir es besser?

Aufzählungen mit enum

- Aufzählungen mit dem Schlüsselwort enum erlauben, die Werte fix definieren und in Form eines Typs bereitstellen zu können.
- Die einzelnen Konstanten werden kommasepariert aufgezählt
- Konstanten per Konvention in Großbuchstaben geschrieben:

```
jshell> enum Jahreszeiten
   ...> {
   ...>     FRÜHLING, SOMMER, HERBST, WINTER
   ...> }
|  created enum Jahreszeiten
```

Aufzählungen mit `Enum`

```
jshell> enum Sizes
   ...> {
   ...>     XS, S, M, L, XL
   ...> }
|  created enum Sizes
```

```
jshell> for (Sizes size : Sizes.values())
   ...>     System.out.println(size);
XS
S
M
L
XL
```

Enum mit Attributen

```
public enum Direction
{
    N(0, -1), NE(1, -1),
    E(1, 0), SE(1, 1),
    S(0, 1), SW(-1, 1),
    W(-1, 0), NW(-1, -1);

    public final int dx;
    public final int dy;

    private Direction(final int dx, final int dy)
    {
        this.dx = dx;
        this.dy = dy;
    }
}
```

DEMO

RandomTraversal

java.util.Arrays

Sonstiges – Die Klasse Arrays

- **Erweiterungen in `java.util.Arrays`:**
  - `equals()` – Vergleicht Arrays auf Gleichheit (bezogen auf Bereiche)
  - `compare()` – Vergleicht Arrays (auch bezogen auf Bereiche)
  - `mismatch()` – Ermittelt die erste Differenz in Array (auch bezogen auf Bereiche)

Sonstiges – Die Klasse Arrays

- **`Arrays.equals()` schon lange im JDK, aber …**

- **man konnte den Vergleich früher leider nicht auf spezielle Bereiche einschränken**

```
final byte[] string1 = "ABC-DEF-GHI".getBytes();
final byte[] string2 = "DEF".getBytes();

System.out.println(Arrays.equals(string1, string2));      => false

System.out.println(Arrays.equals(string1, 4, 7,
                                 string2, 0, 3));         => true
```

Sonstiges – Die Klasse Arrays

- **Arrays.compare()**

- **Vergleicht gemäss Comparator<T> – kann man auch auf spezielle Bereiche einschränken**

```
final byte[] string1 = "ABC-DEF-GHI".getBytes();
final byte[] string2 = "DEF".getBytes();

System.out.println(Arrays.compare(string1, string2));     => ABC < DEF => -3

System.out.println(Arrays.compare(string1, 4, 7,
                                  string2, 0, 3));        => DEF = DEF => 0

System.out.println(Arrays.compare(string1, 8, 11,
                                  string2, 0, 3));        => GHI > DEF => 3
```

Sonstiges – Die Klasse Arrays

- **`Arrays.mismatch()` neu im JDK**

- **Prüft auf Abweichungen im Array – kann man auch auf spezielle Bereiche einschränken**

```
//                          0123401234567890123 4
final byte[] string1 = "ABC-DEF-GHIJKLMNOPQT".getBytes();
final byte[] string2 = "ABC-dEF-GHIJKLMNOPqT".getBytes();

System.out.println(Arrays.mismatch(string1, string2));    => 4

System.out.println(Arrays.mismatch(string1, 5, 20,
                                   string2, 5, 20));      => 13
```

Switch
Syntax-Besonderheiten

Switch-Anweisung

- **Behandlung von Alternativen mit if**

```
jshell> String language = "Java"
language ==> "Java"

jshell> if (language.equals("Java") ||
   ...>     language.equals("Python"))
   ...>     System.out.println("Cool language");
   ...> else if (language.equals("JavaScript"))
   ...>     System.out.println("Ugly language");
   ...> else
   ...>     System.out.println("Unknown language");
Cool language
```

- **Je mehr Fälle abzuprüfen sind, desto unübersichtlicher**

Switch-Anweisung

- Neben if gibt es noch switch zur Behandlung von Alternativen:

```
switch (Ausdruck) {
case x:
    // Block für Fall X
    break;
case y:
    // Block für Fall Y
    break;
default:
    // Block für alle anderen Fälle
}
```

- Abhängig von einem Ausdruck einen von vielen Blöcken auszuführen
- Die möglichen Alternativen werden durch case beschrieben.
- Der in diesem Fall auszuführende Sourcecode findet sich direkt darunter.
- Standardbehandlung mithilfe von defaul, falls kein case passt vorgeben.
- Nach case können Zahlen, Enums oder Strings als Werte genutzt werden.

Switch-Anweisung

- **Einführendes Beispiel, klarer als mit `if`**

```
jshell> String language = "Java"
language ==> "Java"

jshell> switch (language)
   ...> {
   ...>     case "Java":
   ...>     case "Python":
   ...>         System.out.println("Cool language");
   ...>         break;
   ...>     case "JavaScript":
   ...>         System.out.println("Ugly language");
   ...>         break;
   ...>     default:
   ...>         System.out.println("Unknown language");
   ...> }
Cool language
```

Switch-Anweisung mit Fallstricken

```
jshell> String language = "Java"
language ==> "Java"

jshell> switch (language)
   ...> {
   ...>     case "Java":
   ...>     case "Python":
   ...>         System.out.println("Cool language");
   ...>     case "JavaScript":
   ...>         System.out.println("Ugly language");
   ...>     default:
   ...>         System.out.println("Unknown language");
   ...> }
Cool language
Ugly language
Unknown language
```

- **Fallstrick fehlendes break / Fall Through**

Wieso?

Grüden für die Switch Anweisung

- switch-case-Konstrukt noch bei den uralten Wurzeln aus der Anfangszeit von Java
- Kompromisse im Sprachdesign, die C++-Entwicklern den Umstieg erleichtern sollten.
- Relikte wie das `break` und beim Fehlen des solchen das Fall-Through
- Flüchtigkeitsfehler kamen immer wieder vor
- Zudem war man beim `case` recht eingeschränkt bei der Angabe der Werte.
- Das alles ändert sich glücklicherweise mit modernem Java. Dazu wird die Syntax leicht verändert und erlaubt nun die Angabe einer Expression sowie mehrerer Werte beim case.

Switch: Ein Blick zurück

- **Abbildung von Wochentagen auf deren Länge …**

```
DayOfWeek day = DayOfWeek.FRIDAY;
int numOfLetters = -1;

switch (day)
{
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        numOfLetters = 6;
        break;
    case TUESDAY:
        numOfLetters = 7;
        break;
    case THURSDAY:
    case SATURDAY:
        numOfLetters = 8;
        break;
    case WEDNESDAY:
        numOfLetters = 9;
        break;
}
```

Switch Expressions als Abhilfe

- **Abbildung von Wochentagen auf deren Länge … elegant mit modernem Java:**

```
DayOfWeek day = DayOfWeek.FRIDAY;
int numLetters = -1;

switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> numLetters = 6;
    case TUESDAY                -> numLetters = 7;
    case THURSDAY, SATURDAY     -> numLetters = 8;
    case WEDNESDAY              -> numLetters = 9;
};
```

Switch Expressions als Abhilfe

- **Abbildung von Wochentagen auf deren Länge … elegant mit modernem Java:**

Return-Value

```
DayOfWeek day = DayOfWeek.FRIDAY;
int numLetters = switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

Switch Expressions als Abhilfe

- **Abbildung von Wochentagen auf deren Länge … elegant mit modernem Java**

```
DayOfWeek day = DayOfWeek.FRIDAY;
int numLetters = switch (day)
{
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    case THURSDAY, SATURDAY     -> 8;
    case WEDNESDAY              -> 9;
};
```

- **Elegantere Schreibweise beim case:**
  - Neben dem offensichtlichen Pfeil statt des Doppelpunkts
  - auch mehrere Werte
  - Keim break nötig, auch kein Fall-Through
  - switch kann nun einen Wert zurückgeben, vermeidet künstliche Hilfsvariablen

Switch Expressions: Blick zurück … Fallstricke

- **Abbildung von Monat auf deren Namen …**

```
// ACHTUNG: Mitunter ganz übler Fehler: default mitten zwischen den cases
String monthString = "";
switch (month)
{
    case JANUARY:
        monthString = "January";
        break;
    default:
        monthString = "N/A";  // hier auch noch Fall Through
    case FEBRUARY:
        monthString = "February";
        break;
    case MARCH:
        monthString = "March";
        break;
    case JULY:
        monthString = "July";
        break;
}

System.out.println("OLD: " + month + " = " + monthString); // February
```

Switch Expressions als Abhilfe

- **Abbildung von Monaten auf deren Namen … elegant mit modernem Java:**

```
static String monthToName(final Month month)
{
    return switch (month)
    {
        case JANUARY -> "January";
        default -> "N/A";  // hier KEIN Fall Through
        case FEBRUARY -> "February";
        case MARCH -> "March";
        case JULY -> "JULY";
    };
}
```

Switch Expressions: switch-old Fallstrick mit break

- **Gegeben sei eine Aufzählung von Farben:**

```
enum Color { RED, GREEN, BLUE, YELLOW, ORANGE };
```

- **Bilden wir diese auf die Anzahl an Buchstaben ab:**

```
Color color = Color.GREEN;
int numOfChars;

switch (color)
{
    case RED: numOfChars = 3; break;
    case GREEN: numOfChars = 5; /* break; UPS: FALL-THROUGH */;
    case YELLOW: numOfChars = 6; break;
    case ORANGE: numOfChars = 6; break;
    default: numOfChars = -1;
}
```

Switch Expressions: `yield` mit Rückgabe

Mit modernem Java wird wieder alles sehr klar und einfach:

```
public static void switchBreakReturnsValue(Color color)
{
    int numOfChars = switch (color)
    {
        case RED: yield 3;
        case GREEN: yield 5;
        case YELLOW, ORANGE: yield 6;
        default: yield -1;
    };

    System.out.println("color: " + color + " ==> " + numOfChars);
}
```

Switch Expressions

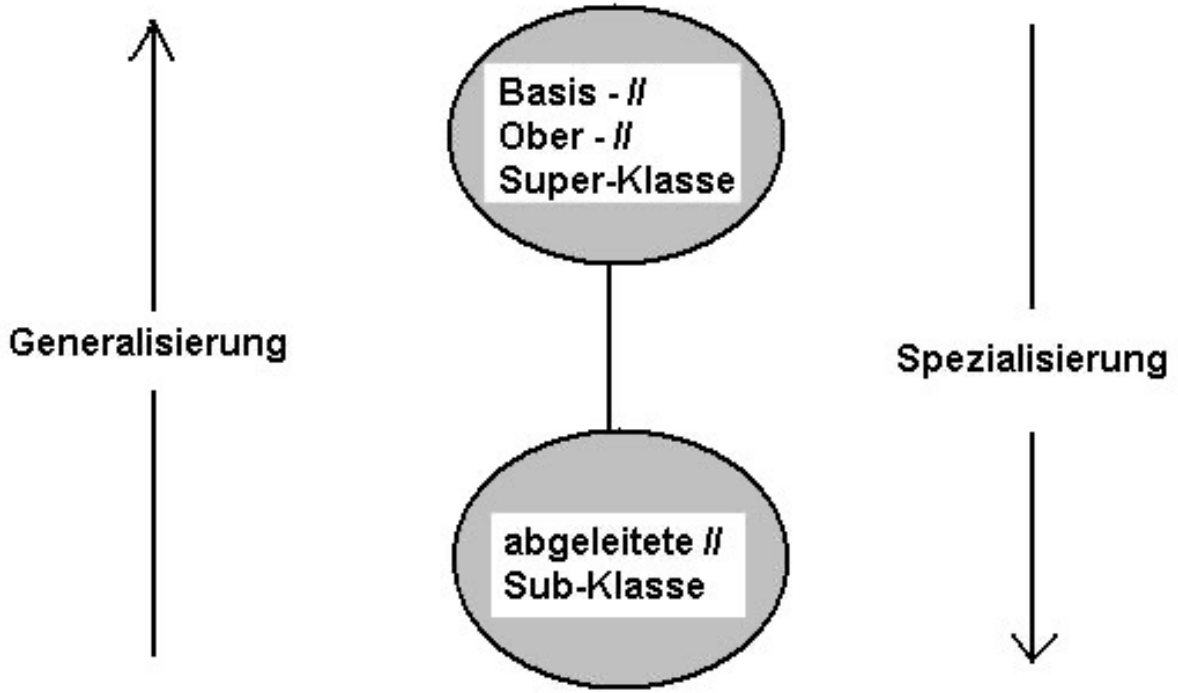
```
public static void main(final String[] args)
{
    DayOfWeek day = DayOfWeek.SUNDAY;

    int numOfLetters = switch (day)
    {
        case MONDAY, FRIDAY, SUNDAY -> {
            if (day == DayOfWeek.SUNDAY)
                System.out.println("SUNDAY is FUN DAY");
            yield 6;
        }
        case TUESDAY                -> 7;
        case THURSDAY, SATURDAY     -> 8;
        case WEDNESDAY              -> 9;
    };
    System.out.println(numOfLetters);
}
```

```
SUNDAY is FUN DAY
6
```

Vererbung, Polymorphie sowie Overloading und Overriding

Vererbung RECAP

- **Eine spezielle Art, neue Klassen basierend auf bestehenden Klassen zu definieren, nennt sich Vererbung**



- **Die neu entstehende Klasse erweitert oder übernimmt (erbt) durch diesen Vorgang das Verhalten und die Eigenschaften der Basis**

Vererbung RECAP

- Vererbungsakt wird durch das Schlüsselwort extends ausgedrückt.
- Eine Subklasse muss lediglich die Unterschiede zu ihrer Basisklasse beschreiben und nicht komplett neu entwickelt werden.

```
class BaseClass
{
    public String method()
    {
    }

    public String otherMethod()
    {
    }
}
```

```
class SubClass extends BaseClass
{
    @Override
    public String method()
    {
        super.method()
        // zusätzliches Verhalten
    }

    public String additionalMethod()
    {
        // zusätzliches Verhalten
    }
}
```

Basisklassen RECAP

- **Beim Entwurf von Klassen entstehen schnell leicht unterschiedliche Namen, etwa neben `draw()` auch die Varianten `drawLine()`, `drawRect()` usw.**

Point	Line	Rect	Circle
+ draw() : void	+ drawLine() : void	+ drawRect() : void	+ draw() : void

Das ist nicht so ganz einheitlich? Aber was könnte daran störend beim Einsatz sein?

Figuren ohne Basisklasse – Einsatz schwierig / unhandlich

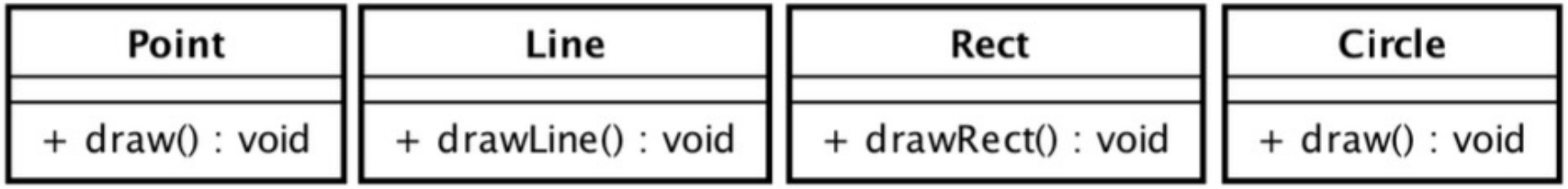
- Handhabung der Figuren für nutzende Klassen umständlich und nicht intuitiv

```
public class Drawer {

 public void drawAll(List<Object> shapes) {

   for (Object shape : shapes) {

       if (shape instanceof Circle) {
              ((Circle) shape).draw();
       }

       if (shape instanceof Rect) {
              ((Rect) shape).drawRect();
       }

       // …

   }

 }
```

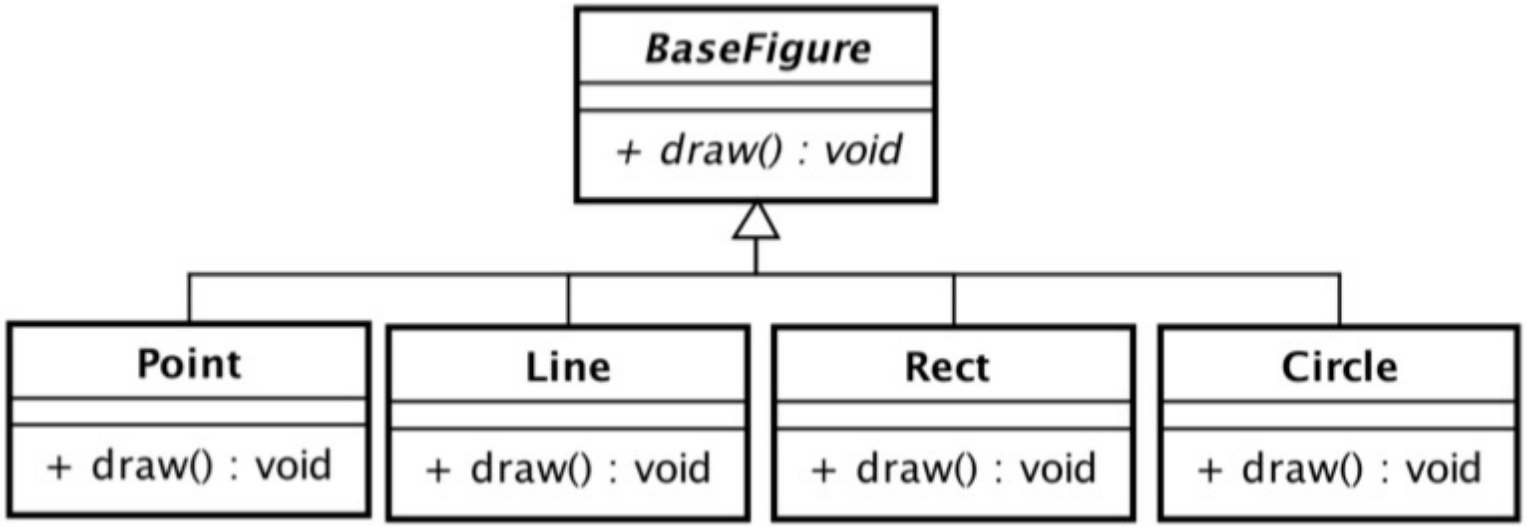
...

Basisklassen RECAP

- Die leicht unterschiedlichen Namen, etwa neben `draw()` auch die Varianten `drawLine()`, `drawRect()` usw.



- werden durch Basisklasse vereinheitlicht:



Basisklasse vereinfacht alles ….

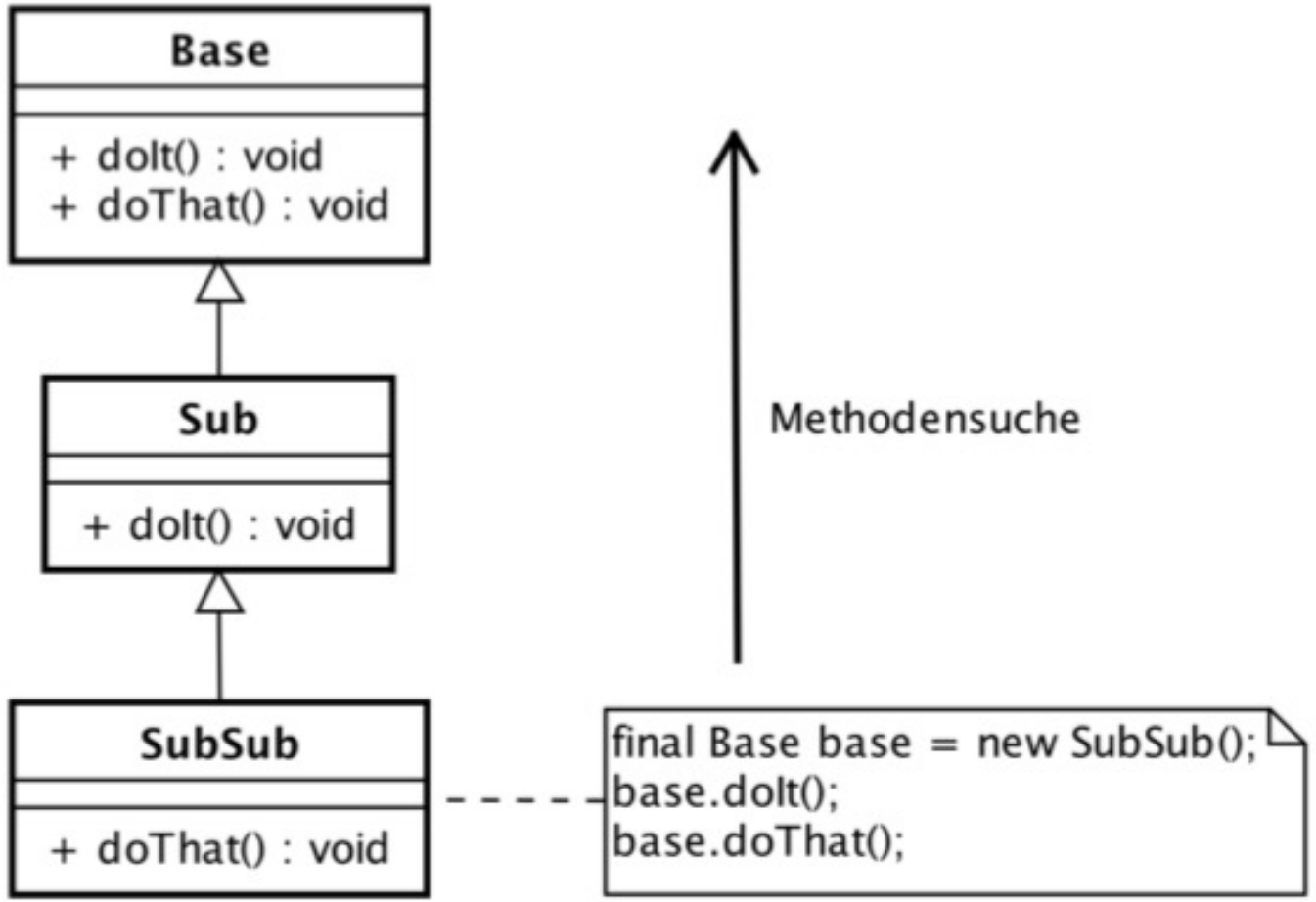
- **Basisklasse kann unterschiedliche Subklassen repräsentieren** … man spricht auch von **Vielgestaltigkeit** oder **Polymorphie**

```
public class Drawer {
 public void drawAll(List<BaseFigure> shapes) {
   for (BaseFigure shape : shapes) {
       shape.draw()
   }
 }
 ...
```

Wieso funktioniert das eigentlich?

Vererbung – Methodensuche RECAP

- „Dynamisches Binden“ => Methodensuche entlang der Typhierarchie

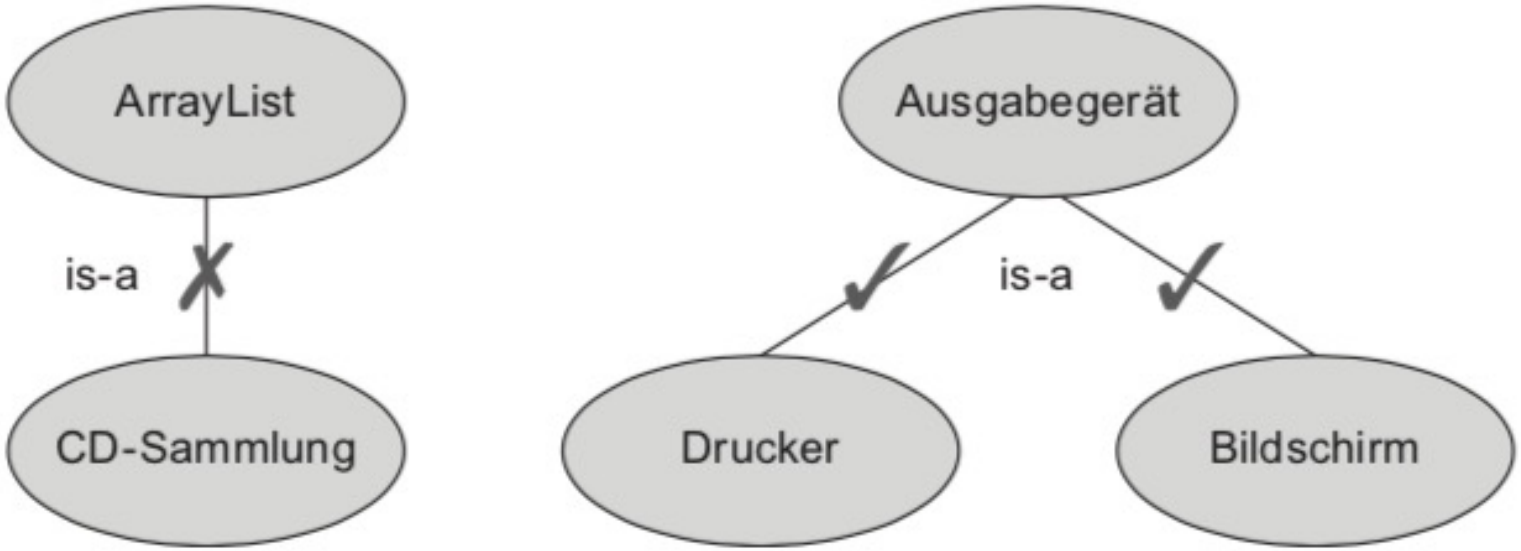


Was sollte man bei Vererbung alles beachten?

Vererbung – is-a-Beziehung RECAP

Vererbung ist ein Mittel, um bereits modelliertes Verhalten vorhandener Klassen wiederzuverwenden und zu erweitern und damit die Wartung zu erleichtern.

Vererbung nur einsetzen, wenn „is-a“-Eigenschaft erfüllt ist



Vorsicht bei der Redefinition von Verhalten!!

```
public class B {

  public String getName() {
    return "Base";
  }

}
```

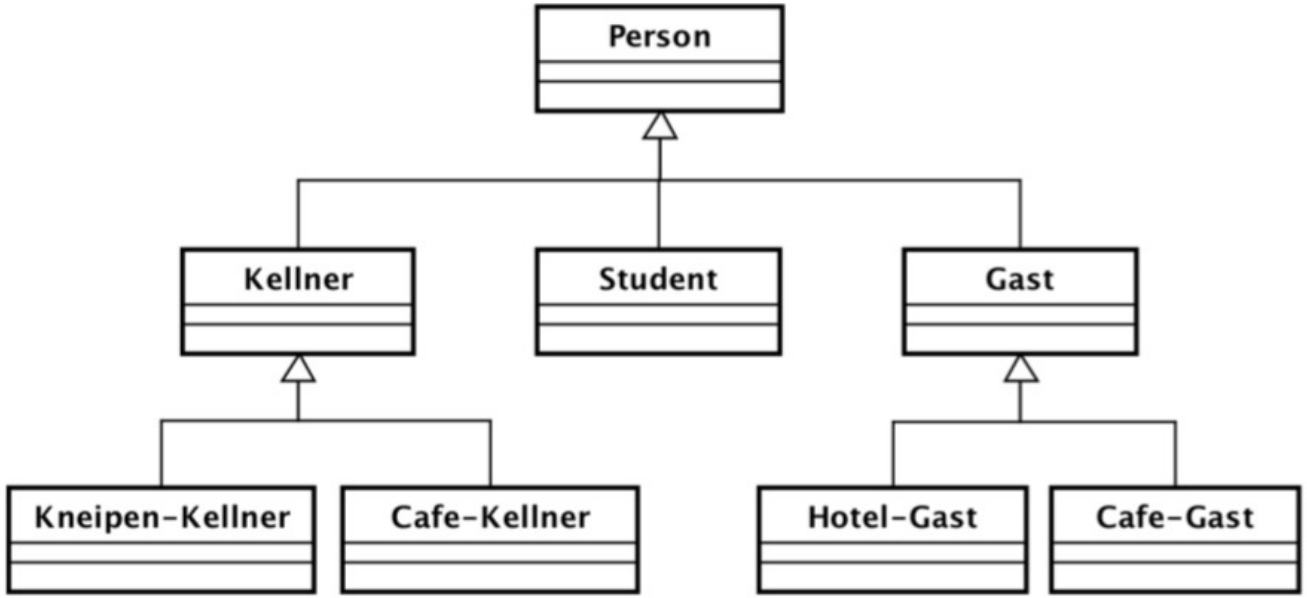
```
public class E extends B {

   public String getName() {
      throw new UnsupportedOperationException();
   }

}
```

```
public class SomeClient {

  @Override

  public String lowerName(B b) {
    return b.getName().toLowerCase();

  }

}
```

SomeClient makes an assumption about B that does not hold for E

Probleme mit Vererbung

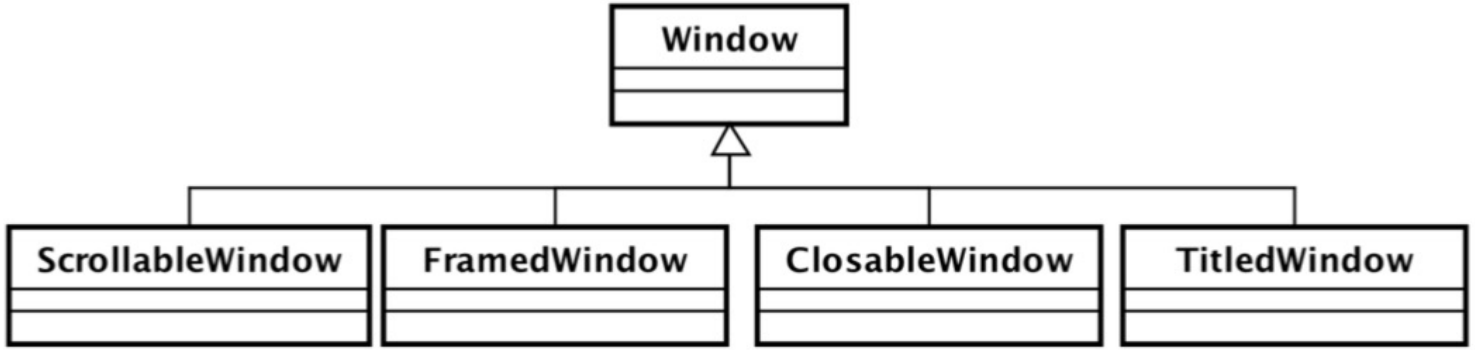
- **Selbst beim Einhalten der „is-a“-Eigenschaft kann es zu Problemen kommen:**



- **Sind alles Spezialisierungen**
- **ABER: Entsprechen mehr einer Rolle/Aufgabe, die zeitweilig ausgeübt wird**

Probleme mit Vererbung

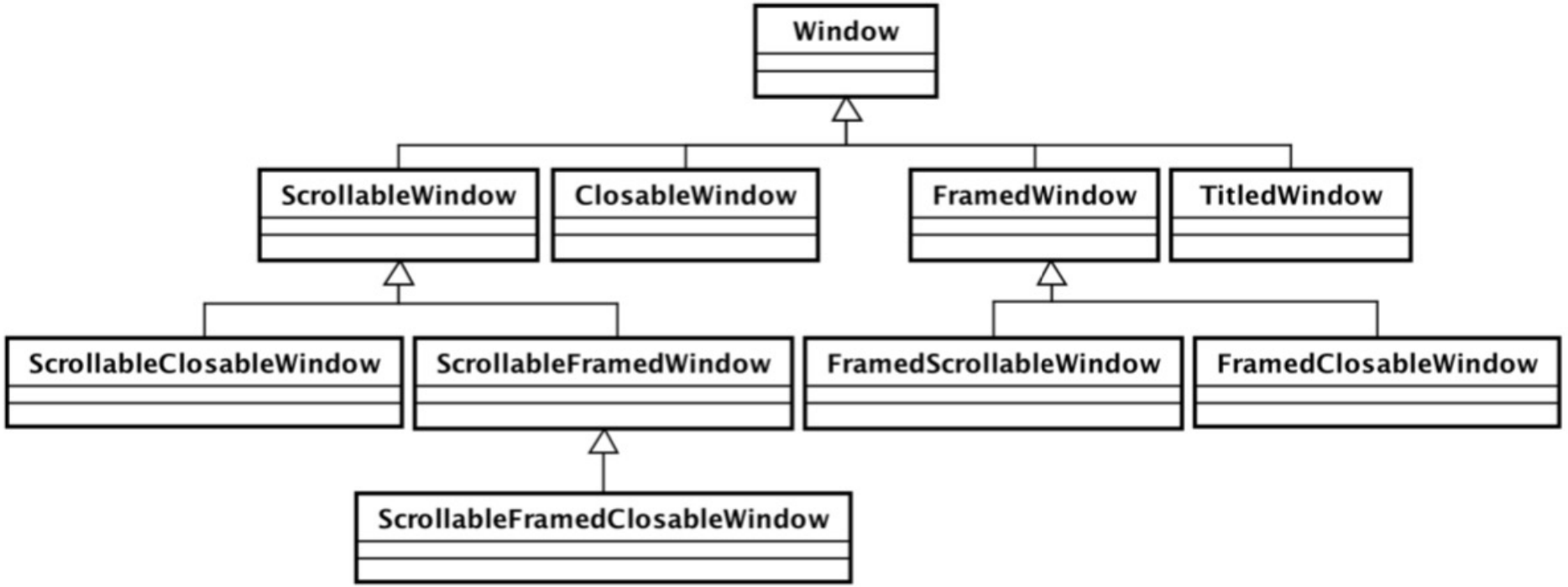
- **Eigenschaften per Vererbung hinzufügen:**



- **Sind alles Spezialisierungen**
- **ABER: Beschreiben eher eine (boolesche) Eigenschaft**
- **Problem: Man möchte die Eigenschaften kombinieren**
  => weitere Ableitungen notwendig
  => Kombinatorische Explosion

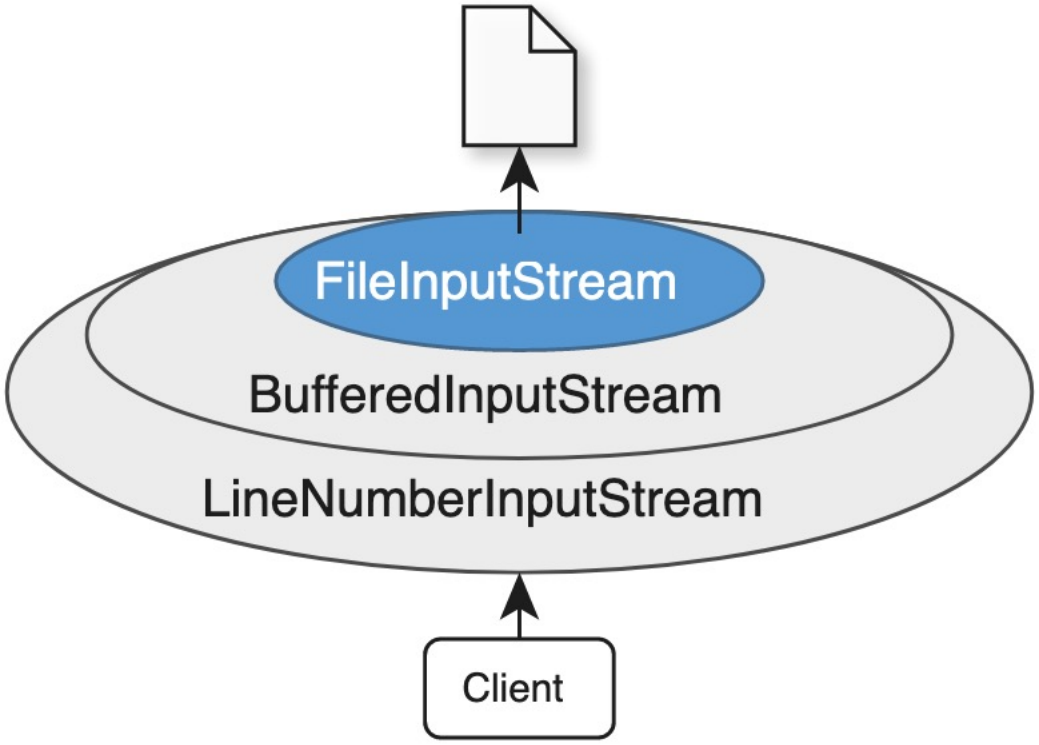
Probleme mit Vererbung

- **Kombinatorische Explosion:**



- **Gibt es Unterschiede bei verschiedenen Ableitungsreihenfolgen?**
- **Für orthogonale (voneinander unabhängige) Eigenschaften: => Decorator-Pattern: Füge Funktionalität dynamisch ohne Vererbung hinzu**

Beispiel Decortator-Pattern aus dem JDK



https://www.philipphauer.de/study/se/design-pattern/decorator.php

Varianten innerer Klassen

Innere Klassen

- In Java sind innere Klassen ein verbreitetes Sprachmittel.
- Sie sind den gewöhnlichen Klassen ähnlich, jedoch innerhalb von Klassen (oder sogar Methoden) definiert, deswegen ihr Name.
- Mithilfe innerer Klassen lassen sich einige Entwurfsprobleme, etwa die Definition von nur innerhalb einer Klasse sichtbaren Hilfsklassen, elegant lösen.
- Nachfolgend sind zwei Varianten gezeigt: normale und statische innere Klassen.

```
public class OuterClass
{
    public class InnerClass
    {
        // ...
    }
    public static class StaticInnerClass
    {
        // ...
    }
}
```

«Normale» Innere Klassen

- Innere Klassen besitzen eine implizite Referenz auf eine Instanz der äußeren Klasse und können dadurch auf deren Elemente zugreifen – sogar auf die privaten.
- Allerdings folgt daraus auch, dass immer ein Objekt der umgebenden Klasse existieren muss, um ein Objekt einer inneren Klasse zu erzeugen. Dadurch ergibt sich eine etwas merkwürdige Syntax zur Objekterzeugung:

```
jshell> // Variante 1
   ...> final OuterClass.InnerClass inner = new OuterClass().new InnerClass();
   ...>
   ...> // Variante 2
   ...> final OuterClass outer = new OuterClass();
   ...> final OuterClass.InnerClass inner2 = outer.new InnerClass();
inner ==> OuterClass$InnerClass@b81eda8
outer ==> OuterClass@27fa135a
inner2 ==> OuterClass$InnerClass@421faab1
```

Statische innere Klassen

- Wenn äußere Klassen lediglich erzeugt werden, um Zugriff auf innere Klassen zu bieten, so ist dies eher unnatürlich.
- Das gilt insbesondere dann, wenn es sich bei den inneren Klassen nur um Hilfsklassen oder Datencontainer handelt.
- Oftmals dienen sie als semantische Strukturierung und sind in der Regel unabhängig von einer Instanz der äußeren Klasse.
- Dafür existiert das Sprachmittel der statischen inneren Klassen.

Statische innere Klassen

- Die Klasse `TripleVO` realisiert einen Datencontainer, der die drei Attribute anbietet:

```
public final class StaticInnerClassExample
{
    public static final class TripleVO
    {
        public final int value1;
        public final int value2;
        public final int value3;

        private TripleVO(final int value1, final int value2, final int value3)
        {
            this.value1 = value1;
            this.value2 = value2;
            this.value3 = value3;
        }
    }

    public TripleVO calculateTriple()
    {
        return new TripleVO(7, 2, 71);
    }
}
```

Methoden lokale Klassen

Innere Klassen können sogar lokal innerhalb von Methoden definiert werden und sind auch nur dort sichtbar und zugreifbar, weshalb sich keine Sichtbarkeit angeben lässt.

```
private void doSomething()
{
    int counter = 100;
    final int constant = 200;
    // Keine Sichtbarkeitsmodifier erlaubt,
    /* public */ class MethodLocalInnerClass
    {
        public void printVar()
        {
            // System.out.println("counter = " + counter ); // => Compile-Error
            System.out.println("constant = " + constant);
        }
    }
    counter++; // Ohne diese Zeile wäre der obige Zugriff auf counter erlaubt
    new MethodLocalInnerClass().printVar();
}
```

Lokale innere Klassen können sowohl auf Attribute der äußeren Klasse als auch auf die in der Methode definierten Variablen und Parameter zugreifen, sofern sich diese nicht mehr ändern, also explizit final oder »effectively final« sind.

Anonyme innere Klassen

- Manchmal sind innere Klassen so speziell und nur für eine einmalige Aufgabe verwendbar, dass man sie weder benennen noch mehrfach erzeugen möchte.
- Das lässt sich mithilfe anonymer innerer Klassen realisieren.
- Diese sind innerhalb von Klassen oder Methoden definiert und besitzen keinen Namen, sind also anonym.
- Ohne Klassennamen können sie allerdings auch keinen Konstruktor bereitstellen.
- Meistens bestehen diese Klassen nur aus wenigen Methoden – häufig sogar lediglich aus einer. Dann spricht man von SAM-Typen (Single Abstract Method).
- Anonyme innere Klassen sind in ihrer Definition dahingehend eingeschränkt, dass sie entweder auf einem Interface basieren oder eine Klasse erweitern.

Anonyme innere Klassen

- Die Schreibweise ist zunächst etwas gewöhnungsbedürftig, da weder das Schlüsselwort `extends` noch `implements` angegeben wird. Stattdessen folgt die Klassendefinition syntaktisch direkt der Instanziierung der Basisklasse bzw. des Interface:

```
final Runnable newRunnable = new Runnable() {
    @Override
    public void run() {
        // ...
    }
}; // ACHTUNG DAS SEMIKOLON IST WICHTIG => SONST COMPILE-ERROR
```

- Der ausschließliche Sinn besteht darin, die Methoden der Basisklasse zu überschreiben bzw. die Methoden eines Interface zu implementieren. Zwar kann man in einer anonymen inneren Klasse zusätzliche Methoden definieren, allerdings können diese niemals von außerhalb der Klasse aufgerufen werden.

Anonyme innere Klassen

```
Comparator<String> byLength = new Comparator<>() {
    @Override
    public int compare(String str1, String str2) {
        return Integer.compare(str1.length(), str2.length());
    }
};
```

- **Derartige SAM-Typen lassen sich kürzer und eleganter als Lambda schreiben**

```
Comparator<String> byLength = (str1, str2) -> Integer.compare(str1.length(),
                                                              str2.length());
```

Records

Wäre es nicht cool, auf einfache Weise Datenbehälterklassen definieren zu können?

Erweiterung Record

```
record MyPoint(int x, int y) { }
```

- simplifizierte Form von Klassen für einfache Datencontainer
- Sehr kurze, kompakte Schreibweise
- API ergibt sich implizit aus den als Konstruktorparameter definierten Attributen

```
MyPoint point = new MyPoint(47, 11);
System.out.println("point x:" + point.x());
System.out.println("point y:" + point.y());
```

- Implementierungen von Accessor-Methoden sowie equals() und hashCode() automatisch und vor allem kontraktkonform

Erweiterung Record

```
record MyPoint(int x, int y) { }
```

```
Michaels-iMac:java14 michaeli$ javap MyPoint
Compiled from "MyPoint.java"
final class java14.MyPoint extends java.lang.Record {
  public java14.MyPoint(int, int);
  public int x();
  public int y();
  public java.lang.String toString();
  public int hashCode();
  public boolean equals(java.lang.Object);
}
```

```
public final class MyPoint
{
    private final int x;
    private final int y;

    public MyPoint(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o)
    {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;

        MyPoint point = (MyPoint) o;
        return x == point.x && y == point.y;
    }

    @Override
    public int hashCode()
    {
        return Objects.hash(x, y);
    }

    @Override public String toString()
    {
        return "MyPoint[x=" + x + ", y=" + y + "]";
    }

    // Zugriffsmethoden auf x und y
}
```

Records und zusätzliche Konstruktoren und Methoden

```
record MyPoint(int x, int y)
{
    public MyPoint(String values)
    {
        this(Integer.parseInt(values.split(",")[0]),
             Integer.parseInt(values.split(",")[1]));
    }

    public String shortForm()
    {
        return "[" + x + ", " + y + "]";
    }
}
```

```
var topLeft = new MyPoint(17, 19);
System.out.println(topLeft);
System.out.println(topLeft.shortForm());
```

```
MyPoint[x=10, y=10]
[10, 10]
```

Records für Data Transfer / Parameter Value Objects

```
record TopLeftWidthAndHeight(MyPoint topLeft, int width, int height) { }

record ColorAndRgbDTO(String name, int red, int green, int blue) { }

record PersonDTO(String firstname, String lastname, LocalDate birthday) { }



record Rectangle(int x, int y, int width, int height)
{
        Rectangle(TopLeftWidthAndHeight pointAndDimension)
        {
                this(pointAndDimension.topLeft().x, pointAndDimension.topLeft().y,
                     pointAndDimension.width, pointAndDimension.height);
        }
}
```

Records für komplexere Rückgabewerte und Parameter

```
record IntStringReturnValue(int code, String info) { }
record IntListReturnValue(int code, List<String> values) { }

record ReturnTuple(String first, String last, int amount) { }
record CompoundKey(String name, int age) { }


IntStringReturnValue calculateTheAnswer()
{
    // Some complex stuff here
    return new IntStringReturnValue(42, "the answer");
}

IntListReturnValue calculate(CompoundKey inputKey)
{
    // Some complex stuff here
    return new IntListReturnValue(201,
                                  List.of("This", "is", "a", "complex", "result"));
}
```

Records für Pairs und Tupel

```
record IntIntPair(int first, int second) {};

record StringIntPair(String name, int age) {};

record Pair<T1, T2>(T1 first, T2 second) {};

record Top3Favorites(String top1, String top2, String top3) {};

record CalcResultTuple(int min, int max, double avg, int count) {};
```

- **Exterm wenig Schreibaufwand**
- **Sehr praktisch für Pair, Tuples usw.**
- **Records funktionieren prima mit primitiven Typen und auch mit Generics**
- **Implementierungen von Accessor-Methoden sowie equals() und hashCode() automatisch und vor allem kontraktkonform**

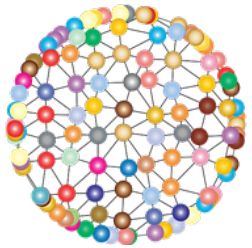
Ist ja cool … ABER: Wie kann ich denn Gültigkeitsprüfungen integrieren?

Records mit Gültigkeitsprüfung

```
record ClosedInterval(int lower, int upper)
{
    public ClosedInterval(int lower, int upper)
    {
        if (lower > upper)
        {
            String errorMsg = String.format("invalid: %d (lower) >= %d (upper)",
                                            lower, upper);
            throw new IllegalArgumentException(errorMsg);
        }


        this.lower = lower;
        this.upper = upper;
    }
}
```

Records mit Gültigkeitsprüfung (Kurzschreibweise)

```
record ClosedInterval(int lower, int upper)
{
    public ClosedInterval
    {
        if (lower > upper)
        {
            String errorMsg = String.format("invalid: %d (lower) >= %d (upper)",
                                            lower, upper);
            throw new IllegalArgumentException(errorMsg);
        }
    }
}
```



Letzte Frage für Records: Ist das alles kombinierbar?

All in Beispiel

```
record MultiTypes<K, V, T>(Map<K, V> mapping, T info)
{
    public void printTypes()
    {
        System.out.println("mapping type: " + mapping.getClass());
        System.out.println("info type: " + info.getClass());
    }
}


record ListRestrictions<T>(List<T> values, int maxSize)
{
    public ListRestrictions
    {
        if (values.size() > maxSize)
            throw new IllegalArgumentException(
                "too many entries! got: " + values.size() +
                ", but restricted to " + maxSize);
    }
}
```

Exercises Part 6

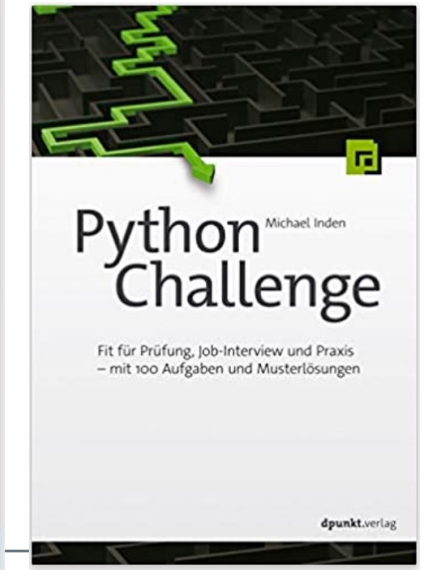
https://github.com/Michaeli71/JAVA_INTRO


HELP
IS ON THE
WAY


Michael Inden
Einfach Java
Gleich richtig programmieren lernen
dpunkt.verlag


5. Auflage
Michael Inden
Der Weg zum Java-Profi
Konzepte und Techniken für die professionelle Java-Entwicklung
dpunkt.verlag


Michael Inden
Java Challenge
Fit für das Job-Interview und die Praxis – mit mehr als 100 Aufgaben und Musterlösungen
dpunkt.verlag


Michael Inden
Python Challenge
Fit für Prüfung, Job-Interview und Praxis – mit 100 Aufgaben und Musterlösungen
dpunkt.verlag

Thank You