VO Einführung in die Programmierung
Teil II

WS 2017/18

H.Hagenauer
FB Computerwissenschaften

UNIVERSITÄT SALZBURG

Kapitel 8

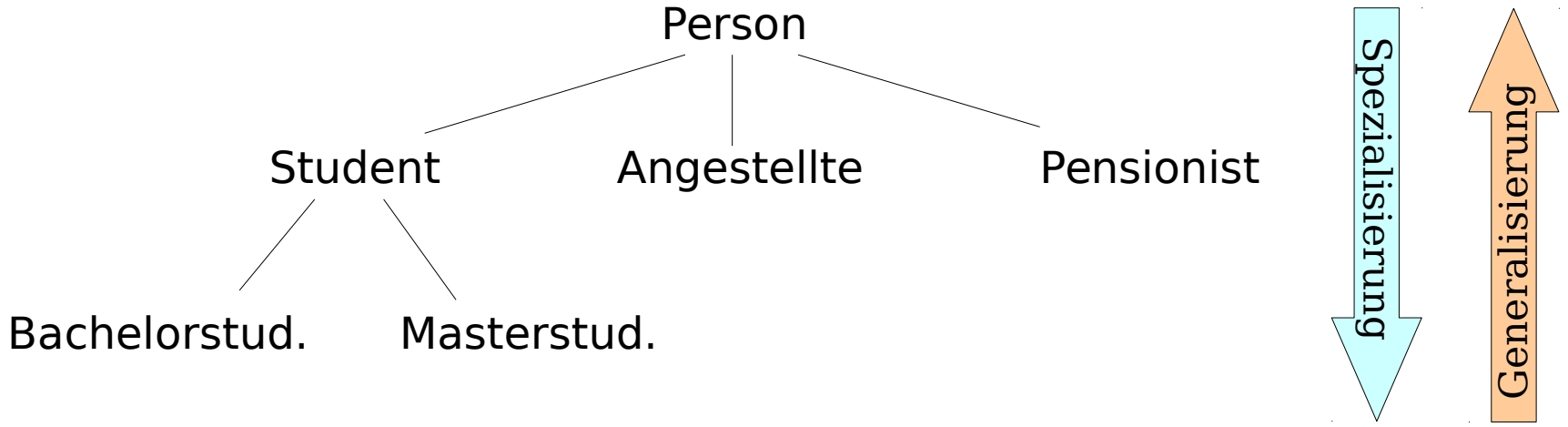
Vererbung, Polymorphismus

UNIVERSITÄT SALZBURG

Oberklasse, Unterklasse

Oft werden verschiedene Dinge (der realen Welt) *klassifiziert*!

Z.B. können Personen eingeteilt werden in: Kinder, Studenten, Angestellte, Selbstständige, Pensionisten. D.h. es wird eine hierarchische Klassifizierung vorgenommen – es gibt Ober-/Unterbegriffe, es erfolgt eine Spezialisierung oder Generalisierung.



Ein Student ist eine Person – aber: nicht jede Person ist ein Student!

UNIVERSITÄT SALZBURG

Oberklasse, Unterklasse (Forts.)

Dieses Konzept wird auch auf die Programmierung übertragen – genauer auf die Definition und den Gebrauch von Datentypen:

→ zu bestehenden Klassen werden Unterklassen gebildet.

Eine **Unterklasse (subclass, child class)**

- *erbt (inherits)* Eigenschaften der **Oberklasse (base class, superclass)**,
- kann weitere Eigenschaften (das sind Instanzvariable, Methoden) hinzufügen (d.h. es wird erweitert)
- und vorhandene Eigenschaften (i.A. Methoden) *überschreiben (overriding)* – d.h. semantisch anpassen.

Dieses Konzept wird **Vererbung (inheritance)** genannt.

UNIVERSITÄT SALZBURG

Oberklasse, Unterklasse (Forts.)

Zweck der Unterklassenbildung

- es können Varianten ohne kopieren von Programmtext erstellt werden – was geerbt wird, muss nicht nochmals hingeschrieben werden

  → *Wiederverwendung von Code (code reuse)*
- Gleichbehandlung „ähnlicher“ Objekte durch rufende Programmteile

  → *Polymorphismus* (siehe später)

Syntax für Unterklassenbildung:

res. Wort

allgemeine Form

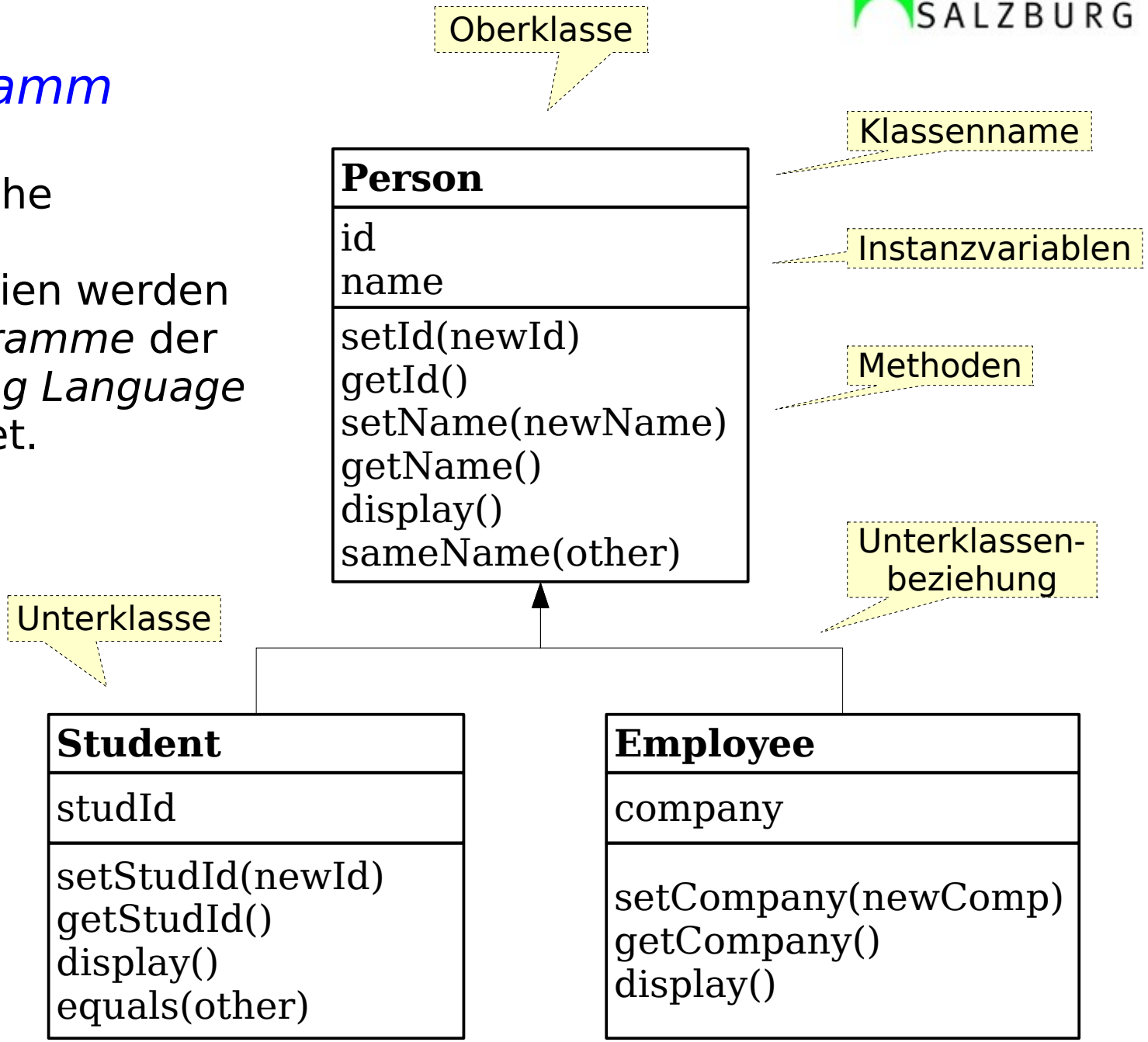
```
class subclass extends superclass {
   body
}
```

der Rumpf (body) der Unterklasse enthält nur die neuen oder überschriebenen Deklarationen und Definitionen!

UNIVERSITÄT SALZBURG

Klassendiagramm

Für die graphische Darstellung von Klassenhierarchien werden oft *Klassendiagramme* der *Unified Modelling Language (UML)* verwendet.



(siehe Klassen `Person, Student, Employee` und Prog. `PersonDemo`)

UNIVERSITÄT SALZBURG

Klassendiagramm (Forts.)

- `Student` erbt von `Person` die Instanzvariablen `id` und `name`, sowie alle Methoden
- `Student` erweitert `Person`: Instanzvariable `studId` und entsprechende Methoden werden hinzugefügt
- `Student` überschreibt Methoden von `Person`: `display` gibt zusätzlich den Wert von `studId` aus

Achtung: direkter Zugriff auf `private`-Instanzvariablen der Oberklasse ist nicht möglich – entsprechende getter- und setter-Methoden sind nötig!

Anmerkung: überschreiben von Methoden oder Unterklassen-bildung allgemein kann mittels dem Modifier `final` verhindert werden (siehe Literatur).

Anmerkung: mehrstufige Hierarchien können durch wiederholte Unterklassenbildung erstellt werden.

UNIVERSITÄT SALZBURG

Konstruktoren und Methoden der Oberklasse

Mittels `super` (und passender Parameterliste) kann auf den entsprechenden Konstruktor der Oberklasse zugegriffen werden.

Dies muss die erste Anweisung im Konstruktor der Unterklasse sein!

```
public Student() {
  super();
  studId = 0;
}
```

Für den Zugriff auf überschriebene Methoden in der Unterklasse wird ebenfalls das reservierte Wort `super` verwendet

```
public void display() {
  super.display();   //display von Person!
  ...println("..." + studId);
}
```

UNIVERSITÄT SALZBURG

Allgemeine Basisklasse Object

In Java ist jede Klasse automatisch ein „Abkömmling“ der vordefinierten Klasse `Object`:

- wird keine Oberklasse angegeben, dann ist `Object` die direkte Oberklasse
- sie enthält Methoden, die von allen Klassen automatisch geerbt werden (z.B. `equals, toString`, ...)
- diese Methoden sind i.A. ohne Änderung nicht nützlich – sie müssen bei Bedarf überschrieben werden

  z.B. `public boolean equals (Object o)`
  liefert true, wenn es sich um das selbe Objekt handelt
  - für Klasse Student überschrieben

```
public boolean equals(Student other){
  return (this.getId() == other.getId()) &&
         (this.studId == other.studId);
}
```

UNIVERSITÄT SALZBURG

Polymorphismus

Polymorphismus (polymorphism) bedeutet „Vielgestaltigkeit“.

Dies heißt in der Programmierung: mehrere Typen sind im selben Kontext erlaubt, jedoch mit möglicherweise unterschiedlicher Wirkung!

In *Java* beinhaltet *Polymorphismus* 2 Bedeutungen:

1. Typfamilien

Sei A eine Klasse und B eine Unterklasse von A – dann gilt:
wo ein A-Objekt erwartet wird (Zuweisung, Parameter), darf ein B-Objekt eingesetzt werden.

Da das B-Objekt alle Eigenschaften des A-Objekts besitzt, ist dies sinnvoll.

```
Person p;
Student s = new Student(...);
p = s;
...println(p.sameName(s));
```

UNIVERSITÄT SALZBURG

Polymorphismus (Forts.)

Die umgekehrte Richtung ist problematisch, da nicht sicher ist, ob die Referenz auf ein Unterklassenobjekt zeigt!

In Java lässt sich mit Hilfe des instanceof-Operators der Typ abfragen:

```
Person p;
Student s;
s = p;  //compile-time error!!
s = (Student)p;  //ok - aber Laufzeitfehler wenn p
  //keine Referenz auf Objekt vom Typ Student ist
```

```
if (p instanceof Student)
  s = (Student)p;
else
  ...
```

UNIVERSITÄT SALZBURG

Polymorphismus (Forts.)

2. Methodenaufruf und dynamische Bindung

Sei wieder B eine Unterklasse von A, und die Methode m aus A wird in B überschrieben – dann gilt (wenn r Variable vom Typ A ist):

beim Aufruf der Methode mittels r.m(...) entscheidet der Typ des von r referenzierten Objekts welche Version der Methode ausgeführt wird (und nicht der Typ von r).

```
Person p = new Person(...);
Student s = new Student(...);
p.display();  //klar: display von Person
s.display();  //klar: display von Student
p = s;
p.display();  //display von Student
```

Damit werden alle Objekte einer Typfamilie gleichartig behandelt und es sind keine expliziten Fallunterscheidungen nötig!

Es wird automatisch die richtige Version der Methode ausgewählt, auch wenn später weitere Unterklassen von Person hinzukommen.

UNIVERSITÄT SALZBURG

Polymorphismus (Forts.)

Der spezifische Typ des referenzierten Objekts ist erst zur Laufzeit bekannt (z.B. abhängig von Eingaben, Berechnungen, …).

Damit kann die entsprechende Ausprägung der Methode auch erst zur Laufzeit ermittelt und mit dem Aufruf verbunden werden

→ **dynamische Bindung (dynamic binding, late binding)**

Im Gegensatz dazu: *statische Bindung (static binding, early binding)* – zuständig dafür ist der Compiler.

UNIVERSITÄT SALZBURG

Kapitel 9

Rekursion

UNIVERSITÄT SALZBURG

Direkte und indirekte Rekursion

Eine Methode kann sich auch *selbst* aufrufen – **rekursiver Aufruf (recursive call, recursive invocation).**

Es gibt 2 Arten der Rekursion (`m()`, `n()` sind Methoden):

direkte Rekursion: `m()` ruft auf `m()`

indirekte Rekursion: `m()` ruft auf `n()` ruft auf `m()`

Beispiel: Berechnung der Fakultät von n (n natürliche Zahl) – n!

Es gilt: n! = 1 * 2 * 3 * … * (n-1) * n

rekursiv definiert: n! = (n-1)! * n für n>1

1! = 1

UNIVERSITÄT SALZBURG

Rekursiver Aufruf, Basisfall

Dies kann mit Java „direkt“ in eine Methode umgesetzt werden:

Basisfall (nicht rekursiv)

rekursiver Aufruf

```
public static long fact(long n){
  if (n == 1)
    return 1;
  else
    return fact(n-1) * n;
}
```

Damit eine Rekursion terminiert, muss dieses allgemeine Muster vorhanden sein:

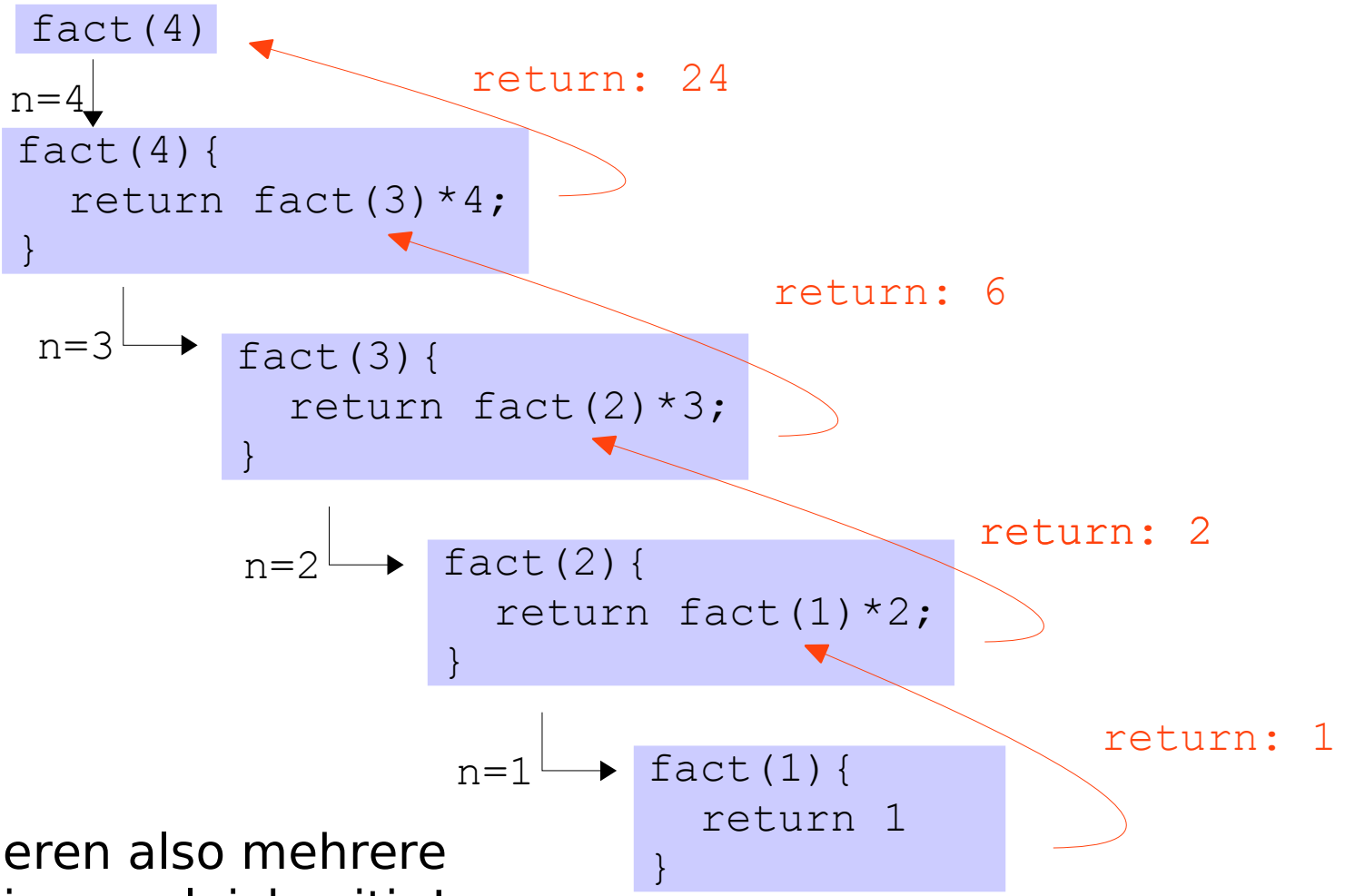
- rekursiver Aufruf, welcher ein „kleineres“ Problem löst
- Basisfall: nicht rekursive Lösung für ein Problem, dass „klein genug“ ist

Achtung: wird der Basisfall nicht erreicht, entsteht eine unendliche Rekursion – diese endet i.A. mit „Stack overflow“ (Speicherüberlauf)!

UNIVERSITÄT SALZBURG

Umsetzung eines rekursiven Aufrufs

Jeder rekursiver Aufruf einer Methode erzeugt eine neue „Inkarnation“ davon (mit eigenen Parametern).



Es existieren also mehrere Inkarnationen gleichzeitig!

UNIVERSITÄT SALZBURG

Bemerkungen zu Rekursion

- Rekursion ist ein Konzept für Wiederholungen (vergleichbar mit Schleifen)
  - Programme mit Schleifen (ohne Rekursion) werden **iterativ** genannt
- Rekursion in der Programmierung
  - Lösung oft einfacher bzw. leichter zu finden als iterative Variante (insbesonders wenn Problem rekursiv beschrieben)
  - rekursive Lösung oft weniger effizient (jeweils neuen Methodenaufruf erzeugen!)
- jede rekursive Lösung (bzw. Algorithmus) kann auch iterativ umgesetzt werden

Insbesonders mathematische Probleme lassen sich oft direkt in Java-Code umsetzten – z.B. Fibonaccizahlen.

Rekursion ist besonders bei rekursiven Datenstrukturen (z.B. Bäume) nützlich (siehe LV Algorithmen und Datenstrukturen)!

UNIVERSITÄT SALZBURG

Beispiel: Türme von Hanoi

Ein klassisches Beispiel zur Rekursion sind die *Türme von Hanoi*.

Ausgangslage: 3 Stangen (Türme), wobei auf der ersten sich n Scheiben (mit Loch in der Mitte) mit abnehmenden Durchmesser befinden.

Aufgabe: die Scheiben sind auf die dritte Stange zu versetzen unter Beachtung folgender Regeln

- Scheiben dürfen nur einzeln bewegt werden
- für die Ablage darf nur eine der 3 Stangen benutzt werden
- von je 2 übereinander liegender Scheiben muss die untere einen größeren Durchmesser haben als die obere

UNIVERSITÄT SALZBURG

Beispiel: Türme von Hanoi (Forts.)

Lösungsidee rekursiv: es wird angenommen, die Lösung für n-1 Scheiben ist bekannt – dann

- löse die Aufgabe für n-1 Scheiben mit Ziel zweite Stange
- versetze größte Scheibe auf dritte Stange
- löse die Aufgabe für n-1 Scheiben, wobei der Start die zweite Stange und das Ziel die dritte Stange ist

Einfache Lösung in Java (ohne graphische Darstellung):

- `void solveIt (int n, int source, int target, int helper)`

  `n` : Anzahl der zu versetzenden Scheiben
  `source`: Ausgangsstange
  `target`: Zielstange
  `helper`: Ablagestange (für Zwischenablage)

damit direkte Umsetzung der oben beschriebenen Lösungsidee, mittels nötiger Methode `move`, die einen Zug (= Versetzung der obersten Scheibe von Stange i nach Stange j) darstellt

UNIVERSITÄT SALZBURG

Beispiel: Türme von Hanoi (Forts.)

- Methode `move (int from, int to)` beschreibt die Versetzung einer Scheibe
  - hier einfache Form: Methode gibt an von welchem zu welchem Stapel wird eine Scheibe versetzt, in der Form „i → j“ (versetze oberste Scheibe von Stange i nach Stange j)
- Schnittstellenmethode solveIt() zum Aufruf von außen
- eigene Klasse `TowersOfHanoi`, davon Instanzen mit der Problemgröße erzeugbar

Anmerkung

Anzahl der Züge der optimalen Lösung für n Scheiben: $2^n - 1$

UNIVERSITÄT SALZBURG

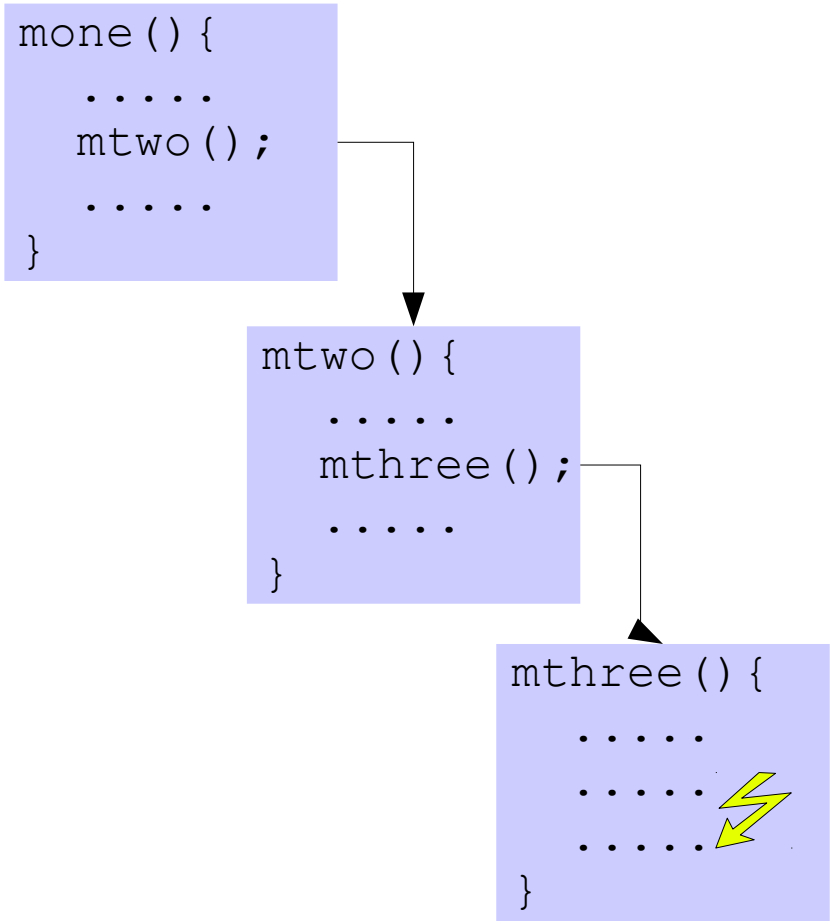
Kapitel 10

Ausnahmen - Exceptions

UNIVERSITÄT SALZBURG

Fehlersituationen

Laufzeitfehler (runtime errors) (jetzt kurz Fehler genannt) können meist nicht an der Stelle behoben werden, wo sie auftreten – wie soll in einer Fehlersituation das Programm reagieren?



mögliche Reaktionen in `mthree`:

- Fehlermeldung ausgeben?
- Programm abbrechen?
- Programmlauf fortsetzen?
- spezielle Korrekturmaßnahmen?
- .....?

eventuell soll `mtwo` oder `mone` reagieren?

Fehler tritt hier auf – wie soll `mthree` reagieren?

UNIVERSITÄT SALZBURG

Anforderungen an Fehlerbehandlung

Z.B.: welcher Teil ist für Reaktion auf Fehler zuständig?

- `charAt()` oder
- rufender Programmteil

```
String s = "EinfProg";
int pos = 11;
char c = s.charAt(pos);
  //Fehler - Index zu gross!!
```

Welche Eigenschaften soll eine gute Fehlerbehandlung besitzen?

- tritt ein Fehler auf, soll eine beliebige Methode in der Aufrufkette darauf reagieren
- die Fehlermeldung soll nicht mittels Rückgabewerte erfolgen
- möglichst gute Trennung von fehlerfreiem Programmablauf und Fehlerbehandlung
- jeder mögliche Fehler soll von irgendeiner Methode in der Aufrufkette behandelt werden – es darf keiner ignoriert werden

UNIVERSITÄT SALZBURG

Ausnahmebehandlung

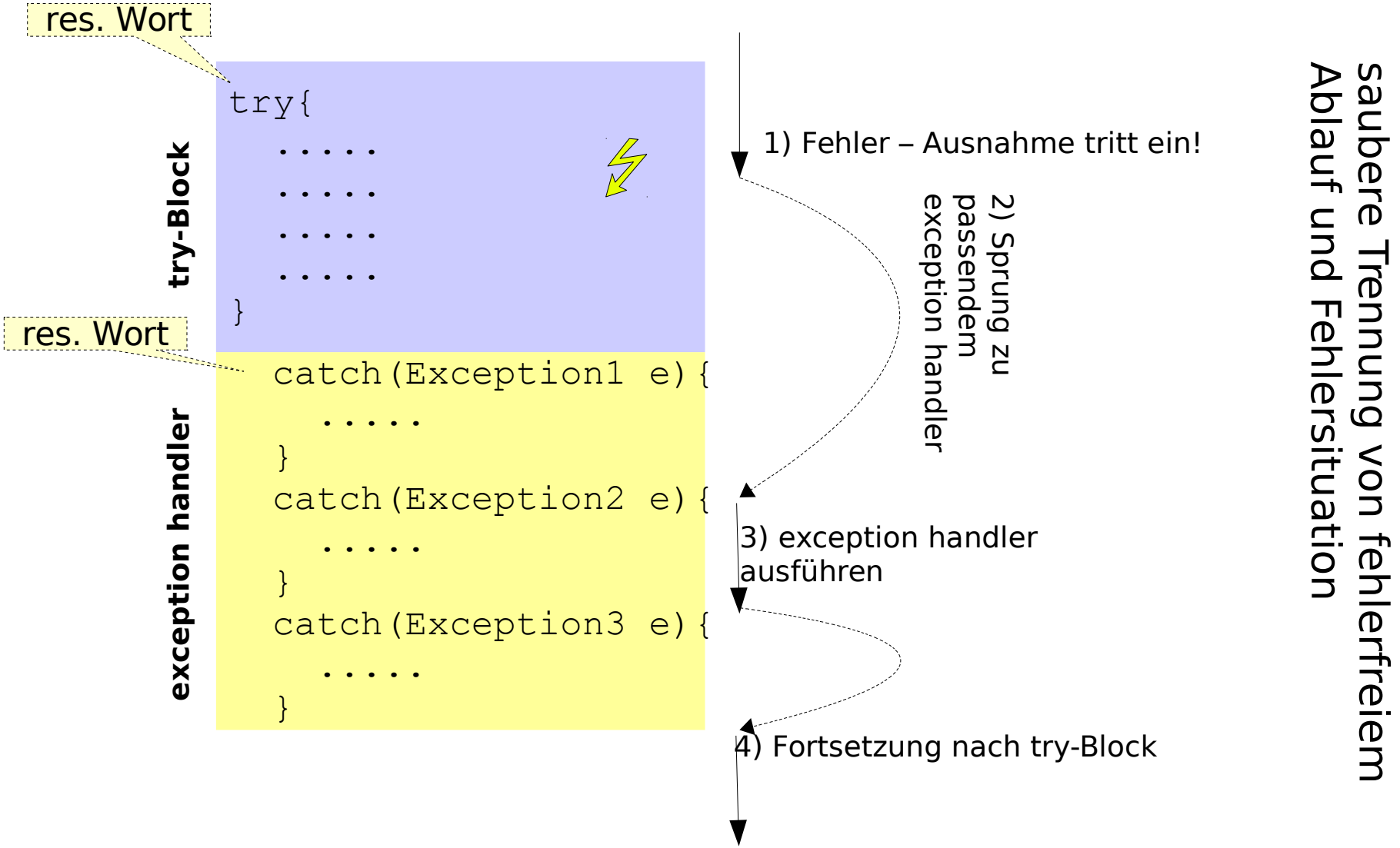
Das Konzept der **Ausnahmebehandlung (exception handling)** in Java besteht aus:

- *Geschütztem Block (try-block)*: gekennzeichnete Anweisungsfolge; tritt darin (oder in aufgerufenen Methoden) ein Fehler auf, wird sichergestellt, dass er behandelt wird.
- *Ausnahmebehandler (exception handler)*: Anweisungsfolge, die im Falle eines Fehler ausgeführt wird (Reaktion auf Fehler); jeder try-Block besitzt einen oder mehrere davon.
- *Ausnahme (exception)*: signalisiert das Auftreten eines Fehlers – es wird ein passender Ausnahmebehandler gesucht und ausgeführt, dann nach dem entsprechenden try-Block fortgesetzt.

Java verwendet dazu die try-Anweisung.

UNIVERSITÄT SALZBURG

try-Anweisung

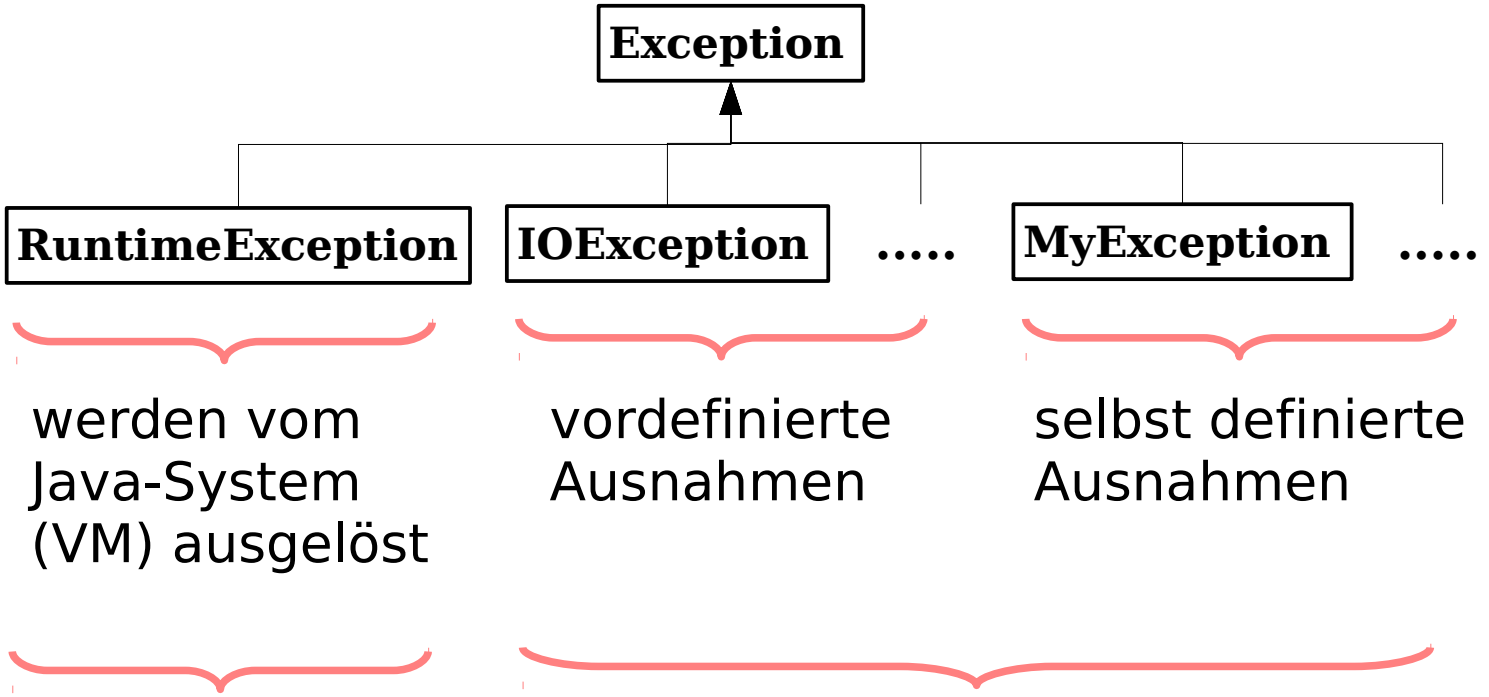


(siehe Programm `ExceptionDemo`)

UNIVERSITÄT SALZBURG

Ausnahmen sind Objekte

In Java sind Ausnahmen als Objekte realisiert. Die Basisklasse dazu heißt `Exception`.



UNIVERSITÄT SALZBURG

Ausnahmen: Beispiele

Runtime Exceptions (Laufzeitfehler), werden von der Java-VM ausgelöst, sind z.B.

- `ArithmeticException`: etwa Division durch 0
- `ArrayIndexOutOfBoundsException`: Index außerhalb des gültigen Bereichs
- `NullPointerException`: null-Zeiger, obwohl Objekt erwartet wird

Ausnahmen, die behandelt werden müssen (checked exceptions) – werden vom Compiler überprüft, somit wird keine Fehlersituation übersehen

- vordefinierte Ausnahmen: z.B. `FileNotFoundException`
- selbst definierte Ausnahmen

UNIVERSITÄT SALZBURG

Klasse Exception und selbst definierte Ausnahmen

Selbst definierte Ausnahmen müssen von der Klasse `Exception` abgeleitet werden. Einige Elemente der Schnittstelle davon sind:

- public Exception(String message) : Konstruktor, der auch eine spezifische Nachricht erhält
- public String getMessage() : liefert die spezifische Nachricht, die mit obigem Konstruktor definiert wurde
- public String toString() : liefert den Namen der entsprechenden Klasse und eventuelle weitere Infos
- public void printStackTrace() : Ausgabe der Methodenaufrufkette bis zur obersten Ebene am Bildschirm (zum Fehlerzeitpunkt)

UNIVERSITÄT SALZBURG

Selbst definierte Ausnahmen und throw-Anweisung

- Definition einer selbstdefinierten Ausnahme als Unterklasse von `Exception`

```
public class ClipException
  extends Exception{

  public ClipException(String s){
    super(s);
  }
}
```

- Auslösung einer Ausnahme mittels der `throw`-Anweisung

```
if (s.length() < 2)
  throw new ClipException("zu kurz");
```

`throw`-Anweisung bewirkt:

- Suche nach einer `catch`-Klausel mit passendem Ausnahmetyp in allen gerade laufenden Methoden
- Abbruch der aktuellen Anweisungsfolge und Sprung zur `catch`-Klausel mit dem Ausnahmeobjekt als Parameter
- Abarbeitung der `catch`-Klausel und anschließender Fortsetzung nach der dazu gehörenden `try`-Anweisung

UNIVERSITÄT SALZBURG

Ausnahmen im Methodenkopf

Ausnahmen sind nicht immer an der Stelle ihres Auftretens sinnvoll behandelbar

→ Lösung: Weiterleitung an rufenden Programmteil

→ dazu nötig: spezifizieren *aller* Ausnahmen, die von einer Methode ausgelöst werden können und nicht abgefangen werden, im Methodenkopf

```
public static String clip (String s) throws ClipException {
  .....
  if (...) throw new ClipException(...);
  .....
```

`throws`-Klausel ist Teil der Methodenschnittstelle

→ Compiler prüft dies

rufender Programmteil muss diese Ausnahme(n) behandeln oder mit throws-Klausel für Weitergabe spezifizieren!

→ Ausnahmebehandlung kann nicht übersehen werden

(siehe Programm `ClipExceptionDemo`)

(weitere Details: siehe Literatur)

UNIVERSITÄT SALZBURG

Kapitel 11

Abstrakte Klassen, Interfaces

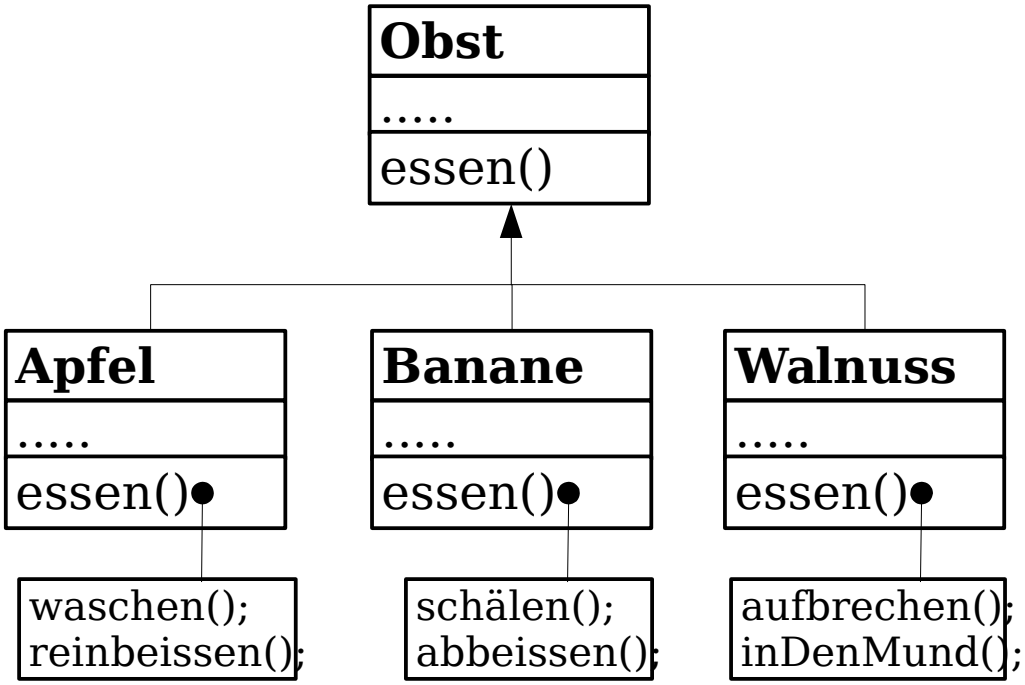
UNIVERSITÄT SALZBURG

Abstrakte Klassen: Motivation

Oberklassen fassen Gemeinsamkeiten anderer Klassen zusammen. Jedoch ist es oft nicht sinnvoll, davon Objekte zu erstellen, weil etwa Methoden nicht universell implementierbar sind.

(Anschauliches) Beispiel – Klasse `Obst`

- Methode `essen()` allgemein nicht sinnvoll umsetzbar, da komplett unterschiedlich für verschiedene Obstsorten
- Methode `essen()` kann erst für eine spezifische Obstsorte sinnvoll definiert werden

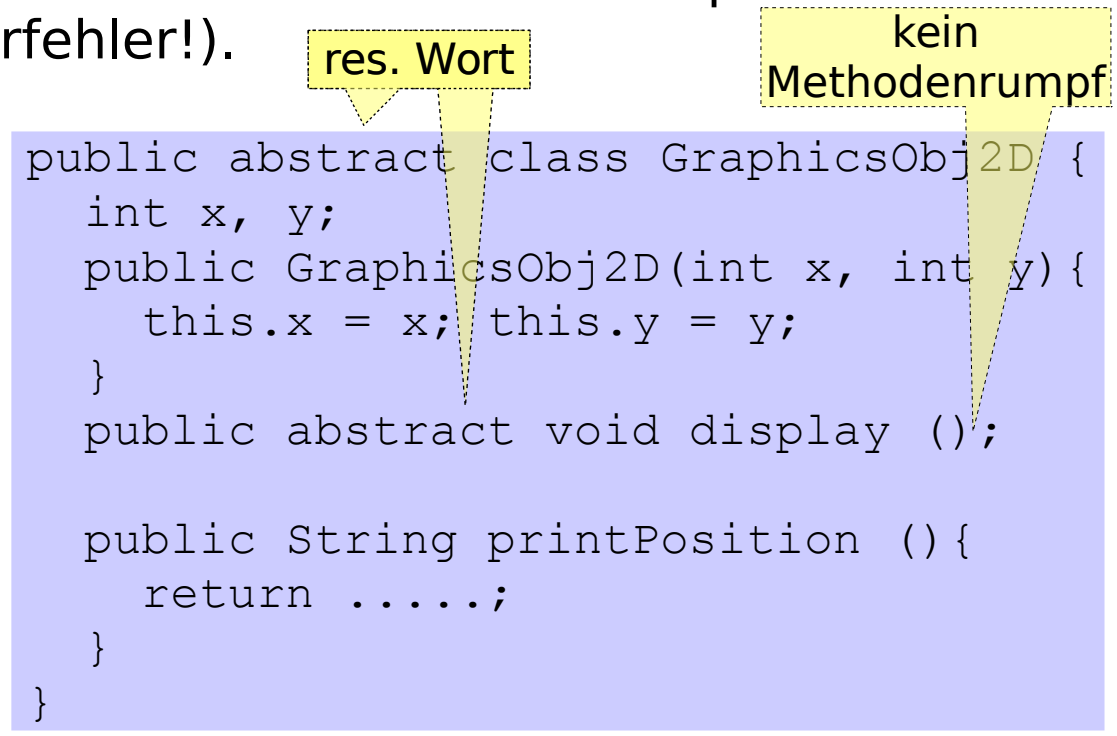


UNIVERSITÄT SALZBURG

Definition von abstrakten Klassen

Methoden ohne Implementierung (d.h. ohne Rumpf) werden **abstrakte Methoden** genannt. Enthält eine Klasse mindestens eine abstrakte Methode, wird sie zu einer **abstrakte Klasse**.

- Von abstrakten Klassen können keine Objekte erzeugt werden!
- Unterklassen von abstrakten Klassen müssen zu allen abstrakten Methoden die Rümpfe implementieren oder sind wieder abstrakt.
- Eine abstrakte Klasse definiert eine *Schnittstelle* für spätere Unterklassen – legt fest, welche Methoden mindestens implementiert werden müssen (sonst Compilerfehler!).
- Eine abstrakte Klasse definiert einen Typ – d.h. Variable, Parameter, ... deklarierbar.
- Mittels Variablen einer abstrakten Klasse sind nur die Methoden dieser Klasse aufrufbar (nicht weitere einer Unterklasse)!



UNIVERSITÄT SALZBURG

Beispiel zu abstrakten Klassen

Beispiel: Verwaltung von 2-dimensionalen graphischen Objekten mittels abstrakter Oberklasse `GraphicsObj2D` . Die abstrakte `display`-Methode dient der Darstellung von objektspezifischen Eigenschaften (vereinfacht hier nur in Textform).

Unterklassen für verschiedene Graphikobjekte (Punkt, Linie, Kreis, ...) ermöglichen beliebige Vielfalt.

```
public class Point2D extends GraphicsObj2D {
  ....
  public void display (){
    System.out.print("Point ...");
  }
}
```

```
public class Circle2D extends GraphicsObj2D {
  ....
  public void display (){
    System.out.print("Circle ...");
  }
}
```

(siehe Klassen `Graphics2DDemo`, `Point2D`, `Line2D`, `Circle2D`)

UNIVERSITÄT SALZBURG

Interfaces: Motivation

Es geht noch abstrakter: die Gemeinsamkeit mehrerer Klassen besteht nur aus abstrakten Methoden!

Z.B. sollen Graphikobjekte in 2D auch verschoben werden können (zu einer neuen Position). Jedoch soll diese Eigenschaft eventuell auch bei weiteren Klassen nützlich sein.

Dann ist ein **Interface** zu implementieren, welches die entsprechenden Methodenköpfe enthält.

- Definition in eigener Datei (wie eine Klasse)
- Interfaces sind so etwas wie „komplett“ abstrakte Klassen: *alle* Methoden sind (implizit) abstrakt
- ein Interface kann auch Konstante (implizit `static final`) enthalten

res. Wort

```
interface Moveable {

  //moves for dx, dy units
  public void move (int dx, int dy);
}
```

implizit abstrakte Methode

UNIVERSITÄT SALZBURG

Verwendung von Interfaces

- Eine Klasse erbt von einem Interface – eigentlich „eine Klasse implementiert ein Interface“
  → res. Wort `implements`
- ein Interface definiert einen Typ, daher sind z.B. Variable deklarierbar - jedoch: keine Objekte davon erzeugbar
- Mittels Variablen eines Interfacetyps sind nur die Methoden des Interfaces aufrufbar!

```
public class Point2DMv ...
    implements Moveable {
  .....
  public void move (int dx, int dy){
  .....
  }
}
```

```
Moveable mv;
mv = new Moveable(); //Compilerfehler!
mv = new Point2DMv(2, 7); //ok
```

- mit Interfaces können Objekte von Klassen, die nicht der selben Hierarchie angehören, gleich behandelt werden
- eine Klasse kann beliebig viele Interfaces implementieren
- Interfaces können erweitert werden (wie Klassen)

(siehe weiters `Moveable, Point2DMv, Line2DMv`)

UNIVERSITÄT SALZBURG

Abstrakte Klassen – Interfaces

Wann Interface – wann abstrakte Klasse?

- ist zum Teil Ansichtssache
- abstrakte Klasse: falls implementierte Methoden oder Instanzvariablen benötigt werden
- Interface: falls sonst sachlich unzusammenhängende Klassen gleiche Funktionalität (von außen gesehen) anbieten sollen

Zusammenfassung: Objekte bieten Dienste (z.B. Methoden) an, die andere Objekte nutzen können
→ Unterscheidung *Schnittstelle – Implementierung* ist wichtig!

- *Schnittstelle* (allg. interface): externe Sicht, angebotene Methoden – beschrieben durch
  - Signatur/Methodenkopf (Ergebnistyp, Parametertyp)
  - semantische Eigenschaften (Spezifikationen, Einschränkungen)
- *Implementierung*: interne Sicht – realisiert mittels Instanzvariable, Methodenrümpfe, Hilfsmethoden

UNIVERSITÄT SALZBURG

Kapitel 10

Zeitkomplexität 12

UNIVERSITÄT SALZBURG

Exkurs: Zeitkomplexität von Algorithmen

Der Zeitbedarf von Programmen hängt von vielen Faktoren ab, z.B.

- Prozessor
- Compiler
- Details bei der Implementierung
- Qualität des Algorithmus (Verfahrens)
- ...

Betrachten hier: *Qualität des Algorithmus*

Somit sind 2 Punkte zu klären:

- Was „kostet“ ein einzelner Schritt?
  - meist nur sehr geringe Unterschiede für verschiedene Lösungen – daher praktisch kaum Auswirkungen
- Wie oft wird dieser wiederholt (Anzahl der Schritte)?
  - hier erzielen unterschiedliche Lösungen oft große Unterschiede

UNIVERSITÄT SALZBURG

Exkurs: Zeitkomplexität von Algorithmen (Forts.)

Beispiel: Vergleich von linearer und binärer Suche wenn Element nicht vorhanden ist

- einzelner Schritt: jeweils Vergleich und arithmetische Operation
- Anzahl der Schritte hängt offensichtlich von der Gesamtanzahl n der zu durchsuchenden Elemente ab

  → Anzahl der Schritte als Funktion von n ist ein Gütemaß des Verfahrens

  d.h. mittels Analyse des Algorithmus ist die Anzahl der Schritte zu bestimmen!

n	linear	binär
10	10	3
20	20	4
50	50	6
100	100	7
1000	1000	10
10000	10000	14
20000	20000	15
100000	100000	17

Für den Fall dass ein Element nicht gefunden wird, gilt:

- linear: alle Elemente müssen überprüft werden → n Schritte
- binär: jeder Schritt führt zu einer Halbierung des Suchbereichs → $log_2(n)$ Schritte (im wesentlichen)

UNIVERSITÄT SALZBURG

Exkurs: Zeitkomplexität von Algorithmen (Forts.)

Beispiele für Funktionen von n, wenn ein Schritt 10^{-4} sec (1/10-Millisekunde) dauert (Angaben in sec, gerundet).

# Schritte	n = 100	n = 1000
$\log_2(n)$	0.001	0.001
n	0.010	0.100
$n*n^{0.5}$	0.100	3.160
n+5	0.010	0.100
2*n	0.020	0.200
n^2	1.000	100.000
$n^2 + n$	1.010	100.100
n^3	100.000	100000.000 (~ 28 Std.)
2^n	$4*10^{18}$ Jahre	?

Daraus ersichtlich

- wenig relevant: konstanter Faktor oder Summand mit kleinerem Exponenten
- großer Unterschied: n groß, verschiedene Potenzen von n
- praktisch unbrauchbar: exponentielles Wachstum!

Das allgemeine Wachstumsverhalten ist ein gutes Vergleichskriterium für Algorithmen.

UNIVERSITÄT SALZBURG

Exkurs: Zeitkomplexität von Algorithmen (Forts.)

Die Funktion `f(n)` für das Wachstumsverhalten eines Algorithmus kann mittels der O-Notation (Groß-O-Notation) beschrieben werden:

`f(n) = O(g(n))` bedeutet „`f` ist von der Ordnung `g`“ (order of growth), auch kurz „`f` ist ein Groß-O von `g`“.

Genauer: `c*g(n)` ist eine obere Schranke von `f(n)`

Formal: $\exists c, n_0 \in \mathbb{N}: \; f(n) \leqslant c * g(n) \;\; \forall n \geqslant n_0$

Dabei ist zu beachten

- dies ist *keine* Gleichung (nicht symmetrisch)
- die Ordnung gibt nur eine von vielen möglichen oberen Schranken an – angestrebt wird natürlich jene, welche eine möglichst enge Charakterisierung ergibt

UNIVERSITÄT SALZBURG

Exkurs: Zeitkomplexität von Algorithmen (Forts.)

Oft benutzte Ordnungen:

$O(1)$	konstant	$5,\ 47,\ 3*x$
$O(log_2(n))$	logarithmisch	$3.5 * log_2(n)$
$O(n)$	linear	$n + 295,\ 3*n + 5$
$O(n^2)$	quadratisch	$5*n^2 + 3*n$
$O(n^3)$	kubisch	$0.5*n^3 - n$
$O(2^n)$	exponentiell	$3*2^n - n^2$

Somit ergibt sich für

- lineare Suche: $O(n)$ - linear
- binäre Suche: $O(log_2(n))$ – logaritmisch
- MinSort: $O(n^2)$ – quadratisch

(genauer: siehe LV über Algorithmen und Datenstrukturen)