

Programmierung 2

Kapitel 4

Mehr zu Vererbung: super

Verständnisfragen



- Was ist ein Nebeneffekt?
- Was ist der Wert von v und u nach der letzten Anweisung?

```
int u = 1;  
int v = u++;
```
- Was ist eine **polymorphe** Methode? Was sind die Vorteile der Polymorphie?
- Was ist eine **überladene** Methode? Wozu ist Überladen gut?
- Wodurch wird beim Aufruf eine Methode eindeutig identifizierbar?
- Muss in Java eine Methode konstant viele Parameter haben?
- Was ist bei varargs zu beachten?
- Wozu hat die main-Methode Parameter?

Verständnisfragen



- Was sind die Vorteile des Vererbungskonzeptes?
- Was ist Subtyping?
- Was wird genau an die Kindklasse vererbt?
- Wie greift man in der Kindklasse auf die (geerbten) privaten Methoden der Elternklasse zu?
- Was ist eine Vererbungshierarchie?
- Welche Java-Klasse ist die erste in der Vererbungshierarchie?
- Ist eine abgeleitete Klasse eine Spezialisierung oder Generalisierung der Basisklasse?

Überblick Programmierung 2



Software-Engineering

- Testen: Unit-Tests mit JUnit
- Strukturieren: Pakete
- Kommentieren: Javadoc, Annotationen

OO- Programmierung

- Vererbung (+ Operatoren)
- **super-Operator**
- Polymorphie von Objekten
- Finale Klassen
- Abstrakte Klassen und Schnittstellen
- Wrapper- Klassen
- generische Klassen

Dynamische Datenstrukturen

- Verkettete Listen
- Stack, Queue, Binärbäume

GUI

- GUI-Programmierung mit Swing

Vererbung und eigene Exceptions-Klassen

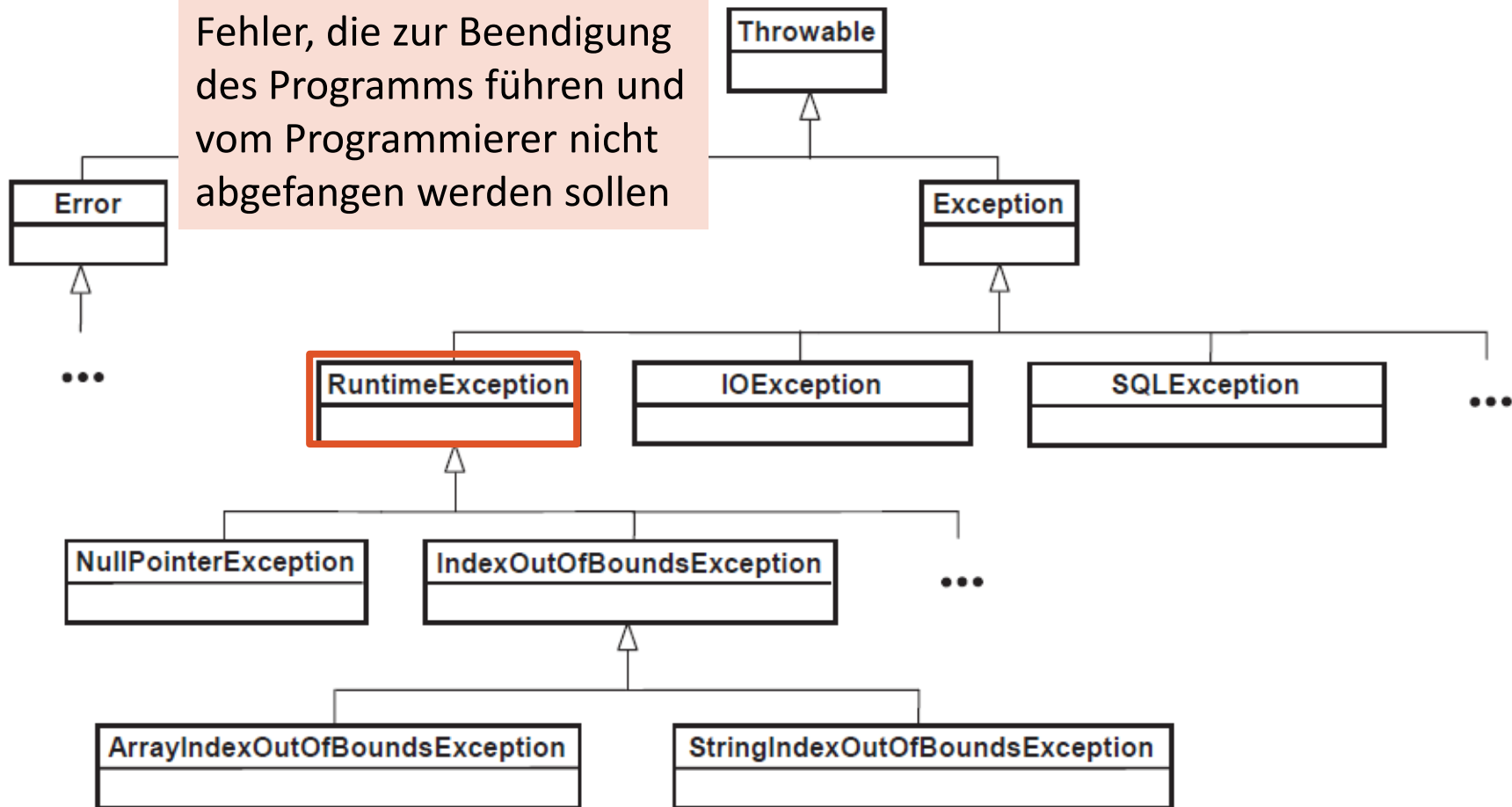
Wiederholung: Exception

in Java:

- jede Exception = **Objekt** einer Exception-Klasse
- Exception-Klasse
 - in Java vordefiniert
 - kann selbst definiert werden (benutzerdefiniert)
- Exception-Klassen stehen in einer *Hierarchie* → Vererbung
- jede Exception-Klasse von der Klasse **Exception** abgeleitet
- Klasse **Exception** wiederum von der Klasse **Throwable** abgeleitet

Exception-Hierarchie

Error = **schwerwiegende** Fehler, die zur Beendigung des Programms führen und vom Programmierer nicht abgefangen werden sollen

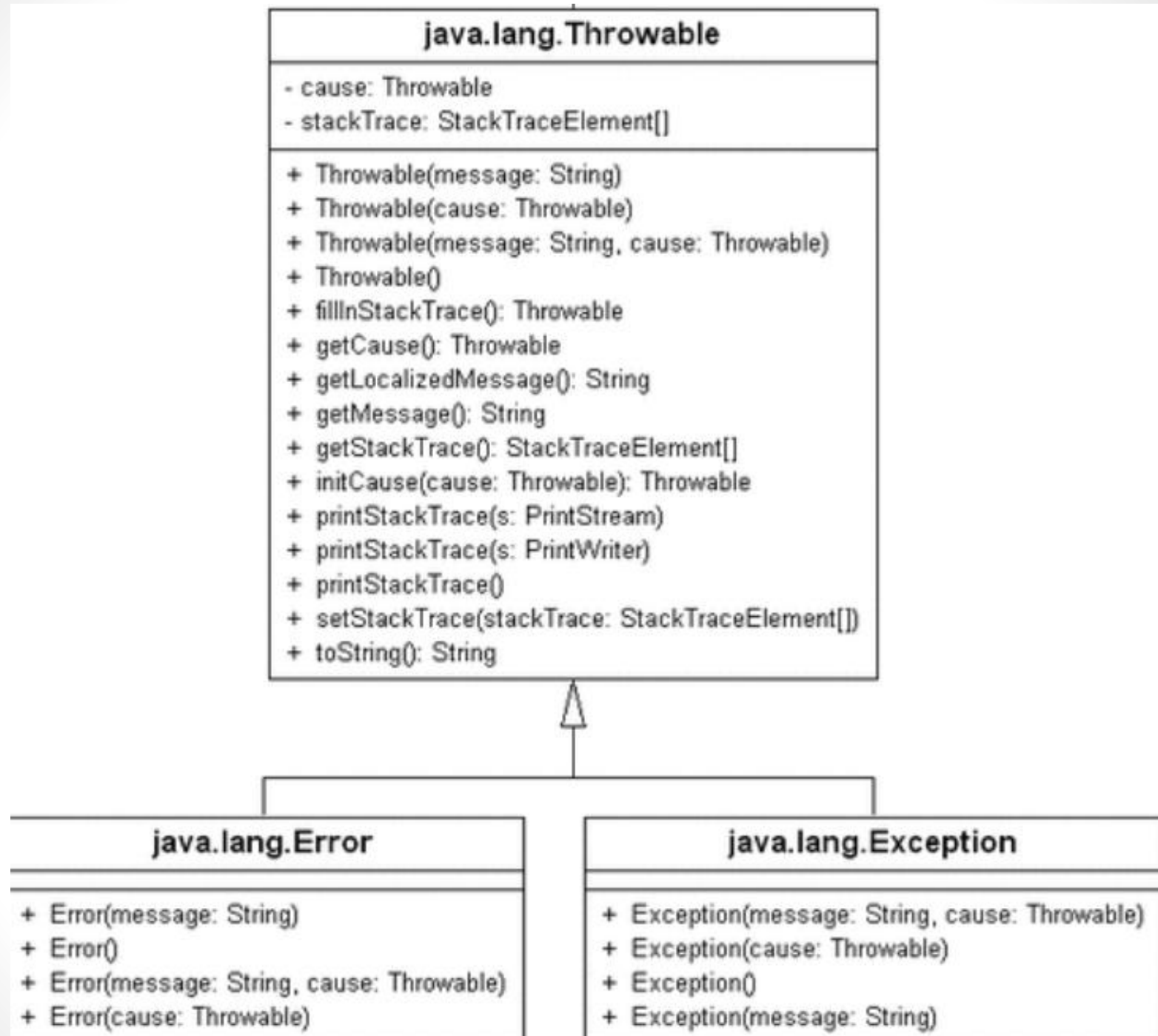


Klasse Throwable

- besitzt privates String-Attribut, um Fehlertext (z.B. Beschreibung des Fehlers) zu hinterlegen → Fehlertext an den Konstruktor der **Exception**-Klasse übergeben

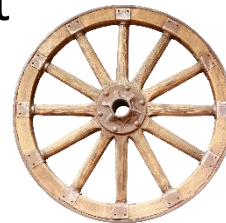
```
throw new ExceptionEinesTyps("Fehlermeldung");
```

- vererbt eine Reihe nützlicher Methoden, wie z.B.
 - **String getMessage()**: liefert einen Fehlermeldungstext
 - **String toString()**: liefert Name der Exception-Klasse + ":" + "detail message"
 - **void printStackTrace()**: zeigt Schachtelung der Methoden, deren Aufrufe zum Auslösen der Exception geführt haben



Vordefinierte Java-Exceptions

- abgeleitet von der Klasse `RuntimeException` bietet Java sehr viele vordefinierte Exceptions
 - vgl. Java-API im Paket **`java.lang`** unter **Exceptions**:
<https://docs.oracle.com/javase/7/docs/api/>
→ *fast* alle gängigen Laufzeitfehler damit abgefangen
 - wenn KEINE vordefinierte Exception passt
und nur dann →
bitte nicht das Rad neu erfinden!
- auch möglich, **eigene** Exception-Klasse zu definieren



Eigene Exception-Klasse: checked

- üblicherweise nur Konstruktoren überschrieben (da sie nicht vererbt werden)

```
class MyException extends Exception {  
    public MyException() {  
        super("Seltener Fehler");  
    }  
  
    public MyException(String message) {  
        super(message);  
    }  
}
```

checked Exception → muss mit **throws** angekündigt und mit **try-catch** abgefangen werden, ansonsten **Compilerfehler**

Eigene Exception-Klasse: unchecked

- üblicherweise nur Konstruktoren überschrieben (da sie nicht vererbt werden)

```
class MyException extends RuntimeException {  
    public MyException() {  
        super("Seltener Fehler");  
    }  
  
    public MyException(String message) {  
        super(message);  
    }  
}
```

unchecked Exception → Compiler prüft nicht, ob sie angekündigt und abgefangen wird, aber wenn nicht gefangen → Programm abgebrochen

Verständnisfragen



- Wann sollte man eine eigene Exception-Klasse implementieren?
- Was ist der Unterschied zwischen **checked** und **unchecked** Exceptions? Geben Sie Beispiele an.
- Löst eine eigene Exceptionklasse eine checked oder unchecked Exception?
- Was passiert, wenn eine Exception im Methodenkopf nicht mit throws angekündigt wird?
- Was ist der Unterschied zwischen einem **robusten** und **defensiven** Programmierstil?

Konstruktoren

- überladener Konstruktor
- eigener parameterloser Konstruktor
- defensiver Konstruktor

Konstruktor

- dient zum Initialisieren eines Objektes
- parameterloser *Default-Konstruktor* für jede Klasse vom Compiler bereitgestellt – nur solange kein eigener Konstruktor existiert

eigener Konstruktor:

- hat den gleichen Namen wie die Klasse selbst und kein Rückgabetyt deklariert → dadurch vom Compiler erkannt
- kann überladen werden
- kann anderen Konstruktor derselben Klasse aufrufen:
 `this(parameterliste)` als 1.Anweisung
- nicht an eine abgeleitete Klasse vererbt

Konstruktor: Beispiel

```
public class FigurA {  
  
    private Punkt anker;  
    private String farbe = "rot";  
  
    public FigurA(Punkt pkt) {  
        anker = pkt;  
    }  
  
    public FigurA(Punkt p, String f) {  
        this(p);  
        farbe = f;  
    }  
  
    ...  
}
```

→ Konstruktor überladen

Konstruktor der Superklasse

- im Konstruktor der abgeleiteten Klasse – Aufruf des Konstruktors der Superklasse möglich:

super(paramliste)

→ im Objekt der abgeleiteten Klasse alle von der Superklasse *geerbten* Datenfelder **initialisiert**

- Aufruf **super(paramliste)** immer **in der 1. Zeile** des Konstruktors der Subklasse

Beispiel: super()

```
public class Kreis extends Figur {  
    private int radius;
```

```
    public Kreis(int r, Punkt ank) {  
        super(ank);  
        radius = r;  
    }  
    ...  
}
```

← Aufruf des Konstruktors
der Superklasse Figur
mit dem Parameter ank:
Figur(ank)

```
public Figur(Punkt p) {  
    anker = p;  
}
```

Automatischer Aufruf

wenn in der 1. Zeile des Konstruktors einer Subklasse der Aufruf **super(paramliste)** fehlt, dann:

- als erste Anweisung im Konstruktor der abgeleiteten Klasse **super()** vom Compiler *automatisch* ausgeführt → Compiler versucht, den parameterlosen **Default-Konstruktor** der Superklasse aufzurufen, um die Datenfelder der Superklasse zu initialisieren *

* parameterloser Default-Konstruktor der Klasse **Object** hat leeren Rumpf (= tut nichts)

Automatischer Aufruf: Beispiel

```
public class Figur {  
    private Punkt anker;  
  
    public Figur(Punkt p) {  
        anker = p;  
    }  
}  
  
public class Kreis extends Figur {  
    private int radius;  
  
    public Kreis(int r, Punkt ank) {  
        super();  
        radius = r;  
    }  
...  
}
```

Problem



Wenn:

- in der Superklasse **Figur**: (nur) ein eigener Konstruktor **Figur(Punkt p)** mit Parameter vorhanden
- in der Subklasse **Kreis**: in der 1. Zeile des Konstruktors KEIN expliziter Aufruf des Konstruktors der Superklasse mit **super(p)**

Dann:

im Konstruktor der Klasse Subklasse **Kreis** vom Compiler automatisch Anweisung **super()** hinzugefügt



Was passiert?

→ **Fehler**, da parameterloser Default-Konstruktor in der Superklasse **Figur** nicht mehr verfügbar! (weil in **Figur** ein Konstruktor mit Parametern implementiert)

Lösung



Robuster Programmierstil:

in jeder Klasse A, die einen eigenen Konstruktor mit Parametern hat – auch einen **eigenen parameterlosen Konstruktor** schreiben

Vorteile:

- wenn im Konstruktor einer Subklasse der explizite Aufruf des A-Konstruktors fehlt, wird durch die Ergänzung des Compilers `super ()` der **eigene parameterlose Konstruktor** von A aufgerufen → **kein Fehler**
- Sichtbarkeit des eigenen parameterlosen Konstruktors kann explizit (selbst) gewählt werden, z.B. auf `private` gesetzt

Beispiel: Klasse *Figur*

```
public class Figur {  
    private Punkt anker;  
  
    public Figur(Punkt p) {  
        anker = p;  
    }  
  
    public Figur() {  
        anker = new Punkt(0,0);  
        System.out.println("Warnung!");  
    }  
    ...  
}
```

← eigener parameterloser Konstruktor

Beispiel: Klasse `Kreis`

```
public class Kreis extends Figur {  
    private int radius;  
  
    public Kreis(int r, Punkt p) {  
        super(p);  
        radius = r;  
    }  
  
    public Kreis() {  
        super();  
        radius = 0;  
        System.out.println("Warnung!");  
    }  
    ...  
}
```

← eigener parameterloser Konstruktor

Konstrukturen: unzulässige Parameter

```
public class Figur {  
    private Punkt anker;  
  
    public Figur(Punkt p) {  
        anker = p;  
    }  
  
    public Figur() {  
        anker = new Punkt(0,0);  
        System.out.println("Parameterloser Konstruktor!");  
    }  
  
    public String toString() {  
        return "Anker:(" + anker.getX() + "," +  
                anker.getY() + ")";  
    }  
}
```

wenn `p = null`, dann
spätestens in `toString()` wirft
Java automatisch eine
`NullPointerException`

WAS TUN?

Defensive Programmierung



- jede Methode (auch der Konstruktor) prüft *alle* Parameter auf **korrekten Wertebereich**
 - arbeitet **normal** für alle Parameterwerte des **Wertebereichs** (insbes. terminiert nicht abrupt)
 - für alle anderen Parameterwerte meldet sie so schnell wie möglich **einen Fehler** (→ **löst eine Ausnahme** aus), anstatt Fehler zu *ignorieren* oder zu *verschleppen* (**fail-fast-Prinzip**)
- defensiv implementierte Methode *misstrauisch* gegenüber allen Eingaben, zeigt *umgehend Verstöße* an und bricht in einem *geordneten* Prozess ab

Defensiver Konstruktor (1)

```
public class Figur {  
    private Punkt anker;  
  
    //anker darf nicht null sein  
    public Figur(Punkt anker) throws NullPointerException {  
        if(anker == null) throw new  
            NullPointerException("anker darf nicht null sein");  
        this.anker = anker;  
    }  
  
    public Figur() {...}  
  
    ...  
}
```

aussagekräftige Fehlermeldung
ist sehr hilfreich!

Defensiver Konstruktor (2)

```
public class Kreis extends Figur {  
    private int radius;
```

```
    //radius muss positiv sein,  
    //anker darf nicht null sein
```

```
    public Kreis(int radius, Punkt anker)  
        throws IllegalArgumentException, NullPointerException {  
        super(anker);  
        if(radius <= 0) throw new  
            IllegalArgumentException("radius muss positiv sein");  
        this.radius = radius;  
    }
```

```
    public Kreis() {...}
```

```
    ...
```

```
}
```

Exception-Ankündigung
weiterreichen!

vom Figur-Konstruktor geworfen



Defensive Setter

```
public class Kreis extends Figur {  
    private int radius;  
  
    public Kreis(int radius, Punkt anker) {...}  
    public Kreis() {...}  
  
    //radius muss positiv sein  
    public int setRadius(int radius)  
        throws IllegalArgumentException {  
        if(radius <= 0) throw new  
            IllegalArgumentException("radius muss positiv sein");  
        this.radius = radius;  
    }  
}
```

Defensive Setter: eigene Exception

```
public class Kreis extends Figur {  
    private int radius;  
  
    public Kreis(int radius, Punkt anker) {...}  
    public Kreis() {...}  
  
    //radius muss zwischen 1 und 10 sein  
    public int setRadius(int radius)  
                                throws IllegalArgumentException {  
        if(!(1 <= radius <= 10)) throw new  
            IllegalArgumentException("radius muss zwischen 1 und  
                                   10 sein");  
        this.radius = radius;  
    }  
}
```

Defensive Setter im Konstruktor

- wenn *defensive Setter* bereits implementiert, dann **im Konstruktor aufrufen** - statt einen defensiven Konstruktor zu implementieren!

```
public class Kreis extends Figur {  
    private int radius;  
  
    public Kreis(int radius, Punkt anker)  
        throws NullPointerException, IllegalArgumentException {  
        super(anker);  
        setRadius(radius);  
    }  
}
```

Prinzip **Don't Repeat Yourself**



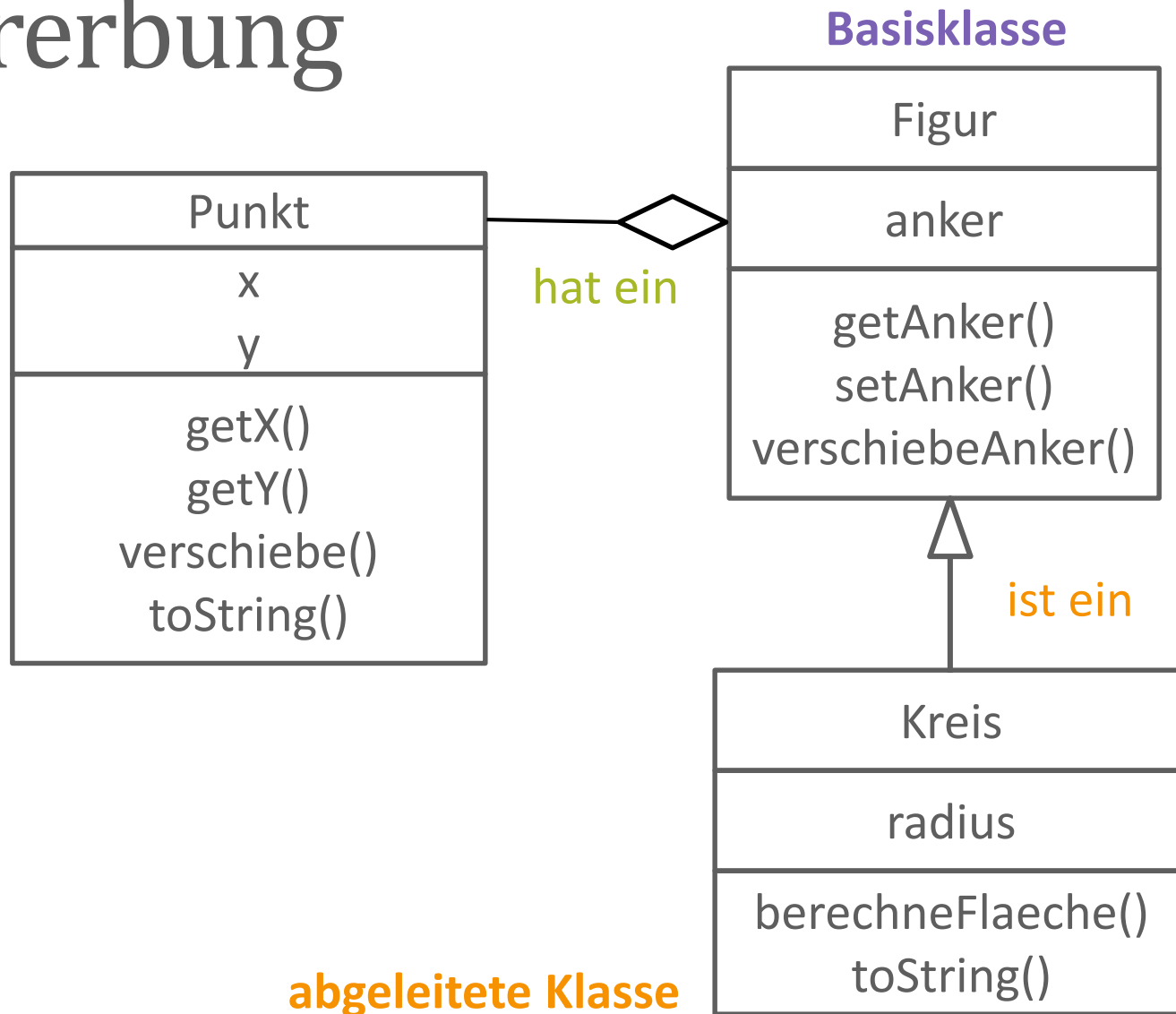
```
//radius muss zwischen 1 und 10 sein  
public int setRadius(int radius)  
    throws IllegalArgumentException {  
    ...  
}  
}
```

Verständnisfragen



- Wie wird der Konstruktor der Vaterklasse im Konstruktor der Sohnklasse aufgerufen?
- Warum macht es Sinn, in jeder Klasse einen eigenen parameterlosen Konstruktor zur Verfügung zu stellen?
- Was ist ein defensiver Konstruktor?
- Müssen Setter defensiv sein?
- Was ist der Vorteil vom Aufruf defensiver Setter im Konstruktor?
- Werden im Konstruktor nur die Exceptions angekündigt, die in diesem Konstruktor durch *throw **explizit*** geworfen werden?

Vererbung



Polymorphie von Methoden: Überschreiben in der Subklasse

Polymorphie von Methoden: Wiederholung

polymorphe Methode:

hat in verschiedenen Klassen identischen Methodenkopf und gleiche Semantik, jedoch unterschiedliche Implementierung, **überschriebene Methode**

→ **Vorteil:** Verständlichkeit des Programms erhöht:

z.B. klar, dass `print()` in jeder Klasse zur Ausgabe des Objektes auf dem Bildschirm verwendet

z.B. in jeder Klasse `toString()` überschrieben

toString() von Kreis

```
public class Kreis extends Figur {  
    private int radius;  
    ...  
    public String toString() {  
        return "Radius: " + radius + " Anker: " +  
                getAnker();  
    }  
}
```

→ Aufruf in der main()-Methode der *Testklasse*:

```
System.out.println(k);
```

→ Ausgabe:

Radius: 5 Anker: (1, 2)

Beispiel: vererbte Attribute

```
public class Kreis extends Figur {  
    private int radius;  
    ...  
    public String toString() {  
        return "Radius: " + radius + " Anker: " +  
                               getAnker();  
    }  
}
```

↑
direkter Zugriff auf eigene
Datenfelder (**radius**)

↑
kein direkter Zugriff auf geerbte
Datenfelder (**anker**) möglich,
da in der Superklasse als
private definiert → nur über
Methode **getAnker()**

Methoden überschreiben: Bsp.

```
public class Figur {  
    private Punkt anker;  
    ...  
    public String toString() {  
        return "Anker:(" + anker.getX() + "," +  
                        anker.getY() + ")";  
    }  
}
```

```
public class Kreis extends Figur {  
    private int radius;  
    ...  
    public String toString() {  
        return "Radius: " + radius + " Anker: " +  
                        getAnker();  
    }  
}
```

überschreibt die Methode
toString() der Superklasse

Instanzmethoden überschreiben

- in der Superklasse: `int methodeA (int p1, char p2)`
- in der Subklasse: `int methodeA (int p1, char p2)`, d.h.
 - gleiche Signatur (Methodenname, Typ, Anzahl und Reihenfolge der formalen Parameter)
 - gleicher Rückgabotyp
 - aber: andere Implementierung (Methodenrumpf)

→ `methodeA()` der Superklasse wurde in der Subklasse überschrieben (**polymorphe Methode**)



→ `methodeA()` der Superklasse in der Subklasse *nicht mehr sichtbar* → beim Aufruf `methodeA()` in der Subklasse immer `methodeA()` aufgerufen

Private Methoden überschreiben?

private Methoden einer Basisklasse in der abgeleiteten Klasse nicht sichtbar



→ **private Methoden** der Basisklasse nicht überschreibbar

Zugriff über super

Zugriff in der Tochterklasse auf die „ursprüngliche“ Methode **methodeA()** der Mutterklasse:

```
super.methodeA()
```

Was passiert?

`super.methode()` in `((Mutterklasse)this).methode()` vom Compiler umgesetzt →

`this` = Referenz auf das eigene Objekt, kann auf den Typ Mutterklasse gecastet werden

Beispiel: super (1)

```
public class Figur {  
    private Punkt anker;  
    ...  
    public String toString() {  
        return "Anker:(" + anker.getX() + "," +  
                    anker.getY() + ")";  
    }  
}
```

```
public class Kreis extends Figur {  
    private int radius;  
    ...  
    public String toString() {  
        return "Radius: " + radius + " Anker: " +  
                getAnker();  
    }  
}
```

soll die Methode **toString()** der Superklasse erledigen!

Beispiel: super (2)

```
public class Figur {  
    private Punkt anker;  
    ...  
    public String toString() {  
        return "Anker:(" + anker.getX() + "," +  
                    anker.getY() + ")";  
    }  
}  
  
public class Kreis extends Figur {  
    private int radius;  
    ...  
    public String toString() {  
        return "Radius: " + radius + " " +  
                    super.toString();  
    }  
}
```

Vorteile von super

Aufgaben, die Superklasse bereits erledigt hat, in der Subklasse **nicht nochmals** gemacht →

Prinzip **Don't Repeat Yourself**



- **effizienter**, da *Wiederholungen* in Programmcode *vermieden*
- **wartbarer**, da ev. Änderungen an *einer* Stelle (in der Superklasse) durchzuführen
- **robuster**, da Veränderungen in der Methode der Superklasse – in der Subklasse auch gleich sichtbar

Zugriff auf Großvater-Methoden?

Instanzmethode mit Hilfe von `super.methode()` in einer **Vererbungshierarchie** angesprochen →

ausgehend von der aktuellen Klasse alle darüber liegenden Klassen des Vererbungsbaums der Reihe nach solange durchsucht, bis *zum ersten Mal* eine Instanzmethode `methode()` gefunden

d.h. Aufruf einer Instanzmethode `methode()` der Großvaterklasse über `super.methode()` nur dann möglich, wenn `methode()` in der Vaterklasse **nicht überschrieben**
→ ansonsten durch Aufruf `super.methode()` die Methode `methode()` der Vaterklasse (*erste* in der Hierarchie) aufgerufen

Klassenmethoden überschreiben

überschriebene Klassenmethode `methode()` der Superklasse –
in der Subklasse aufgerufen durch:

`Superklasse.methode()`

Beispiel:

- in **Figur**: `static int methodeA (int p1, char p2)`
- in **Kreis**: `static int methodeA (int p1, char p2)`

Aufrufe in der Klasse **Kreis**:

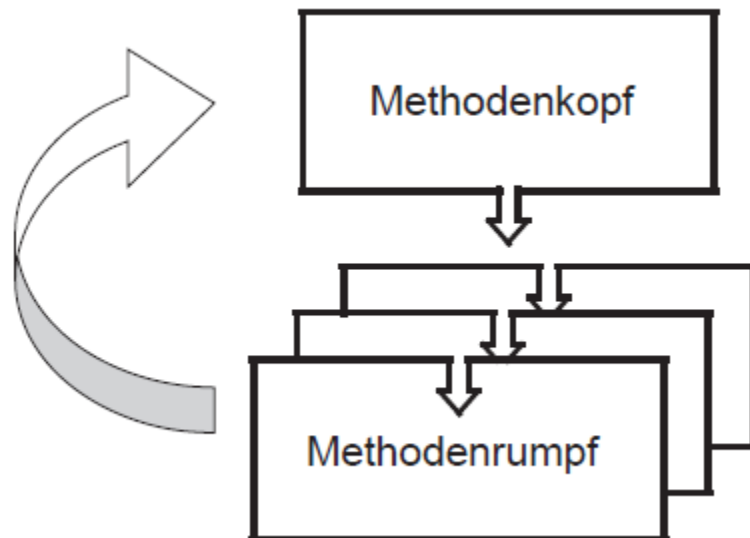
```
Figur.methodeA(2, 'a');  
methodeA(2, 'a');
```

Bindung

Bindung:

Zuordnung eines Methodenrumpfes zu einem aufgerufenen Methodenkopf

Aufruf einer Methode (Methodenkopf) → der Programmcode des entsprechenden Methodenrumpfes gesucht und ausgeführt



Frühe vs. späte Bindung

Zuordnung Methodenkopf → Methodenrumpf bei OO-Sprachen zu unterschiedlichen Zeitpunkten:

- **frühe** (**statische**) Bindung: Zuordnung **zur Kompilierzeit**
- **späte** (**dynamische**) Bindung: Zuordnung **zur Laufzeit**

Bindung in Java

- kein direkter Einfluss auf Bindung in Java
- **private**-Instanzmethoden: **statisch gebunden**
 - nur in der eigenen Klasse aufgerufen → Zuordnung zur Klasse für **Compiler** bereits **zur Kompilierzeit** klar
- Klassenmethoden: **statisch gebunden**
 - stets eindeutig, zu welcher Klasse eine Klassenmethode gehört → **Compiler** entscheidet **zur Kompilierzeit**, welche Methode aufgerufen wird
- **final**-Methoden: **statisch gebunden**
 - von Subklassen nicht überschreibbar → Zuordnung zur Klasse für **Compiler** bereits **zur Kompilierzeit** klar
- **public**-Instanzmethoden: **dynamisch gebunden**
 - **Interpreter** entscheidet **zur Laufzeit**, von welchem Typ das aufrufende Objekt ist und welche Methode aufgerufen wird

Verständnisfragen

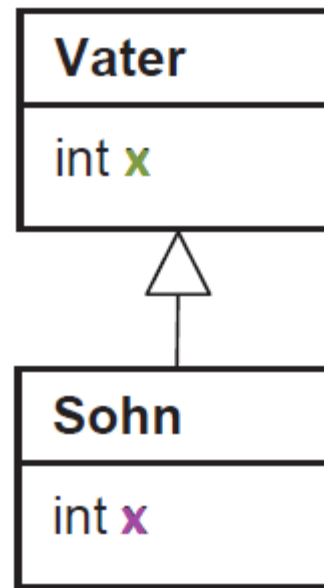


- Was ist eine polymorphe Methode?
- Was passiert, wenn innerhalb der Vererbungshierarchie polymorphe Methoden verwendet werden?
- Wie können private Methoden der Superklasse in der Subklasse überschrieben werden?
- Wird `methodeA()` in der Subklasse A überschrieben, welche Methode wird in A durch `super.methodeA()` aufgerufen?
- Was sind die Vorteile des Aufrufs der Methoden über `super`?
- Werden überschriebene Klassenmethoden der Superklasse in der Subklasse auch mithilfe von `super` aufgerufen?
- Was ist frühe und späte Bindung der Methoden?
- Wie werden Instanzmethoden in Java gebunden und was sind die Konsequenzen davon?

Verdecken von Instanz- und Klassenvariablen

Verdecken von Instanzvariablen

- in der Sohnklasse eine Instanzvariable **mit gleichem Namen** (z.B. x) definiert, wie eine von der Vaterklasse vererbte Instanzvariable → Instanzvariable **x** der Sohnklasse **verdeckt** die Instanzvariable **x** der Vaterklasse

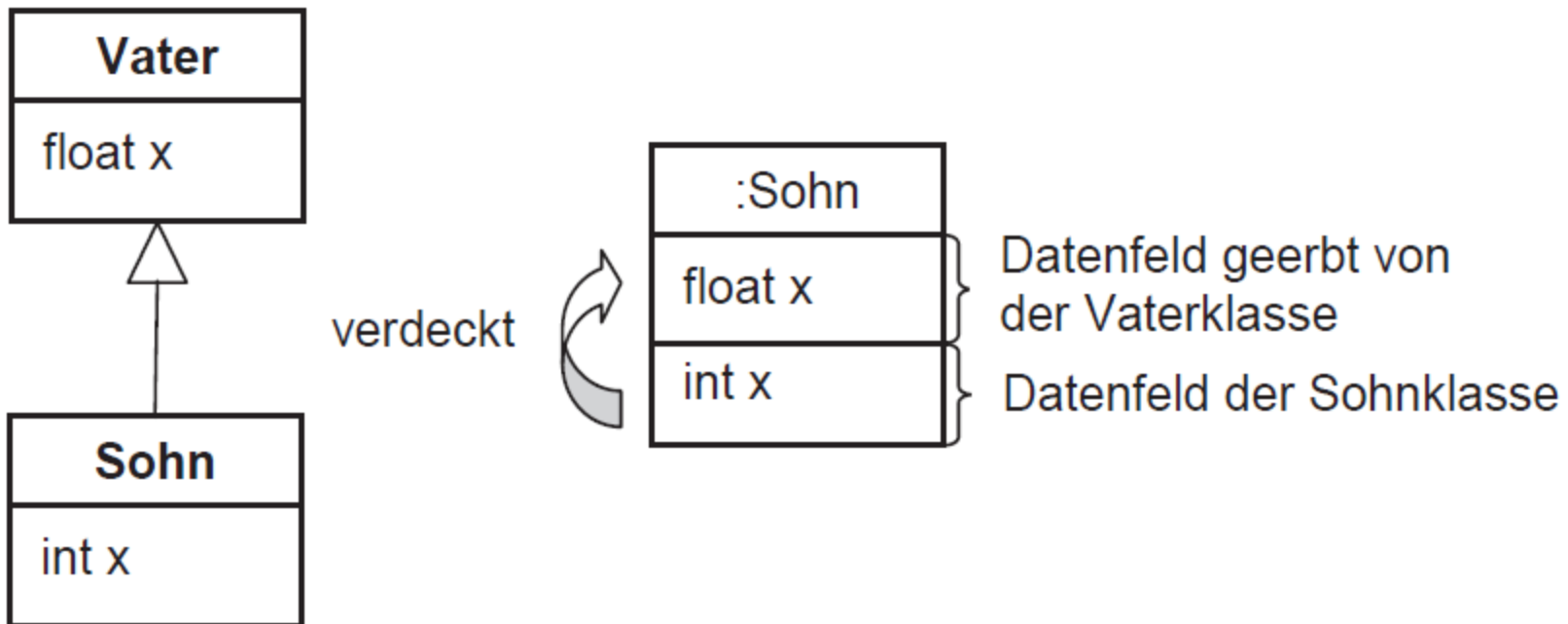


→ über x (oder `this.x`) in der Sohnklasse immer auf **x** zugegriffen

Verdecken nur über Namen



Datenfelder auch bei *unterschiedlichen Datentypen* verdeckt!



Zugriff über super

Zugriff in der Sohnklasse auf das Datenfeld **x** der Vaterklasse:

`super.x`

oder: `((Vaterklasse)this).x`

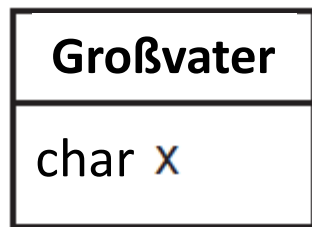
→ `super.x` in `((Vaterklasse)this).x` vom Compiler automatisch umgesetzt →

`this` = Referenz auf den Typ Vaterklasse gecastet



funktioniert nur dann, wenn Datenfeld **x** der Vaterklasse nicht **private**!

Zugriff auf verdeckte Großvater-Instanzvariablen



`super.super.x` gibt es nicht!

`((Großvater)this).x`

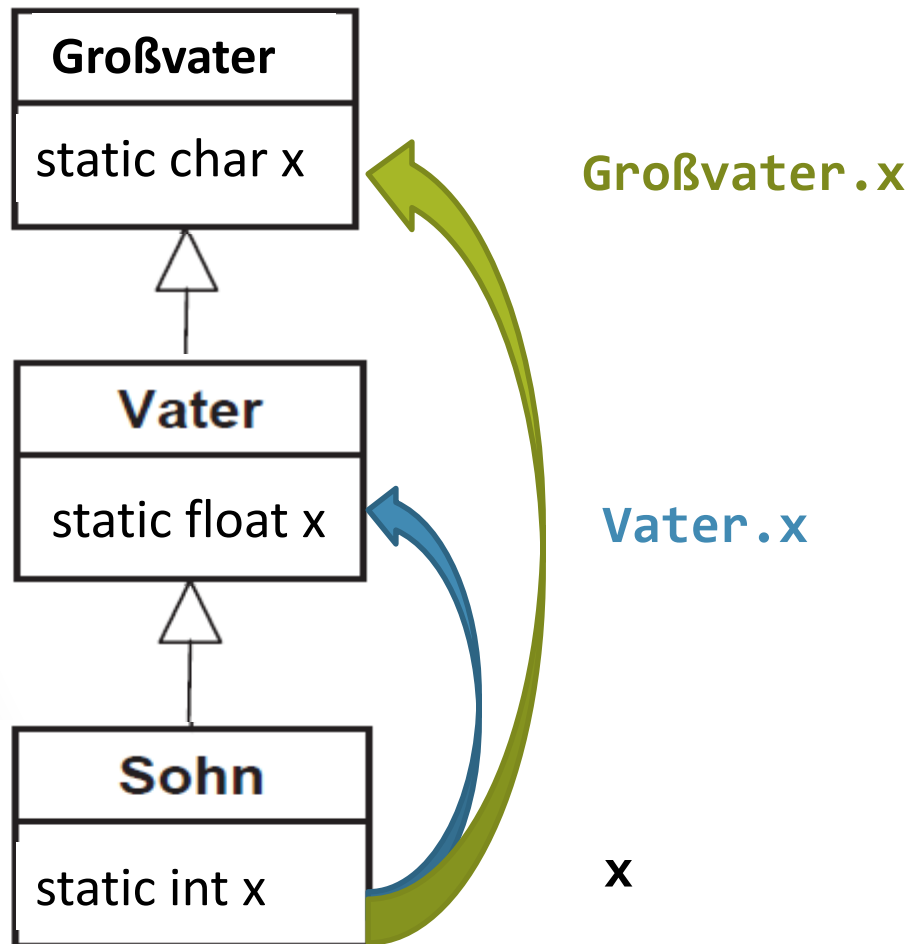
`super.x` oder `((Vater)this).x`

`x` oder `this.x`

Verdecken von Klassenvariablen

- Klassenvariablen – wie Instanzvariablen – können genauso **verdeckt** werden

Zugriff auf verdeckte Klassenvariablen

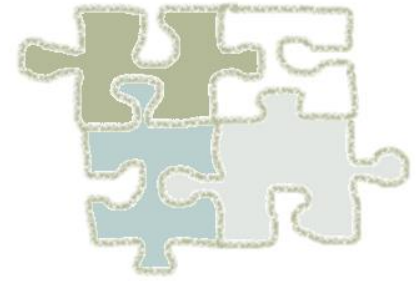





Verständnisfragen



- Wie können Datenfelder der Superklasse in der Subklasse verdeckt werden?
- Wie greift man auf verdeckte Instanz- und Klassenvariablen der Superklasse(n) zu?

Zusammenfassung



- Konstruktoren können in einer Klasse überladen werden.
- **Eigener** *parameterloser* Konstruktor soll in einer Klasse implementiert werden. 
- Überschriebene *Instanzmethode* können mit Hilfe von **super**.methode() in einer Vererbungshierarchie angesprochen werden. 
- Instanz- und Klassenvariablen können in einer Vererbungshierarchie *verdeckt* werden (**zu vermeiden!**) → angesprochen mithilfe von super oder dem Klassennamen
- Vererbung und Exceptions: *eigene* Exceptionklassen können (**nur!**) bei Bedarf implementiert werden: abgeleitet von Klasse Exception (**checked**) oder RuntimeException (**unchecked**)
- (insbes.) Konstruktoren und Setter *defensiv* implementieren 

Links

- 37 - Super keyword in Java
<https://www.youtube.com/watch?v=jUJ4CMPFkM4>
- Was ist Polymorphie in Java?
<https://www.youtube.com/watch?v=j9QfB4Sf1pA>
- Java Super Keyword Tutorial - Super Keyword for Variables, Constructors and Methods
<https://www.youtube.com/watch?v=hLYOpvoM4vk>