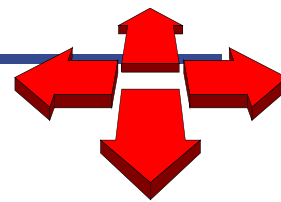




Übersicht Implementationsvererbung

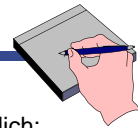


- Motivation
- Erben und Erweitern
- Erben und Anpassen
- Abstrakte Methoden und abstrakte Klassen
- Selbstaufrufe
- Vererbung und Konstruktoren
- Die Erbenschnittstelle

Motivation: Wiederverwendung

- **Wiederverwendung** von bereits entwickelter Software ist ein wichtiger Aspekt effizienter Softwareentwicklung.
 - „Das Rad nicht jedesmal neu erfinden.“
- In objektorientierten Systemen gibt es zwei grundsätzliche Formen von Wiederverwendung:
 - Einfache Benutzung – „**Use**“
 - systematisch in Software-Bibliotheken
 - Implementationsvererbung – „**Reuse**“
 - Erben zum Anpassen
 - Erben zum Vermeiden von Redundanz
 - Erben zum Vervollständigen
- **Implementations-** oder **Code-Vererbung** (engl.: inheritance) ist das Organisieren von **ausführbaren** Software-Elementen in Hierarchien.

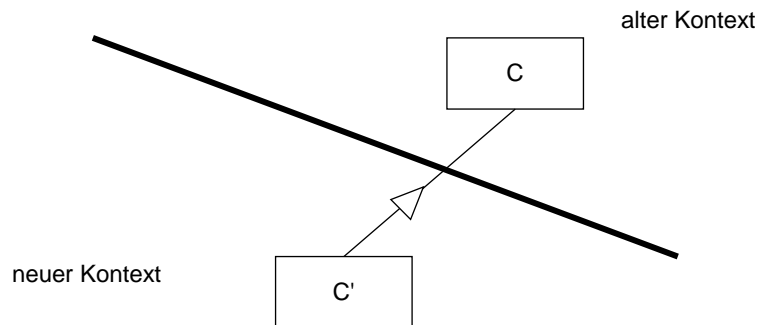
Generalisierung und Spezialisierung



- Zwei prinzipielle Sichtweisen auf Implementationshierarchien sind möglich:
 - **Generalisierung** (bottom-up):
 - Gleiche Eigenschaften mehrerer Klassen lassen sich generalisieren und als eine eigene Software-Einheit formulieren.
 - Gemeinsamkeiten werden zusammengefasst.
 - Die Oberklasse wird aus bereits existierenden Klassen „herausgezogen“.
 - Dient dem Vermeiden von Redundanz.
 - **Spezialisierung** (top-down):
 - Neue Unterklassen werden als Spezialisierungen von bestehenden Klassen formuliert.
 - Als Unterklassen oder abgeleitete Klassen verfeinern/erweitern sie bereits existierende Konzepte.
 - Wiederverwendung im Sinne der Anpassung für andere/neue Kontexte.

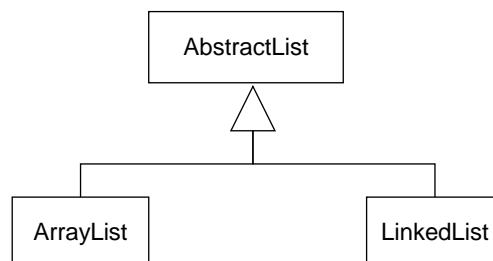
Erben zum Anpassen

- In einen neuen Kontext passt eine bestehende Klasse nicht vollständig hinein, kann aber mit kleineren **Modifikationen** auf die neuen Anforderungen zugeschnitten werden.
- Meist eine 1-zu-1-Beziehung



Erben zum Vermeiden von Redundanz

- Statt gleiche oder ähnliche Teile eines Systems redundant (an mehreren Stellen) zu realisieren, sollen **Gemeinsamkeiten in Oberklassen** zusammengefasst werden.
- Typischerweise eine 1-zu-n-Beziehung
- Beispiel: Java Collections Framework

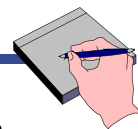


Erben zum Vervollständigen

- Der überwiegende Teile eines gewünschten Verhaltens wurde in einer Reihe von Klassen festgelegt, mit Unterklassen sollen offene Stellen ergänzt werden (engl. auch **code injection**); der Weg zu **Rahmenwerken** (engl.: **frameworks**).
- Typisches Beispiel: Grafische Benutzungsoberflächen

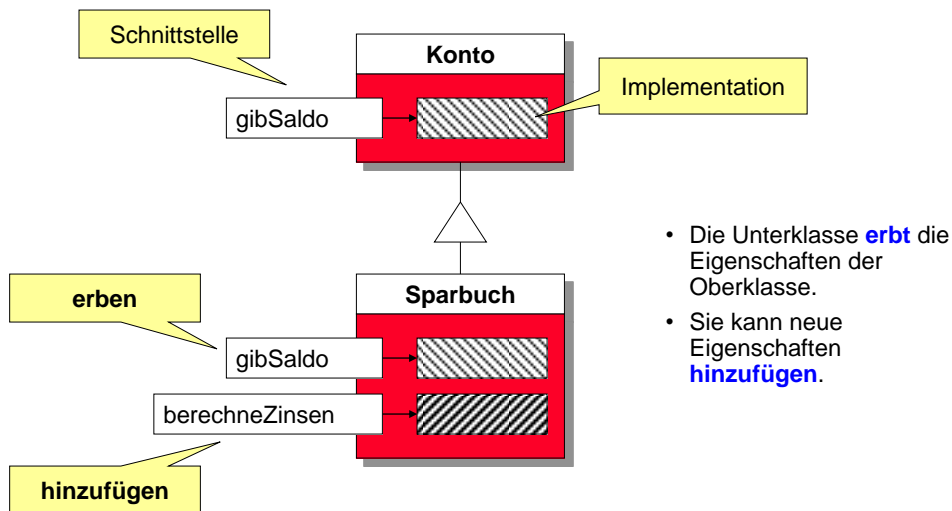


Übersicht: Umgang mit geerbten Eigenschaften



- **Ober- und Unterklasse**: Alle Eigenschaften der Oberklasse sind durch Vererbung zunächst auch Eigenschaften einer Unterklasse; sie bilden einen festen Bestandteil der Unterklasse.
 - Die geerbten **Felder** der Oberklasse werden **vollständig** und (üblicherweise) **unverändert** übernommen; weitere können in Unterklassen hinzugefügt werden.
 - Die geerbten **Methoden** können in einer Unterklasse spezialisiert werden; dabei werden **Operationen**
 - **hinzugefügt**, wenn noch keine signaturgleiche Operation in einem Supertyp existiert;
 - **redefiniert**, wenn eine neue Methode in der Unterklasse eine gleichnamige Methode einer Oberklasse ersetzt (**überschreibt** oder **erweitert**);
 - **definiert** (durch eine Methode implementiert), für die in der Oberklasse keine Methode angegeben war.

Erben und hinzufügen



SE2 – OOPM – Teil 1

9

Vererbung zwischen Klassen in Java

- In Java geschieht die Implementationsvererbung ausschließlich über Beziehungen zwischen Klassen, die mit **extends** formuliert werden.
- Vererbungsbeziehung herstellen:
 - Sparbuch** erbt alle Operationen und Felder der Klasse **Konto** durch das Schlüsselwort **extends**.
 - Sparbuch** ergänzt die geerbten Operationen um zwei neue: **gibZinssatz** und **berechneZinsen**.

```

class Konto {
    public Konto() {...}
    public void einzahlen(float b) {...}
    public void auszahlen(float b) {...}
    public float gibSaldo() {...}
}
  
```

```

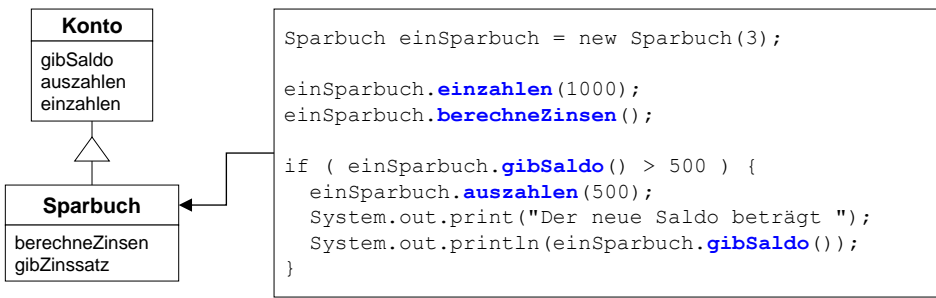
class Sparbuch extends Konto {
    public Sparbuch(Zinssatz z) {...}
    public Zinssatz gibZinssatz() {...}
    public void berechneZinsen() {...}
}
  
```

SE2 – OOPM – Teil 1

10

Verwendung von Geerbtem

- Benutzung von geerbten Eigenschaften:
 - Wird ein Exemplar der Klasse **Sparbuch** erzeugt, **enthält** dieses **alle Zustandsfelder** von **Sparbuch** und **Konto**.
 - An einem Exemplar der Klasse **Sparbuch** können neben allen Operationen, die in **Sparbuch** definiert werden, **auch alle Operationen, die die Klasse Konto anbietet**, aufgerufen werden.

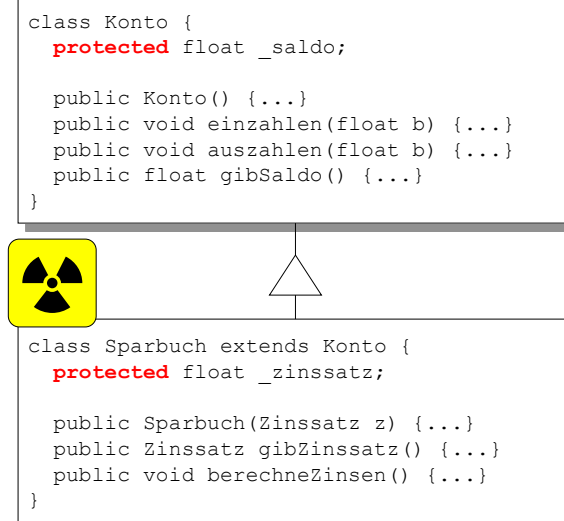


SE2 – OOPM – Teil 1

11

Private Eigenschaften und Vererbung

- Private Eigenschaften (Methoden und Felder) werden zwar mitvererbt, sie sind in einer Unterklasse jedoch nicht zugreifbar.
- Im Beispiel:
 - Damit das Feld **_saldo** in der Klasse **Sparbuch** zugreifbar ist, deklarieren wir es nicht **private**, sondern nur **protected**.
 - Es ist dann für erbende Klassen zugreifbar, aber nicht für „normale“ Klienten.
 - Sparbuch** erweitert die geerbten Felder um das Feld **_zinssatz**.



SE2 – OOPM – Teil 1

12

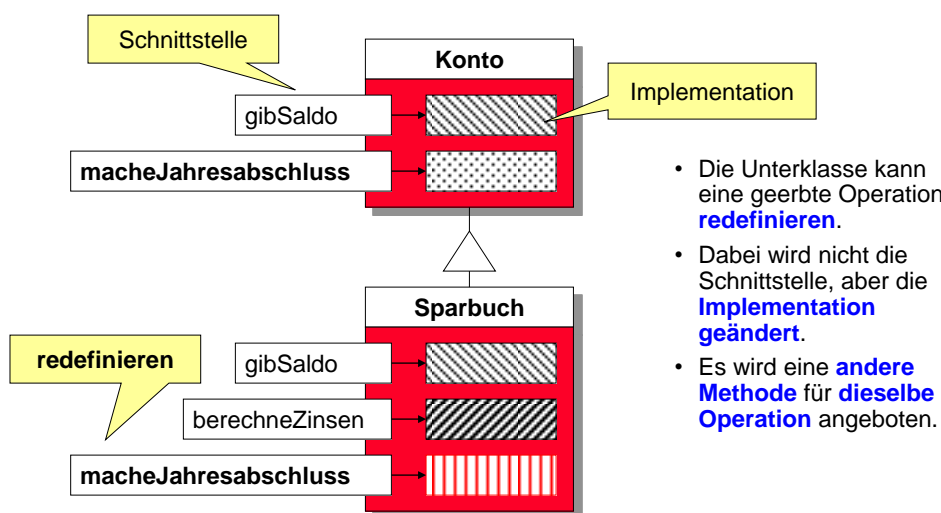
Klasse Object als Oberklasse

- Und wenn ich nicht erbe?
 - Jede Klasse, die nicht explizit (durch eine `extends`-Klausel) von einer anderen Klasse erbt, **beerbt implizit die Klasse Object**, die Teil des Java-Sprachkerns ist.
 - Die Klasse **Object** ist damit direkte oder indirekte **Oberklasse aller anderen Klassen**.
 - Der **Typ Object** definiert somit **Operationen**, die aufgrund dieser impliziten Vererbungsbeziehung **jeder Typ** hat.

```
public String toString() // Darstellung als String
public boolean equals(Object other) // Vergleich
public int hashCode() // Hashcode berechnen
...
```

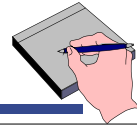
- Die **Klasse Object** implementiert diese Operationen durch ihre Methoden, die in erbenden Klassen verändert werden können.

Erben und redefinieren



- Die Unterklasse kann eine geerbte Operation **redefinieren**.
- Dabei wird nicht die Schnittstelle, aber die **Implementation geändert**.
- Es wird eine **andere Methode** für **dieselbe Operation** angeboten.

Redefinieren: Überschreiben



Eine Operation der Oberklasse kann redefiniert werden, indem die Operation in der Unterklasse **erneut deklariert** und **implementiert** wird. Wird dabei die redefinierte Methode **nicht** aufgerufen, sprechen wir von **Überschreiben**.

```
class Konto {
    public void macheJahresabschluss()
    {
        _saldo = _saldo - _gebuehren;
    }
    ...
}

class Sparbuch extends Konto {
    @Override
    public void macheJahresabschluss()
    {
        _saldo = _saldo - _gebuehren;
        _saldo = _saldo + berechneZinsen();
    }
    ...
}
```

Seit Java 1.5: Die Annotation `@Override` zum Deklarieren, dass redefiniert werden soll.

Die Operation aus Sparbuch wird aufgerufen!

```
Sparbuch einSparbuch;
einSparbuch = new Sparbuch(3.0f);

einSparbuch.macheJahresabschluss();
```

Redefinieren: Erweitern (mit super-Aufruf)

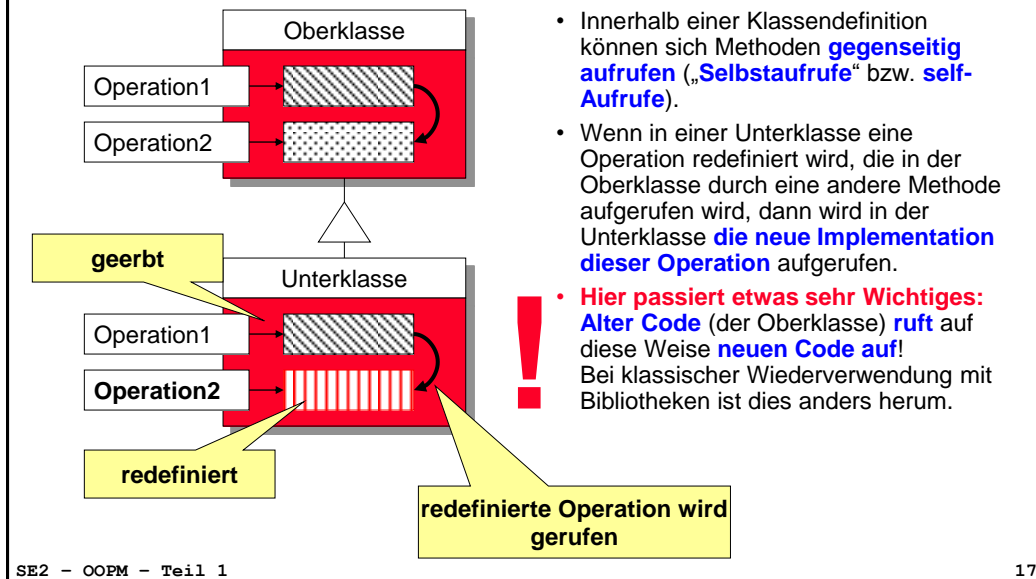


- Aufruf von Methoden der Oberklasse:
 - Mit dem **Schlüsselwort** **super** wird bei der Redefinition einer Operation die Methode der Oberklasse gerufen; auf diese Weise wird die Methode nicht komplett überschrieben, sondern **erweitert**.
 - Einen „**super.super**-Aufruf“ gibt es in Java nicht, deshalb kann man **nur die** Implementation **der direkten Oberklasse** erreichen!

```
class Konto {
    public void macheJahresabschluss()
    {
        _saldo = _saldo - _gebuehren;
    }
    ...
}

class Sparbuch extends Konto {
    @Override
    public void macheJahresabschluss()
    {
        super.macheJahresabschluss();
        _saldo = _saldo + berechneZinsen();
    }
    ...
}
```


Aufrufe redefinierter Operationen



17

Aufruf einer redefinierten Operation: Beispiel

- Die Klasse **Konto** definiert eine Operation **druckeJahresbericht**:
 - Diese ruft unter anderem **makeJahresabschluss** auf.
- In der Klasse **Sparbuch** wird **makeJahresabschluss** redefiniert, aber **druckeJahresbericht** nicht verändert.
- Bei einem Aufruf von **druckeJahresbericht** an einem **Sparbuch** wird jedoch aus der geerbten Methode die überschriebene Methode **makeJahresabschluss** aufgerufen!

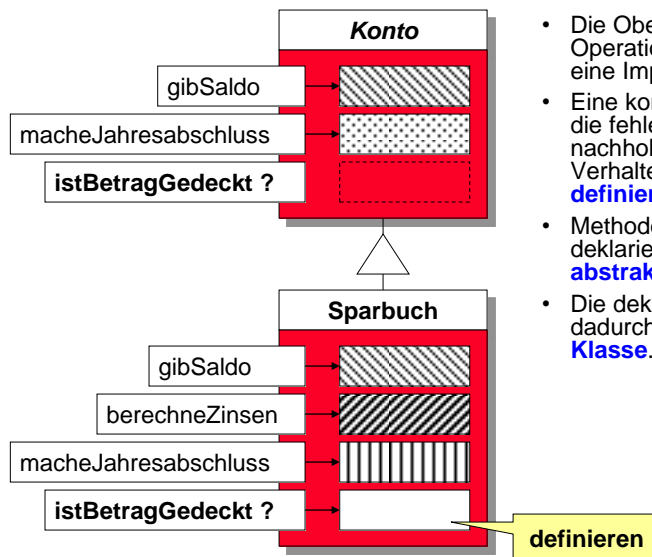
```
class Konto {
    public void druckeJahresbericht()
    {
        makeJahresabschluss();
        // Ergebnisse geeignet drucken
        ...
    }
    ...
}

class Sparbuch extends Konto {
    @Override
    public void makeJahresabschluss()
    {
        super.makeJahresabschluss();
        _saldo = _saldo + berechneZinsen();
    }
    ...
}
```

SE2 – OOPM – Teil 1

18

Abstrakte Methoden: Deklarieren ohne Definieren



- Die Oberklasse kann eine Operation **deklarieren**, ohne eine Implementation anzugeben.
- Eine konkrete Unterklasse muss die fehlende Implementation nachholen, indem sie das Verhalten in einer Methode **definiert**.
- Methoden, die Operationen nur deklarieren, nennt man **abstrakte Methoden**.
- Die deklarierende Klasse wird dadurch zu einer **abstrakten Klasse**.

SE2 – OOPM – Teil 1

19

Abstrakte Klassen

- Szenario:
 - Die Klasse **Konto** deklariert eine Operation **IstBetragGedeckt**, die sie **nicht implementiert**. Ob ein auszuzahlender Betrag gedeckt ist oder nicht, hängt von der jeweiligen Kontoart ab und kann daher nur **in den konkreten** abgeleiteten Klassen **Sparbuch** und **Girokonto** **sinnvoll ausimplementiert** werden.
- **Abstrakte Methoden** sind in Java mit dem Schlüsselwort **abstract** gekennzeichnet.
- Eine **Klasse**, die eine abstrakte Methode enthält, muss ebenfalls mit dem Schlüsselwort **abstract** deklariert werden.


```

abstract class Konto
{
    public abstract boolean istBetragGedeckt( float b );
    ...
}
  
```

SE2 – OOPM – Teil 1

20

Abstrakte Methoden und ihre Definition



```

abstract class Konto
{
    protected float _saldo;
    ...
    public abstract boolean istBetragGedeckt(float b);
}

class Sparbuch extends Konto
{
    ...
    @Override
    public boolean istBetragGedeckt(float b)
    {
        return (_saldo >= b);
    }
}

```

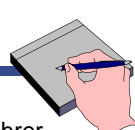
Methoden, die in einer Klasse nur eine Operation **deklarieren**, sie aber nicht implementieren, werden mit dem Schlüsselwort **abstract** markiert.

Die Klasse Sparbuch **definiert** die in Konto zunächst nur deklarierte Operation **istBetragGedeckt**, indem sie sie mit einer Methode **implementiert**.

Da die Exemplarvariable **_saldo** in der Oberklasse **protected** deklariert ist, kann in allen abgeleiteten Klassen, also auch in Sparbuch, **direkt darauf zugegriffen** werden.

SE2 – OOPM – Teil 1 21

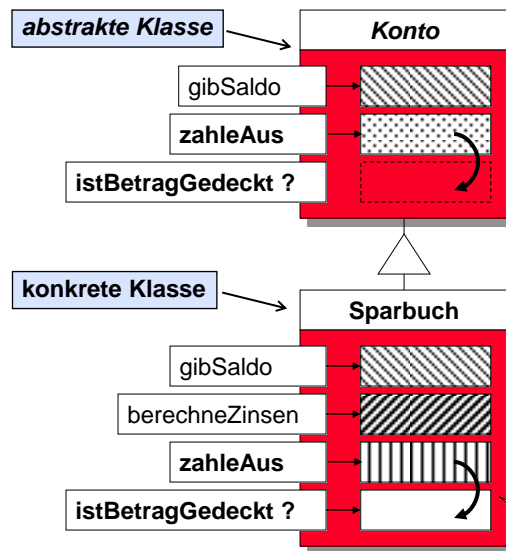
Eigenschaften abstrakter Klassen

- 
- Eine abstrakte Klasse kann Operationen deklarieren, die lediglich in ihrer Schnittstelle festgelegt sind.
 - Wenn ein Exemplar einer solchen Klasse erzeugt würde, würden bei einem Aufruf dieser Operationen **keine Implementierungen existieren**.
 - Um diesen Fehlerfall zu vermeiden, darf man in Java **keine Exemplare** von abstrakten Klassen **erzeugen**.
 - Eine Unterklasse einer abstrakten Klasse muss **auch abstrakt deklariert sein**, wenn sie nicht alle Operationen implementiert.
 - Eine Klasse ohne abstrakte Methoden wird auch eine **konkrete Klasse** genannt.



In Java kann jede Klasse als abstrakt deklariert werden, auch wenn sie keine abstrakten Methoden enthält!

Abstrakte Methoden sind aufrufbar!



- Eine **abstrakte Methode** kann aus einer nicht abstrakten Methode heraus **aufgerufen werden**.
- Aufgrund der Regeln für abstrakte Klassen ist dies **kein Konflikt zur Laufzeit**, da nur Exemplare von konkreten Klassen erzeugt werden dürfen, in denen garantiert alle Operationen implementiert sind.
- Der Aufruf einer abstrakten Methode wird zur Laufzeit also immer funktionieren.



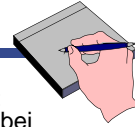
alle Operationen definiert!

Wie kann das überhaupt funktionieren?

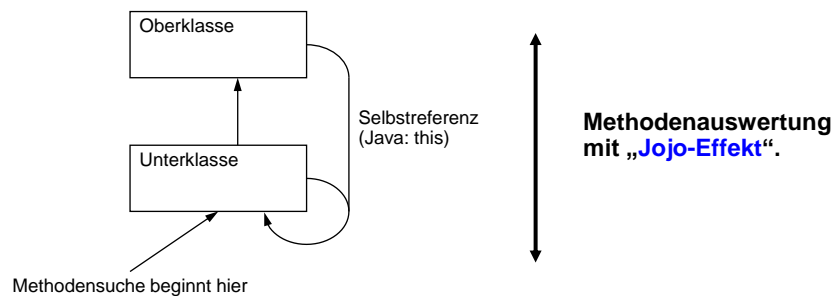
- Zwei **mentale Modelle** erleichtern das Verständnis:
 - „Erben“ heißt „Quelltext kopieren und neu übersetzen“
 - Das Jojo-Modell
- **Quelltext kopieren und neu übersetzen:**
 - Unmittelbar einsichtig: Wenn eine aufgerufene Methode in einer Unterklasse redefiniert wurde, wird dort die redefinierte Methode aufgerufen; wenn die Methode nicht redefiniert wurde, wird die ursprüngliche (kopierte) Methode aufgerufen.
 - Als mentales Modell nützlich, aber selten in Programmiersprachen realisiert (z.B. in Sather).
- Das **Jojo-Modell:**
 - Auch Selbst-Aufrufe werden dynamisch gebunden. Die Suche nach einer geeigneten Implementation beginnt dabei immer in der Klasse des aktuellen Objekts und sucht dann in der direkten Oberklasse, wenn in der durchsuchten Klasse keine Implementation der Operation vorliegt.
 - Unter anderem in Java realisiert.



Auswertung von Selbst-Aufrufen in Java



- Ein Selbst-Aufruf wird **dynamisch** (also zur Laufzeit) **gebunden** (auch: **spät gebundene Selbstreferenz**, engl.: late-bound self-reference). Dabei wird immer in der Klasse des aktuellen Objektes mit der Methodensuche begonnen.



- Es kann bei der Methodensuche immer wieder dazu kommen, dass eine Methode einer Oberklasse gewählt wird. Aber auch von dort wird jeder Selbstaufwurf wieder in der Klasse des aktuellen Objektes begonnen.

Schablonenmethode und Einschubmethode



- Das Verhältnis zwischen einer aufrufenden (also nicht abstrakten) Methode und der aufgerufenen abstrakten Methode wird im **Schablonenmuster** abstrahiert:
 - Eine **Schablonenmethode** (engl.: template method) ist eine konkrete Methode, die eine oder mehrere abstrakte Methoden aufruft. Sie legt üblicherweise einen **Ablauf**, einen **Algorithmus** oder ein **Teilverhalten** fest.
 - Eine **Einschubmethode** (engl.: hook method) ist eine abstrakte Methode, die meist zusammen mit einer Schablonenmethode vorkommt. Die erbende Klasse implementiert die Einschubmethode und konkretisiert so einen **Teil des vorgegebenen abstrakten Verhaltens**. Die Unterklasse „schiebt“ quasi konkretes Verhalten ein, daher der Name.
 - Einschubmethoden werden für den Zweck der **Code Injektion** definiert und sind eine wichtige Grundlage für **objektorientierte Rahmenwerke**.

Verwendung von abstrakten Oberklassen

```
abstract class AbstractTitelListe
{
    public abstract void append(Titel t);
    public abstract void start();
    public abstract void next();
    public abstract boolean empty();
    public abstract boolean off();
    public abstract Titel item();

    public boolean contains(Titel t){
        boolean ergebnis = false;

        if ( !empty() ) {
            start();
            while (!off() && (item() != t)){
                next();
            }
            ergebnis = !off();
        }
        return ergebnis;
    }
}
```

- Abstrakte Implementationen
 - Oft kann man auf Basis von abstrakten Methoden schon vollständige Algorithmen formulieren.
 - **contains** kann von der abstrakten Oberklasse **AbstractTitelListe** unter Rückgriff auf ihre abstrakten Methoden für die gesamte Klassenfamilie der Titel-Listen implementiert werden – in diesem Fall als **lineare Suche**.

Abstrakte Oberklassen zur Spezifikation

- Abstrakte Klassen können also für den Zweck definiert werden, **Teile des Verhaltens** von Unterklassen festzulegen.
- In Java kann dieses Verhalten den Unterklassen auch **zwingend vorgeschrieben** werden:

- Wenn eine **Operation** als **final** deklariert wird, dann kann sie in einer Unterklasse nicht redefiniert werden:

```
final void gibSaldo() { ... }
```

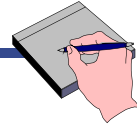
- Wenn eine ganze **Klasse** als **final** deklariert wird, dann können von ihr **keine Unterklassen** definiert werden:

```
final class String {
    ...
}
```



Die Klasse **String** ist in Java **final** deklariert.

Vererbung und Konstruktoren in Java



- Konstruktoren werden in Java **nicht vererbt**.
 - Eine ererbende Klasse bietet deshalb nicht automatisch die gleichen Konstruktoren wie ihre Oberklasse an!
- Bei der Erzeugung eines Objekts einer abgeleiteten Klasse werden jedoch die Konstruktoren **sämtlicher Oberklassen** - von der entferntesten bis zur direkten - **gerufen**.
- Ein Programmierer kann dies **explizit** anstoßen, indem er einen super-Aufruf als **erste Anweisung im Konstruktor** der Unterklasse formuliert; eventuell müssen dabei Parameter übergeben werden, wenn der gerufene Konstruktor der Oberklasse diese fordert.
- Fehlt in einem Konstruktor ein expliziter super-Aufruf, dann fügt der Compiler **automatisch einen parameterlosen Aufruf** (`super()`) ein – auch wenn dieser Aufruf möglicherweise fehlschlägt!
- Wenn in einer Klasse **A** gar kein Konstruktor angegeben ist, dann fügt der Compiler automatisch den folgenden Standard-Konstruktor ein:

```
public A() { super(); }
```

Beispiel: Eigene Exception-Klasse definieren

- Die Klasse **Exception** verfügt über ein String-Attribut, das mit einem Konstruktor gesetzt werden kann. Wir können es über einen super-Aufruf zum Ablegen eines **Fehlertextes** nutzen.
- Mit der von **Exception** geerbten Operation `getMessage()` kann ein Klient diesen Text auslesen.

```
class RangeException extends Exception
{
    public RangeException() { super(); }
    public RangeException(String s){ super(s); }
    public RangeException(int i)
    {
        super("Access at position " + i + " was invalid.");
    }
}
```

Weiteres Beispiel für Konstruktoren und Vererbung

```
class Punkt {
    private double _x, _y;

    public Punkt() {
        this(0.0, 0.0);
    }

    public Punkt(double x, double y) {
        _x = x;
        _y = y;
    }
    ...
}

class Pixel extends Punkt {
    private Farbe _farbe;
    ...
    public Pixel(double x, double y, Farbe f) {
        super(x,y);
        _farbe = f;
    }
}
```

eigener Standardkonstruktor, der den 2-stelligen Konstruktor **Punkt** ruft.

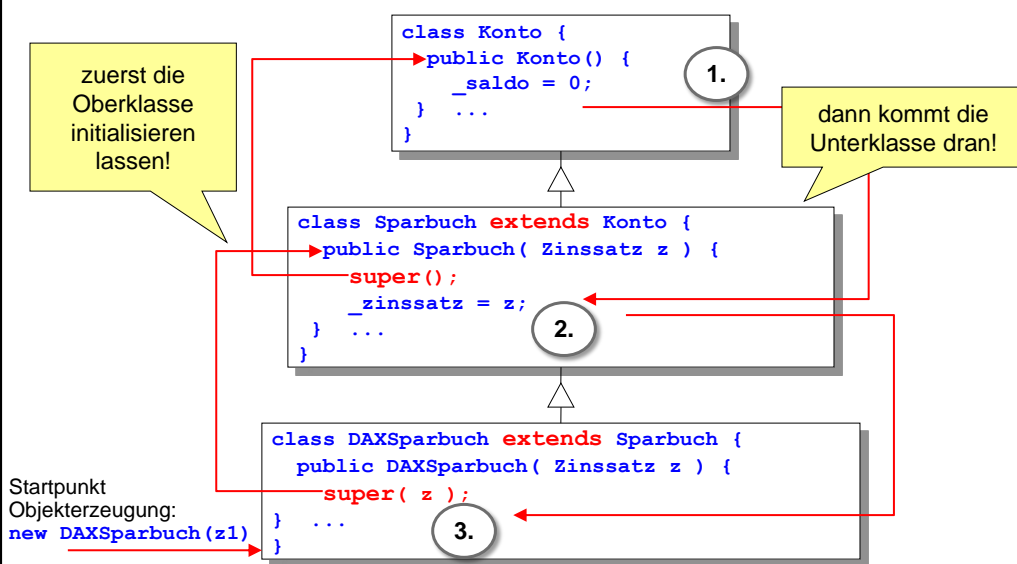
2-stelliger Konstruktor **Punkt** initialisiert die Exemplarvariablen.

Konstruktor **Pixel** ruft den Konstruktor der Oberklasse **Punkt**.

SE2 – OOPM – Teil 1

31

Konstruktoren und Vererbung, mehrstufiges Beispiel

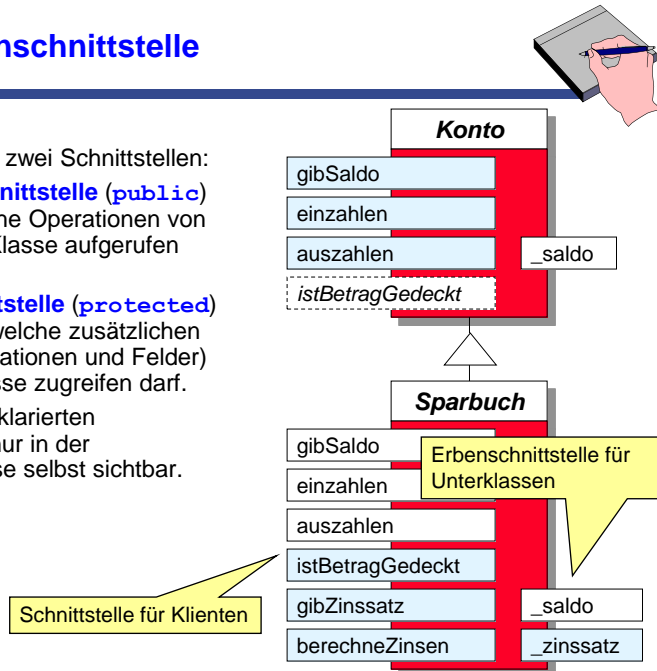


SE2 – OOPM – Teil 1

32

Klienten- und Erbenschnittstelle

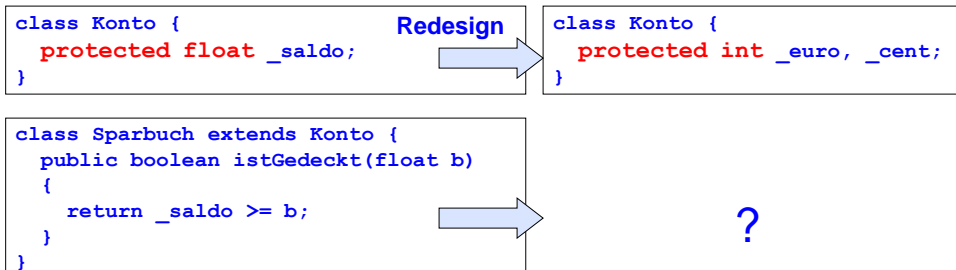
- Eine Klasse verfügt über zwei Schnittstellen:
 - An der **Klientenschnittstelle** (**public**) wird festgelegt, welche Operationen von einem Klienten der Klasse aufgerufen werden können.
 - An der **Erbenschnittstelle** (**protected**) wird festgelegt, auf welche zusätzlichen Eigenschaften (Operationen und Felder) eine abgeleitete Klasse zugreifen darf.
 - Alle als **private** deklarierten Eigenschaften sind nur in der deklarierenden Klasse selbst sichtbar.



SE2 – OOPM – Teil 1

33

Modellierung der Erbenschnittstelle



- Problem:
 - Bei unserem aktuellen Entwurf haben **Sparbuch** und **Girokonto** **direkten Zugriff** auf die Exemplarvariable **_saldo** der Oberklasse.
 - Auf diese Weise **durchbrechen wir die Datenkapselung** in **Konto** und machen unseren Entwurf **änderungsanfälliger**.
 - Die **Erbenschnittstelle** sollte die interne Speicherstruktur von **Konto** deshalb **genauso gut kapseln**, wie es auch für die Klientenschnittstelle gefordert wird.

SE2 – OOPM – Teil 1

34

Modellierung der Erbenschnittstelle

```
class Konto {
    private float _saldo;
    protected float gibSaldo() {
        return _saldo;
    }
}
```

Redesign

```
class Konto {
    private int _euro, _cent;
    protected float gibSaldo() {
        return _euro + (float)_cent/100;
    }
}
```

```
class Sparbuch extends Konto {
    public boolean istGedeckt(float b)
    {
        return gibSaldo() >= b;
    }
}
```

Die Unterklasse bleibt unverändert - trotz der Umstellung der internen Struktur!

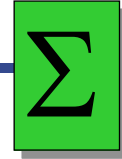


Werden die Exemplarvariablen **durch Zugriffsoperationen** auch gegenüber den erbenenden Klassen **gekapselt**, so kann die Repräsentation in der Oberklasse geändert werden, ohne dass die Unterklasse von dieser Änderung betroffen ist!

Aufgepasst: Der Operator instanceof in Java

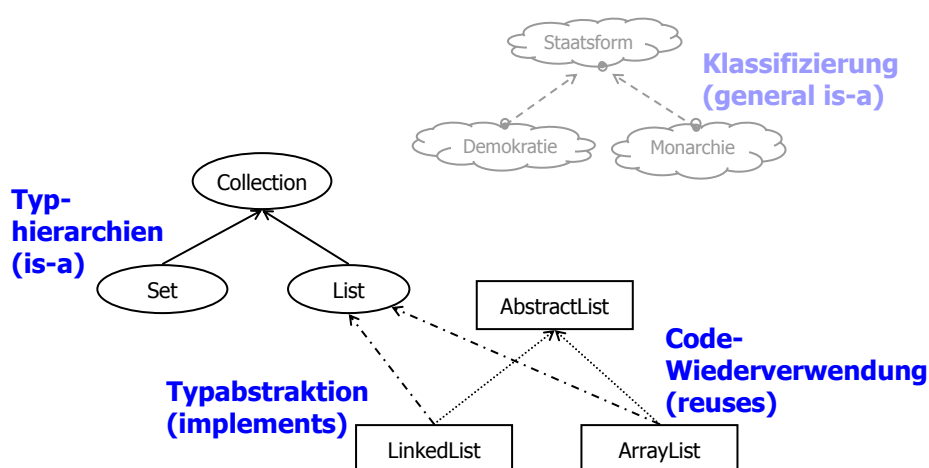
- In SE1 haben wir gesagt, dass der Operator `instanceof` genau dann `true` liefert, wenn eine gegebene Referenz auf ein **Exemplar der genannten Klasse** verweist.
- Dies ist angesichts der zusätzlichen Möglichkeiten durch Subtyping und Implementationsvererbung nicht mehr ganz korrekt. Eigentlich wird getestet, ob eine gegebene Referenz auf ein **Exemplar des genannten Typs oder eines seiner Subtypen** verweist.
- Beispielsweise liefert
`ref instanceof Object`
 für jedes Exemplar in Java `true`, auch für solche, die nicht direkte Exemplare der Klasse `Object` sind!
- Immerhin haben wir den Operator in SE1 unter dem Begriff **Typtest** bereits korrekt eingeordnet...

Zusammenfassung Implementationsvererbung

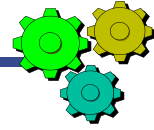


- Mit Implementationsvererbung wird ermöglicht,
 - ... **gemeinsames Verhalten** verschiedener Klassen in einer Oberklasse **zusammenzufassen**.
 - ... dass eine Unterklasse den Code, den sie erbt, um eigene Operationen **erweitert**.
 - ... dass eine Unterklasse den Code, den sie erbt, auch für ihre speziellen Zwecke **anpasst**, indem sie Operationen redefiniert.
- **Abstrakte Klassen** deklarieren für einige Operationen nur ihre Schnittstelle, geben aber nicht ihre Implementation vor.
- Durch Implementationsvererbung wird eine Unterscheidung zwischen **Klientenschnittstelle** und **Erbenschnittstelle** notwendig.
- Java-spezifisch:
 - Alle **Typen** (Klassen und Interfaces) erben die **Operationen** des Typs **Object**, alle **Klassen** erben die Methoden der Klasse **Object**.
 - **Konstruktoren** werden nicht vererbt.

Übersicht revisited: Zentrale „Vererbungs“-konzepte

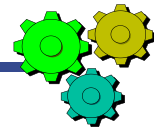


Diskussion Vererbung et al.



- Die Vererbungskonzepte in objektorientierten Programmiersprachen werden methodisch eingesetzt zur
 - Modellierung von Begriffshierarchien (in Subtyp-Hierarchien),
 - Trennung von Spezifikation und Implementation (Typabstraktion),
 - inkrementellen Übernahme und Veränderung vorhandener ausführbarer Software-Einheiten (Implementationsvererbung),
 - Abstraktion gemeinsamer Merkmale (Redundanzvermeidung auf Code-Ebene).

Diskussion Vererbung et al. (2)



- Auf Implementationsebene hat sich Einfachvererbung durchgesetzt, während auf die flexiblen Entwurfsmöglichkeiten durch multiples Subtyping (siehe Interfaces in Java) kaum noch verzichtet werden kann.
- Objektorientierte Vererbungskonzepte spielen schon beim fachlichen Entwurf eine große Rolle.
- Bei der Konstruktion großer Systeme nutzen wir
 - Interfaces zur Spezifikation von Schnittstellen,
 - Klassen zur abstrakten und konkreten Implementierung.