

Programmiertechnik 1

Teil 5: Java Objektorientierung
Kapselung, Vererbung, Polymorphie, Dynamische Bindung

Programmiertechnik 1 - Teil 5

1) Programmierparadigmen

- 2) Kapselung und Klassenvererbung
- 3) Blick in die Java Klassenbibliothek: Oberklassen
- 4) Schnittstellenvererbung
- 5) Blick in die Java Klassenbibliothek: Schnittstellen
- 6) Polymorphie und dynamische Bindung
- 7) Empfehlungen

Programmierparadigmen: Prozedurale Programmierung



Am Anfang der Programmiertechnik stand die **prozedurale Programmierung**

Kennzeichen:

- Algorithmen und Funktionsbibliotheken

Vorgehen:

- Programmierer denken in Abläufen, entwerfen Algorithmen
- gesucht ist der Algorithmus mit dem geringsten Laufzeit- und Speicherbedarf, der ein gegebenes Problem löst

Erhoffter Nutzen:

- Rechner können alles berechnen, wofür Menschen zu langsam sind

Suchen in großen Datenbeständen

Sortieren großer Datenbestände

Lösen komplizierte Gleichungssysteme

Optimieren von Lösungen

...

Programmierparadigmen: Strukturierte Programmierung

Höhere Programmiersprachen brachten die **strukturierte Programmierung**

Kennzeichen:

- Kontrollstrukturen und Datentypen

Vorgehen:

- Programmierer strukturieren Abläufe innerhalb der Funktionen nach vorgegebenen Mustern (*siehe Ablaufsteuerung in Teil 3*) 
auch mit dem Schlagwort "goto-freie Programmierung" bezeichnet
- Programmierer strukturieren Daten mit Hilfe von Datentypen 

Erhoffter Nutzen:

- bessere Lesbarkeit der Programme
textuelle Reihenfolge und Ausführungsreihenfolge stimmen weitgehend überein
- mehr Sicherheit beim Programmieren
Datentypen erlauben Fehlerentdeckung schon zum Übersetzungs-Zeitpunkt

Programmierparadigmen: Modulare Programmierung

Immer größer werdende Programme führen zur **modularen Programmierung**

Kennzeichen:

- **Module** sowie **Lokalitäts- und Geheimnisprinzip** (*Information Hiding*)

Vorgehen:

- Programmierer gruppieren logisch zusammengehörende Funktionen und Daten in getrennt übersetzbaren Modulen
in Java umsetzbar mit Paketen und Utility-Klassen
- Programmierer verbergen Implementierungsdetails, die für die Benutzung eines Moduls unerheblich sind, hinter der Modul-Schnittstelle
in Java umsetzbar mit Zugriffsrechten

Erhoffter Nutzen:

- bessere Überschaubarkeit der Software durch Strukturierung im Grossen
- größere Änderungsfreundlichkeit der Software
die verborgene Implementierung eines Moduls kann jederzeit geändert werden

Programmierparadigmen: Objektorientierte Programmierung

Stand der Technik ist die **objektorientierte Programmierung**
(heute oft in Kombination mit funktionaler Programmierung)

Kennzeichen:

- **Kapselung** (*encapsulation*)
- **Vererbung** (*inheritance*)
- **Polymorphie** (*polymorphism*)
- **Dynamische Bindung** (*dynamic dispatch*)



Vorgehen:

- Programmierer denken in Objekten und Objektbeziehungen statt in Abläufen
Objekte haben einen Zustand (→ Instanzvariablen) und ein Verhalten (→ Methoden)
gleichartige Objekte gehören zur selben Klasse,
ähnliche Klassen haben gemeinsame Oberklassen
- Gesucht ist ein System von Klassen, das die Struktur einer gegebenen Problemdomäne angemessen widerspiegelt (→ *Domänenmodell*)
aus Programmierern werden Modellierer

Erhoffter Nutzen:

- bessere Wiederverwendbarkeit der Software



Programmiertechnik 1 - Teil 5

- 1) Programmierparadigmen
- 2) Kapselung und Klassenvererbung**
- 3) Blick in die Java Klassenbibliothek: Oberklassen
- 4) Schnittstellenvererbung
- 5) Blick in die Java Klassenbibliothek: Schnittstellen
- 6) Polymorphie und dynamische Bindung
- 7) Empfehlungen

Kapselung: Klassen mit privaten Instanzvariablen

- für in Objekten gekapselte Daten können konsistente Werte garantiert werden:

```
public class Termin {   
    private Datum wann; } private Instanzvariablen sind gekapselt, weil sie nur in den Methoden der Klasse zugreifbar sind  
    private final String was; }  
  
    public Termin(Datum wann, String was) {  } Konstruktoren garantieren konsistente Initialisierung der Instanzvariablen  
        ...  
    }  
  
    public final void verschieben(Datum wohin) {   
        ...  
    } öffentliche Methoden lassen nur Konsistenz erhaltende Zugriffe zu  
    public final Datum wann() {  
        ...  
    }  
    ...  
}
```

Klassenvererbung: Oberklassen und Unterklassen

Klassenvererbung erlaubt die **Definition ähnlicher Klassen**, indem man die Gemeinsamkeiten in Oberklassen zusammenfasst und die Eigenheiten in davon abgeleiteten Unterklassen ergänzt

- **Oberklassen** vererben alle Variablen und nicht-privaten Methoden an ihre Unterklassen

Objekte einer Oberklasse können deshalb durch Objekte der Unterklassen ersetzt werden

- **Unterklassen** können ihre Oberklasse(n) erweitern, indem sie weitere Variablen und Methoden hinzufügen

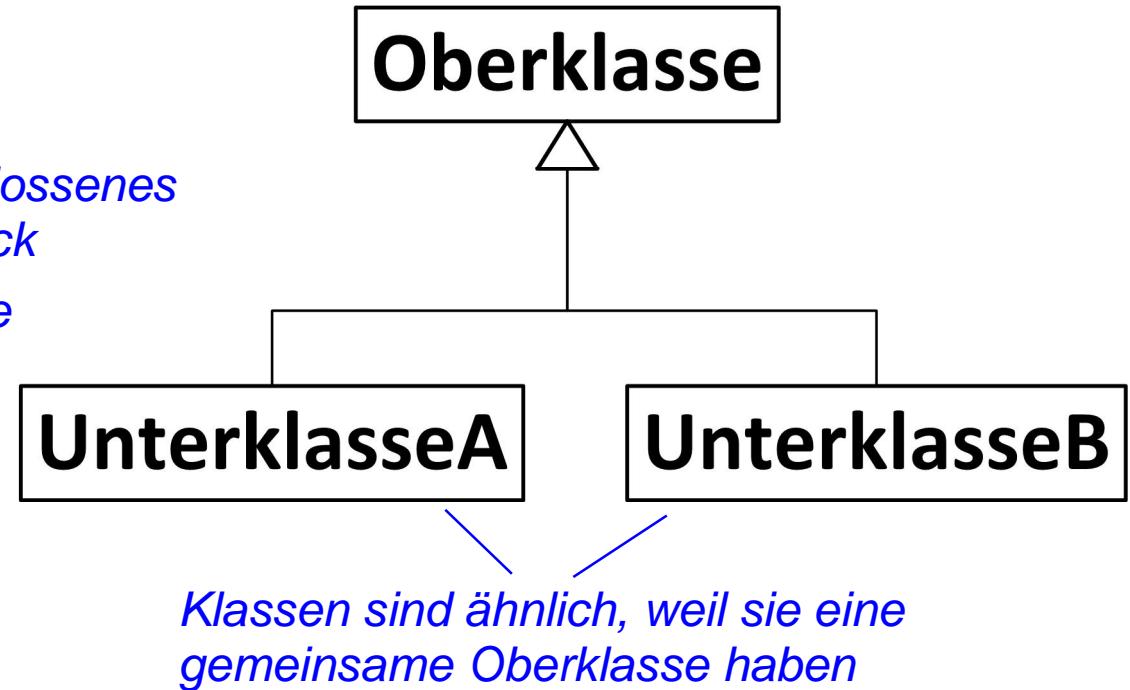
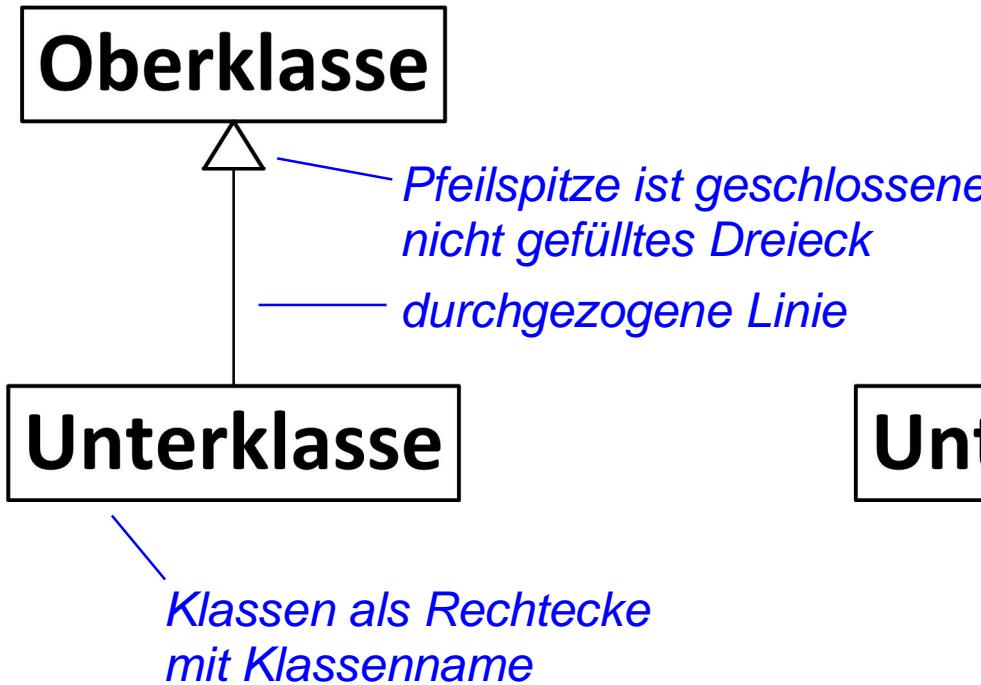
Objekte einer Unterklasse können deshalb nicht durch Objekte der Oberklasse ersetzt werden

- **Unterklassen** können Instanzmethoden ihrer Oberklasse(n) überschreiben, d.h. anders implementieren (engl. Overriding)

insbesondere können in der Oberklasse weggelassene Implementierungen in Unterklassen nachgeholt werden

Klassenvererbung: Grafische Darstellung mit UML

In der grafischen Modellierungssprache UML (**Unified Modeling Language**) wird Klassenvererbung als Pfeil von der Unterkasse zur Oberklasse gezeichnet:



Java Oberklassen: Eigenschaften und Syntax (1)

- Klassen ohne **final**-Markierung sind optional als Oberklasse verwendbar:

```
hier kein final
public class Oberklassename {
    Zugriffsrecht Typ Variable;
    Zugriffsrecht Rückgabetyp ÜberschreibbareMethode(...) {
        ...
    }
    Zugriffsrecht final Rückgabetyp NichtÜberschreibbareMethode(...) {
        ...
    }
    ...
}
```

Implementierung ist endgültig,
d.h. sie darf in Unterklassen nicht ersetzt werden

Java Oberklassen: Eigenschaften und Syntax (2)

- **Abstrakte Klassen** sind ausschließlich als Oberklasse verwendbar: 

```
public abstract class Oberklassenname {  
    ... // alles wie zuvor ebenfalls erlaubt  
  
    // zusätzlich darf es abstrakte Methoden ohne Rumpf geben:  
    Zugriffsrecht abstract Rückgabetyp NichtImplementierteMethode(...);  
    ...  
}
```

*hier abstract statt final
(dadurch kein new Oberklasse(...) mehr möglich)*

*Implementierung der Methode
wird an die Unterklassen delegiert* 

*hier Semikolon
statt Rumpf*

Java Oberklassen: Eigenschaften und Syntax (3)

- Zugriffsrechte für Oberklassen in aufsteigender Ordnung:

private	<i>nur innerhalb der Oberklasse zugreifbar</i>
<i>/* ohne */</i>	<i>zusätzlich aus allen Klassen des gleichen Pakets zugreifbar</i>
protected	<i>zusätzlich aus allen Unterklassen dieser Oberklasse zugreifbar</i>
public	<i>aus allen Klassen zugreifbar</i>

*Zugriffsrecht **private** ist bei **abstract** markierten Methoden nicht erlaubt*

Java Unterklassen: Eigenschaften und Syntax (1)

- in Java nur **Einfachvererbung** (Unterklassen erweitern nur eine Oberklasse):

 ohne final könnte die Unterklasse selbst wieder Oberklasse sein

```
public final class Unterklassename extends Oberklassename {  
    Zugriffsrecht Typ zusätzlicheVariable;  
  
    Zugriffsrecht Rückgabetyp zusätzlicheMethode(...) {  
        ...  
    }  
    @Override  
    Zugriffsrecht Rückgabetyp ÜberschreibbareMethode(...) {  
        ...  
    }  
    ...  
}
```

 nur mit dieser **Annotation** prüft der Compiler nach, dass tatsächlich eine Methode der Oberklasse überschrieben wird

 das Zugriffsrecht darf nicht strenger als in der Oberklasse sein

Java Unterklassen: Eigenschaften und Syntax (2)

- in Java ist eine Klasse direkte Unterklasse von **java.lang.Object**, wenn keine andere Oberklasse angegeben ist:

```
public final class Klassename extends Object {  
    ...  
}
```

*wird vom Compiler ergänzt,
wenn nicht explizit angegeben*

- indirekt ist damit jede Klasse Unterklasse von `java.lang.Object`

java.lang.Object vererbt unter anderem die Methoden `toString()`, `equals(Object)` und `hashCode()` (siehe Teil 4) 

Java Unterklassen: Eigenschaften und Syntax (3)

- eine Unterklasse kann Ihre Oberklasse über **super** ansprechen:

super();	}	<i>Aufruf eines Konstruktors der Oberklasse (nur als erste Anweisung in Konstruktoren erlaubt)</i>
super(Argumentliste);		<i>Aufruf einer überschriebenen Methodenimplementierung der Oberklasse</i>
super.Methode(Argumentliste);		

- jeder Konstruktor, dessen Rumpf nicht mit einem **this(...)**-Aufruf beginnt, muss als erste Anweisung einen Konstruktor der Oberklasse aufrufen:

```
public final class Unterklassename extends Oberklassename {  
    Zugriffsrecht Unterklassename( Parameterliste ) {  
        super( Argumentliste );  
        ...  
    }  
    ...  
}
```

*fehlt der **super(Argumentliste)** -Aufruf,
ergänzt der Compiler einen Aufruf des
Standardkonstruktors der Oberklasse:
super();*

Beispiel-Programm Vererbung



Unterklasse *Oberklasse*

```
public final class OrtsTermin extends Termin {  
    private final String wo; // zusätzliche Instanzvariable  
    public OrtsTermin(String wo, Datum wann, String was) {  
        super(wann, was); // geerbte Instanzvariablen der Oberklasse initialisieren  
        ... // zusätzliche Instanzvariable der Unterklassse initialisieren  
    }  
    public String wo() {  
        return this.wo; } // zusätzliche Methode der Unterklassse  
    @Override  
    public String toString() {  
        return String.format("%s, %s",  
            this.wo, super.toString()); } // überschriebene Methode der Oberklasse  
}
```

Programmiertechnik 1 - Teil 5

- 1) Programmierparadigmen
- 2) Kapselung und Klassenvererbung
- 3) Blick in die Java Klassenbibliothek: Oberklassen**
- 4) Schnittstellenvererbung
- 5) Blick in die Java Klassenbibliothek: Schnittstellen
- 6) Polymorphie und dynamische Bindung
- 7) Empfehlungen

Java Standard-Bibliothek: Oberklasse `java.lang.Throwable` (1)

Die Klasse **`java.lang.Throwable`** ist Oberklasse aller Ausnahmen. 

- nur Objekte der Klasse Throwable bzw. Objekte von deren Unterklassen können als Ausnahmen mit **`throw`** geworfen und mit **`catch`** gefangen werden

*Ausnahmen können eine Fehlermeldung in Form eines Strings, eine Fehlerursache in Form einer anderen Ausnahme sowie unterdrückte Folge-Ausnahmen enthalten
(außerdem speichern Sie die Programmstelle, an der das `throw` passiert ist)*

- Konstruktoren:

```
public Throwable()
public Throwable( String message )
public Throwable( Throwable cause )
public Throwable( String message, Throwable cause )
```

*Ausnahme ohne Meldung
und ohne Ursache*

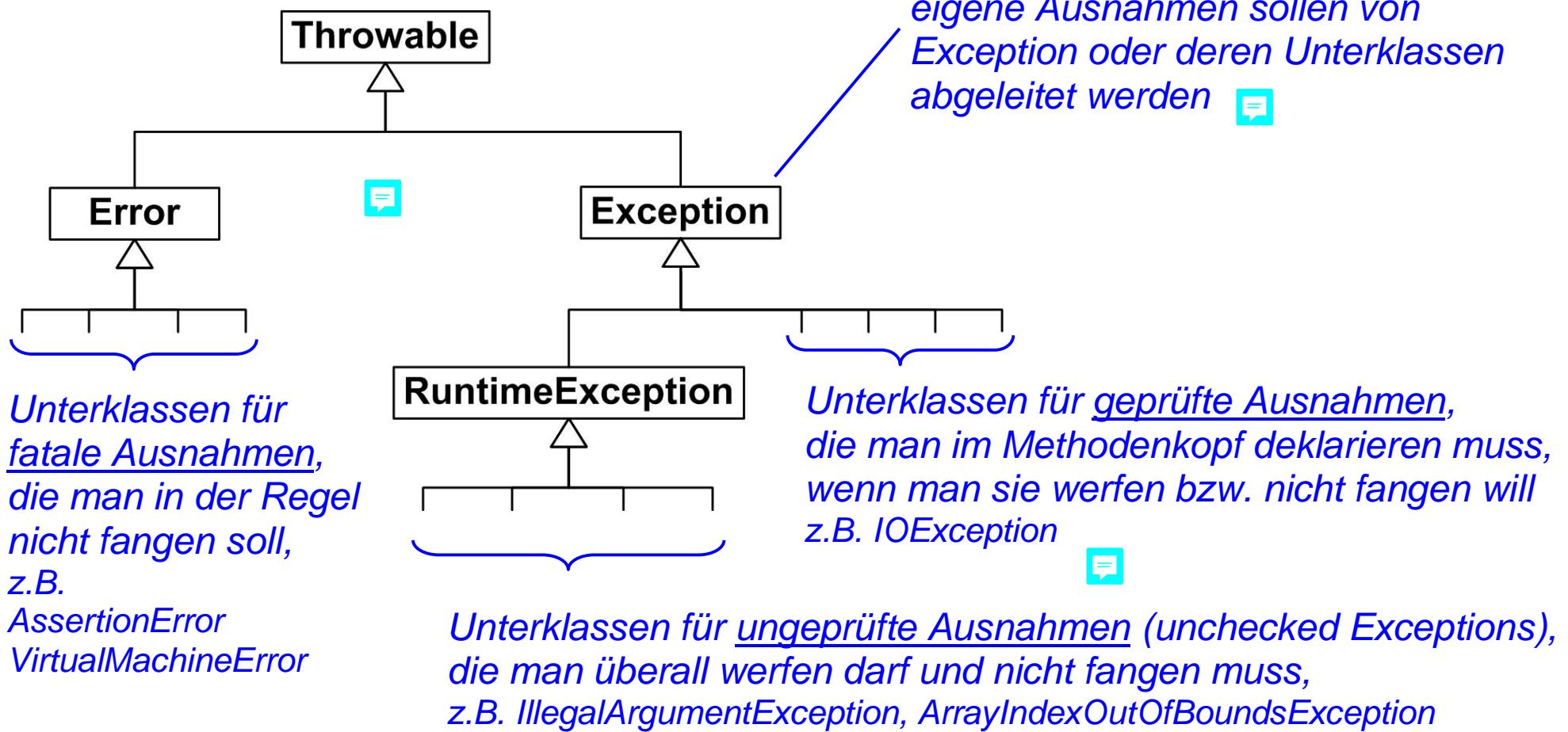
- Instanzmethoden:

```
public Throwable getCause()
public String getMessage()
public void printStackTrace() 
...
```

*wird bei nicht gefangenen Ausnahmen
vor dem Programmabbruch aufgerufen*

Java Standard-Bibliothek: Oberklasse `java.lang.Throwable` (2)

- Unterklassen im Paket `java.lang`:



Java Standard-Bibliothek: Oberklasse `java.lang.Number`

Die Klasse `java.lang.Number` ist Oberklasse der Wrapper-Klassen für Zahlen:

- Klassendefinition im Paket `java.lang`:

```
public abstract class Number {  
    public abstract int intValue();  
    public abstract long longValue();  
    public abstract float floatValue();  
    public abstract double doubleValue();  
  
    public byte byteValue() {  
        return (byte) intValue();  
    }  
    public short shortValue() {  
        return (short) intValue();  
    }  
}
```

abstrakte Methoden müssen in den Unterklassen implementiert werden

Implementierungen der konkreten Methoden werden an die Unterklassen vererbt

- Unterklassen im Paket `java.lang`:

Wrapper-Klassen `Byte`, `Double`, `Float`, `Integer`, `Long`, `Short`

Java Standard-Bibliothek: Oberklasse `java.text.NumberFormat`

Die Klasse **`java.text.NumberFormat`** ist eine abstrakte Oberklasse für die landes- und sprachabhängige String-Darstellungen von Zahlen:

- statische Fabrikmethoden:

```
public static final NumberFormat getInstance()
public static NumberFormat getInstance(Locale inLocale)
public static final NumberFormat getCurrencyInstance()
public static NumberFormat getCurrencyInstance(Locale inLocale)
```

*z.B. Locale.GERMAN
für deutsche Sprache*

*Formatierer für
Währungsbeträge*

...

je nach Fabrikmethode wird in der Regel ein unterschiedlich konfiguriertes Objekt der Unterklasse DecimalFormat geliefert

- Instanzmethoden:

```
public final String format(double number)
public abstract StringBuffer format(double n, StringBuffer sb, FieldPosition fp)
...
```

die `final`-Methode ruft die Unterklassenimplementierung der `abstract`-Methode auf

Programmiertechnik 1 - Teil 5

- 1) Programmierparadigmen
- 2) Kapselung und Klassenvererbung
- 3) Blick in die Java Klassenbibliothek: Oberklassen
- 4) Schnittstellenvererbung**
- 5) Blick in die Java Klassenbibliothek: Schnittstellen
- 6) Polymorphie und dynamische Bindung
- 7) Empfehlungen

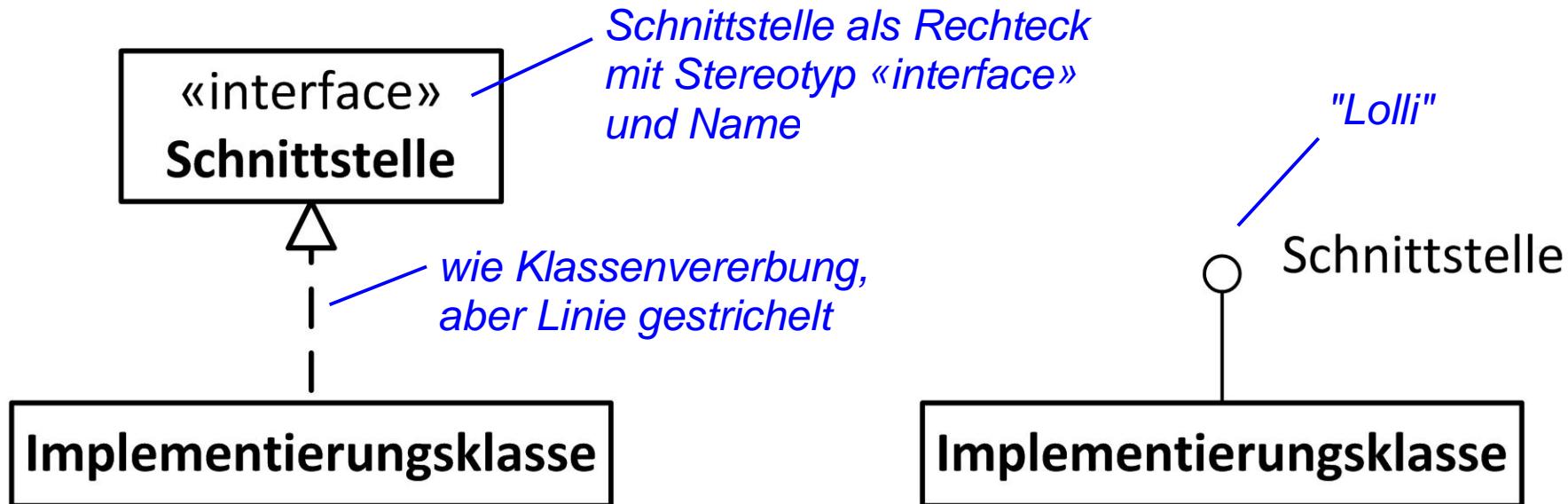
Schnittstellenvererbung: Schnittstellen und Implementierungen

Schnittstellenvererbung erlaubt die **Definition einheitlich benutzbarer Klassen**,
indem man öffentliche Methoden in Schnittstellen zusammenfasst
und diese Methoden in abgeleiteten Klassen unterschiedlich implementiert.

- **Schnittstellen** sind abstrakte Oberklassen mit ausschließlich öffentlichen Methoden sowie ohne Instanzvariablen und ohne Konstruktoren
Klassenvariablen sind auch nur erlaubt, wenn sie konstant sind (`static final`)
Instanzmethoden müssen entweder abstrakt sein oder die Implementierung muss mit `default` markiert sein
- **Implementierungsklassen** sind Unterklassen, die die Methoden von einer oder mehreren Schnittstellen implementieren
werden nicht alle Schnittstellenmethoden implementiert, so ist die Implementierungsklasse wiederum abstrakt und muss per Klassenvererbung vervollständigt werden
- Schnittstellenvererbung und Klassenvererbung können gemischt werden

Schnittstellenvererbung: Grafische Darstellung mit UML

In UML wird Schnittstellenvererbung als Pfeil von der Implementierungsklasse zur Schnittstelle oder als "Lolli" gezeichnet:



Java Schnittstellen: Eigenschaften und Syntax (1)

- Schnittstellendefinition mit Schlüsselwort **interface** statt **class**:

```
public interface Schnittstellenname {  
    Rückgabetyp SchnittstellenMethode(...);  
    ...  
}
```

public abstract wird vom Compiler automatisch ergänzt, wenn es fehlt

- Schnittstellenvererbung mit Schlüsselwort **implements** statt **extends**:

```
public final class Klassename implements Schnittstellenname {  
    @Override  
    public Rückgabetyp SchnittstellenMethode(...) {  
        ... // Implementierung  
    }  
    ...  
}
```

hier können mit Komma getrennt weitere Schnittstellen folgen

Java Schnittstellen: Eigenschaften und Syntax (2)

- Implementierungsklassen für sehr einfache Interfaces werden oft in Kurzschreibweise innerhalb eines `new`-Ausdrucks definiert:

```
Methodenkopf {  
    ...  
    Schnittstellename o = new Schnittstellename( ) {  
        // anonyme Implementierungsklasse = lokale Klasse ohne Name  
        @Override  
        public Rückgabetyp SchnittstellenMethode( ... ) {  
            ...  
        }  
        ...  
    };  
    ...  
}
```



Lokale Klassen sind eingebettete Klassen, die innerhalb einer Methode definiert werden. Sie haben Zugriff auf die konstanten Parameter und konstanten lokale Variablen dieser Methode sowie auf die Variablen und Methoden der umgebenden Klasse.

Beispiel-Programm Schnittstellen-Vererbung



```
public interface Formatter {  
    /* public abstract */ String format(int n);  
}
```

```
public final class DecimalFormat implements Formatter {  
    @Override  
    public String format(int n) {  
        return Integer.toString(n);  
    }  
}
```

*Formatierung als
Dezimalzahl mit Vorzeichen*

```
public final class UnsignedHexFormat implements Formatter {  
    @Override  
    public String format(int n) {  
        return Integer.toHexString(n);  
    }  
}
```

*Formatierung als Hexadezimalzahl
(negative Zahlen als Zweierkomplement)*

Programmiertechnik 1 - Teil 5

- 1) Programmierparadigmen
- 2) Kapselung und Klassenvererbung
- 3) Blick in die Java Klassenbibliothek: Oberklassen
- 4) Schnittstellenvererbung
- 5) **Blick in die Java Klassenbibliothek: Schnittstellen****
- 6) Polymorphie und dynamische Bindung
- 7) Empfehlungen

Java Standard-Bibliothek: Interface `java.lang(CharSequence`

Die Schnittstelle `java.lang(CharSequence` ermöglicht einen einheitlichen Lesezugriff auf Zeichenfolgen:

- Schnittstellenmethoden:

`char charAt(int index)`

`int length()`



`CharSequence subSequence(int start, int end)`

`String toString()`



seit Java 8 weitere Methoden (mit Default-Implementierung)



- Implementierungsklassen:

z.B. `java.lang.String` und `java.lang.StringBuilder`



Java Standard-Bibliothek: Interface `java.lang.Appendable`

Die Schnittstelle **`java.lang.Appendable`** ermöglicht ein einheitliches Aneinanderhängen von Zeichenfolgen:

- Schnittstellenmethoden:

- Schnittstellenmethoden:
 - Appendable `append(char c)`
 -  Appendable `append(CharSequence csq)`
 - Appendable `append(CharSequence csq, int start, int end)`

- Implementierungsklassen:

- z.B. `java.lang.StringBuilder` und `java.io.PrintStream`

Java Standard-Bibliothek: Interface `java.lang.Comparable<T>`

Die generische Schnittstelle `java.lang.Comparable<T>` ermöglicht ein  einheitliches Vergleichen von Objekten:

- Schnittstellenmethode:

 `int compareTo(T anotherObject)`

T ist ein Typparameter und wird mit dem Namen der jeweiligen Implementierungsklasse belegt

- Implementierungsklassen:

z.B. `java.lang.String` und alle Wrapper-Klassen (`java.lang.Float` usw.)

Eine Klasse C, die `Comparable<C>` implementiert, definiert eine Totalordnung (= lineare Ordnung) all ihrer Objekte. Diese Ordnung wird auch als die natürliche Ordnung der Klasse bezeichnet. 

Felder `C[]` können dann z.B. mit `java.util.Arrays.sort` sortiert werden.

Eine natürliche Ordnung ist in der Regel nur für Wertobjekte sinnvoll, nicht für Entitäten (für Entitäten besser Sortierkriterien mit Comparator-Funktionsobjekten definieren). 

Java Standard-Bibliothek: Interface `java.util.Comparator<T>`

Die generische Schnittstelle `java.util.Comparator<T>` ermöglicht die Definition unterschiedlicher Sortierkriterien für Objekte:

- Schnittstellenmethode:

 ✉ `int compare(T o1, T o2)`

- Implementierungsklassen:

 z.B. `java.text.Collator` ✉

T ist ein Typparameter und wird mit dem Klassennamen der zu sortierenden Objekte belegt

Seit Java 8 zusätzliche Methoden zum Erzeugen von Implementierungsobjekten in der Schnittstelle ✉

Collator definiert für Strings Totalordnungen, die von der natürlichen Ordnung abweichen, z.B. eine Ordnung, die deutsche Umlaute und Groß-/Kleinschreibung ignoriert.

Klassen, die Comparator<T> implementieren, können als Baupläne für Funktionsobjekte (function objects) verwendet werden. Funktionsobjekte sind im Prinzip Wertobjekte, die als "Wert" die Implementierung einer abstrakten Methode enthalten, hier compare(T, T).

Java Standard-Bibliothek: Interface `java.lang.Iterable<T>`

Die generische Schnittstellen `java.lang.Iterable<T>` und `java.util.Iterator<T>` ermöglichen Iterationen über Sammlungen mit Elementtyp T per for-each-Schleife:

- `java.lang.Iterable<T>` ist die Schnittstelle für die Erzeugung von Iteratoren:

 `Iterator<T> iterator()`

- `java.util.Iterator<T>` ist die Schnittstelle für die Benutzung von Iteratoren

 `boolean hasNext()`
`T next()`

- Implementierungsklassen von `java.lang.Iterable<T>`:

z.B. Listen-Klassen im Paket `java.util`

- Implementierungsklassen von `java.util.Iterator<T>`:

z.B. `java.util.Scanner`, anonyme Klassen im Paket `java.util`

*`T` ist ein Typparameter und wird mit dem Namen der Elementklasse der jeweiligen Sammlung belegt
Seit Java 8 in beiden Schnittstellen weitere Methoden (jeweils mit Default-Implementierung)*

Beispiel-Programm `java.lang.Iterable<T>` (1)

```
public final class IntList implements Iterable<Integer> { [Talk]
    private Element head = null;
    public IntList insert(int n) { ... }
    private static final class Element { ... } } [Talk] siehe Beispiel IntList  
in Teil 4

@Override
public Iterator<Integer> iterator() {
    return new Iterator<Integer>() { [Talk] anonyme Implementierungsklasse  
für die Iteration über die Listenelemente
        private Element current = IntList.this.head; [Talk]

@Override
public boolean hasNext() {
    return this.current != null;
}

...
```

Beispiel-Programm `java.lang.Iterable<T>` (2)

```
...

@Override
public Integer next() {
    if (this.current == null) {
        throw new NoSuchElementException();
    }
    Element e = this.current;
    this.current = this.current.next;
    return Integer.valueOf(e.n);
}
}; // Ende der return-Anweisung mit Instanziierung der anonymen Implementierungsklasse
} // Ende der Methode iterator
} // Ende der Klasse IntList
```

Beispiel-Programm `java.lang.Iterable<T>` (3)

```
public final class ListVar {  
    private ListVar() { }  
    public static void main(String[] args) {  
        int[] anIntArray = {3421, 3442, 3635, 3814};  
        IntList anIntList = new IntList();  
        for (int i = anIntArray.length; i > 0; --i) {  
            anIntList.insert(anIntArray[i - 1]);  
        }  
        // Liste ausgeben  
        for (int n : anIntList) {  
            System.out.println(n);  
        }  
        Iterator<Integer> i = anIntList.iterator();  
        while (i.hasNext()) {  
            int n = i.next();  
            System.out.println(n);  
        }  
    }  
}
```

*siehe Beispiel ListVar
in Teil 4*

Programmiertechnik 1 - Teil 5

- 1) Programmierparadigmen
- 2) Kapselung und Klassenvererbung
- 3) Blick in die Java Klassenbibliothek: Oberklassen
- 4) Schnittstellenvererbung
- 5) Blick in die Java Klassenbibliothek: Schnittstellen
- 6) Polymorphie und dynamische Bindung**
- 7) Empfehlungen

Polymorphie: Referenzen auf Oberklassen oder Schnittstellen

Polymorphie (*Vielgestaltigkeit*) bedeutet, dass ein und dieselbe Variable zur Laufzeit Objekte unterschiedlicher Klassen referenzieren kann:

- In der Literatur zur Unterscheidung von anderen Formen der Polymorphie auch **Subtyp-Polymorphie** oder kurz **Subtyping** genannt.
- in Java ist Polymorphie an Vererbung gekoppelt (**eingeschränkte Polymorphie**)
eine Variable kann nur Objekte referenzieren, deren Klasse genau dem Variablenotyp entspricht oder eine Unterklasse bzw. Implementierungsklasse des Variablen Typs ist

Number n; *polymorphe Variable, weil Number eine Oberklasse ist* 

```
if (...) {  
    n = Integer.valueOf(1);  
} else {  
    n = Double.valueOf(2.3);  
}
```

beides ok, weil sowohl Integer als auch Double Unterklassen von Number sind

Dynamische Bindung: polymorphe Methodenaufrufe



Dynamische Bindung (*dynamic dispatch*) bedeutet, dass sich erst zur Laufzeit entscheidet, welche Methodenimplementierung aufgerufen wird

- Aufrufe von Instanzmethoden über polymorphe Variablen kann der Compiler nicht auflösen:

```
Number n;  
if (...) {  
    n = Integer.valueOf(1);  
} else {  
    n = Double.valueOf(2.3);  
}  
int i = n.intValue();
```

zur Laufzeit muss die Klasse des Objekts *n* bestimmt werden, um entscheiden zu können, ob *Integer.intValue()* oder *Double.intValue()* aufzurufen ist

- bei Klassenmethoden und privaten Methoden gilt **statische Bindung**
die aufzurufende Methodenimplementierung steht hier schon zur Übersetzungszeit fest

Java Objektorientierung: Klasse versus Typ

Die Konzepte Klasse und Typ sind eng verwandt, aber nicht bedeutungsgleich:

- **Klassen** haben mit der Implementierung von Objekten zu tun
gleichartige Objekte gehören zur selben Klasse
ähnliche Objekte habe eine gemeinsame Oberklasse

*die Klasse eines Objekts bestimmt,
welche Instanzvariablen das Objekt enthält,
wie die Instanzvariablen mit Konstruktoren initialisiert werden und
wie sich das Objekt bei Methodenaufrufen verhält*



- **Typen** haben mit der Benutzung von Objekten zu tun
*der Typ einer Variablen bestimmt,
Objekte welcher Klassen referenziert werden können und
welche Methoden bei diesen Objekten aufgerufen werden können*
*wird vom Typ eines Objekts gesprochen, ist damit gemeint,
dass Variablen dieses Typs das Objekt referenzieren können*

Java Objektorientierung: Typanpassung

zwischen Referenzvariablen sind gewisse Typanpassungen (*Typecasts*) erlaubt:

- **Upcast** von Referenz auf UnterkLASSE zu Referenz auf OberKLASSE
findet bei Bedarf implizit statt (gegebenenfalls Fehlermeldung des Compilers)
- **Downcast** von Referenz auf OberKLASSE zu Referenz auf UnterkLASSE
muss explizit angegeben werden (gegebenenfalls zur Laufzeit ClassCastException)
- **Crosscast** von Referenz auf OberKLASSE zu Referenz auf andere OberKLASSE
muss explizit angegeben werden (gegebenenfalls zur Laufzeit ClassCastException)



Beispiel:

```
public final class C implements A, B { ... }
```

...

```
A a = new C(); // Upcast von C nach A
```

```
B b = (B) a; // Crosscast von A nach B
```

```
C c = (C) b; // Downcast von B nach C
```

*mit Operator instanceof
ClassCastException verhindern:
if (a instanceof B) {
 b = (B) a;*



Programmiertechnik 1 - Teil 5

- 1) Programmierparadigmen
- 2) Kapselung und Klassenvererbung
- 3) Blick in die Java Klassenbibliothek: Oberklassen
- 4) Schnittstellenvererbung
- 5) Blick in die Java Klassenbibliothek: Schnittstellen
- 6) Polymorphie und dynamische Bindung
- 7) Empfehlungen**

Java Objektorientierung: Empfehlungen

- Kapselung für Konsistenzsicherung nutzen:

*Instanzvariablen immer **private** und wenn möglich **final** deklarieren, mit Konstruktoren (bzw. Fabrikmethoden) konsistent initialisieren und in Instanzmethoden nur Konsistenz erhaltend manipulieren.*

- Klassenvererbung bei Entitäten vorsichtig einsetzen, bei Wertklassen vermeiden:

*Klassen entweder gezielt als Oberklassen entwerfen (dann in der Regel **abstract**) oder Unterklassen verbieten (**final**).*

- Schnittstellenvererbung möglichst bei allen nichttrivialen Klassen nutzen:

Es können dann weitere Klassen entwickelt werden, die zu den bisherigen Klassen kompatibel sind.

- Polymorphie und dynamische Bindung nutzen:

Entitäten möglichst nur über Oberklassen- oder noch besser Schnittstellen-Referenzen benutzen, denn Programmteile mit solchen Referenzen funktionieren ohne Änderung mit verschiedenen Unter- bzw. Implementierungsklassen.

Downcasts vermeiden!

Java Objektorientierung: Index

@Override 5-11	java.lang.Exception 5-19	statische Bindung 5-33
abstract 5-9	java.lang.Iterable<T> 5-28,5-29	strukturierte Programmierung 5-2
abstrakte Klasse 5-9	java.langIterator<T> 5-28,5-29	Typ versus Klasse 5-34
Annotation 5-9	java.lang.Object 5-12,5-15	Überschreiben 5-6,5-11
anonyme Klasse 5-23	java.lang.RuntimeException 5-19	UML 5-7,5-21
Crosscast 5-35	java.lang.Throwable 5-19	Unterklasse 5-6,5-7,5-11 bis 5-13
Downcast 5-35	Kapselung 5-4,5-5	Upcast 5-34
dynamische Bindung 5-4,5-33	Klassenvererbung 5-6,5-7	Vererbung 5-4,5-6,5-20
Einfachvererbung 5-11	Lokalitäts- und Geheimnisprinzip 5-3	
extends 5-11	Modul 5-3	
function object 5-27	modulare Programmierung 5-3	
Funktionsobjekt 5-27	natürliche Ordnung 5-26	
implements 5-22	Oberklasse 5-4,5-6 bis 5-10	
Information Hiding 5-3	objektorientierte Programmierung 5-4	
interface 5-22,5-24	Polymorphie 5-4,5-32	
java.lang.Appendable 5-25	Programmierparadigma 5-1 bis 5-4	
java.lang.Comparable<T> 5-26	protected 5-10	
java.lang.Comparator<T> 5-27	prozedurale Programmierung 5-1	
java.lang.Error 5-19	Schnittstellenvererbung 5-20,5-21	