

---

# Arten von Klassen-Beziehungen

**Untertypbeziehung:** Ersetzbarkeit

Vererbung von Code aus Oberklasse irrelevant

**Vererbungsbeziehung:**

Klasse entsteht durch Abänderung anderer Klassen

Ersetzbarkeit irrelevant

**Reale-Welt-Beziehung:** Beziehung zw. Einheiten im Entwurf

intuitiv klar, ohne Details zu kennen

oft zu Untertypbeziehung weiterentwickelbar

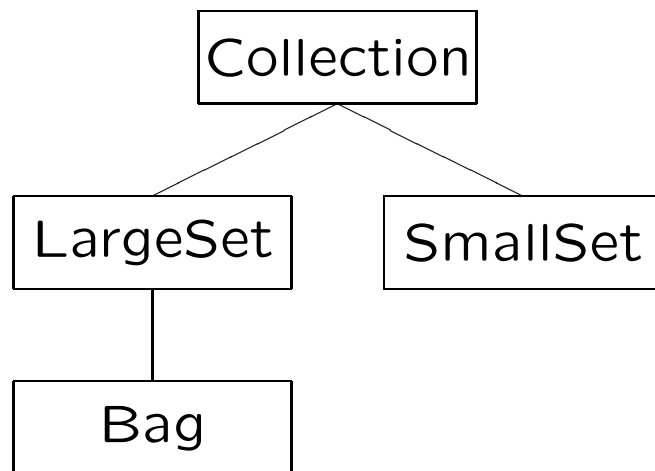
---

# Untertypen versus Vererbung in Java

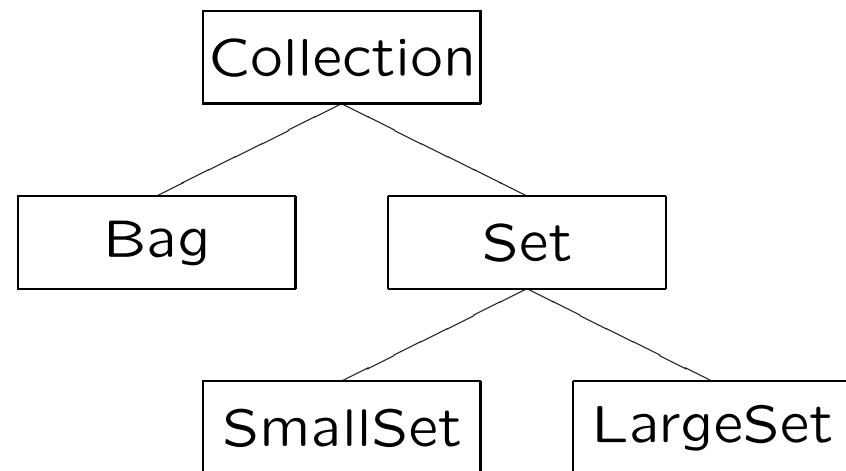
- Untertypbeziehung setzt Vererbung voraus
- Vererbung setzt Untertypbeziehung voraus, soweit vom Compiler überprüft
- daher Untertyp- und Vererbungsbeziehung nur durch (nicht vom Compiler überprüfte) Zusicherungen unterscheidbar
- trotzdem oft einfach erkennbar, was angestrebt wird:
  - Vererbung: Ähnlichkeiten im Code ausgenützt
  - Untertypen: ersetzbares Verhalten beschrieben

---

## Beispiel: Untertypen versus Vererbung



reine Vererbungsbeziehung



Untertypbeziehung

---

# Untertypen versus Vererbung (Tip)

- Vererbung = direkte Codewiederverwendung (sichtbar)
- Untertypbeziehung = indirekte Codewiederverwendung (manchmal nicht gleich sichtbar)
- Untertypen bedeuten oft weniger direkte Codewiederverwendung als Vererbung, da Zusicherungen berücksichtigt
- indirekte Codewiederverwendung langfristig viel wichtiger als direkte (lokale Programmänderungen möglich)
- Tip: immer auf Ersetzbarkeit achten, direkte Wiederverwendung nur wenn Zusicherungen (zufällig) passen  
⇒ jede Vererbungsbeziehung soll Untertypbeziehung sein

---

# Direkte Codewiederverwendung

- direkte Codewiederverwendung durch Vererbung ist auch wichtig (solange Untertypbeziehung nicht verletzt):
  - Code nur einmal geschrieben
  - Änderungen nur an einer Stelle
- durch Möglichkeit des Überschreibens kaum Nachteile

---

# Vererbung, 1. Beispiel

```
class A {  
    public void foo() { ... }  
}
```

```
class B extends A {  
    private boolean b;  
    public void foo() {  
        if (b) { ... }  
        else { super.foo(); }  
    }  
    ...  
}
```

---

## Vererbung, 2. Beispiel

```
class A {  
    public void foo() {  
        if (...) { ... }  
        else { ...; x = 1; ... }  
    }  
}
```

```
class B extends A {  
    public void foo() {  
        if (...) { ... }  
        else { ...; x = 2; ... }  
    }  
}
```

---

## Vererbung, 3. Beispiel

```
class A {
    public void foo() {
        if (...) { ... }
        else { fooX(); }
    }
    protected void fooX() { ...; x = 1; ... }
}

class B extends A {
    protected void fooX() { ...; x = 2; ... }
}
```



---

## Vererbung, 4. Beispiel

```
class A {  
    public void foo() { fooY(1); }  
    protected void fooY (int y) {  
        if (...) { ... }  
        else { ...; x = y; ... }  
    }  
}
```

```
class B extends A {  
    public void foo() { fooY(2); }  
}
```

---

# Variablen in Java

- Instanzvariablen (über Instanz zugreifbar):

```
int x = 2, y[] = new int[32], z;
```

- Klassenvariablen (über Klasse zugreifbar):

```
static int maxRadius = 1023;
```

- Konstanten (nicht schreibbar):

```
static final int MAX_SIZE = 1024;  
final int sizeOfThis = MAX_SIZE - n;
```

---

# Methoden in Java

- Instanzmethoden (über Instanz zugreifbar):

```
void foo (String[] args) { ... }
```

dynamisches Binden

- statische Methoden (über Klasse zugreifbar):

```
static void main (String[] args) { ... }
```

kein dynamisches Binden

- static initializer (zur Initialisierung von Klassenvariablen):

```
static { ... }
```

Code nur einmal ausgeführt

---

# Konstruktoren in Java

- Default-Konstruktor, falls kein expliziter Konstruktor:

```
class Classname {  
    // public Classname() {};  
    ...  
}
```

- `super(...)` führt Konstruktor der Oberklasse aus
- `this(...)` führt anderen Konstruktor derselben Klasse aus
- Konstruktor beginnt nicht mit `super(...)` oder `this(...)`:  
`super()` implizit am Anfang des Konstruktors eingefügt

---

# Überladen

```
class Circle {
    private int r;
    Circle(int r) { this.r = r; }           // 1
    Circle(Circle c) { this.r = c.r; }     // 2
    Circle() { r = 1; }                    // 3
    ...
}

...
Circle a = new Circle(2); // Konstruktor 1
Circle b = new Circle(a); // Konstruktor 2
Circle c = new Circle();  // Konstruktor 3
```

---

# Verdecken versus Überschreiben

Variablen gleichen Namens in Ober- und Unterklasse:

- Variable in Unterklasse verdeckt Variable in Oberklasse
- verdeckte Variable zugreifbar: `super.var`  
`((Oberklasse)this).var`

Methoden gleichen Namens in Ober- und Unterklasse:

- Unterklassenmethode überschreibt Oberklassenmethode
- überschriebene Methode zugreifbar: `super.method(...)`
- kein Zugriff über `((Oberklasse)this).method(...)`

---

# Final

Überschreiben einer Methode verhindertbar:

```
final method ( ... ) { ... }
```

final Methoden sollen meist vermieden werden

Ableitung einer Unterklasse verhindertbar:

```
final class FinalClass { ... }  
final class FinClass extends NonFinalClass { ... }
```

in einigen oo Programmierstilen sind final Klassen häufig:

- Instanzen werden nur von final Klassen erzeugt
- dadurch Ersetzbarkeit einfacher zuzusichern

---

# Statische geschachtelte Klassen

- gehören zu umschließender Klasse selbst

```
class EnclosingClass {  
    ...  
    static class StaticNestedClass { ... }  
    ...  
}
```

- nur Klassenvariablen und statische Methoden umschließender Klasse zugreifbar
- Erzeugung: `new EnclosingClass.StaticNestedClass()`



---

# Innere Klassen

- gehören zu Instanzen umschließender Klasse

```
class EnclosingClass {  
    ...  
    class InnerClass { ... }  
    ...  
}
```

- nur Instanzvariablen und Instanzmethoden umschließender Klasse direkt zugreifbar
- Erzeugung: `a.new InnerClass()`  
(wobei `a` eine Instanz von `EnclosingClass` ist)

---

# Pakete

- eine public Klasse pro Datei
- Paket umfaßt alle Dateien bzw. Klassen im selben Ordner
- explizite Paketdeklaration: `package paketName;`
- Aufruf von `foo()` in der Datei `myclasses/test/AClass.java`:

```
myclasses.test.AClass.foo()
```

- Kürzer durch Import-Deklaration (am Dateianfang)

```
import myclasses.test;           ... test.AClass.foo() ...  
import myclasses.test.AClass;   ... AClass.foo() ...  
import myclasses.test.*;       ... AClass.foo() ...
```

---

# Sichtbarkeit

	<code>public</code>	<code>protected</code>	<code>default</code>	<code>private</code>
lokal sichtbar	ja	ja	ja	nein
global sichtbar	ja	nein	nein	nein
lokal vererbbar	ja	ja	ja	nein
global vererbbar	ja	ja	nein	nein

lokal = im selben Paket, auch außerhalb der Klasse  
global = auch außerhalb des Pakets

---

# Anwendung der Sichtbarkeit

**Public:** Bei der Verwendung der Klasse und deren Instanzen benötigte Methoden, Konstanten und (selten) Variablen

**Private:** Methoden und Variablen, die nur innerhalb der Klasse verwendet werden sollen — vor allem, wenn Bedeutung außerhalb der Klasse nicht klar

**Protected:** Methoden und Variablen für Verwendung nicht benötigt, aber hilfreich bei Erweiterungen der Klasse

**Default:** Eng zusammenarbeitende Klassen im selben Paket sollen auf Methoden und Variablen zugreifen können, die sonst privat wären