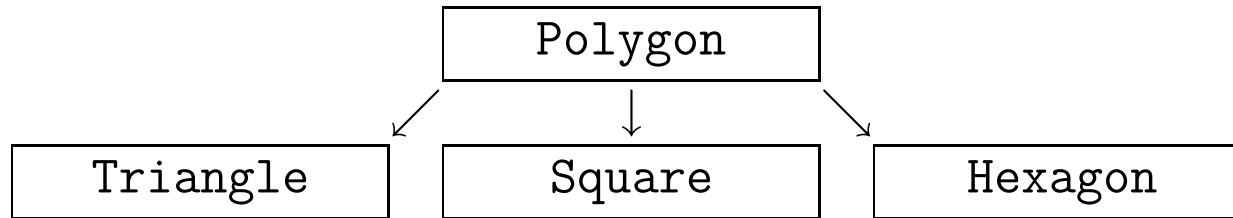


OOP

Untertypen, Vererbung, Sichtbarkeit

Abstrakte Klassen



```
public abstract class Polygon {
    public abstract void draw(); // draw polygon on screen
}

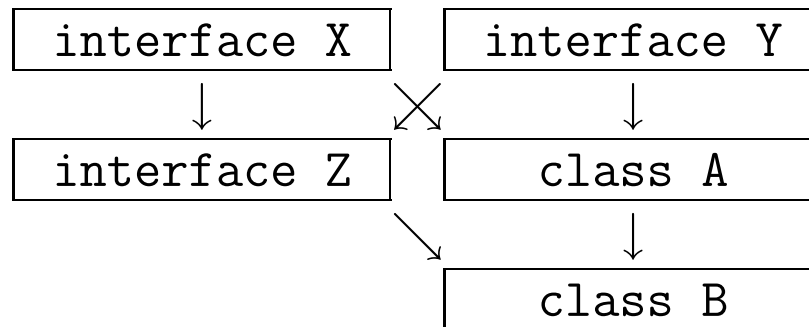
public class Triangle extends Polygon {
    public void draw() { ... } // draw triangle on screen
}
```

Zusicherungen auf abstrakten Methoden besonders wichtig

auch weil sie sich von denen in Unterklassen unterscheiden

Interfaces

Interfaces für Untertyprelationen sehr gut geeignet



Klassen gegenüber Interfaces nur bevorzugen wenn Code vererbt werden soll

Zusicherungen auf Interfaces besonders wichtig

Abstr. Klassen und Interfaces verwenden

Implementierungen oft nur in Klassen ohne Unterklassen

abstrakte Klassen und Interfaces eher stabil

Parametertypen sollen möglichst stabil sein
und nicht mehr ausdrücken als nötig ist – keine unnötigen Abhängigkeiten

abstrakte Klassen bzw. Interfaces meist leicht hinzuzufügen
etwa als Parametertypen für jeden Bedarf

Arten von Klassen-Beziehungen

Untertypbeziehung: Ersetzbarkeit

wofür Vererbung von Code aus Oberklasse irrelevant

Vererbungsbeziehung:

Klasse entsteht durch Abänderung anderer Klassen

wofür Ersetzbarkeit irrelevant

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

ist intuitiv klar ohne Details zu kennen

und oft zu Untertypbeziehung weiterentwickelbar

Untertypen versus Vererbung in Java

Untertypbeziehung setzt Vererbung voraus und
Vererbung setzt Untertypbeziehung voraus, soweit vom Compiler überprüft

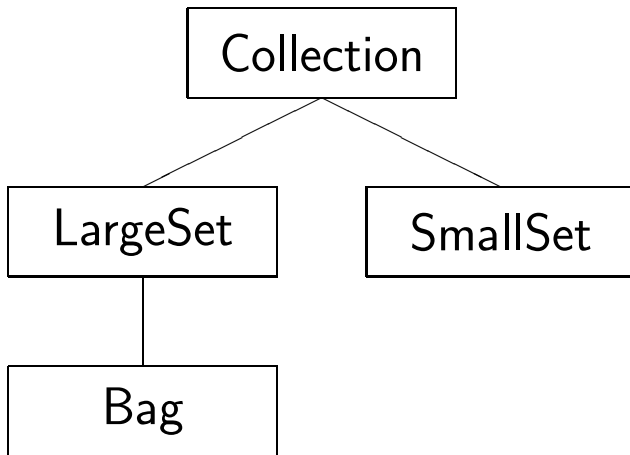
Untertyp- und Vererbungsbeziehung nur durch Zusicherungen unterscheidbar

trotzdem oft einfach erkennbar, was angestrebt wird:

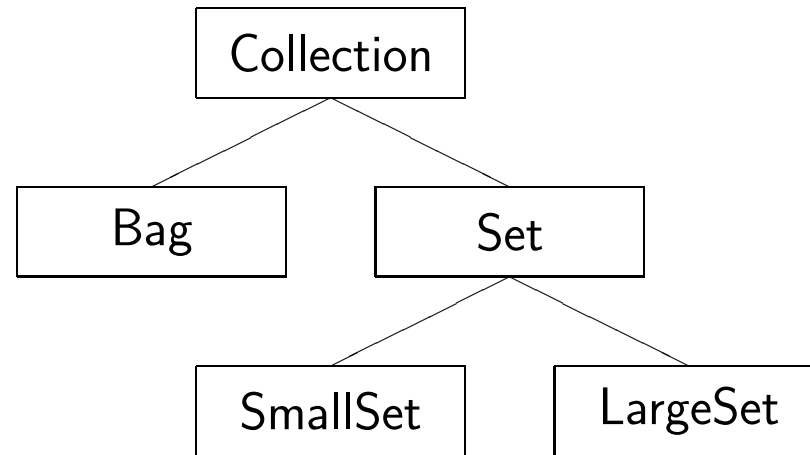
Vererbung: Ähnlichkeiten im Code ausgenützt

Untertypen: ersetzbares Verhalten beschrieben

Beispiel: Untertypen versus Vererbung



reine Vererbungsbeziehung



Untertypbeziehung

Tip zu Untertypen

Vererbung = direkte Codewiederverwendung (leicht sichtbar)

Untertypbeziehung = indirekte Codewiederverwendung (nur schwer sichtbar)

Untertypen bedeuten oft weniger direkte Codewiederverwendung als Vererbung, da Zusicherungen berücksichtigt werden müssen

indirekte Codewiederverwendung langfristig viel wichtiger als direkte, da lokale Programmänderungen ohne Kenntnis der Aufrufstellen möglich

Tip: immer auf Ersetzbarkeit achten,
direkte Wiederverwendung nur wenn Zusicherungen zufällig passen
→ jede Vererbungsbeziehung soll Untertypbeziehung sein

Aufgabe: Bedeutung von Untertypen

Such Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Frage:

Warum ist Ersetzbarkeit deutlich wichtiger als Vererbung?

Zeit: 2 Minuten

Direkte Codewiederverwendung

direkte Codewiederverwendung durch Vererbung auch wichtig da

Code nur einmal geschrieben und
nötige Änderungen nur an einer Stelle

Nachteil: starke Abhängigkeit zwischen Unter- und Oberklasse

→ nur Code von stabiler Oberklasse erben

Vererbung, 1. Beispiel

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

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

Vererbung, 2. Beispiel

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

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

Vererbung, 3. Beispiel

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

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

Vererbung, 4. Beispiel

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

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

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(...)`

aber kein Zugriff über `((Oberklasse)this).method(...)`

Final

Überschreiben einer **Methode** verhinderbar:

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

final Methoden sollen meist vermieden werden

Ableitung einer **Unterklasse** verhinderbar:

```
public final class FinalClass { ... }  
public final class FinClass extends NonFinClass { ... }
```

in einigen oo Programmierstilen sind final Klassen häufig:

Instanzen werden nur von final Klassen erzeugt
wodurch Ersetzbarkeit einfacher zuzusichern

Statisch geschachtelte Klasse

gehört zu umschließender **Klasse**

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

nur Klassenvariablen und statische Methoden umschließender Klasse zugreifbar

Objekterzeugung: `new EnclosingClass.StaticNestedClass()`

Innere Klasse

gehört zu einem **Objekt** der umschließenden Klasse

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

Objektvariablen und nicht-statische Methoden umschließender Klasse direkt zugreifbar, statische Teile (x) indirekt durch `EnclosingClass.x`

Objekterzeugung: `a.new InnerClass()`
wobei a eine Instanz von `EnclosingClass` ist

Pakete

nur 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

	public	protected	(default)	private
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 Sichtbarkeitskontrolle

Public: für allgemeine Verwendung benötigte Methoden, Konstruktoren und Konstanten; Variablen verpönt

Private: alles, was außerhalb der Klasse nicht verständlich zu sein braucht; ideal für Variablen

Protected: nicht für allgemeine Verwendung gedachte Methoden, Konstruktoren und Konstanten (möglichst keine Variablen) wenn in Unterklassen tatsächlich benötigt

Default: nur bei tatsächlichem Bedarf für enge Zusammenarbeit zwischen Klassen im Paket, möglichst keine Variablen