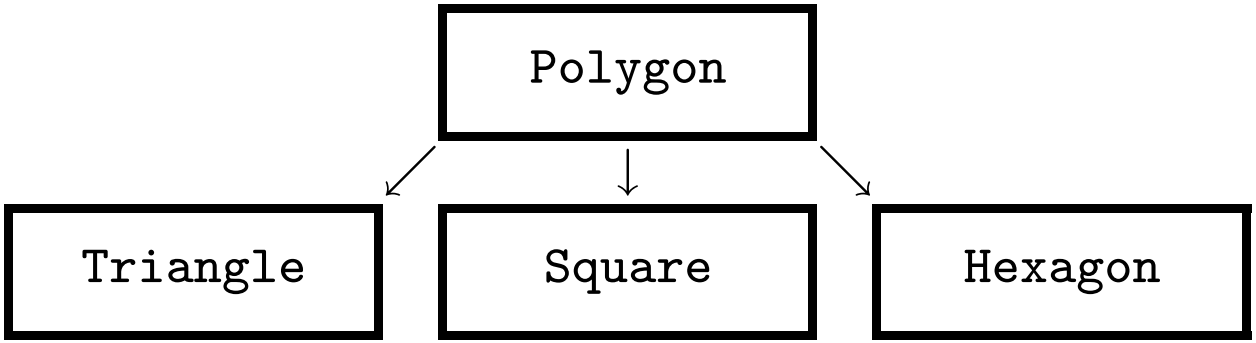
Faustregeln zu Zusicherungen

Zusicherungen sollen

- stabil sein (vor allem an Wurzel der Typhierarchie)
- keine unnötigen Details festlegen
- explizit im Programm stehen
- unmissverständlich formuliert sein
- während Programmentwicklung ständig überprüft werden

Abstrakte Klassen

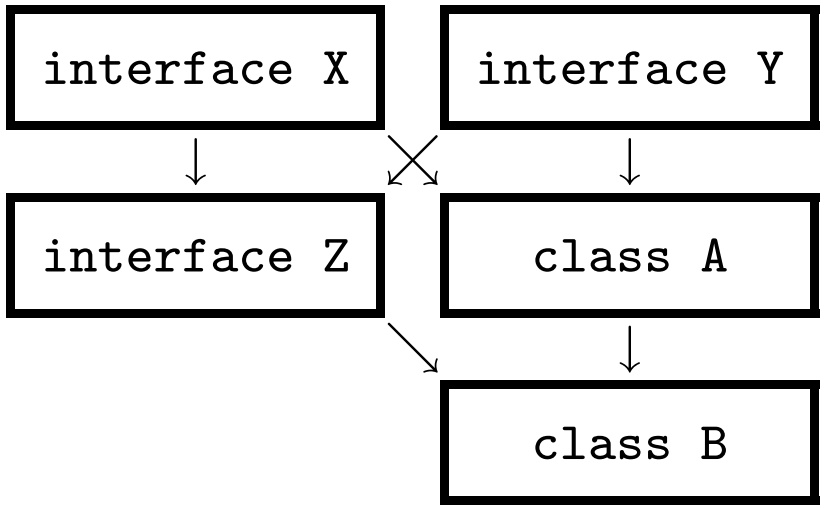



```
public abstract class Polygon {
    public abstract void draw(); // draw polygon on screen
}
public class Triangle extends Polygon {
    public void draw() { ... }  // draw triangle on screen
}
```

Zusicherungen auf abstrakten Methoden besonders wichtig!

Interfaces

Interfaces für Untertyprelationen sehr gut geeignet



Abstrakte Klassen gegenüber Interfaces nur dann zu bevorzugen, wenn Code vererbt werden soll

Zusicherungen auf Interfaces besonders wichtig!

Abstr. Klassen u. Interfaces verwenden

- Implementierungen oft nur in Klassen ohne Unterklassen
- abstrakte Klassen und Interfaces eher stabil
- Parametertypen sollen stabil sein

  $\rightarrow$ sie sollen nicht mehr ausdrücken, als nötig
  (keine unnötigen Abhängigkeiten)
- abstrakte Klassen bzw. Interfaces leicht hinzuzufügen
  (z.B. verwendbar als Parametertypen für jeden Bedarf)
- Zusicherungen auf abstrakten Einheiten besonders wichtig

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

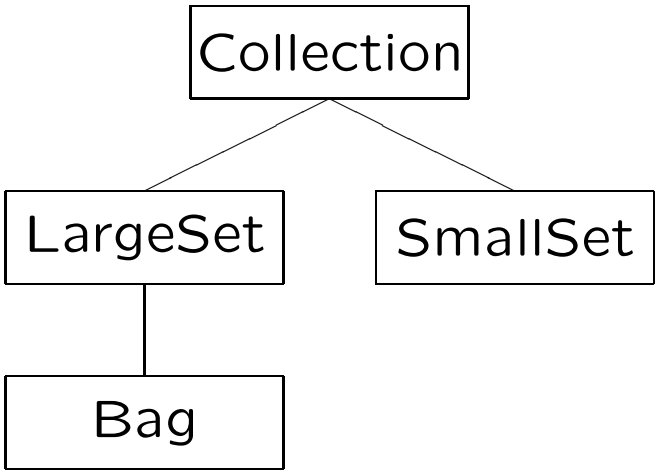
intuitiv klar, ohne Details zu kennen

oft zu Untertypbeziehung weiterentwickelbar

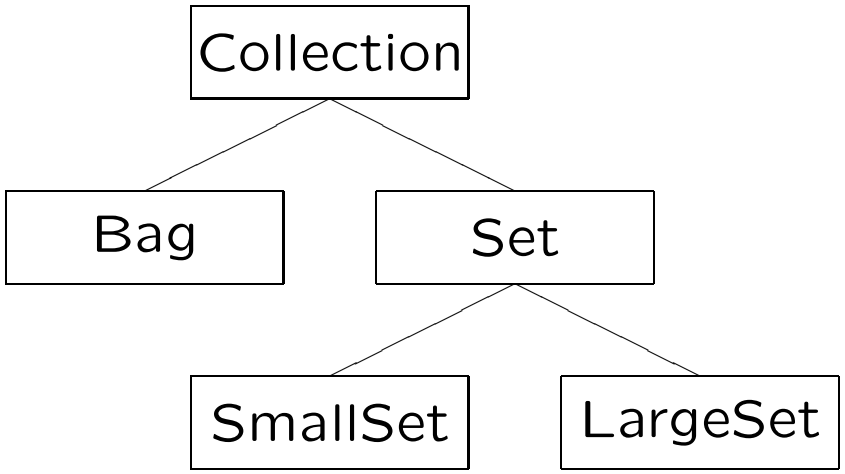
Untertypen versus Vererbung in Java

- 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

  $\rightarrow$ 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

```
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(...)`
- 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
- 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

	`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