Tagesprogramm

Beziehungen zwischen Typen

Vererbung

Sichtbarkeit

Abstrakte Klassen und Interfaces

```
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

weil sie sich von denen in Unterklassen unterscheiden

Klassen gegenüber Interfaces nur bevorzugen wenn Code vererbt werden soll

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 zwischen Einheiten im Entwurf,

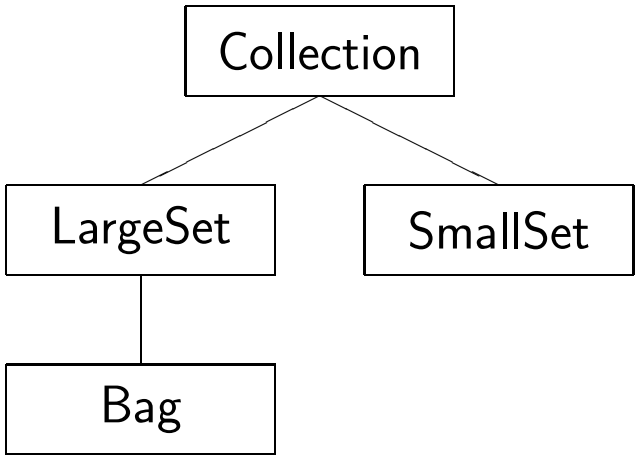
intuitiv klar ohne Details zu kennen,

oft zu Untertypbeziehung weiterentwickelbar

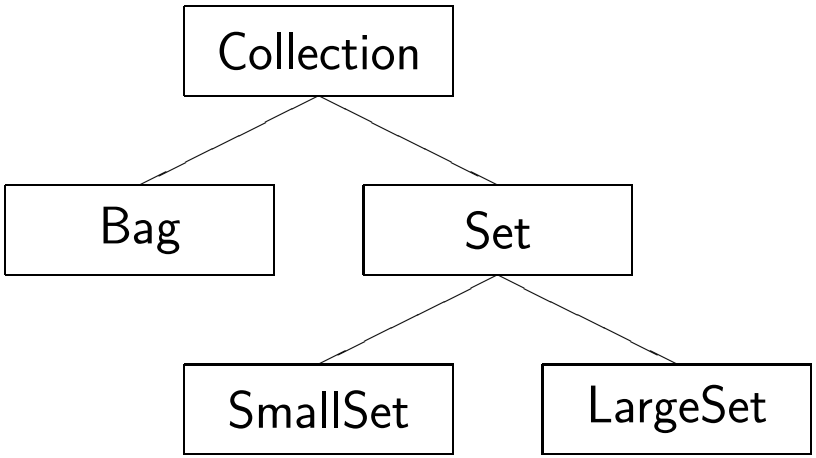
Untertypen versus Vererbung

Untertypbeziehung setzt Vererbung voraus,
Vererbung setzt Untertypbeziehung voraus (soweit vom Compiler überprüfbar)

Untertyp- und Vererbungsbeziehung nur durch Zusicherungen unterscheidbar,
trotzdem oft einfach erkennbar, was angestrebt wird



reine Vererbungsbeziehung



Untertypbeziehung

Tipp zu Untertypen

Vererbung = direkte Codewiederverwendung (leicht sichtbar)
Untertypbeziehung = indirekte Codewiederverwendung (nur schwer sichtbar)

Untertypbeziehung = weniger direkte Codewiederverwendung als Vererbung
(da Zusicherungen zu berücksichtigen)

indirekte Codewiederverwendung **langfristig** viel wichtiger als direkte

Aufgabe: Bedeutung von Untertypen

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

Warum ist Vererbung kurzfristig leichter sichtbar?

Warum ist Ersetzbarkeit langfristig wichtiger?

Zeit: 2 Minuten

Direkte Codewiederverwendung

direkte Codewiederverwendung auch von Bedeutung da

> Code nur einmal geschrieben und
> nötige Änderungen nur an einer Stelle

Nachteil: starke Abhängigkeit zwischen Unter- und Oberklasse

$\longrightarrow$ nur von stabiler Klasse erben

Vererbung durch super

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

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

Vererbung verhindert

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

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

Vererbung durch Zerlegung

```
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 durch Parametrisierung

```
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