Beziehungen zwischen Typen
Vererbung
Steuerung der Sichtbarkeit
Ausnahmebehandlung

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
}
```

Ist jedes Dreieck auch ein Polygon?

Ist ein Objekt von `Polygon` durch eines von `Triangle` ersetzbar?

Ist Implementieren von `Triangle` durch Erben von `Polygon` einfacher?

Zusicherungen auf abstrakten Methoden besonders wichtig
weil sie sich von denen in Unterklassen unterscheiden

Klassen gegenüber Interfaces nur bevorzugen wenn Objektvariablen unvermeidbar sind

Arten von Klassen-Beziehungen

Untertypbeziehung: Objekt von T_2 ersetzt Objekt von T_1

Ersetzbarkeit,
Vererbung von Code aus Oberklasse irrelevant

Vererbungsbeziehung: T_1 vereinfacht Implementierung von T_2

Klasse entsteht durch Abänderung anderer Klassen,
Ersetzbarkeit irrelevant

Reale-Welt-Beziehung: Begriff_2 ist auch ein Begriff_1

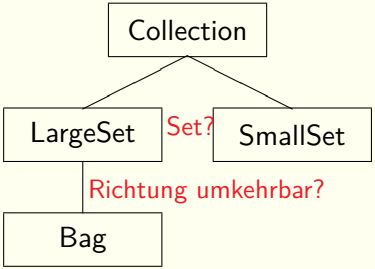
Beziehung zwischen Einheiten im Entwurf,
intuitiv klar ohne Details zu kennen, $\text{Begriff}_x \sim$ Objekt von T_x
oft zu Untertypbeziehung weiterentwickelbar aber $\text{Begriff}_x \neq$ Objekt von T_x

Reale-Welt-Beziehung $\not\Rightarrow$ Untertypbeziehung

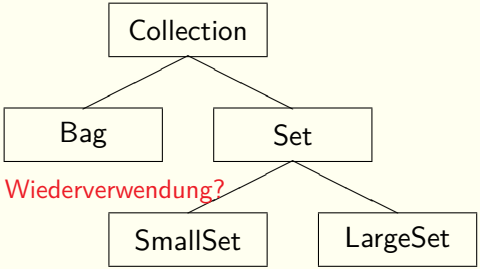
Untertypen versus Vererbung

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 (schlecht)



Untertypbeziehung (gut)

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

für indirekte Wiederverwendung (wenn notwendig) auf sichtbare Wiederverwendung verzichten

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
$\rightarrow$ 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 eher vermieden werden

Ableitung einer **Unterklasse** verhinderbar:

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

in einigen objektorientierten Programmierstilen sind final Klassen häufig:

Instanzen werden nur von final Klassen erzeugt — jede Klasse ist final oder abstract
wodurch Ersetzbarkeit einfacher zuzusichern — ideal: nur Interfaces und final Klassen

Statisch geschachtelte Klasse

gehört zu umschließender **Klasse**

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

auch private 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

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

Anonyme innere Klasse – Lambda-Ausdruck

Erzeugung **eines Objekts** einer anonymen inneren Klasse als Java-Ausdruck:

```
new T(...) {  // Parameter für Konstruktor falls T Klasse
    ...       // Implementierung eines Untertyps von T
}             // Ergebnis ist Objekt des Untertyps von T
```

Functional Interface = Interface mit genau einer abstrakten Methode

Lambda-Ausdruck ∼ vereinfachte Syntax für Objekt von Functional Interface:

```
T x = (a, b) -> a + b;  // wobei T Funcional Interface
```

entspricht für Methode `int f(int x, int y)` in `T`

```
new T() { int f(int a, int b) { return a + b; } }
```

Pakete

nur eine public Klasse pro Datei

Paket umfasst alle Dateien bzw. Klassen im selben Ordner

explizite Paketdeklaration: `package paketName;` nicht im Default-Paket

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 (wegen Design-by-Contract)

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

Protected: nicht für allgemeine Verwendung gedachte Methoden, Konstruktoren und Konstanten (möglichst keine Variablen) wenn in Unterklassen benötigt (historisch gewachsen, sollte heute weitgehend vermieden werden)

Default: nur bei tatsächlichem Bedarf für enge Zusammenarbeit zwischen Klassen im Paket, möglichst keine Variablen (selten sinnvoll, meist falsch verwendet)

Ausnahmebehandlung in Java

```
class A {
    void foo() throws Help, SyntaxError { ... }
}
class B extends A {
    void foo() throws Help {
        if (helpNeeded())
            throw new Help();
    }
}

... try { ... }
    catch (Help e) { ... }
    catch (Exception e) { ... }
    finally { ... }
```

Ausnahmebehandlung und Ersetzbarkeit

Methode in Unterklasse soll nur dann eine Exception werfen,
wenn Aufrufer der Methode in Oberklasse dies erwartet

Einschränkungen durch `throws`-Klausel nicht hinreichend
$\rightarrow$ Zusicherungen beachten (**Nachbedingung**)

Einsatz von Ausnahmebehandlungen

Ursachen unvorhergesehener Programmabbrüche finden (kaum vermeidbar)	gut
kontrolliertes Wiederaufsetzen nach Fehlern (notwendig, aber schwierig)	gut
vorzeitiger Ausstieg aus Sprachkonstrukten (fehleranfällig, vermeidbar)	schlecht
Rückgabe alternativer Ergebniswerte (schlechte Programmstruktur, vermeidbar)	schlecht

Ganz schlechtes Einsatzbeispiel

Ohne Ausnahmebehandlung: pro Iteration zwei Vergleiche mit `null`

```
while (x != null)
    x = x.getNext();
```

Mit Ausnahmebehandlung: pro Iteration nur ein Vergleich mit `null`

```
try {
    while (true)
        x = x.getNext();
}
catch (NullPointerException e) {}
```

nicht-lokal und fehleranfällig, Ausnahme auch in `getNext` auslösbar — gefährlich

Schlechtes Einsatzbeispiel

trickreiche Verwendung von Ausnahmen:

```
                                    try { throw x }
if (x instanceof T1) {...}          catch (T1 x) {...}
else if (x instanceof T2 {...}      catch (T2 x) {...}
...                                 ...
else if (x instanceof Tn {...}      catch (Tn x) {...}
else {...}                          catch (Exception x) {...}
```

nur lokale Ausnahmen (daher weniger fehleranfällig)
aber beide Varianten schwer wartbar

Empfohlenes Einsatzbeispiel

Ohne Ausnahmebehandlung:

```
public static String addA (String x, String y) {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else return "Error";
}
```

Mit Ausnahmebehandlung:

```
public static String addB (String x, String y)
        throws NoNumberString {
    if (onlyDigits(x) && onlyDigits(y)) { ... }
    else throw new NoNumberString();
}
```

sinnvoller Einsatz, da Fehlerabfragen vermieden werden

Überprüfte versus unüberprüfte Ausnahmen

Java verwendet in einigen Standardbibliotheken überprüfte (checked) Ausnahmen

Problem: überprüfte Ausnahmen nur bei nominaler Abstraktion gut geeignet

→ zusammen mit funktionalen (strukturellen) Programmteilen ungeeignet

→ unüberprüfte Ausnahmen meist zu bevorzugen

→ nötigenfalls überprüfte in unüberprüfte Ausnahmen umwandeln
(überprüfte Ausnahmen abfangen und unüberprüfte Ausnahmen auslösen)