

**OOP**

# Grundlagen der Ersetzbarkeit

## Das Ersetzbarkeitsprinzip

**U ist Untertyp von T wenn eine Instanz von U überall verwendbar ist wo eine Instanz von T erwartet wird**

Ersetzbarkeitsprinzip benötigt man für

den Aufruf einer Routine mit einem Argument,  
dessen Typ Untertyp des formalen Parametertyps ist

die Zuweisung eines Objekts an eine Variable,  
wobei der Objekttyp Untertyp des deklarierten Variablentyps ist

## Untertypen und Schnittstellen

Typ U ist Untertyp von Typ T wenn

für jede Konstante in T (vom Typ A)  
existiert Konstante in U (vom Typ B), wobei B Untertyp von A

für jede Variable in T (vom Typ A)  
existiert eine Variable in U (vom Typ B), wobei A und B äquivalent

für jede Methode in T existiert Methode in U, wobei

Parameteranzahlen und Parameterarten gleich

Parametertypen passen zueinander (je nach Art)

Ergebnistyp in U ist Untertyp von Ergebnistyp in T

Methode in U wirft nicht mehr Exceptions als die in T

## Untertypen nach Parameterarten

Methodenparameter in Obertyp T vom deklarierten Typ A und

Methodenparameter in Untertyp U vom deklarierten Typ B

### Eingangsparameter:

beim Aufruf wandert Argument von Aufrufer zu aufgerufener Methode

→ A Untertyp von B

### Ausgangsparameter:

am Methodenende wandert Ergebnis von aufgerufener Methode zu Aufrufer

→ B Untertyp von A

### Durchgangsparameter:

gleichzeitig Ein- und Ausgangsparameter

→ A und B äquivalent

# Varianz von Typen

## Kovarianz:

Typ von Element im Untertyp ist Untertyp des Elementtyps im Obertyp  
→ Typ von Konstante, Ergebnis, Ausgangsparameter

## Kontravarianz:

Typ von Element im Untertyp ist Obertyp des Elementtyps im Obertyp  
→ Typ von Eingangsparameter

## Invarianz:

Typ von Element im Untertyp ist äquivalent zu Elementtyp im Obertyp  
→ Typ von Variable, Durchgangsparameter

## Beispiel für Varianz

Annahme: variante Parametertypen erlaubt (nicht in Java)

```
class T {  
    public T meth(U p) { ... }  
}  
class U extends T { // U ist Untertyp von T  
    public U meth(T p) { ... }  
} // in Java ueberladen, nicht ueberschrieben
```

entspricht Bedingungen für Untertypbeziehungen

**ACHTUNG: funktioniert in Java nicht so**

(Überladen statt Überschreiben, Parametertypen invariant)

## Wann und warum Kovarianz?

Ersetzbarkeit nur bei **lesendem** Zugriff auf Konstante, Ergebnis oder Ausgangsparameter (bei Methodenaufruf)

nur Elementtyp A im Obertyp T statisch bekannt

Lesezugriff kann tatsächlich auf entsprechendes Element vom Typ B im Untertyp U erfolgen

gelesener Wert soll trotzdem vom erwarteten Typ A sein

- jede Instanz von B soll auch Instanz von A sein
- B soll Untertyp von A sein

bei Initialisierung von Konstante, Ergebnis, Ausgangsparameter in Methode ist genauer Typ bekannt, daher keine Ersetzbarkeit nötig

## Wann und warum Kontravarianz?

Ersetzbarkeit nur bei Schreibzugriff auf Eingangsparameter bei Methodenaufruf

nur Parametertyp A im Obertyp T statisch bekannt

Schreibzugriff kann tatsächlich auf entsprechenden Parameter vom Typ B im Untertyp U erfolgen

tatsächlich geschriebener Wert soll vom Typ B sein  
obwohl Werte vom Typ A geschrieben werden können

- jede Instanz von A soll auch Instanz von B sein
- A soll Untertyp von B sein

bei Lesezugriff auf Eingangsparameter in Methode ist deklarierter Typ bekannt,  
daher keine Ersetzbarkeit nötig



## Wann und warum Invarianz?

Ersetzbarkeit bei **schreibendem und lesendem** Zugriff benötigt  
(Variable, Durchgangparameter)

daher sowohl Kovarianz als auch Kontravarianz gefordert

nur Invarianz erfüllt beide Forderungen

## Aufgabe: Explizite Untertypbeziehungen

Regeln für Untertypbeziehungen sind durch die Theorie vollständig und unumstößlich vorgegeben. Sie sind nicht verhandelbar.

Suchen Sie in Gruppen zu zwei bis drei Personen Antworten auf folgende Fragen:

1. Kann man Untertypbeziehungen aus den vorgegebenen Regeln ableiten und dadurch auf das explizite Anschreiben von Untertypbeziehungen verzichten?

Wann ja, nennen Sie eine solche objektorientierte Programmiersprache.

2. Warum ist es in Sprachen wie Java nötig Untertypbeziehungen explizit in den Programmcode zu schreiben?

Zeit: 3 Minuten

## Theorie und Praxis

Theorie vollständig und widerspruchsfrei auf Signaturen

→ für strukturelle Typen keine Untertypdeklarationen nötig

in der Praxis: nominale Typen

→ explizite Vererbungsbeziehung vorausgesetzt

→ Vererbung entsprechend eingeschränkt

→ Abstraktion verhindert „zufällige“ Untertypbeziehungen

weitere Einschränkung in Java und ähnlichen Sprachen:

Ergebnistypen kovariant, aber alle anderen Typen invariant  
da intuitiv und unterscheidbar von Überladen

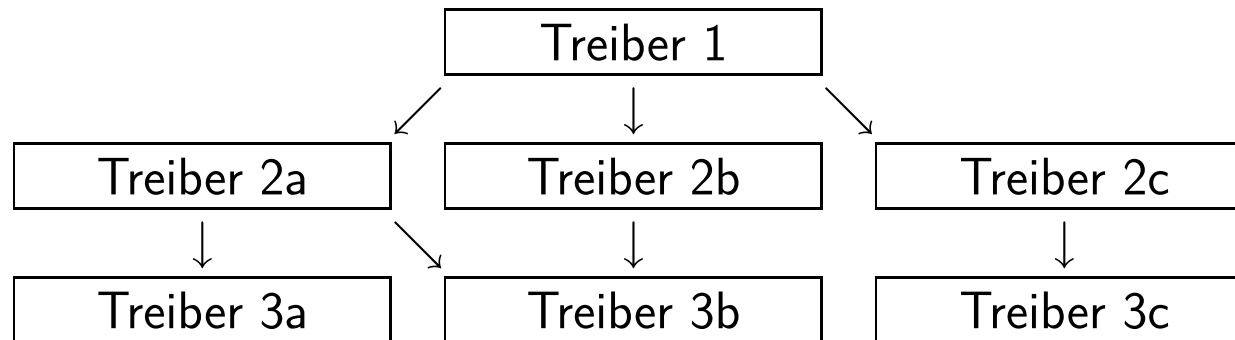
## Grenzen von Untertypbeziehungen

```
public class Point2D {
    protected int x, y;
    public boolean equal(Point2D p) {
        return x == p.x && y == p.y;
    }
}

public class Point3D extends Point2D {    // FALSCH
    protected int z;
    public boolean equal(Point3D p) {
        return x == p.x && y == p.y && z == p.z;
    } // equal ueberladen, nicht ueberschrieben
}
```

## Untertypen – Codewiederverwendung

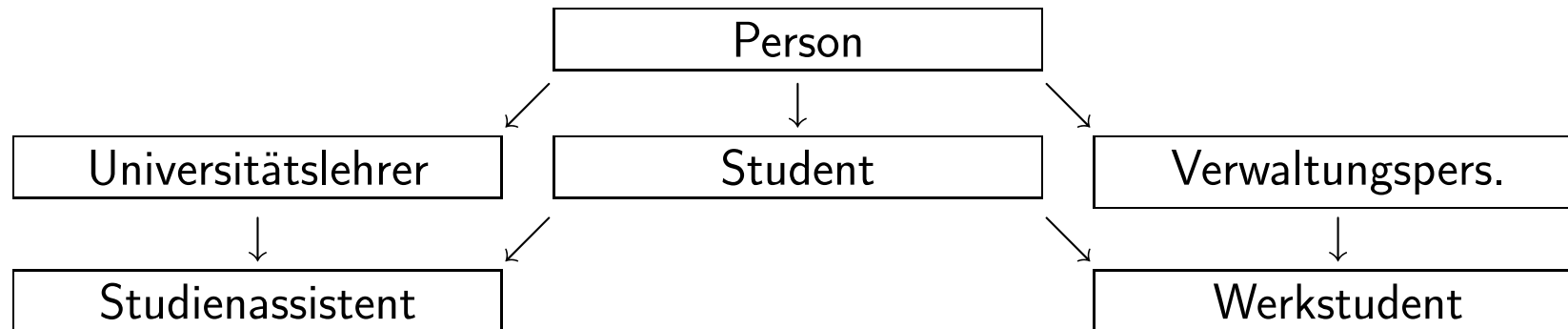
Software-Generationen und -Versionen



Typen sollen unverändert bleiben, Erweiterungen möglich

## Untertypen – Codewiederverwendung

Wiederverwendung innerhalb eines Programms



Typen sollen stabil sein – vor allem weiter oben in Typhierarchie

## Dynamisches Binden

```
class A {
    public String foo1() { return "foo1A"; }
    public String foo2() { return fooX(); }
    protected String fooX() { return "foo2A"; } }
class B extends A {
    public String foo1() { return "foo1B"; }
    protected String fooX() { return "foo2B"; } }
class DynamicBindingTest {
    public static void test(A x)
        { System.out.println(x.foo1());
          System.out.println(x.foo2()); }
    public static void main(String[] args)
        { test(new A()); test(new B()); }
}
```

## Beispiel mit Switch

```
public void gibAnredeAus(int anredeArt, String name) {  
    switch(anredeArt) {  
        case 1:    // weiblich  
            System.out.print ("S.g. Frau " + name);  
            break;  
        case 2:    // maennlich  
            System.out.print ("S.g. Herr " + name);  
            break;  
        default:   // unbekannt  
            System.out.print ("S.g. " + name);  
    }  
}
```



## Beispiel ohne Switch

```
public class Adressat {
    protected String name;
    public void gibAnredeAus()
        { System.out.print("S.g. " + name); }
    ... // Konstruktoren und weitere Methoden
}

public class WeiblicherAdressat extends Adressat {
    public void gibAnredeAus()
        { System.out.print ("S.g. Frau " + name); }
}

public class MaennlicherAdressat extends Adressat {
    public void gibAnredeAus()
        { System.out.print ("S.g. Herr " + name); }
}
```