Inhalt

Motivation: Minimumsuche

Beispiel: Minimum eines Integer-Arrays bestimmen:

```
public static Integer min(Integer[] arr) {
    Integer m = arr[0];
    for(int i = 1; i < arr.length; i++) {
        if (arr[i] < m)
            m = arr[i];
    }
    return m;
}
```

Frage/Ziel: (Wie) Lässt sich diese Methode generisch machen?

Warum ist das keine Lösung?

```
public static <T> T minGen(T[] arr) {
    T m = arr[0];
    for(int i = 1; i < arr.length; i++) {
        if (arr[i].compareTo(m) < 0)
            m = arr[i];
    }
    return m;
}
```

Es müsste garantiert werden können,

- dass es für den Typ T eine compareTo()-Methode gibt
- das heißt, dass der Typ T das Interface Comparable<T> implementiert
- das heißt, dass T ein Untertyp von Comparable<T> ist

Typeinschränkung mit extends

Genau das leistet die Typeinschränkung:

```
public static <T extends Comparable<T>> T minGen(T[] arr) {
    T m = arr[0];
    for(int i = 1; i < arr.length; i++) {
        if (arr[i].compareTo(m) < 0)
            m = arr[i];
    }
    return m;
}
```

- Der Ausdruck `<T extends Comparable<T>>` schränkt die für T erlaubten Argumente ein
- auf Typen, die über eine compareTo()-Methode verfügen.

Einschränkung auf mehrere Obertypen

Es kann auch erforderlich sein, dass ein Typ T mehrere Bedingungen erfüllen muss, zB

- Unterklasse einer Klasse K zu sein
- ein Interface *Ifc*1 zu implementieren
- ein Interface *Ifc*2 zu implementieren

Die (nicht-generische) Klasse C würde diese Bedingungen erfüllen:

```
class C extends K implements Ifc1, Ifc2 {
    // ...
}
```

Generisch: Forderung an das Typargument T:

```
class  GenClass<T extends K & Ifc1 & Ifc2>  {
    // ...
}
```

Inhalt

Begriffsbildung

Situation:

- Wir haben Klassen, die sich durch Vererbung in eine Klassenhierarchie einordnen.
- Wir benutzen diese Typen, um andere Typen daraus abzuleiten
  - Array-Typen
  - Generische Klassen oder Interfaces

Die Frage ist:

- (Wie) Übertragen sich Vererbungsbeziehungen zwischen Grundtypen auf die abgeleiteten Typen?

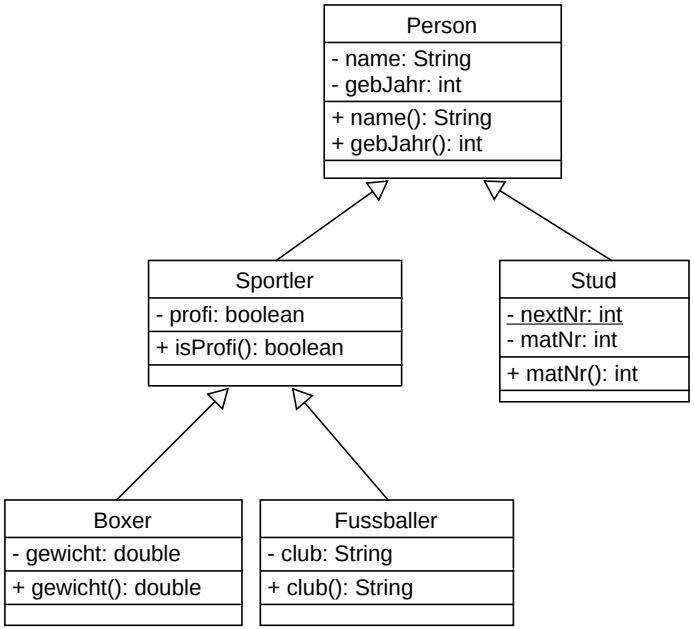
Kovarianz, Kontravarianz, Invarianz

Man spricht von

- Kovarianz, wenn sich die Vererbungsrichtung auf die abgeleiteten Typen in gleicher Richtung überträgt
- Kontravarianz, wenn sich die Vererbungsrichtung auf die abgeleiteten Typen in umgekehhrter Richtung überträgt
- Invarianz, wenn sich die Vererbungsrichtung nicht (in keiner Richtung) auf die abgeleiteten Typen überträgt

Erinnerung: Kovarianz von Arrays

Diese Klassenhierarchie



erlaubt diese Zuweisung:

```
Person[] pArr = new Person[3];
Sportler[] spArr = new Sportler[3];
// ...
pArr = spArr;
```

Wenn U eine Unterklasse von K ist, dann ist $U[\,]$ ein Untertyp von $K[\,]$.

Keine Kovarianz

Ein entsprechender Code für EVL<Person> und EVL<Sportler> erzeugt einen Compiler-Fehler:

```
EVL<Person> pListe = new EVL<>();
EVL<Sportler> spListe = new EVL<>();
// ...
pListe = spListe; // Compiler-Fehler!
```

EVL<Sportler> ist kein Untertyp von EVL<Person> !

Generische Klassen verhalten sich invariant!

Arrays: kovariant - Generics: invariant

- Bei Arrays besteht die Typinformation noch zur Laufzeit!
- (Programmier)-Fehler könn(t)en durch Exceptions abgefangen werden.
- Der folgende Code wir fehlerfrei übersetzt, führt aber zu Absturz:

```
public static void foobar(Person[] pArr, Person p) {
    pArr[0] = p;
}
public static void testFooBar() {
    Stud[] studArr = new Stud[1];
    Person p =  new Boxer("Bubi", 1999, 90.0);
    foobar(studArr, p);
}
```

- Bei generischen Typen ist die Typinformation durch Typlöschung zur Laufzeit verloren!
- (Programmier)-Fehler werden bereits vom Compiler verhindert.

Inhalt

Beispiel: leere Kisten

```
public class Kiste<T> {
    private T inhalt;
    public Kiste() {
        inhalt = null;
    }
    public boolean isEmpty() {
        return inhalt == null;
    }
    // get und set fuer inhalt
}
```

Wir wollen: eine statische Methode **boolean** beideLeer("Kiste" k1, "Kiste" k2), die zwei Kisten beliebigen (auch verschiedenen) Inhalt-Typs darauf prüft, ob beide leer sind.

Aufruf-Beispiel:

```
Kiste<Integer> intK = new Kiste<>();
Kiste<String> strK = new Kiste<>();
boolean b = beideLeer(intK, strK);
```

Beispiel: leere Kisten (1)

1. Versuch: Generische Methode mit zwei verschiedenen Typparametern

```
static <T, U> boolean beideLeer(Kiste<T> k1, Kiste<U> k2) {
    return k1.isEmpty() && k2.isEmpty();
}
```

- liefert das gewünschte Ergebnis, aber
- die beiden Typ-Variablen werden an keiner Stelle benötigt!
- schlecht auf mehr als zwei Kisten (beliebig viele Kisten? Array von Kisten?) adaptierbar

Beispiel: leere Kisten (2)

Bessere Lösung: Verwendung von „wildcards“

```
static boolean beideLeer(Kiste<?> k1, Kiste<?> k2) {
    return k1.isEmpty() && k2.isEmpty();
}
```

Die wildcard ?

- steht für einen festen, aber zur Compile-Zeit noch unbekannten Typ
- liefert Typinformation zur Laufzeit!
- ist keine Typvariable! Die Methode ist nicht generisch!
- Ein mit wildcard parametrisierter Typ ist Obertyp für alle konkret parametrisierten Typen!

Beispiel: viele leere Kisten

```
public static boolean alleLeer(EVL<Kiste<?>> kListe) {
    boolean b = true;
    for(Kiste<?> k: kListe) {
        b = b && k.isEmpty();
    }
    return b;
}
```

Aufruf möglich mit

```
Kiste<Integer> intK = new Kiste<>();
Kiste<String> strK = new Kiste<>();
Kiste<Person> pK = new Kiste<>();
EVL<Kiste<?>> kl = new EVL<>();
kl.append(intK);
kl.append(strK);
kl.append(pK);
System.out.println("Alle leer: " + alleLeer(kl));
```

Best Practice

```
public <T extends Person> void bspMenthod(EVL<T> liste) {...}
```

und

```
public void bspMenthod(EVL<? extends Person> liste) {...}
```

sind semantisch identisch! Wann benutzen wir was?

Typparameter <T>, wenn

- der Typparameter an andere Stelle (zB als Rückgabetyp der Methode) benötigt wird
- die Methode zwei oder mehr Eingabe-Parameter hat, die denselben Typ T meinen

Wildcard „?“ wenn

- der Typparameter innerhalb der Methode nicht benötigt wird

Inhalt

Motivation

Beispiel

- gegeben: Liste von Sportlern
- gesucht: Methode zur Berechnung des Durchschnittsalters (im Jahr 2024)

```
public static int dAlter(EVL<Sportler> spListe) {
    int sum = 0;
    for(Sportler sp: spListe) {
        sum += sp.alter(2024);
    }
    return sum/spListe.size();
}
```

Problem: Invarianz der generischen Klassen

- Der Methode kann zB keine Liste von Boxer-Objekten übergeben werden
- EVL<Boxer> ist kein Untertyp von EVL<Sportler>

Upper Bounded Wildcards

Wir wollen

- den Typ des Eingabeparameters möglichst allgemein definieren
  - es sollen sowohl Listen von Boxer als auch Listen von Fussballern oder Listen von Sportlern akzeptiert werden
- den Typ des Eingabeparameters einschränken
  - auf Listen von Sportlern oder
  - Listen von Untertypen von Sportlern

Genau das leistet die Typeinschränkung `<? extends Typ>`:

```
public static int dAlter(EVL<? extends Sportler> pListe) {
    // wie oben
}
```

Upper Bounded Wildcard: Kovarianz

Upper Bounded Wildcards <? **extends** Typ> erzeugen Kovarianz: statt des angegebenen Typs kann beim Aufruf auch ein Untertyp angegeben werden.

```
EVL<Sportler> spListe = new EVL<>();
EVL<Boxer> bListe = new EVL<>();
int da;
// ... den Listen Objekte hinzufuegen ...
// Methode aufrufen fuer Listen von unterschiedlichem Typ
da = dAlter(spListe);
da = dAlter(bListe);
}
```

Upper Bounded Wildcard: ReadOnly

Konsequenz:

- Auf Objekte eines „nach oben“ beschränkten Typs kann nur lesend zugegriffen werden.
- Schreibzugriffe sind nicht erlaubt
  Ausnahme: setzen auf **null**

```
public static void
      insert(EVL<? extends Sportler> spListe, Sportler sp) {

   spListe.append(sp); // geht nicht
   spListe.append(null); // Ausnahme: das geht
}
```

Lower Bounded Wildcards

Der umgekehrte Fall: wir wollen

- den Typ des Parameters einschränken
- auf Listen von Sportlern oder Listen von Obertypen von Sportler

Genau das leistet die Typeinschränkung `<? super Typ>`

```
public static void
      insert(EVL<? super Sportler> spListe, Sportler sp) {
   spListe.append(sp);
}
```

Lower Bounded Wildcard: Kontravarianz

Lower Bounded Wildcards <? **super** Typ> erzeugen Kontravarianz: statt des angegebenen Typs kann beim Aufruf auch ein Obertyp angegeben werden.

```
Sportler s = new Sportler("Susi", 1999);
EVL<Sportler> spListe = new EVL<>();
EVL<Person> pListe = new EVL<>();

insert(spListe, s);
insert(pListe, s);
```

Lower Bounded Wildcard: WriteOnly

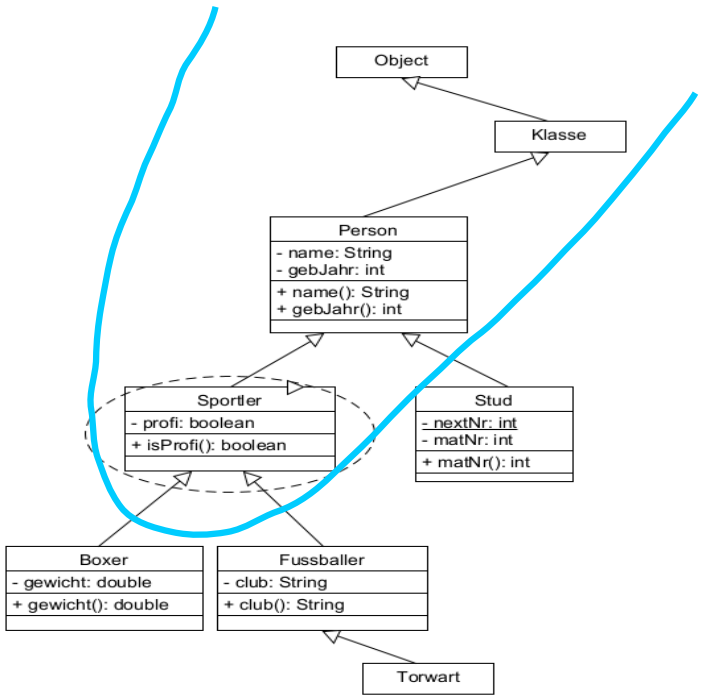
Konsequenz:

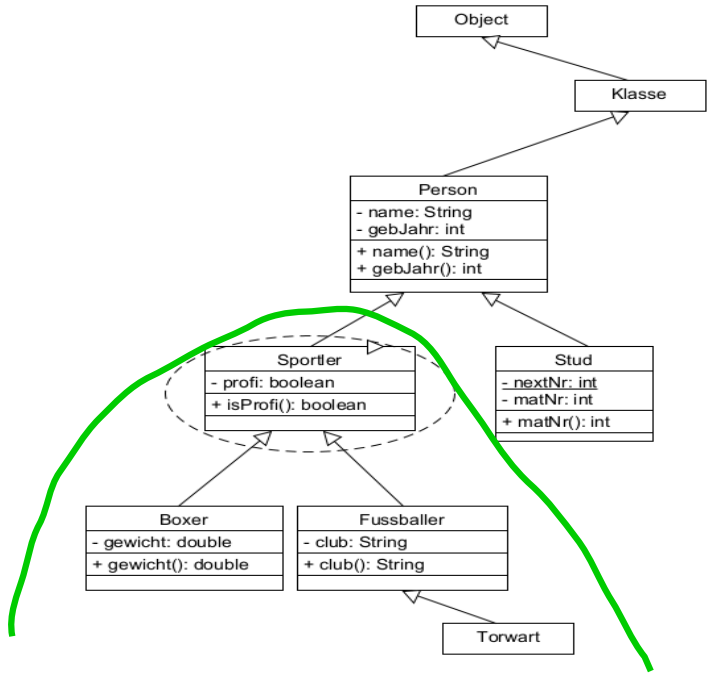
- Auf Objekte eines „nach unten“ beschränkten Typs kann nur schreibend zugegriffen werden.
- Lesezugriffe sind nicht erlaubt

```
public static void
      insert(EVL<? super Sportler> spListe, Sportler sp) {
   Sportler s = spListe.getFirst(); // geht nicht
}
```

Lower Bounded - Upper Bounded

<? super Sportler>





<? extends Sportler>

Merkregel LESS

LESS steht für

- „Lesen $\hat{=}$ Extends“
  Wenn eine Methode nur lesend auf Elemente eines generischen Parameters <T> zugreift, kann dieser Parameter auch durch jeden Untertyp ersetzt werden: <? **extends** T>

- „Schreiben $\hat{=}$ Super“
  Wenn eine Methode nur schreibend auf Elemente eines generischen Parameters <T> zugreift, kann dieser Parameter auch durch jeden Obertyp ersetzt werden: <? **super** T>