Realisierungsmöglichkeiten für parametrische Polymorphie

- **heterogen**
  Für jeden konkreten Typ wird individueller Code erzeugt (type expansion).
  Diesen Ansatz verfolgt z.B. C++ (templates directory).
  Vorteile: Typinstanziierung zur Laufzeit bekannt, mehr Optimierungsmöglichkeiten
  Nachteil: Erzeugter Objektcode wird aufgebläht (code bloat).
- **homogen**
  Kein individueller Bytecode, `Object` statt der Typvariablen, Typanpassungen für einen konkreten Typ, Typüberprüfungen nur zur Übersetzungszeit (type erasure).
  Realisierung für Java.
  Vorteile: schlanker Bytecode, einfachere Verwendung
  Nachteil: Typinstanziierung zur Laufzeit nicht bekannt.

Typlöschung (type erasure)

Zur Laufzeit ist der konkrete Typ der Typvariablen für ein Objekt nicht bekannt. Dies hat Konsequenzen für `instanceof`.

Folgendes liefert einen Übersetzungsfehler:

```
Stapel<String> stapel = new Stapel<String>();

if ( stapel instanceof Stapel<String> ) ... // Compiler-Fehler
if ( stapel instanceof Stapel ) ...         // OK
```

Grund: Zur Laufzeit liegt nur die Typinformation `Stapel` vor.

```
Stapel<String> stapel1 = new Stapel<String>();
Stapel<Integer> stapel2 = new Stapel<Integer>();
System.out.println(stapel1.getClass() == stapel2.getClass()); // true!
```

Raw-Type

Eine Instanziierung mit einem konkreten Typ muss nicht unbedingt stattfinden. Ohne Instanziierung gilt implizit `Object` als konkreter Typ.

```
Stapel stapel = new Stapel<String>();
Stapel ostapel = new Stapel();
```

Solche Typen heissen *Raw-Type*.

Raw-Typen können wir wie parametrisierten Typen verwendet werden, es findet aber keine Typüberprüfung bzgl. des Komponentenyps statt.

Eventuell gibt der Compiler Warnungen aus.

☞ Vorsicht vor Typproblemen zur Laufzeit.

Generische Felder

- Konsequenz der Typlöschung: Es können keine Felder mit instanziierten Typvariablen angelegt werden. So liefert z.B. die folgende Anweisung einen Übersetzungsfehler:

```
Stapel<String>[] stapelFeld = new Stapel<String>[10];
```

- Stattdessen kann für die Erzeugung des Feldes ein Raw-Type als Komponententyp verwendet werden. Die Zuweisung an eine Variable mit instanziiertem Typ ist möglich, liefert aber je nach Compiler eine Warnung:

```
Stapel<String>[] stapelFeld = new Stapel[10];
```

- Zugriffe auf das Feld sind wieder typsicher:

```
stapelFeld[0] = new Stapel<String>();  // OK
stapelFeld[1] = new Stapel<Integer>(); // Fehler bei Compilierung
```

- Bei der Implementierung eines generischen Typs kann die Typvariable auch nicht als Komponententyp für ein Feld verwendet werden!

- Folgendes liefert einen Übersetzungsfehler:

```
public class Stapel<T> {
    private T[] stapel = new T[4];
    ...
}
```

- Das Problem ist wieder nur die Felderzeugung, nicht die Variablendeklaration!
- Alternative: `Object` als Basistyp mit Downcast:

```
private T[] stapel = (T[]) new Object[4];
```

oder Verzicht auf Felder in eigenen generischen Klassen und stattdessen Verwendung der generischen Klassen des Java-API (z.B. `ArrayList<T>` statt `T[]`).

Kovarianz bei Kollektionen

- In Java verhalten sich Felder kovariant.
- Wenn `Fussballer` eine Unterklasse von `Sportler` ist, dann ist `Fussballer[]` ein Untertyp von `Sportler[]`.
- Daher ist folgenden problemlos möglich:

```
Fussballer[] fussballer = new Fussballer[11];
Sportler[] sportler = fussballer;
```

- In der objektorientierten Programmierung bezeichnet man dies als *Kovarianz*: Ein Aspekt (hier: Feldbildung) wird gleichartig zur Vererbungsrichtung (Fussballer ist Unterklasse von Sportler) eingesetzt.
- Kovarianz stellt bei lesendem Zugriff kein Problem dar und garantiert für diesen Fall Substituierbarkeit.

- Beispiel:

```
class Sportler { ...  public void gibAus() { ... } ... }
class Fussballer extends Sportler { ... }

public static void gebeSportlerAus(Sportler[] sportler) {
    for (int i=0 ; i<sportler.length ; i++)
        sportler[i].gibAus();
}

public static void irgendeineMethode() {
   ...
   Fussballer[] fussballer = new Fussballer[11];
   fussballer[0] = new Fussballer(...);
   ...
   gebeSportlerAus(fussballer);    // impliziter Up-Cast
   ...
}
```

Generics und Vererbung (1)

- Generics sind untereinander nicht kovariant sondern invariant!
- D.h. die Typhierarchie der Typvariablen überträgt sich nicht auf den generischen Typ.
- Beispiel: `ArrayList<Fussballer>` ist kein Untertyp von `ArrayList<Sportler>`.
- Daher liefert der Compiler für folgende Anweisungen einen Fehler:

```
ArrayList<Sportler> sportler = new ArrayList<Fussballer>();
```

- Begründung: Kovarianz ist bei schreibendem Zugriff auf eine Kollektion problematisch.

Problem der Kovarianz

- Wegen der Kovarianz bei Feldern liefert der Java-Compiler für die folgenden Anweisungen keinen Fehler bei der Übersetzung:

```
class Sportler { ... }
class Fussballer extends Sportler { ... }
class Handballer extends Sportler { ... }
...
Sportler[] sportler = new Fussballer[11];   // impliziter Up-Cast
sportler[0] = new Handballer();             // ArrayStoreException
```

- Zur Laufzeit würde aber eine `ArrayStoreException` ausgelöst, denn eine `Handballer`-Instanz kann natürlich nicht in ein Feld von `Fussballern` eingefügt werden.
- Durch den impliziten Upcast nach `Sportler[]` geht die Typinformation für das `Fussballer`-Feld verloren.

Keine Kovarianz bei Generics

- Im Gegensatz zu Feldern verhalten sich Generics nicht kovariant.
- Dies verhindert den Laufzeitfehler auf der vorangegangenen Folie:

```
class Sportler { ... }
class Fussballer extends Sportler { ... }
class Handballer extends Sportler { ... }
...
                                          // Compiler meldet Fehler
ArrayList<Sportler> sportler = new ArrayList<Fussballer>();
sportler.add(new Handballer());       // deshalb kann es hier nicht zu ei
                                          // Laufzeitfehler kommen
```

Generics und Vererbung (2)

- Wir haben gesehen, dass Kovarianz bei schreibendem Zugriff problematisch sein kann!
- Dagegen wäre folgendes vom Prinzip her unproblematisch:

```
class Mensch { ... }
class Sportler extends Mensch { ... }
class Handballer extends Sportler { ... }
...
Sportler[] sportler = new Mensch[5];
sportler[0] = new Handballer();
sportler[1] = new Sportler();
...
ArrayList<Sportler> sportler = new ArrayList<Mensch>();
sportler.add(new Handballer());
sportler.add(new Sportler());
```

würde aber nicht vom Compiler akzeptiert, weder für Felder noch für Generics.

- *Kontravarianz*: Verwendung eines Aspektes entgegen der Vererbungsrichtung
- Bei schreibendem Zugriff ist Kontravarianz typsicher.

☞ Statt festeingebauter Kovarianz bieten Generics in Java durch Typeinschränkungen und Wildcards die Möglichkeit, je nach Situation Kovarianz, Kontravarianz oder Invarianz zu definieren.

Typeinschränkungen bei Generics

Die zugelassenen konkreten Datentypen für eine Typvariable können mit `extends` auf Untertypen eines Obertyps eingeschränkt werden.

Syntax:

```
<T extends O>
```

Für die Typvariable T sind dann O und alle Untertypen von O zulässig.

Beispiel: Ein typsicheres generisches `max`:

```
public static <T extends Comparable<T>> T max(T o1, T o2)
{
   return o1.compareTo(o2) >= 0 ? o1 : o2;
}
```

- ☞ **extends** auch für Untertypen von Schnittstellen!
- ☞ Das ist besser als

```
public static <T> Comparable<T> max(Comparable<T> o1, Comparable<T> o2)
```

Warum?

Typeinschränkung mit `super`

- Mit `super` statt `extends` kann eine Einschränkung auf Obertypen statt auf Untertypen erfolgen.
- `<T super O>`
- Damit dürfen für die Typvariable T O oder Obertypen von O eingesetzt werden.
- Diese Einschränkung wird z.B genutzt, wenn eine mit T parametrisierte generische Datenstruktur manipuliert wird (Kontravarianz).

Kombinationen von Obertypen

- Soll die Typvariable T zu mehreren Obertypen passen, lassen sich Kombinationen bilden.
- Man beachte: Für T kann nur eine Oberklasse angegeben werden, da es in Java nur einfache Vererbung bei Klassen gibt.
  Weitere Obertypen müssen Schnittstellen sein.
- Beispiel: `<T extends O & I1 & I2 & I3>`
  Bedeutung: Für die Typvariable T sind alle Typen erlaubt, die Untertypen der Klasse O und der Schnittstellen I_1, I_2 und I_3 sind (also diese Schnittstellen implementieren).
  genauer: T muss von O abgeleitet sein, und I_1, I_2 und I_3 implementieren.

Generische Variablen: Wildcards

Folgendes funktioniert nicht (weil sich Generics nicht kovariant verhalten):

```
class Sportler { ... }
class Fussballer extends Sportler { ... }
class Handballer extends Sportler { ... }
class Biertrinker { ... }
...
ArrayList<Sportler> sportler;
ArrayList<Fussballer> fussballer = new ArrayList<Fussballer>();
ArrayList<Handballer> handballer = new ArrayList<Handballer>();
sportler = fussballer;     // Compiler liefert Fehler!
```

- Es ist jetzt aber möglich, statt konkreter Typangaben Wildcards anzugeben und
- diese mit zusätzlichen Einschränkungen (`extends, super`) zu versehen.

Generische Variablen: Wildcards (2)

Beispiel:

```
ArrayList<? extends Sportler> sportler;
ArrayList<Fussballer> fussballer = new ArrayList<Fussballer>();
ArrayList<Handballer> handballer = new ArrayList<Handballer>();
ArrayList<Biertrinker> biertrinker = new ArrayList<Biertrinker>();
sportler = fussballer; sportler = handballer;     // OK!
sportler = biertrinker;                           // Fehler!
```

Generische Variablen: Wildcards (3)

Wildcard mit `extends` liefert uns also Kovarianz: Für

```
public static void gebeSportlerAus(ArrayList<? extends Sportler> sportler) {
    for (Sportler s : sportler) {
        s.gibAus();
    }
}
```

ist

```
ArrayList<Fussballer> fussballer = new ArrayList<Fussballer>();
fussballer.add(new Fussballer(...)); ...
gebeSportlerAus(fussballer);
```

kein Problem mehr.

Generische Variablen: Wildcards (4)

Wildcard mit super liefert uns Kontravarianz: Für

```
public static void packEinenFussballerDrauf(Stapel<? super Fussballer> stapel) {
    stapel.push(new Fussballer());
}
```

ist

```
Stapel<Sportler> sportler = new Stapel<Sportler>();
packEinenFussballerDrauf(stapel);
```

kein Problem.

Generische Variablen: Wildcards (5)

- Kovarianz und Kontravarianz können natürlich auch zusammen auftreten.
- Beispiel: Eine Methode, die alle Elemente einer Liste `src` in eine Liste `dest` kopiert:

```
public static <T> void copy(List<? super T> dest, List<? extends T> src) {
    ...
}
```

- Kovarianz für die Liste `src`, aus der gelesen wird.
- Kontravarianz für die Liste `dest`, in die geschrieben wird.
- Nutzung:

```
List<Fussballer>  fussballer  = new ArrayList<Fussballer>();
List<Fussballer>  fussballer2 = new ArrayList<Fussballer>();
List<Sportler>    sportler    = new LinkedList<Sportler>();
List<Biertrinker> biertrinker = new ArrayList<Biertrinker>();
```

```
copy(fussballer2, fussballer);   // OK fuer T=Fussballer
copy(sportler, fussballer);      // OK fuer T=Sportler oder T=Fussballer
copy(fussballer, sportler);      // Fehler
copy(sportler, biertrinker);     // Fehler
copy(biertrinker, sportler);     // Fehler
```

The Get and Put Principle:

Use an `extends` wildcard when you only *get* values out of a structure, use a `super` wildcard when you only *put* values into a structure, and don't use a wildcard when you *both* get and put.

M. Naftalin, P. Wadler, *Java Generics*, O'Reilly, 2006.