Generics

Typen und Klassen

Erinnerung: jedes Objekt hat einen *Typ* = Klasse

Typsystem wird benutzt, um Fehler zur Compilezeit anzufangen, z.B. Zugriff auf nicht-definierte Instanzvariablen

Nachteil von Typ = Klasse: allgemeinere Funktionalitäten können nicht mit Typsystem modelliert werden

seit Java 1.5: Erweiterung des Typsystems um *Generics* = parametrisierte Typen

Beispiel: Datenstrukturen

Aufgabe: verwalte zu einem Spielbrett die Menge aller darauf angemeldeten Ameisenstämme

benötigt Datentyp "Menge von Ameisenstämmen"

kann z.B. durch `Ant[][]` modelliert werden, hat aber Nachteile:

- schlechte Zugriffszeit beim Einfügen und Suchen
- keine symbolischen Namen
- ...

bessere Ansätze siehe Vorlesung Algorithmen und Datenstrukturen

hier: besserer Ansatz erfordert eigene Klasse mit Methoden zum Einfügen, Suchen, etc.

was macht man, wenn man auch noch Mengen von anderen Objekten verwaltet möchte?

Casting

bis Java 1.4: einzig Array-Typ `[]` war generisch (aber nicht typsicher!), z.B. in

```
String[] namen = new String[10];
```

selbstverständlich:

- nur Strings können eingetragen werden
- Zugriffe liefern Objekte vom Typ `String` (oder `null`)

nicht so bei anderen Datenstrukturen, z.B.

```
List namen = new LinkedList();
namen.add("Hans Wurst");
String name = (String) namen.get(0);
```

Nachteile:

- Datenstrukturen können Elemente verschiedener Typen halten
- Laufzeitfehler, falls Cast nicht möglich
- fehleranfälliger Code (Cast vergessen)

Generics

ab Java 1.5: Typen können parametrisiert werden (sind generisch)

Bsp: nicht Typ *Liste*, sondern *Liste von Strings*

Notation in Java: `List<String>`, z.B.

```
List<String> namen = new LinkedList<Namen>();
namen.add("Hans Wurst");
String name = namen.get(0);
```

offensichtliche Vorteile, insbesondere Fehlerabfang zu Compilezeit

Variablen für Parameter

Verwendung von Objekten generischen Typs normalerweise mit konkretem Parametertyp

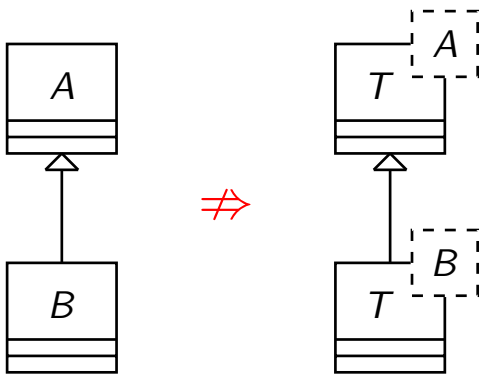
aber Typvariablen nötig für Definition; soll ja mit allen (oder bestimmten) Typen benutzt werden können, z.B.

- *Listen von Objekten* (beliebigen aber gleichen Typs)
- *Listen von Zahlen*, also sowohl `Integer`, als auch `Double`, etc.

```
public interface List<E> extends Collection<E> {
  ...
  boolean add(E o) {
    ...
  }
}
```

Generics und Vererbung

Achtung! Typkontexte erhalten Vererbungsbeziehung nicht




```
class B extends A { ... }
class C extends A { ... }

  ...
  Vector<B> bVector = new Vector<B>();
  bVector.add(0, new B());
  Vector<A> aVector = bVector;
  aVector.add(1, new C());
```

Grund: Zuweisungskompatibilitäten

Wildcards und Kovarianz

nicht weiter verwendete Typparameter können auch mit der Wildcard ? gekennzeichnet werden

```
Vector<B> bVector = new Vector<B>();
Vector<?> aVector = bVector;
```

Kovarianz = Einschränkung auf Typ und seine Untertypen

```
class T<E extends Comparable<E>> { ... }
class U<? extends A & B> { ... }
```

Zu beachten

- generische Typen sind (fast) *first-class citizens*, können also auch z.B. in Methodendefinitionen verwendet werden
- Arrays über generischen Typen können nicht angelegt werden, verwende stattdessen z.B. `ArrayList`
- schreibende Zugriffe bei kovarianten Wildcardtypen sind unzulässig
- in Kovarianz `extends` sowohl bei Vererbung wie auch Implementation von Interfaces
- Konstruktoren von Typargumenten sind nicht sichtbar

```
class T<E> {
  T() {
    ...
    new E();   // verboten
    ...
  }
}
```