PM2 Java: Generics

Generische Klassen
Generische Interfaces
Generische Typen
Raw-Types
Generische Methoden

Liste und Node

MOTIVATION

Motivation (1)

Probleme mit der Klasse *Liste*

- Die Methoden der rechts definierten ***Liste*** arbeiten mit beliebigen Objekten. Der Komponententyp in ***Liste*** ist ***Object***.
- Damit ist sie in der Verwendung sehr flexibel. Wir können sie zur Speicherung von Kunden, Verträgen, Integer etc. verwenden.
- **Probleme** bekommen wir:
  - wenn wir den Komponententyp in ***Liste*** auf einen speziellen Typ einschränken wollen.
  - wenn wir für die externen Nutzer Objekte speziellen Typs typsicher zurückgeben wollen. Bei der allgemeinen Lösung sind externe Nutzer zu einem Downcast gezwungen.

```
public class Liste {
    ...
    public Liste() {...}
    public void add(Object o){...}
    public boolean hasNext(){...}
    public Object next(){…}
    public void reset(){}
    public String toString() {…}
}
```

Motivation (1)

Typsicherheit Komponententyp

1. Wir müssten für **jeden** Komponententyp eine Subklasse von ***Liste*** erzeugen. Also ***KundeListe, VertragListe, IntegerListe, StringListe*** etc.
2. Da wir die Methode ***add*** überschreiben, können wir in der Signatur von ***add*** die Typeinschränkung zur **Compilezeit** nicht erzwingen. (keine Typsicherheit)
3. In der ***add*** Methode müssen wir zur **Laufzeit** eine Typprüfung auf den Komponententyp durchführen und ggf. eine ***Exception*** generieren.
4. In der ***next*** Methode können wir über kovariante Ergebnistypen für externe Nutzer Typsicherheit herstellen. Wir müssen dazu das Ergebnis des ***super*** Aufrufs auf den Komponententyp casten.

```
public class StringListe extends Liste {
     public void add(Object o) throws
        Exception{
          if (!(o instanceof String))
             throw new Exception("Falscher
                      Komponententyp");
          super.add(o);
     }

     public String next(){
        return (String)super.next();
     }
}
```

Motivation (1)

Aber

- Es ist nicht akzeptabel für **jeden** Komponententyp eine Subklasse von ***Liste*** zu definieren.
- Wir brauchen eine Lösung, die es erlaubt den Komponententyp in ***Liste*** bei der Typdeklaration festzulegen, so wie wir dies von Arraytypen kennen.
- Genau diese Möglichkeit erhalten wir durch **generische Typen.**
- Dazu müssen wir zuvor in **generischen Klassen** einen konkreten Komponententyp durch eine Variable ersetzen. Diese Variable heißt auch **Typvariable**.

```
public class Liste<T> {
    private T[] elements;
    public Liste() {...}
    public void add(T elem){...}
    public T next() {...}
    public boolean hasNext() {...}
    public void reset() {…}
}
```

Motivation (1)

Generische Klasse *Liste<T>*

- Rechts ist die Klasse ***Liste<T>*** als generische Klasse implementiert. ***T*** ist die Typvariable.
- Auf Basis dieser generischen Klasse erzeugen wir unten zwei generische Typen ***Liste<String>*** und ***Liste<Integer>***.
- Der erste Fehler zeigt, dass Typsicherheit für das Hinzufügen von Elementen bereits zur Compilezeit garantiert wird.
- **Anmerkung:** Durch Substitution der Typvariablen ***T*** durch einen konkreten Typ werden Methoden unterschiedlich instanziiert, aber nicht überladen.

```
public class Liste<T> {
    private T[] elements;
    public Liste() {...}
    public void add(T elem){...}
    public T next() {...}
    public boolean hasNext() {...}
    public void reset() {…}
}
```

```
Liste<String> ls = new
    Liste<String>();
ls.add(34); // Compilerfehler
String s = ls.next();
Liste<Integer> li = new
    Liste<Integer>();
li.add(34);
String s = li.next(); //
    Compilerfehler
```

Motivation(2): Node Beispiel

- siehe Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT) Kap. 13

GENERISCHE KLASSEN / INTERFACES / TYPEN

Motivation(2): Node Beispiel

- siehe Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT) Kap. 13

TYPEBOUNDS

Typebounds

- siehe Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT) Kap. 13

Typebounds mit Typvariablen: Beispiel *Regal*

- **Aufgabe:**
  - Ihr Unternehmen betreibt ein eigenes Warenlager. Das Lager besteht aus mehreren Regalen. Jedes Regal nimmt Kisten mit Waren auf. Jedes Regal enthält nur Kisten eines Warentyps.
- **Lösung**:
  - Um den Warentyp in den Kisten variabel zu halten, definieren wir eine generische Klasse ***Kiste<W>***.
  - Die generische Klasse ***Regal*** schränkt den Typ der Waren ein. Diese Einschränkung formulieren wir über die Typvariable ***C***.
  - ***Regal*** hat eine zweite Typvariable für die aufzunehmenden Kisten (***K extends Kiste< C>***). Die Waren in den Kisten müssen alle vom gleichen Typ ***C*** sein.
- **Source**: ***generics.typeboundtypevar***

```
public class Kiste<W> {
      Liste<W> liste;
      public Kiste(){
          liste = new Liste<W>();
      }
      public void add(W ware){
          liste.add(ware);
      }
      //...
}
public class Regal<C, K extends Kiste<C>>
    {
      private Liste<K> kisten;
      public Regal(){
          kisten = new Liste<K>();
      }
      void add(K kiste){
          kisten.add(kiste);
      }
}
```

Erzeugen und Befüllen von Regalen

- Ihr Baumarkt hat eine Regalabteilung nur für Fliesen.
- Wir erzeugen eine Regalinstanz, in der nur Fliesen zugelassen sind ***rf***.
- Nur Kisten, die Fliesen enthalten (***kf***), dürfen in das Regal.
- Kisten mit Lebensmitteln (***kl***) sind nicht zulässig.

```
Regal<Fliesen,Kiste<Fliesen>> rf = new
    Regal<Fliesen, Kiste<Fliesen>>();
Kiste<Fliesen> kf = new Kiste<Fliesen>();
kf.add(new Fliesen());
Kiste<Lebensmittel> kl = new
    Kiste<Lebensmittel>();
kl.add(new Lebensmittel());

rf.add(kf); // ok

// Compilerfehler
// The method add(Kiste<Fliesen>) in the
// type Regal<Fliesen,Kiste<Fliesen>> is
// not applicable for the arguments
//(Kiste<Lebensmittel>)
rf.add(kl);
```

TYPKOMPATIBILITÄT

Typkompatibilität

- siehe Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT) Kap. 13

Typkompatibilität und Varianzen der generischen (Wildcard)Typen

- **Invarianz**
  - generische Typen ohne Wildcards **C<T>** sind invariant.
  - sind nur kompatibel, wenn die Typargumente vom selben Typ sind.
  - Lesen zulässig
  - Schreiben zulässig
- **Bivarianz**
  - Wildcard-Typen **C<?>** ohne Einschränkungen sind bivariant.
  - Alle generischen Typen der gleichen generischen Klasse wie der Wildcard-Typ sind kompatibel zum Wildcard-Typen.
  - Schreiben unzulässig
  - Lesen unzulässig
- **Kovarianz**
  - Wildcard-Typen mit **Upper-Typebound** ***C<? extends B>***
  - alle generischen Typen, deren Typargument zu **B** kompatibel ist, sind kompatibel zum Wildcard-Typ.
  - Schreiben unzulässig
  - Lesen zulässig
- **Kontravarianz**
  - Wildcard-Typen mit Lower-Typbounds ***C<? super B>***
  - alle generischen Typen, deren Typargument Supertyp von ***B*** ist, sind kompatibel zum Wildcard-Typ.
  - Schreiben zulässig
  - Lesen unzulässig

Das vollständige Beispiel

im Package **generics.sortednode** des Projektes **v12-Generics**

bivariante kovariante kontravariante

WILDCARDTYPEN

Wildcardtypen

- siehe Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT) Kap. 13

Typkompatibilität und Varianzen der generischen (Wildcard)Typen

- **Invarianz**
  - generische Typen ohne Wildcards **C<T>** sind invariant.
  - sind nur kompatibel, wenn die Typargumente vom selben Typ sind.
  - Lesen zulässig
  - Schreiben zulässig
- **Bivarianz**
  - Wildcard-Typen **C<?>** ohne Einschränkungen sind bivariant.
  - Alle generischen Typen der gleichen generischen Klasse wie der Wildcard-Typ sind kompatibel zum Wildcard-Typen.
  - Schreiben unzulässig
  - Lesen unzulässig
- **Kovarianz**
  - Wildcard-Typen mit **Upper-Typebound** ***C<? extends B>***
  - alle generischen Typen, deren Typargument zu **B** kompatibel ist, sind kompatibel zum Wildcard-Typ.
  - Schreiben unzulässig
  - Lesen zulässig
- **Kontravarianz**
  - Wildcard-Typen mit Lower-Typbounds ***C<? super B>***
  - alle generischen Typen, deren Typargument Supertyp von ***B*** ist, sind kompatibel zum Wildcard-Typ.
  - Schreiben zulässig
  - Lesen unzulässig

GENERISCHE METHODEN

Generische Methoden

- siehe Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT) Kap. 13

MapUtils mit generischen statische Methode

- **Aufgabe**: Schreiben Sie die Methoden ***intersect***, ***union***, ***isSubMap***, als generische statische Methoden der Klasse ***MapUtils***

- **Lösung in 2 Schritten**:
  - Lösung 1 mit Typvariablen, die bei der Konkretisierung zu invarianten Typen werden:
    - Typvariablen für die Key-Value Typen der ***Map*** Parameter und den Rückgabewert
    - **Nachteil**: es lassen sich nur zwei Maps zusammenführen, die identische Typen für Schlüssel und Werte haben
  - Lösung 2 allgemeiner mit kovarianten Wildcard-Ausdrücken:
    - kovariante Wildcardtypen für Key-Value Typen der ***Map*** Parameter und Typvariablen für den Rückgabewert
    - **Vorteil**: Es lassen sich auch Maps zusammenführen, deren Key-Value Paare jeweils einen gemeinsamen Supertyp haben.

- Vorgehen erläutert am Beispiel der Methode ***intersect***

Generische statische Methode *intersect* (1)

Um die Variablen zu binden müssen diese dem Rückgabewert vorangestellt werden.

Das Ergebnis ist eine generische Map mit Typvariablen T,U

```
public static <T,U> Map<T,U> intersect1(
    Map<T,U> m1, Map<T,U> m2){


    Map<T,U> intersection = new HashMap<>();

    for(Map.Entry<T,U> entry: m1.entrySet()) {
        T m1key = entry.getKey();
        if (m2.containsKey(m1key) && m2.get(m1key).equals(entry.getValue())) {
            intersection.put(m1key,entry.getValue());
        }
    }
    return intersection;
}
```

Anstelle fester Typen werden für Key und Values beider Maps die Typvariablen T,U verwendet

Beim Iterieren über Map m1 haben dann die Entries die gleichen Typvariablen T,U

Generische statische Methode *intersect* (2)

Um die Variablen zu binden müssen diese dem Rückgabewert vorangestellt werden.

Das Ergebnis ist eine generische Map mit Typvariablen T,U

```
public static <T,U> Map<T,U> intersect2(
        Map<? extends T,? extends U> m1,
        Map<? extends T,? extends U> m2){

    Map<T,U> intersection = new HashMap<>();

    for(Map.Entry<? extends T,? extends U> entry: m1.entrySet()) {
        T m1key = entry.getKey();
        if (m2.containsKey(m1key) && m2.get(m1key).equals(entry.getValue())) {
            intersection.put(m1key,entry.getValue());
        }
    }
    return intersection;
}
```

Da aus m1 und m2 nur gelesen wird, können wir kovariante Typen für die Key, Value Paare verwenden.

Beim Iterieren über Map m1 haben dann die Entries die gleichen kovarianten Typen ? extends T, ? extends U

Aufrufbeispiele für Lösung 1

```
Map<Integer,Integer> mii1 = new HashMap<>();
Map<Integer,Integer> mii2 = new HashMap<>();
Map<Double,Integer> mdi1 = new HashMap<>();
Map<Double,Integer> mdi2 = new HashMap<>();
Map<Float,Integer> mdf = new HashMap<>();
Map<Polar,Polar> mpp1 = new HashMap<>();
Map<Polar,Polar> mpp2 = new HashMap<>();
Map<Cartesian,Polar> mcp = new HashMap<>();

// ok, da in allen Fällen die Typen für Schlüssel und Wert identisch sind
MapUtils.intersect1(mii1,mii2);
MapUtils.intersect1(mdi1,mdi2);
MapUtils.intersect1(mpp1,mpp2);
// Fehler, da die Typen der Schlüssel unterschiedlich sind
//MapUtils.intersect1(mdi1,mdf);
// MapUtils.intersect1(mpp1,mcp);
```

Nur, wenn die Typen für Schlüssel und Wert der Maps m1 / m2 paarweise identisch sind, ist der Aufruf von intersect1 möglich.

Auch wenn die Typen für Schlüssel und Wert der Maps m1 / m2 paarweise einen gemeinsamen Supertyp haben, ist der Aufruf von intersect1 nicht möglich.

Aufrufbeispiele für Lösung 2

```
Map<Integer,Integer> mii1 = new HashMap<>();
Map<Integer,Integer> mii2 = new HashMap<>();
Map<Double,Integer> mdi1 = new HashMap<>();
Map<Double,Integer> mdi2 = new HashMap<>();
Map<Float,Integer> mdf = new HashMap<>();
Map<Polar,Polar> mpp1 = new HashMap<>();
Map<Polar,Polar> mpp2 = new HashMap<>();
Map<Cartesian,Polar> mcp = new HashMap<>();

// ok, da in beiden Fällen die Typen für Schlüssel und Wert identisch
sind
MapUtils.intersect2(mii1,mii2);
MapUtils.intersect2(mdi1,mdi2);
// ok, da die Typen der Schlüssel den gemeinsamen Supertyp Number haben
Map<Number,Integer> mnn = MapUtils.intersect2(mdi1,mdf);
// ok, da die Typen der Schlüssel und Wert den gemeinsamen Supertyp
AbstractComplex / Complex haben
Map<Complex, Complex> mcc = MapUtils.intersect2(mpp1,mcp);
```

Wenn die Typen für Schlüssel und Wert der Maps m1 / m2 paarweise identisch sind, ist der Aufruf von intersect2 möglich.

Aufgrund der Kovarianz aber auch mit Typargumneten, die einen gemeinsamen Supertyp haben

TYPE-ERASURE

Type-Erasure

- siehe Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT) Kap. 13

GRENZEN GENERISCHER TYPEN

Grenzen generischer Typen

- siehe Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT) Kap. 13

Grenzen generischer Typen: keine generischen Array-Objekte

- Obwohl generische Array-Typen erlaubt sind - ***T[] tAry*** ist eine gültige Typdeklaration -
- ist es unmöglich generische Array-Objekte zu erzeugen: ***new T[100]*** erzeugt einen Compilerfehler.
- **Grund:** Typdeklarationen werden vom Compiler ausgewertet. Objekterzeugung ist ein Vorgang zur Laufzeit. Zu diesem Zeitpunkt ist die Information über den generischen Typ bereits verloren.
- **Problem**: Unsere generische ***Liste<T>*** muss intern ein generisches Array mit Komponententyp ***T*** erzeugen können.
- Heißt das, dass wir keine generische Liste implementieren können?

```
public class Liste<T> {
      T[] elements;
      ...
      int capacity;

      public Liste() {
            capacity = 10;
            ...
            //Compilerfehler
            //Cannot create a generic array
            // of T
            elements = new T[capacity];
      }
}
```

Grenzen generischer Typen: keine generischen Array-Objekte

- **Nein!** Alle Implementierungen von ***List<T>*** der Java Collection API verwenden intern ein Array.
- **Lösung 1:** Wir casten ein ***Object*** Array auf ***T[].***
- **Problem**: Damit unterlaufen wir allerdings die statische Typprüfung, was durch eine Compiler-Warnung angezeigt wird.

➔ Die generische Klasse muss Typfehler mit dem privaten Array verhindern.

- **Lösung 1:** Die Klasse stellt in den Zugriffsmethoden (z.B. ***add,get,set***) beim Eintragen und Lesen der Elemente den Komponententyp sicher.

```
public class Liste<T> {
      T[] elements;
      ...
      private int capacity;
      public Liste() {
            capacity = 10;
            ...
            // Compiler-Warnung:
            // Type safety: Unchecked cast
            // from Object[] to T[]
            elements =
                  (T[])new Object[capacity];
      }
      public void add(T x){    ...}
      }
      public T get(int index){...}
      public void set(int index,T x){...}
}
```

Grenzen generischer Typen: keine generischen Array-Objekte

- **Lösung 1**: Die Klasse stellt in den Zugriffsmethoden (z.B. ***add, get(int i)***) beim Eintragen und Lesen der Elemente den Komponententyp sicher.
- **Vorsicht:** Im Beispiel wurde die Sichtbarkeit des internen Arrays ***elements*** mit „böser" Absicht auf package private gesetzt.
- Darüber ist es nun möglich die typsicheren Zugriffsmethoden zu unterwandern und Laufzeitfehler zu erzeugen.
- Die im Folgenden gezeigten ***Exceptions*** stammen alle aus der Inkonsistenz des ***Object[]*** Arrays der generischen Klasse ***Liste<T>***, das nach Type-Erasure übrigbleibt und dem Cast auf ***Integer[]***, der durch Type-Erasure bei der Verwendung des generischen Typen in den Quelltext eingefügt wird.
- Das schauen wir uns jetzt genauer anhand von Beispielen an.
- Zuerst betrachten wir das Ergebnis der Type-Erasure auf der generischen Klasse ***Liste<T>.***
- Dann den Effekt von Type-Erasure bei der Verwendung des generischen Typen ***Liste<Integer>.***

Liste<T> nach Type-Erasure

- Der Cast auf (*T[]*) ist wirkungslos, da Type-Erasure die Typvariable *T* in *Object* übersetzt.
- Im vorbearbeiteten Quelltext steht an allen Stellen ein *Object*-Array.

```
public class Liste {
      Object[] elements;
      ...
      private int capacity;
      public Liste() {
            capacity = 10;
            ...
            // Compiler-Warnung vorher
            // Type safety: Unchecked cast
            // aus T wird Object
            elements = (Object[])
                        new Object[capacity];
      }
      public void add(Object x){...}
      public Object get(int index){...}
      public void set(int index, x){...}
}
```

Type-Erasure auf dem generischen Typ *Liste<Integer>*

- Nach der statischen Typprüfung ist ***li.elements*** in ***Liste<Integer>*** deklariert als ***Integer[] elements*** also kompatibel mit dem Typ von ***ielems***.
- Zur Laufzeit aber, nach Type-Erasure in der Klasse ***Liste<T>*** ist ***li.elements*** immer vom Typ ***Object[]***.
- Daher kommt es zur Laufzeit immer zu einer ***ClassCastException***.

```
Liste<Integer> li = new Liste<Integer>();

// statischer Typ li.elements: Integer[]
// nach Type-Erasure: Object[]
// Zuweisung erlaubt, aber Laufzeitfehler
// ClassCastException

Integer[] ielems = li.elements;
```

Type-Erasure auf dem generischen Typ *Liste<Integer>*

- Wir umgehen das Problem, in dem wir ***li.elements*** einer Variable oi vom kompatiblen Typ ***Object[]*** zuweisen.
- Die Zuweisung ist ok. Das Einfügen des Wertes ***35*** in ***oi*** ebenfalls.
- Die Probleme beginnen, wenn wir jetzt auf Element des Arrays ***li*** lesend oder schreibend zugreifen.
- Ursache: Type-Erasure beim Zugriff auf Elemente des Arrays fügt einen Cast auf ***Integer[]*** vor ***li.elements*** ein.

```
Liste<Integer> li = new Liste<Integer>();
// Erlaubt Integer Subtyp von Object
Object[] oi = li.elements;
oi[0]  = 35;

// Laufzeitfehler: ClassCastException
p(li.elements[0]);

// Grund: Type-Erasure übersetzt in
// p(((Integer[])li.elements)[0]);
// li.elements liefert immer Object[]
// wegen Type-Erasure in Liste<T>
// Object[] nicht kompatibel zu Integer[]

// Laufzeitfehler: ClassCastException
li.elements[1]= 56;

// Grund: Type-Erasure übersetzt in
//((Integer[])li.elements)[1]= 56;
// Argumentation wie bei Elementzugriff
```

Generische Arrays als Rückgabewerte von Methoden

- Im Beispiel ***Liste<T>*** geben wir ungeschützt eine Referenz auf ein internes Array zurück. Dies verletzt die Regeln der Datenkapselung und wäre eh schlechter Stil.
- Es gibt allerdings Situationen, in denen als Ergebnis einer Methode ein generisches Array gefordert ist. (z.B. die Methode ***toAry()***)
- Dabei muss sicher gestellt sein, dass der Arraytyp zur Laufzeit mit dem Typ zur Compilezeit verträglich ist. Wir erreichen das, indem wir den Komponententyp des Arrays erst zur Laufzeit festlegen.
- Dazu übergeben wir den Komponenten Typ als **Typetoken** bei der Erzeugung eines ***GenericArray***.
- Zuerst betrachten wir das alte Lösungsmuster:

```
public class GenericArray<T> {
      private T[] ary;
      public T[] toAry(){
         return ary.clone();
      }
      public GenericArray(int cap) {
         ary = (T[])new Object[cap];
      }
      public void set(int index, T x){
         ...}

      public T get(int index){...}
}
```

Generische Arrays als Rückgabewerte von Methoden

- Mit dem alte Lösungsmuster, das ein ***Object[]*** Array auf (***T[]***) castet, erhalten wir mit der Klasse ***GenericArray<T>*** Laufzeitfehler.
- **Lösung:** Wir verschieben das Erzeugen des Arrays und die Festlegung des Komponententyps auf die Laufzeit.
- Da wir jetzt erst zur Laufzeit den Typ des Array-Objekts festlegen, treten die Typ-Konflikte zur Laufzeit nicht mehr auf.

```
GenericArray<String> ga = new
          GenericArray<String>(10);

//String[] sAry = ga.toAry();
// erlaubt
Object[] oi = ga.toAry();
oi[0]  = "foo";

// Laufzeitfehler: ClassCastException
p(ga.toAry()[0]);
ga.toAry()[1]= "bar";

// beides ok vorher die fehlerhaften
// auskommentieren!!!!
ga.set(1,"foo");
p(ga.get(1));
```

Generische Arrays als Rückgabewerte von Methoden

- Festlegen des Komponententyps zur Laufzeit:
- Wir erzeugen das Array mit ***Array.newInstance*** und einem Komponententyp, der Parameter des Konstruktors ist (Übergabe eines Type-Tokens).

1. ***Array.newInstance(Class<T> type, int sz)*** erzeugt ein Array ***gaOk*** der Länge ***sz*** mit Komponententyp ***T***.
2. Den Komponententyp ***String*** legen wir beim Erzeugen des Array-Objektes dynamisch fest: ***new GenericArray<String>(10,String.class)***
3. Dann ist das interne Array in ***gaOk*** vom Typ ***String[]*** und **nicht** vom Typ ***Object[]*** wie in ***gaCorrupt.***

```
public class GenericArray<T> { ...
     public T[] toAry(){return
        ary.clone();}
     public GenericArray(int sz,
                         Class<T> type) {
        ary =
          (T[])Array.
              newInstance(type, sz);
     }
     ...
}

GenericArray<String> gaCorrupt = new
    GenericArray<String>(10);
GenericArray<String> gaOk = new
    GenericArray<String>(10,String.class);
p(gaCorrupt.toAry());
p(gaOk.toAry().getClass());
```

```
[Ljava.lang.Object;@19821f
class [Ljava.lang.String;
```

Generische Arrays als Rückgabewerte -- die Alternative

- **Damit jetzt keine Panik ausbricht!**

- In den meisten Anwendungsfällen, in den Objektsammlungen notwendig sind, sollten Sie die Collection Klassen des Java APIs anstelle von Arrays verwenden.

- Dann stellt die Java Bibliothek Typsicherheit her und Sie müssen sich nicht mit den Tücken des generischen Typsystems im Zusammenhang mit Arrays und Type-Erasure „herumschlagen“.

- Zu weiteren Details dieser Materie verweise ich auf ***Bruce Eckel, Thinking in Java, Kap. Generics.***

Grenzen generischer Typen: keine generischen Basistypen und Exceptions

Generische Basistypen

- Typvariablen dürfen **nicht als Basistypen** verwendet werden
- **Grund**:
  - Jeder Konstruktor einer abgeleiteten Klasse muss den Konstruktor der Basisklasse aufrufen
  - wenn die Basisklasse unbekannt ist, ist auch der Konstruktor nicht bekannt

Generische Exception

- Generische Typen ***dürfen nicht*** von `Throwable` ableiten:
- Generische Typen ***können nicht*** in den `catch` Klauseln „gefangen“ werden
- ***Grund***: der exakte Typ einer Exception muss sowohl zur Compilezeit als auch zur Laufzeit bekannt sein.
- Typvariablen ***dürfen*** in den `throws` Klauseln einer Methoden Deklaration verwendet werden.

ZUSAMMENFASSUNG

Zusammenfassung

- **Generische Klassen und Interfaces** sind Vorlagen für parametrische Typen. Parametrisierung erfolgt über **Typvariablen**.
- **Generische Klassen und Interfaces** sind selbst noch **keine gültigen Typen**. Sie definieren eine Typfamilie.
- Aus generischen Klassen und Interfaces entstehen **generische Typen**, indem die Typvariablen über **Typargumente** konkretisiert werden. Typargumente müssen **gültige Typausdrücke** sein.
- Generische Typen sind **invariant**: Generische Typen der gleichen Klasse / des gleichen Interfaces sind nur **kompatibel**, wenn die Typargumente identischen Typ haben.
- Bei **invarianten** generischen Typen ist das **Lesen und Schreiben** der Typargumente erlaubt.

Zusammenfassung

- **Wildcard-Typen**
  - sind generische Typen mit dem Typargument **?**
  - Zu einem Wildcard-Typ sind alle generischen Typen der gleichen Klasse / des gleichen Interfaces kompatibel.
  - sind **bivariant**: Weder Lesen noch Schreiben der Typargumente ist erlaubt

- **Wildcard-Typen mit Upper Typebound**
  - → sind generische Typen mit dem Typargument **? extends <Typausdruck>**
  - → **Typausdruck** beschränkt den Typ der Typvariable
  - → Zu einem Wildcard-Typen mit Upper Typebound sind alle generischen Typen kompatibel, deren Typargument ein Subtyp des Upper Typebound ist.
  - → sind **kovariant**: Lesen der Typargumente ist erlaubt, Schreiben ist nicht erlaubt.

- **Wildcard-Typen mit Lower Typebound**
  - → sind generische Typen mit dem Typargument **? super <Typausdruck>**
  - → **Typausdruck** beschreibt den Typ, die die Typvariable minimal einnehmen darf.
  - → Zu einem Wildcard-Typen mit Lower Typebound sind alle generischen Typen kom-patibel, deren Typargument ein Supertyp des Lower Typebound ist.
  - → sind **kontravariant**: **Schreiben** der Typargumente ist erlaubt, Lesen ist nicht erlaubt.

Zusammenfassung

- **Generische Methoden** parametrisieren Methodenparameter über Typvariablen. Statische Methoden müssen generische Methoden sein, wenn Sie Typvariablen enthalten.

- Generische Typen gibt es in Java nur zur **Compilezeit.** Die **Laufzeit** kennt **keine** generischen Typen. Die generischen Typen werden vom Compiler in **Raw-Types** übersetzt. Nur in den Signaturattributen bleiben die Typvariablen erhalten. Dieser Vorgang wird **Type-Erasure** genannt.

- Type-Erasure ist der Grund für viele **Einschränkungen** in der Verwendung von generischen Typen.

- Vorsicht **bei Array-Attributen mit generischem Komponententyp**: Kein direkter Zugriff auf das Array von außen. ***set*** und ***get*** Me-thoden müssen in der Signatur sicherstellen, dass nur kompatible Elemente eingetragen / gelesen werden.

- Soll ein Array mit generischem Komponententyp nach außen gegeben werden, dann darf der Komponententyp des Arrays nur zur Laufzeit festgelegt werden.

Quelle

- Reinhard Schiedermeier: *Programmieren mit Java I*, (Pearson Studium IT)
- Bruce Eckel: *Thinking in Java*, Kap. Generics.