```
public static <T extends Object & Comparable<? super T>>
T max(Collection<? extends T> coll)
```

```
/** Add an element t to a Set s */
public static <T> void
addToSet(Set<T> s, T t) {...}
```

Polymorphie in Java

oder

Generics und Co.

Ein Vortrag über Neuerungen in Java 1.5
von Stanley Rost

```
TreeSet(Comparator<? super E> c)
```

Gliederung

- Was sind Generics?
- Java Collections Framework
- Entwicklung der Generics
- **“Tutorial**
- Integration
- Weitere Neuerungen in Java 1.5

Was bedeutet “Generics”?

Generisch?

Generation?

Genmanipuliert?

Bedeutung

- ge·ner·ic [ʼdʒɪˈnerɪk] *adj*: ~ ***term*** *(od.* ***name****)*
  biol. Gattungsname *m*; *allg.* Oberbegriff *m*.
- In Java:
  - Abstraktion über Typen
  - Typsicherheit z.B. im Collections Framework

Einsatz von Generics

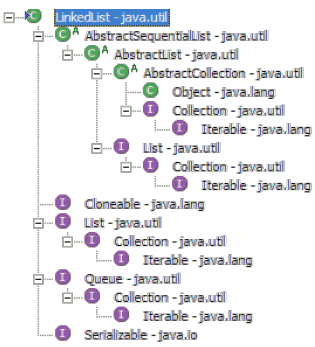
- Container-Typen

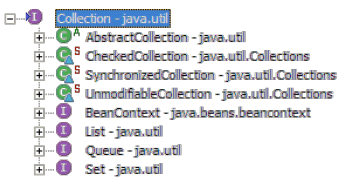
```
List myIntList = new LinkedList()
myIntList.add(new Integer(0));
Integer x = (Integer) myIntList.iterator().next();
```

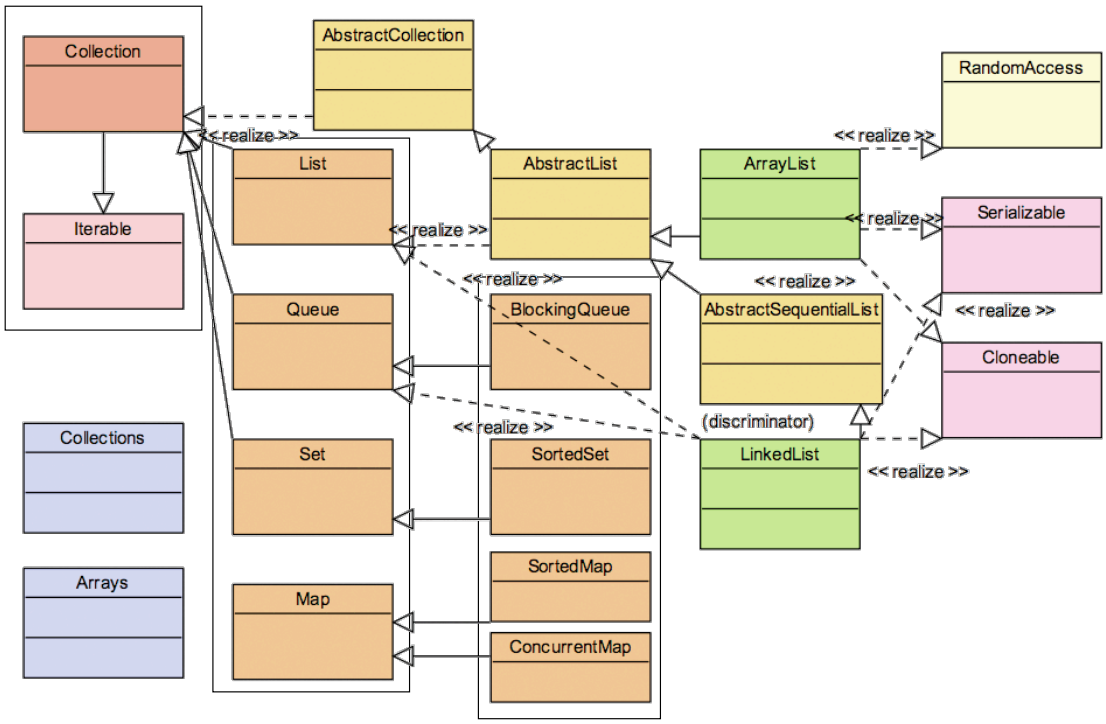
```
List<Integer> myIntList = new LinkedList<Integer>();
myIntList.add(0);
Integer x = myIntList.iterator().next();
```

Das Collections-Framework

Ein Überblick







Historie

der Generics

1848 1857 1899 1912 - 26

Historie der Generics

- Mai 1999 JSR Approval (Java Specification Request)
- September 2000 Community Review
- Mai - August 2001 Public Review
- als Basis GJ (Pizza), PolyJ (MIT)
- Gilad Bracha, Sun Microsystems Inc.

Tutorial

public static <T extends Object & Comparable<? super T>>
T max(Collection<? extends T> coll)

Simple Generics

```
public interface List<E> {
   void add(E x);
   Iterator<E> iterator();
}
public interface Iterator<E> {
   E next();
   boolean hasNext();
}
```

- E = formaler Typparameter

```
List<Integer> l = new ArrayList<Integer>();
```

Subtypen

```
List<String> ls = new ArrayList<String>();
```

```
List<Object> lo = ls;
```
 Fehler

```
lo.add(new Object());
```

```
String s = ls.get(0);
```

Wildcards (1)

```
void printCollection(Collection c) {
   Iterator i = c.iterator();
   for (k = 0; k < c.size(); k++) {
       System.out.println(i.next());
   }
}
```

```
void printCollection(Collection<Object> c) {
   for (Object e : c) {
       System.out.println(e);
   }
}
```

Wildcards (2)

```
void printCollection(Collection<?> c) {
   for (Object e : c) {
      System.out.println(e);
   }
}
```

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // compile time error
```

 Fehler

Bounded Wildcards (1)

```
public abstract class Shape {
    public abstract void draw(Canvas c);
}
public class Circle extends Shape {
    private int x, y, radius;
    public void draw(Canvas c) { ... }
}
public class Rectangle extends Shape {
    private int x, y, width, height;
    public void draw(Canvas c) { ... }
}
```

```
public class Canvas {
    public void draw(Shape s) {s.draw(this); }
}
```

```
public void drawAll(List<Shape> shapes) {
    for (Shape s: shapes) { s.draw(this); }
}
```

```
public void drawAll(List<? extends Shape> shapes) { ... }
```

Bounded Wildcards (2)

```
public void addRectangle(List<? extends Shape>
shapes) {
    shapes.add(0, new Rectangle());
}
```

 Fehler

Generic Methods (1)

```
static void fromArrayToCollection(Object[] a,
Collection<?> c) {
   for (Object o : a) {
       c.add(o); // compile time error
   }
}
```

Fehler

```
static <T> void fromArrayToCollection(T[] a,
Collection<T> c) {
   for (T o : a) {
       c.add(o); // correct
   }
}
```

- Universelle Methode!

Generic Methods (2)

- Generische Methoden oder Wildcards?

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

```
class Collections {
   public static <T, S extends T> void
   copy(List<T> dest, List<S> src) {...}
}
```

Legacy Code

- `void abc(Collection c) {...} // raw type`
- `abc(new Collection<String>()); // unchecked warning`
-
- ```
  void abc(Collection<String> c) {
     for(String s: c) {...}
  }
  ```
- `abc(new Collection()); // unchecked warning`
- Aufruf von legacy code aus generic code unsicher!

Integration


  ```

Integration

- Collection Framework 100%
- java.lang.Class<T>
  - Typ von String.class = Class<String>, Typ von Serilizable.class = Class<Serializable>
  - Verbessert Typsicherheit in Reflection
  - Class.newInstance() gibt jetzt den aktuellen Typ zurück (Factory-Methoden)

Integration

Erfahrungsbericht

- ```
  new javax.swing.table.TableModel(Vector  data,
  Vector columnNames);
  new javax.swing.table.TableModel(Object[][]  data,
  Object[] columnNames);
  ```
- ```
  new javax.swing.table.TableModel(List<List<?>  data,
  List<?> columnNames);
  ```
- ```
  Map<String, Integer> typePool = new HashMap<String,
  Integer>();
  Integer typeCount = typePool.get(className);
  // int typeCount = typePool.get(className);
  typeCount++;
  ```
- Fazit: Man hätte es vorher genauso programmieren können, nur waren an einigen Stellen Arrays praktischer, weil typsicher und ich hatte viel java.util.Vektor benutzt.
  ```

Erasure (1)

```
public String loophole(Integer x) {
   List<String> ys = new LinkedList<String>();
   List xs = ys; xs.add(x); // compile-time unchecked warning
   return ys.iterator().next();
}
```

- zur Laufzeit:

```
public String loophole(Integer x) {
   List ys = new LinkedList();
   List xs = ys;
   xs.add(x);
   return (String) ys.iterator().next(); // run time error
}
```

- ClassCastException

Erasure (2)

- Löscht alle generischen Typinformationen
- Z.B. Eine parametrisierte List<String> wird List
- Alle verbleibenden Auftreten von Typvariablen werden durch ihre “upper bound” ersetzt (normalerweise Object)
- Einfügen von Casts an den entsprechenden Stellen

Zur Laufzeit

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

```
true!!!
```

- Alle Instancen einer generischen Klasse haben dieselbe Runtime-Klasse
- Generisch heißt: Die Klasse verhält sich für alle Parameter-Typen gleich

Casts

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) ...// illegal
```

```
Collection<String> cstr = (Collection<String>) cs;
// unchecked warning
```

```
<T> T badCast(T t, Object o) {
   return (T) o; // unchecked warning
}
```

- Arrays können nicht parametrisiert werden (außer durch “unbounded wildcards”)

Zusammenfassung

- Typ-Variablen existieren zur Laufzeit nicht!
- Das bedeutet: Kein Performance Overhead
- Aber: Keine typsicheren Casts
- “If your entire application has been compiled without unchecked warnings using `javac -source 1.5`, it is type safe.”

Generics & Co

Autoboxing, Neue for-Schleife, Typsichere Enumerationen, Varargs, Statische Importe, Annotations

Autoboxing

- Wrapper für primitive Typen werden automatisch eingesetzt.

```
List<Integer> list = new ArrayList<Integer>();
list.add(1);
int i = list.iterator().next();
```

```
Integer i1 = new Integer(5);
int i2 = 8;
int i3 = ++i1 + i2;
```

Erweiterte for Schleife

```
void cancelAll(Collection c) {
   for(Iterator i = c.iterator(); i.hasNext();) {
       TimerTask tt = (TimerTask) i.next();
       tt.cancel();
   }
}
```

```
void cancelAll(Collection<TimerTask> c) {
   for(TimerTask tt: c) {
       tt.cancel();
   }
}
```

- Funktioniert auch mit Arrays und verschachtelten Iterationen

Typsichere Enumerationen

- ```
  enum  Farbe {EICHEL, GRUEN, HERZ, SHELL}
  enum Wert {SIEBEN, ACHT, NEUN, ZEHN< BUBE, DAME,
  KOENIG, ASS}
  ```

- ```
  List<Card> spiel = new ArrayList<Card>();
  for(Farbe farbe: Farbe.values())
     for(Wert wert: Wert.values())
         spiel.add(new Card(farbe, wert));
  Collections.shuffle(spiel);
  ```

- Würde ohne Java 1.5 mehrere Seiten Code benötigen!!!

Varargs

- ```
  public static String format(String pattern,
  Object... arguments)
  ```

- ```
  String result = MessageFormat.format(”At {1, time}
  on {1,date}, there was {2} on planet “ +
  “{0,number,integer}.”, 7, new Date(), “a disturbance
  in the Force”);
  ```

- Parametertyp von `arguments` ist `Object[]`

- Aufrufer muß neue Syntax nicht benutzen

Statische Importe

- Nützlich z.B. für mathematische Klassen

```
import static java.lang.Math.*;
...
double x = sin(PI / 2);
```

Annotations (1)

Metadata

- Beispiel JAX-RPC Web Service

```
public interface CoffeeOrderIF extends java.rmi.Remote {
    public Coffee[] getPriceList()
        throws java.rmi.RemoteException;
    public String orderCoffee(String name, int quantity)
        throws java.rmi.RemoteException;
}
```

```
public Class CoffeeOrderImpl implements CoffeeOrderIF {
    public Coffee[] getPriceList() ...
    public String orderCoffee(String name, int quantity) ...
}
```

Annotations (2)

Metadata

- mit Metadaten

```
import java.xml.rpc.*;

public Class CoffeeOrder {
   @Remote public Coffee[] getPriceList() {...}
   @Remote public String orderCoffee(String name, int
       quantity) {...}
}
```

Other Stuff

- (noch) keine RawSockets
- noch keine Isolation API, aber Class Sharing (wie unter Mac OS X)
- Java Management Extensions
- Bessere Unterstützung für Multithreading
- XML, OpenGL, LDAP, ...

Zusammenfassung

Zusammenfassung

- Generics, Erweiterte for-Schleife, Enumerationen, Autoboxing
  - Typsicherheit, “schöner” Code
  - relativ schnelles Einlernen (Anwendung)
- Binärkompatibilität ist gewährleistet
- Zur Zeit keine IDE-Unterstützung

Quellen

- htpp://java.sun.com
- Gilad Bracha: *Generics Tutorial*
- Latest JSR14 draft specification
- Gilad Bracha: *Adding Generics to the Java Programming Language,* Slides from JavaOne 2003 presentation
- iX 03/2004: *Der Tiger ist los*