Generics

- Motivation
- Definition
- Typebounds
- Wildcards

Motivation Generics

Ein Java-Programm: mit Vektoren

```
class Konto
{private int nr, stand;   // Kontostand in €

public Konto(int nr, int stand)
   {this.nr = nr;
    this.stand = stand;}

   public void drucken()
   {System.out.println(nr + ",
    Kontostand: " + stand);}
}

class Kunde
{private String name, vorname, ort;

public Kunde(String n, String v, String o)
   {name = n; vorname = v; ort = o;}

   public void drucken()
   {System.out.println(vorname + " " + name +
                       " wohnt in " + ort);
   }
}
```

Motivation Generics

Ein Java-Programm: mit Vektoren

```
import java.util.Vector;

public class Motivation
{
 public static void main(String[] argv)
 { Vector v = new Vector();
  // Kunden-Element an v anfuegen
  v.addElement(new Kunde("Maier", "Franz",
  "Ulm"));
  // Konto-Element einfügen
  v.addElement(new Konto(4711, 1234));
  // Integer-Element einfügen
  v.addElement(new Integer(5));
  System.out.println("Alles ausgeben\n-------");
      for (int i = 0; i < v.size(); i++)
      {
         Object o = v.elementAt(i);
         System.out.print(i + ": ");
         if (o instanceof Kunde)
            ((Kunde)o).drucken();
         else if (o instanceof Konto)
            ((Konto)o).drucken();
         else
    System.out.println(((Integer)o).intValue());
      }
   }
}
```

Motivation Generics

Nachteile dieser Vorgehensweise:

- Die Objekte, welche in einen Vektor aufgenommen werden, können unterschiedlichen Klassen angehören.
- Beim Zugriff auf Elemente des Vektors muss immer eine explizite Typkonvertierung von *Object* auf die eigentliche Klasse stattfinden.

Lösungsmöglichkeit:

Benutzung von generischen Datentypen. Bei der Definition wird der Datentyp als Parameter angegeben. Bei der Verwendung wird der Datentyp festgelegt. Der Compiler kann den Typ statisch überprüfen.

Definition:

Generics sind Schablonen für Klassen, Interfaces oder Methoden , in denn Komponententypen bei der Definition noch unbekannt und durch Typvariablen vertreten sind.

Ein Beispiel zur Verwendung von Generics

Ein Vektor, der nur Integer Objekte aufnimmt

```
Vector<Integer> v = new Vector<Integer>();
```

Folgender Befehl ist korrekt:

```
v.addElement(new Integer(5));
```

Folgender Befehl führt zu einer Fehlermeldung:

```
v.addElement(new Kunde("Maier", "Franz",
  "Ulm"));

Motivation2.java:10:
  addElement(java.lang.Integer) in
  java.util.Vector<java.lang.Integer>
  cannot be applied to (Kunde)
```

Ein Beispiel zur Verwendung von Generics

Ein Vektor, der nur Integer Objekte aufnimmt

```
import java.util.Vector;

public class Motivation2
{ public static void main(String[] argv)
   {
Vector<Integer> v = new Vector<Integer>();
// Integer-Element einfügen
v.addElement(new Integer(5));

System.out.println("Alles ausgeben\n----");

      for (int i = 0; i < v.size(); i++)
      {
         Integer elem = v.elementAt(i);
         System.out.print(i + ": ");

  System.out.println(elem.intValue());
      }
   }
}
```

Generics: Begriffe

Eine generische Klasse ist eine Klassendefinition, in der unbekannte Typen durch Typvariablen vertreten sind. Für generische Interfaces gilt Entsprechendes

Ein generischer Typ ist eine Typangabe, in der eine generische Klasse mit einem konkreten Typargument versehen wird.

Anmerkung

Während bei nicht-generischen Klassen ein 1:1 Beziehung zwischen Klassen und Interfaces zu Typen besteht, ist es bei generischen Klassen und Interfaces eine 1:n Beziehung zu Typen

Definition einer generischen Klasse

```
class Punkt <T>
{
   private T x;
   private T y;
}
```

Bei der Klassendefinition wird dem Klassennamen eine Typvariable in spitzen Klammern nachgestellt.

Per Konvention werden für die Typen Großbuchstaben in der Nähe des „T“ gewählt..

Ein ausführlicheres Beispiel

```
class Punkt <T>
{
  private T x;
  private T y;

  public Punkt(T x, T y)
  {this.x=x;
  this.y=y;}

  public T getX()
  {return x;}

  public Punkt<T> getPunkt()
  {return this;}
}


public class Generic1
{
   public static void main(String[] argv)
   {
   Integer x=new Integer(5);
   Integer y=new Integer(7);
   Punkt<Integer> p =
      new Punkt<Integer>(x, y);
    System.out.println(p);
    System.out.println(p.getX());
   }
}
```

Weitere generische Klassen

```
class Obst <T, U>
{
   private T x;
   private U y;
}
```

Es können auch mehrere Typvariablen genutzt werden. In diesem Beispiel sind die Typvariablen T und U unabhängig voneinander und repräsentieren beliebige, unbekannte Typen.

```
class Obst <T, U>
{  public T name;
   public U anzahl;
   public Obst (T name, U anzahl){
      this.name=name;
      this.anzahl=anzahl;}
}
public class Generic2
{public static void main(String[] argv)
   { Obst<String, Integer> p;
     p = new Obst<String, Integer>
             ("Birne", new Integer(5));
    System.out.println("Es sind " + p.anzahl +
" Stueck " + p.name + " vorhanden");
   }
}
```

Typebounds mit Typenvariablen

Covarianz

Generische Klasse akzeptiert (bisher) beliebige Typargumente

Oft sinnvoll: Einschränkung der Typargumente

Lösung: "Typebound" bei der Definition einer generischen Klasse:

```
class Someclass<T extends U>
{...}
```

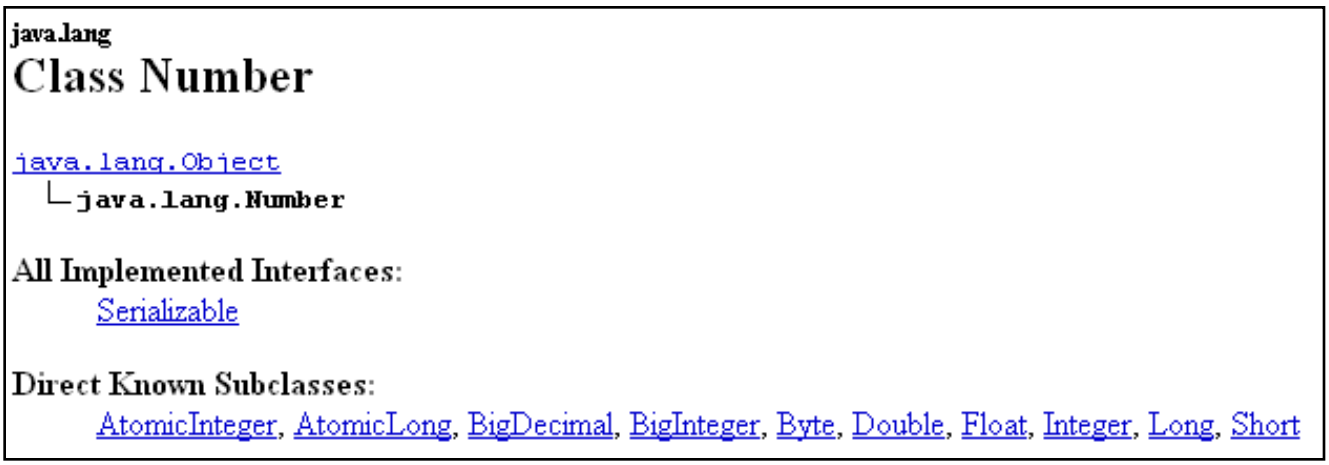
Folge: späterer Typargumente müssen zum Typebound U kompatibel sein

"extends" hier für Klassen und Interfaces

Beispiel: Typebounds mit Typenvariablen

Beispiel: Knoten mit ausschließlich numerischen Informationen:

```
class Node<T extends Number>
{...}
```



java.lang

Class Number

java.lang.Object
  └ **java.lang.Number**

All Implemented Interfaces:
Serializable

Direct Known Subclasses:
AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short

Number ist Basisklasse von Integer, Double etc., nicht aber von beispielsweise String, Object

Compiler lehnt unpassende Typargumente ab:

```
Node<Integer> ni;  // ok
Node<Double> nd;   // ok
Node<Object> no;   // Fehler!
Node<String> ns;   // Fehler!
```

Beispiel: Typebounds mit Typenvariablen

```
class Figur
{  protected String name;}

class Rechteck extends Figur
{  protected int laenge;
   protected int breite;}

class Quadrat extends Rechteck
{  public Quadrat(int dim, String name)
   {  laenge = dim;
      breite = dim;
      this.name=name; }
}

class Element<T extends Figur>
{  T elem;
   Element(T elem)
   {this.elem = elem;}
}

public class Generic3
{  public static void main(String[] argv)
   {
    Element<Quadrat> el;
    el=new Element<Quadrat>
       (new Quadrat(3, "box"));
    System.out.println("Ein Objekt der: " +
el.elem.getClass() + " mit der Dimension " +
el.elem.laenge);  }
}
```

Wildcards mit Typenvariablen

Jeder generische Typ derselben generischen Klasse ist kompatibel zum Wildcardtyp

Allgemein (→ heißt "ist kompatibel zu"): C<A> → C<?>

für jedes A

Beispiel:

```
Node<?> nx;
nx = new Node<String>("foo");
nx = new Node<Integer>(1);
nx = new Node<Double>(3.14);
```

Typebounds mit Typenvariablen

Contravarianz

Neue Form der Einschränkung auf Basisklassen

Syntax:

```
C<? super B>
```

Nur generische Typen kompatibel, zu deren Typargument B kompatibel ist (Vorsicht: Verwechslungsgefahr)

Beispiel:

```
 Node<? super Number> n;
n = new Node<Integer>(23);              //
Fehler
n = new Node<Object>(new Object());      // ok
```

Ein Beispiel aus dem Collections Framework

```
import java.util.*;

public class Iterator2
{
  public static void main(String args[])
  {
    List<String> list =
          new ArrayList<String>();
    list.add("abc");
    list.add("def");
    list.add("ghi");

    ListIterator<String> iter =
      list.listIterator(list.size());

    while (iter.hasPrevious())
      System.out.println(iter.previous());
  }
}
```