

Praktische Informatik 1

Generisches Programmieren

Thomas Röfer

- Generische Klassen und Interfaces
- Generische Typen
- Typebounds
- Wildcard-Typen
- Übersetzung generischer Klassen
- Grenzen generischer Typen
- Polymorphe Methoden

Bivarianz generischer Typen

- *Wildcard (Joker)* bei Typangaben
- Generische Typen können unbestimmte Typargumente nennen
 - *Node<?> nx;*
 - *nx = new Node<String>("foo");*
 - *nx = new Node<Integer>(1);*
 - *nx = new Node<Double>(3.14);*
- Kein Zugriff auf Methoden oder Attribute, die Typargument verwenden
 - *double d = nx.getInfo(); // Fehler*
 - *nx.setInfo(2.72); // Fehler*

```
static int count(Node<?> n) {  
    if (n == null) {  
        return 0;  
    } else {  
        return 1  
            + count(n.getLeft())  
            + count(n.getRight());  
    }  
}
```

Covarianz generischer Typen

- Zu einem Wildcard-Typ mit Typargument ? sind alle generischen Typen der betreffenden generischen Klasse kompatibel
- Manchmal möchte man dies durch einen sog. *Upper-Typebound* einschränken (der *allgemeinste* erlaubte Typparameter)
 - *Node<? extends Number> nb;*
 - *nb = new Node<Integer>(23);*
 - *nb = new Node<Object>(new Object()); // Fehler!*

```
static double sum(Node<? extends Number> n) {  
    if (n == null) {  
        return 0;  
    } else {  
        return n.getInfo().doubleValue()  
            + sum(n.getLeft()) + sum(n.getRight());  
    }  
}
```

Covarianz generischer Typen

- Mit *Upper-Typebounds* wird *Covarianz* für generische Typen eröffnet
- Allgemein gilt
 - $C\langle A \rangle$ ist kompatibel zu $C\langle ? \text{ extends } B \rangle$, wenn A ist kompatibel zu B
- Problem
 - Arrays kennen *Covarianz*, aber statische Typprüfung versagt
 - `Number[] a = new Integer[23];`
 - `a[0] = new Double(3.14);` // `ArrayStoreException`
 - Lösung bei generischen Typen: nur Lesen erlaubt
 - `Node<? extends Number> nb = new Node<Integer>(23);`
 - `Number n = nb.getInfo();`
 - `nb.setInfo(3.14);` // Fehler!

Contravarianz generischer Typen

- Zu einem Wildcard-Typ mit Typargument ? sind alle generischen Typen der betreffenden generischen Klasse kompatibel
- Manchmal möchte man dies durch einen sog. *Lower-Typebound* einschränken (der *speziellste* erlaubte Typparameter)
 - `Node<? super Number> nb;`
 - `nb = new Node<Object>(new Object());`
 - `nb = new Node<Integer>(0); // Fehler!`

```
static int subNodes(Node<? super Integer> n) {  
    if (n == null) {  
        return 0;  
    } else {  
        int s = subNodes(n.getLeft()) + subNodes(n.getRight());  
        n.setInfo(s);  
        return s + 1;  
    }  
}
```

Contravarianz generischer Typen

- Mit *Lower-Typebounds* wird *Contravarianz* für generische Typen eröffnet
- Allgemein gilt
 - $C\langle A \rangle$ ist kompatibel zu $C\langle ? \text{ super } B \rangle$, wenn B ist kompatibel zu A
- Nur Schreibzugriff
 - `Node<? super Number> nb = new Node<Object>(new Object());`
 - `nb.setInfo(1);`
 - `Number n = nb.getInfo(); // Fehler!`

Zusammenfassung Varianzen

- Invarianz
 - Verschiedene generische Typen sind zueinander inkompatibel, unabhängig von der Kompatibilität ihrer Typargumente
- Bivarianz
 - *Wildcard*-Typen ohne Einschränkung ($C<?>$) sind immer zueinander kompatibel
- Covarianz
 - Zu *Wildcard*-Typen mit *Upper-Typebound* ($C<? \text{ extends } B>$) sind alle generischen Typen kompatibel, deren Typargument zu B kompatibel ist
- Contravarianz
 - Zu *Wildcard*-Typen mit *Lower-Typebound* ($C<? \text{ super } B>$) sind alle generischen Typen kompatibel, zu deren Typargument B kompatibel ist

Zusammenfassung Varianzen

	Typ	Lesen	Schreib.	Kompatible Typargumente
Invarianz	$C<T>$	ja	ja	T
Bivarianz	$C<?>$	nein	nein	Alle
Covarianz	$C<? \text{ extends } B>$	ja	nein	B und abgeleitete Typen
Contravarianz	$C<? \text{ super } B>$	nein	ja	B und Basistypen

Übersetzung generischer Klassen

- Generische Datentypen werden in Java ausschließlich vom Compiler verarbeitet
- Das Laufzeitsystem weiß nichts von generischen Datentypen
- Mit *Type-Erasure* wird „generischer Code“ mit Typvariablen und Typargumenten auf normalen, nicht-generischen Java-Quelltext reduziert
- Der nicht-generische Java-Quelltext wird weiterverarbeitet wie bisher
- Aus jeder generischen Klasse wird eine nicht-generische Klasse generiert und in eine *.class*-Datei übersetzt
 - In C++ wird dagegen jede Instanziierung einer Klasse mit Typargumenten getrennt übersetzt
 - Dadurch langsames Übersetzen und größere Kompilate, aber bessere Optimierungsmöglichkeiten und weniger Einschränkungen

Type-Erasure für generische Klassen

- Typ-Variablen in spitzen Klammern werden gelöscht
- Alle Vorkommen von Typvariablen mit einem oder mehreren *Typebounds* werden durch den einzigen bzw. ersten *Typebound* ersetzt
- Alle Vorkommen von Typvariablen ohne *Typebounds* werden durch *Object* ersetzt

Type-Erasure – Beispiel

Generische Klasse

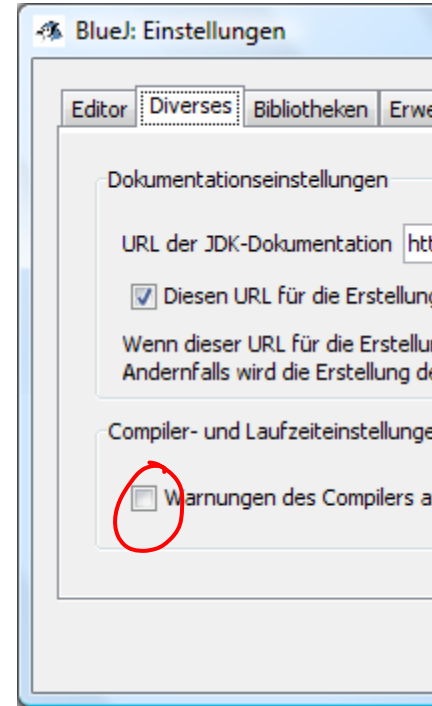
```
class Node<T> {  
    private T info;  
    private Node<T> left, right;  
  
    Node(T i) {info = i;}  
    Node(T i, Node<T> l, Node<T> r) {  
        info = i; left = l; right = r;  
    }  
    T getInfo() {return info;}  
    Node<T> getLeft() {return left;}  
    Node<T> getRight() {return right;}  
    void setInfo(T i) {info = i;}  
}
```

Nach Type-Erasure (Rawtype)

```
class Node {  
    private Object info;  
    private Node left, right;  
  
    Node(Object i) {info = i;}  
    Node(Object i, Node l, Node r) {  
        info = i; left = l; right = r;  
    }  
    Object getInfo() {return info;}  
    Node getLeft() {return left;}  
    Node getRight() {return right;}  
    void setInfo(Object i) {info = i;}  
}
```


Type-Erasure für generische Typen

- Die Typ-Korrektheit wird statisch geprüft (d.h. zum Übersetzungszeitpunkt)
 - Typargumente müssen allen *Typebounds* genügen
 - Generische Typen müssen auch untereinander korrekt verwendet werden, insbesondere bei *Wildcard*-Typen
- Typargumente, einschließlich Wildcards, in spitzen Klammern werden gelöscht
- *Typecasts* werden eingeschoben, wo der Wert eines Typarguments benutzt wird
- *Rawtypes* lassen sich auch direkt benutzen, aber die Typsicherheit geht verloren (opt. Compiler-Warnung)



Generische Typen

```
Node<String> n = new Node<String>("foo");  
String s = n.getInfo();
```

Nach Type-Erasure (Rawtype)

```
Node n = new Node("foo");  
String s = (String) n.getInfo();
```


Typebounds – Beispiel

```
class SortedSet<T extends Comparable<T>> extends Set<T> {  
    public void insert(T element) {  
        if (!contains(element)) {  
            if (first == null || element.compareTo(first.element) <= 0) {  
                first = new Element<T>(element, first);  
            } else {  
                Element<T> e = first;  
                while (e.next != null && element.compareTo(e.next.element) > 0) {  
                    e = e.next;  
                }  
                e.next = new Element<T>(element, e.next);  
            }  
        }  
    }  
}
```

Grenzen generischer Typen

- Primitive Typargumente
 - *Node<int> ni = new Node<int>(23); // Fehler!*
 - Aber:
Node<Integer> ni = new Node<Integer>(23); // Autoboxing
- Statische Elemente
 - *class Broken<T> {
 static T data; // Fehler!
}*
 - Grund: Alle Klassen *Broken<T>* teilen sich das Klassenattribut *data*. Welchen Typ soll es haben?
- Dynamische Typprüfung
 - *class Node<T> {
 boolean isCompatible(Object o) { return o instanceof T; }
} // Fehler!*
 - Type-Erasure: *o instanceof T* \rightarrow *o instanceof Object*

Grenzen generischer Typen

- Konstruktoraufrufe

- ```
class Node<T> {
 T info;
 Node() { info = new T(); } // Fehler
}
```
- Woher soll Java wissen, dass *T* einen Standard-Konstruktor hat?
- Beispiel:  

```
Node<Integer> ni = new Node<Integer>();
```
- Ausweg:  

```
class Node<T> {
 T info;
 Node(T i) { info = i; }
}
```

```
Node<Integer> ni = new Node<Integer>(23);
```

# Grenzen generischer Typen

- Typecasts
  - `class Node<T> {  
    T info;  
    void setInfo(Object o) { info = (T) o; } // sinnlos  
}`
  - Type-Erasure:  $(T) o \rightarrow (Object) o$
  - Compiler erzeugt: „warning: unchecked cast of type T“
- Generische Basistypen
  - `import java.util.Date;  
class Timestamped<T> extends T { // Fehler  
    Date timestamp = new Date();  
}`
  - Generische Klasse muss Konstruktor der Basisklasse aufrufen können. Dieser kann hier aber nicht zur Kompilierzeit ermittelt werden!

# Grenzen generischer Typen

- Exceptions
  - *class UniversalException<T> extends Exception {  
    T reason;  
} // Fehler*
  - Generische Typen können nicht für *Exceptions* verwendet werden, da das Fangen mit *catch* auf dem Ermitteln des Typs des geworfenen Objekts basiert. Dieser geht aber bei der *Type-Erasure* verloren:
    - *throw new UniversalException<String>("Hallo")  
    → throw new UniversalException("Hallo")*
    - *catch (UniversalException<String> e) → catch  
    (UniversalException e)*
  - Compiler erzeugt: „a generic class may not extend `java.lang.Throwable`“

# Grenzen generischer Typen

- Arrays von Typvariablen
  - `class Container<T> {  
    T[] a = new T[100]; // Fehler  
}`
  - Ausweg:  
`class Container<T> {  
    T[] a = (T[]) new Object[100]; // optionale Warnung  
}`
  - Wenn aktiviert, Warnung „uses unchecked or unsafe operations“
  - Man kann trotzdem typsichere Klassen erstellen:

```
class Container<T> {
 private T[] a = (T[]) new Object[100];

 void set(int i, T t) {a[i] = t;}
 T get(int i) {return a[i];}
}
```

## Blick über den Tellerrand – C++

- C++ übersetzt generische Typen getrennt, d.h. die Typparameter werden in die generischen Klassen eingesetzt und für jede Einsetzung wird ein eigenes Kompilat erzeugt
- *Keine* der genannten Einschränkungen gilt für C++
- Generische Klassen können auch andere als Typparameter haben
- Bessere Optimierung
- Keine expliziten *Typebounds*
- Nur Invarianz generischer Typen

```
template<typename T, int size>
 class Stack {
private:
 T buffer[size];
 int length;

public:
 Stack() : length(0) {}

 void push(T t) {
 buffer[length++] = t;
 }

 T pop() {
 return buffer[--length];
 }
};

Stack<int, 10> s;
s.push(5);
```



# Blick über den Tellerrand – C++ Auswertung zur Übersetzungszeit

```
template<int n> class Factorial {
public:
 static const int f = n * Factorial<n - 1>::f;
};

template<> class Factorial<0> {
public:
 static const int f = 1;
};

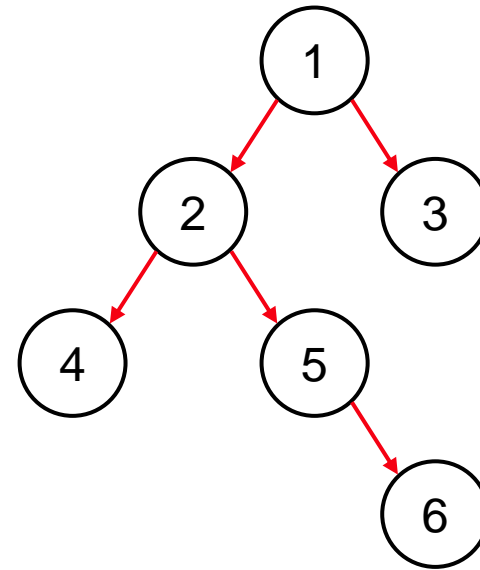
int f5 = Factorial<5>::f;
```



# Motivation

```
class Node {
 private Object info;
 private Node left, right;

 Node(Object i) {info = i;}
 Node(Object i, Node l, Node r) {
 info = i; left = l; right = r;
 }
 Object getInfo() {return info;}
 Node getLeft() {return left;}
 Node getRight() {return right;}
 void setInfo(Object i) {info = i;}
}
```



```
Node n1 = new Node("foo");
Node n2 = new Node(3.14);
Node n0 = new Node(23, n1, n2);

String s = (String) n1.getInfo();
```

- Problem: Zugriff nicht Typ-sicher!

# Generische Klassen

- Generische Klassen haben *Typvariablen*, die im Klassenrumpf verwendet werden können
  - *class Node<T> { T info; ... }*
- Typvariablen können (fast) wie ein normaler Typ benutzt werden
- Mehrere Typvariablen sind

```
class Pair<T, U> {
 private T first;
 private U second;
 Pair(T t, U u) {first = t; second = u;}
 T getFirst() {return first;}
 U getSecond() {return second;}
}
```

```
class Node<T> {
 private T info;
 private Node<T> left, right;

 Node(T i) {info = i;}
 Node(T i, Node<T> l, Node<T> r) {
 info = i; left = l; right = r;
 }
 T getInfo() {return info;}
 Node<T> getLeft() {return left;}
 Node<T> getRight() {return right;}
 void setInfo(T i) {info = i;}
}
```

# Generische (polymorphe) Methoden

- Polymorphe Methoden sind unabhängig von generischen Typen
- Sie können auch in nicht-generischen Klassen definiert werden
- Klassen- und Objektmethoden sowie Konstruktoren können polymorph sein
- Beim Aufruf mit expliziter Angabe des Typparameters muss *immer* ein Punkt vor dem Typparameter stehen
  - *Klassenname.<Typ>methode(...)*
  - *referenz.<Typ>methode(...)*
- Ansonsten wird der Fehler „illegal

```
class C {
 <T> T vote(T x, T y, T z) {
 if (x.equals(y)) {
 return x;
 } else if (y.equals(z)) {
 return y;
 } else if (z.equals(x)) {
 return z;
 } else {
 return null;
 }
 }
}
```

# Generische (polymorphe) Methoden

- Aufruf
  - *String s =*  
*this.<String>vote("foo","foo","bar");*
  - *int i = this.<Integer>vote(1,2,2);*
  - *s = this.<String>vote(1,2,2); // Fehler!*
- Typ-Inferenz
  - *String s = vote("foo", "foo","bar");*
  - Typparameter werden automatisch mit dem „untersten“ gemeinsamen Typ belegt
  - *Double d = vote(1, 3.14, 1); // Fehler!*
  - entspricht  
*Double d = vote(new Integer(1), new Double(3.14), new Integer(1));*
  - *Number n = vote(1, 3.14, 1); // ok*

```
class C {
 <T> T vote(T x, T y, T z) {
 if (x.equals(y)) {
 return x;
 } else if (y.equals(z)) {
 return y;
 } else if (z.equals(x)) {
 return z;
 } else {
 return null;
 }
 }
}
```

# Funktionen höherer Ordnung

- Funktionen höherer Ordnung bekommen selbst Funktionen als Parameter
- In Java sind Funktionen als Parameter nicht möglich, wohl aber Objekte von Klassen, die bestimmte Interfaces implementieren
- Beispiel: Faltung
  - Eine Faltung „faltet“ eine Liste (oder ein Array) zu einem einzigen Wert zusammen
  - Dazu wird eine binäre Funktion der Reihe nach auf alle Elemente angewendet
  - In Haskell:  
$$\text{foldl } (+) \ 0 \ [1,2,3,4,5] = (((((0 + 1) + 2) + 3) + 4) + 5)$$



# Funktionen höherer Ordnung

```
interface FoldlFn<A, B> {
 A fn(A a, B b);
}
```

```
static <A, B> A foldl(FoldlFn<A, B> f, A a, B[] b) {
 for (B b0 : b) {
 a = f.fn(a, b0);
 }
 return a;
}
```

```
static String concat(Object... objects) {
 return foldl(new FoldlFn<String, Object>() {
 public String fn(String s, Object o) {
 return s + o.toString();
 }
 }, "", objects);
}
```

# Funktionen höherer Ordnung

- Beispiel: Abbildung
  - Die Methode *map* wendet eine Funktion auf alle Elemente einer Reihung an
  - In Haskell:  
 $\text{map } (* 2) [1,2,3,4,5] = [2,4,6,8,10]$
  - Eigentlich sollten Eingabetyp und Ergebnistyp unterschiedlich sein können, aber Java kann keine Reihung für einen generischen Ergebnistyp anlegen

```
interface MapFn<A> {
 A fn(A a);
}
```

```
static void square(Integer[] a) {
 map(new MapFn<Integer>() {
 public Integer fn(Integer i) {
 return i * i;
 }
 }, a);
}
```

```
static <A> void map(MapFn<A> f, A[] a) {
 for (int i = 0; i < a.length; ++i) {
 a[i] = f.fn(a[i]);
 }
}
```