

13. Generics

Prof. Dr. Harald Gall
Institut für Informatik
Universität Zürich
<http://seal.ifi.uzh.ch>



Lernziele

- Wie kann man mehr Flexibilität hinsichtlich der Typen erreichen?
- Wie kann man Typen als Variablen verwenden?
- Was sind Generics und wie setzt man diese ein?
- Was ist der Unterschied zu Vererbung und Interfaces?
- Was muss bei Generics beachtet werden?

2

Motivation

- Container und Elemente
- Beispiel: Knotenklasse
- Probleme
- Generische Typen
- Verwechslungsgefahr
- Begriffe

3

Container und Elemente

- **Container** = Klassen zum Speichern anderer Objekte als Elemente
- Beispiele:
 - Strings (eingeschränkt), Arrays, Collections
- Arbeitsweise eines Containers unabhängig vom konkreten Elementtyp
- Problem: Definition neuer Containertypen

4

Beispiel: Knotenklasse

- Idee
- Knoteninformation
- Knotenklasse
- Konstruktoren
- Aufbau eines Baums
- Getter und Setter

5

Knotenklasse: Problem

- Elementtyp Object erlaubt beliebige Knoteninformation (via Autoboxing auch primitive Typen)
- Aber: selber Knotentyp innerhalb eines Baums oft sinnvoll oder notwendig
- Keine Unterstützung durch Knotenklasse. Beispiel:

```
Node n1 = new Node("foo");
Node n2 = new Node(3,141592);
Node n0 = new Node(23, n1, n2);
```
- Auslesen von Knoteninformationen erfordert Typecast, weil getInfo als Ergebnis nur Object liefert:

```
String s = (String)n.getInfo();
```
- Typecasts immer problematisch

6

Generische Typen

- Lösung: Generische Klassen
 - Definition: Elementtyp bleibt unbestimmt, vertreten durch eine Typvariable
 - Verwendung: konkretes Typargument ersetzt die Typvariable der Definition \Rightarrow neuer Typ
- Beispiel Knotenklasse:
 - Definition: Typvariable für Knoteninformation (statt Object)
 - Verwendung: Angabe eines konkreten (Referenz-)Typs für die Typvariable, fixiert Typ der Knoteninformation

7

Verwechslungsgefahr

- Begrifflich ähnlich wie Argumente und Parameter bei Methodenaufrufen
- Aber:
 - Methodenaufrufe werden zur Laufzeit abgewickelt (von der JVM),
 - Ersetzen von Typargumenten beim Übersetzen (vom Compiler)
- Völlig verschiedene Mechanismen!

8

Begriffe

- Generische Klasse = Klasse mit Typvariablen
- Generischer Typ = Typangabe aus generischer Klasse + konkretem Typargument
- 1 generische Klasse \Rightarrow viele generische Typen
- Interfaces in diesem Zusammenhang unter dem Begriff "Klasse" subsumiert

9

Definition und Anwendung

- Typvariable
- Typvariable im Rumpf
- Generische Typen
- Typprüfung
- Generische Interfaces
- Implementierung generischer Interfaces
- Mehrere Typvariablen
- Mehrere Typargumente

10

Typvariable

- Definition einer generischen Klasse \approx normale, nicht-generische Klasse
- Aber: Typvariable in spitzen Klammern nach dem Klassennamen:

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

- Typvariable = beliebiger Java-Identifizier
- Per Konvention einzelner Großbuchstabe in alphabetischer Nähe zum "T"

11

Typvariable im Rumpf

- Im Klassenrumpf: Typvariable (weitgehend) wie konkreter Typ

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

- Konstruktoren akzeptieren als Parameter nur gleiche Typvariablen (Knoten mit gleichen Typvariablen):

```
Node(T i) {
    this(i, null, null);
}
Node(T i, Node<T> l, Node<T> r) {
    info = i;
    left = l;
    right = r;
}
```

12

Typvariable im Rumpf (2)

- Ergebnis der Getter und Parameter der Setter beziehen sich auf die Typvariable:

```
...
T getInfo()           {...}
Node<T> getLeft()    {...}
Node<T> getRight()   {...}
void setInfo(T i)     {...}
void setLeft(Node<T> l) {...}
void setRight(Node<T> r) {...}
```

13

Generische Typen

- "Anwendung" einer generischen Klasse: Konkretes Typargument für die Typvariable
- Generische Klasse + Typargument = generischer Typ
- Jeder generische Typ ist eigenständig, gleichrangig zu anderen Javatypen. Beispiel: String-Knoten

```
Node<String> ns;
ns = new Node<String>("foo");
```

- In einem Programm *verschiedene generische Typen derselben generischen Klasse* erlaubt:

```
...
Node<Integer> ni;
ni = new Node<Integer>(1);
Node<Rational> nr;
nr = new Node<Rational>(new Rational(2, 3));
```

14

Typprüfung

- Node<String> und Node<Integer> sind inkompatibel:

```
Node<Integer> ni;
ni = new Node<String>("foo"); // Typfehler
```

- Nur Bäume mit Knoten desselben Typs konstruierbar:

```
Node<Integer> ni = new Node<Integer>(1);
Node<String> ns = new Node<String>("foo", ni, null);
//Typfehler
```

- Typecasts nicht mehr nötig:

```
String s = ns.getInfo(); // ohne Typecast
Node<String> l = ns.getLeft();
```

- Compiler sichert korrekte Verwendung ab, keine Tests zur Laufzeit

15

Generische Interfaces

- Generische Interfaces entsprechend zu generischen Klassen
- Beispiel: Generisches Interface `Taggable` mit zwei Methoden zum
 - Anbringen einer Markierung (engl. "tag") und
 - Ablesen der Markierung
- Definition:

```
interface Taggable<T> {  
    void setTag(T tag);  
    T getTag();  
}
```

16

Implementierung generischer Interfaces

- Nicht-generische Klasse kann generisches Interface durch Angabe eines Typarguments implementieren
- Beispiel:

```
class Someclass implements Taggable<String> {  
    private String tag;  
  
    public void setTag(String t) {...}  
    public String getTag() {...}  
  
    ...  
}
```

17

Implementierung generischer Interfaces (2)

- Generische Klasse kann generisches Interface durch "Übergabe" der Typvariablen implementieren
- Beispiel:

```
class Node<T> implements Taggable<T> {  
    private T tag;  
  
    public void setTag(T t)    {...}  
    public T getTag()         {...}
```
- Knoteninformation und Tag haben hier denselben Typ

18

Mehrere Typvariablen

- Beliebige viele Typparameter einer generischen Klasse
- Beispiel: generische Klasse `Pair` verknüpft zwei Objekte

```
class Pair<T, U> {  
    private final T first;  
    private final U second;  
}
```

- Typvariablen `T, U` = unbekannte, unabhängige Typen

```
class Pair<T, U> {  
    Pair(T fst, U snd) {  
        first = fst;  
        second = snd;  
    }  
    T getFirst() {  
        return first;  
    }  
    U getSecond() {  
        return second;  
    }  
}
```

19

Mehrere Typargumente

- Generische Klasse mit mehreren Typvariablen \Rightarrow mehrere Typargumente, eines für jede Typvariable

```
Pair<String, Integer> psi;
```

- Compiler verhindert Typfehler

```
Pair<String, Integer> p;  
p = new Pair<String, Integer>("foo", 1);  
int i = p.getFirst(); // Fehler!  
int j = p.getSecond(); // ok
```

- Generische Typen = vollwertige Typen \rightarrow selbst als Typargumente zulässig:

```
Pair<Integer, Pair<String, Integer>> pp;  
pp = new Pair<Integer, Pair<String, Integer>>(23, p);  
int k = pp.getSecond().getSecond(); // ok
```

20

Typebounds

- Idee
- Beispiel
- Mehrfache Typebounds
- Beispiel
- Typebounds mit Typvariablen
- Problem

21

Typebounds: Idee

- 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: spätere Typargumente müssen zum Typebound U kompatibel sein
- "extends" hier für Klassen und Interfaces

22

Typebounds: Beispiel

- Beispiel: Knoten mit ausschließlich numerischen Informationen:

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

- 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!
```

23

Mehrfache Typebounds

- Liste von Typebounds ⇒ mehrfache Einschränkungen
- Typvariablen müssen zu jedem Typebound kompatibel sein

- Syntax:

```
class Someclass<T extends U1 & U2 & U3...>
{...}
```

- Einschränkung:
 - **Erster** Typebound U1 darf ein **beliebiger** Typ sein
 - **Weitere** Typebounds U2, U3, ... **müssen Interfaces** sein
- Reihenfolge der weiteren Typebounds nach dem ersten irrelevant

24

Beispiel

- Bäume sollen auf Streams geschrieben werden können ⇒ müssen das Interface `Serializable` implementieren
- Bäume sollen kopierbar sein ⇒ müssen das Interface `Cloneable` implementieren
- Gilt ebenso für Knoteninformation
- Einfordern mit Typebounds:

```
class Node<T extends Serializable & Cloneable>
{...}
```

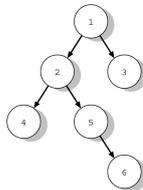
- Knoten damit selbst serialisierbar und kopierbar:

```
class Node<T extends Serializable & Cloneable>
    implements Serializable, Cloneable
{...}
```

25

Typebounds mit Typvariablen

- Beispiel: geordnete Bäume
- Bezüglich jedes Knotens: alle Informationen im linken Teilbaum kleiner und alle Informationen im rechten Teilbaum größer oder gleich
- Binärer Baum:



26

Binärer Baum

- Voraussetzung: Knoteninformationen kann verglichen werden
- Technisch: Knotentyp implementiert das Interface `Comparable`
- `Comparable` ist ein generisches Interface
- Definition einer generischen Knotenklasse für geordnete Bäume:

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

- Compiler stellt Konsistenz sicher:

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

27

Binärer Baum (2)

- Lösung noch mangelhaft: Von Comparable abgeleitete Klasse wird nicht als Typargument akzeptiert
- Beispiel:

```
class Base implements Comparable<Base>
{...}

class Derived extends Base
{...}
```

```
...
Node<Base> nb;    // ok
Node<Derived> nd; // Fehler
```

- Problem: Typebound verlangt, dass das Typargument selbst direkt Comparable implementiert \Rightarrow zu streng
- Lösung: contravariante Wildcardtypen

28

Wildcardtypen

- Motivation
- Invarianz
- Bivarianz
- Covarianz
- Contravarianz
- Gegenüberstellung Varianzen
- Varianzen und Objekte
- Wozu Wildcardtypen?

29

Motivation

- Kompatibilität eines Typs T zu einem Typ U (kurz $T \rightarrow U$) heisst:
 - Werte vom Typ T können Variablen vom Typ U zugewiesen werden.
T t = ...;
U u = t;
- Kompatibilität zwischen Typen bisher:
 - Implizite Typkonversion: für bestimmte primitive Typen, zB int \rightarrow double
 - Implementierung: Klasse zu Interface, einschließlich Arrays
 - Vererbung: Abgeleitete Klasse zu Basisklasse, zB String \rightarrow Object, einschließlich Arrays
 - Autoboxing: Primitiver Typ zu Wrapperklasse, zB int \rightarrow Integer
 - Auto-Unboxing: Wrapperklasse zu primitivem Typ, zB Integer \rightarrow int
- Generische Typen der gleichen generischen Klasse bisher inkompatibel
- Mit Wildcardtypen Einführung von Kompatibilitäten \Rightarrow mehr Flexibilität

30

Invarianz

- Generische Typen der gleichen generischen Klasse bisher inkompatibel
- Kompatibilität zwischen Typargumenten wird ignoriert
- Beispiel: `Integer` ist kompatibel zu `Number`, aber `Node<Integer>` nicht kompatibel zu `Node<Number>`

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

- Bezeichnung als "Invarianz":
Eine Variation des Typargumentes bzgl. der Vererbungshierarchie wird nicht auf die generischen Typen übertragen.

31

Bivarianz

- Generischer Typ mit Wildcardzeichen "?" als Typargument = Wildcardtyp
- Syntax: `C<?>`
- Beispiel: `Node<?> nx;`
- Wildcardtyp erlaubt zur Definition von Variablen, nicht zu Instanziierung von Objekten (siehe Interfaces und ABCs)
- Beispiel:

```
Node<?> nx;
nx = new Node<?>(); // Fehler!
```

32

Kompatibilität

- Jeder generische Typ derselben generischen Klasse ist kompatibel zum Wildcardtyp
- Allgemein (\rightarrow heißt "ist kompatibel zu"):
 - `C<A>` \rightarrow `C<?>`
 - für jedes A
- Beispiel:

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

- Bezeichnung als "Bivarianz"
Zu einem Wildcardtyp sind alle generischen Typen der gleichen generischen Klasse kompatibel.

33

Ähnlichkeit zur Ableitung

- Kompatibilität konkreter generischer Typ → Wildcardtyp ist asymmetrisch:

```
Node<?> nx;  
...  
Node<String> ns = nx; // Fehler
```

- Verhältnis formal ähnlich wie Ableitung:
 - konkreter generischer Typ → Wildcardtyp
 - abgeleitete Klasse → Basisklasse

34

Anwendung

- Wildcardtypen sinnvoll, wenn das konkrete Typargument keine Rolle spielt
- Beispiel: Methode nodesCount zum Zählen der Knoten in einem Baum
- Inhalt der Knoten irrelevant, nur Baumstruktur wichtig

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

35

Einschränkungen

- Compiler kennt konkreten Typ des Wildcardargumentes nicht
- Kein lesender Zugriff mit konkretem Typargument:

```
Node<?> nx;  
...  
Integer i = nx.getInfo(); // Fehler
```

- Kein schreibender Zugriff mit konkretem Typargument:

```
Node<?> nx;  
...  
nx.setInfo(1); // Fehler
```

- Das tatsächliche, konkrete Typargument zur Laufzeit spielt keine Rolle:

```
Node<?> nx;  
nx = new Node<Integer>(23);  
Integer i = nx.getInfo(); // Fehler  
nx.setInfo(1); // Fehler
```

36

Begrenzte Zugriffe

- Compiler kennt das Typargument zur Laufzeit nicht
- Jeder Referenztyp ist kompatibel zu Object ⇒ Lesen als Object immer möglich:

```
Node<?> nx;  
...  
Object x = nx.getInfo(); // ok
```

- null ist kompatibel zu jedem Referenztyp => Schreiben von null immer möglich:

```
Node<?> nx;  
...  
nx.setInfo(null); // ok
```

37

Covarianz

- Wildcard ? führt zu Kompatibilität mit generischen Typen aller Typargumente
- Einschränken auf bestimmte Typargumente mit Typebound
- Syntax: `C<? extends B>`
- Nur generische Typen kompatibel, deren Typargument zu B kompatibel ist
- Beispiel:

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

38

Covarianz: Kompatibilität

- Bezeichnung als "Covarianz":
Die Kompatibilität der generischen Typen folgt der Kompatibilität des Typargumentes.
- Allgemein (" \rightarrow " heißt "ist kompatibel zu"):
 - $C<A> \rightarrow C<? \text{ extends } B>$
 - wenn $A \rightarrow B$
- Typebound B = "Upper-Typebound": Aus Sicht einer Vererbungshierarchie die "Obergrenze" für konkrete Typargumente

39

Covariante Arrays

- Arrays verhalten sich covariant:

- A[] → B[]
- wenn A → B

- Problem: Lücke im statischen Typsystem!

```
Number[] a = new Integer[23];  
a[0] = new Double(3.14); // ArrayStoreException
```

- Ohne weitere Schutzmassnahmen: Fehler übertragen auf covariante Wildcardtypen:

```
Node n = new Node(23);  
n.setInfo(3.14); // ??
```

40

Gesperrte Schreibzugriffe

- Schutz gegen Lücke im statischen Typsystem: Keine schreibenden Zugriffe über covariante Wildcardtypen. Z.B

- Beispiel:

```
Node<? extends Number> n;  
...  
n.setInfo(1); // Schreiben - unzulässig
```

- Ratio: "? extends Number" ist irgendein zu Number kompatibler Typ, vielleicht Byte; Integer muss zu diesem Typ nicht kompatibel sein
- Ausnahme: null passt zu jedem Referenztyp, wird immer akzeptiert

```
Node<? extends Number> n;  
...  
n.setInfo(null); // Schreiben von null - ok
```

41

Gesperrte Schreibzugriffe (2)

- Lesender Zugriff kein Problem:

```
Node<? extends Number> n;  
...  
Number x = n.getInfo(); // Lesen - ok
```

- Ratio: Welcher konkrete Typ sich immer hinter "? extends Number" verbirgt, er ist mit Sicherheit kompatibel zu Number

42

Beispiel Covarianz

- Methode `copyFrom` der generischen Klasse `Node<T>`: Erwartet einen anderen Knoten als Parameter, kopiert dessen Information in den eigenen Knoten
- Information im eigenen Knoten hat Typ `T`, kann ein Objekt vom Typ `T` oder einem dazu kompatiblen Typ aufnehmen
- Parameter von `copyFrom`: Wildcard mit `T` als Upper-Typebound
- Definition von `copyFrom`:

```
class Node<T> {  
    ...  
    void copyFrom(Node<? extends T> other)  
    {  
        info = other.getInfo();  
    }  
}
```

43

Beispiel Covarianz (2)

- Aufruf von `copyFrom`:

```
Node<Number> n;  
Node<Integer> i;  
...  
n.copyFrom(i);    // ok  
i.copyFrom(n);    // Fehler
```

44

Contravarianz

- Einschränkung des Typarguments
- 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
```

45

Contravarianz: Definition

- **Contravarianz:**
Die Kompatibilität der generischen Typen läuft der Kompatibilität des Typargumentes entgegen.
- Allgemein (" \rightarrow " heißt "ist kompatibel zu"):
 - $C < A > \rightarrow C < ? \text{ extends } B >$
 - wenn $A \leftarrow B$
- Typebound $B = \text{Lower-Typebound}$: Aus Sicht einer Vererbungshierarchie die "Untergrenze" für konkrete Typargumente

46

Gesperrte Lesezugriffe

- Keine lesenden Zugriffe über contravariante Wildcardtypen, zB
- Ratio: "? super Number" ist Number selbst oder irgendeine Basisklasse von Number. Der Typ kann nicht einfach als Number behandelt werden.
- Ausnahme: An Object kann jeder Referenztyp zugewiesen werden:

```
Node<? super Number> n;  
...  
Number x = n.getInfo(); // Lesen - unzulässig
```

```
Node<? super Number> n;  
...  
Object x = n.getInfo(); // Lesen als Object - ok
```

47

Gesperrte Lesezugriffe (2)

- Schreibender Zugriff unproblematisch
- Ratio: Was immer sich hinter "? extends Number" verbirgt, eine Number kann mit Sicherheit zugewiesen werden

```
Node<? extends Number> n;  
...  
n.setInfo(1); // Schreiben - ok
```

48

Beispiel Contravarianz

- Methode `copyTo` der generischen Klasse `Node<T>`: Erwartet einen anderen Knoten als Parameter, kopiert die eigene Information in den anderen Knoten
- Information im eigenen Knoten hat Typ `T`, kann in jeden fremden Knoten geschrieben werden, der auch den Typ `T` oder eine Basisklasse davon hat
- Parameter von `copyTo`: Wildcard mit `T` als Lower-Typebound
- Definition von `copyTo`:

```
class Node<T> {  
    ...  
    void copyTo(Node<? super T> other)  
    {  
        other.setInfo(info);  
    }  
}
```

49

Beispiel Contravarianz (2)

- Aufruf von `copyTo`:

```
Node<Number> n;  
Node<Integer> i;  
...  
n.copyTo(i);    // Fehler  
i.copyTo(n);    // ok
```

50

Knoten geordneter Bäume

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

- Von `Comparable` abgeleitete Klasse wird nicht als Typargument akzeptiert -> Lösung: irgendeine Basisklasse des Typarguments muss `Comparable` implementieren:

```
class Node<T extends Comparable<? super T>>  
{...}
```

- Beispiel:

```
class Base implements Comparable<Base>  
{...}
```

```
class Derived extends Base  
{...}
```

```
...  
Node<Base> nb;    // ok  
Node<Derived> nd; // ok
```

51

Varianzen: Gegenüberstellung

- Vier Arten der Varianz:

	Art	Lesen	Schreiben	kompatible Typargumente
Invarianz	<code><T></code>	erlaubt	erlaubt	<code>T</code>
Bivarianz	<code><T></code>	verboten	verboten	alle
Covarianz	<code><? extends B></code>	erlaubt	verboten	<code>B</code> und abgeleitete
Contravarianz	<code><? super B></code>	verboten	erlaubt	<code>B</code> und Basistypen

52

Varianzen und Objekte

- Varianz: Einschränkungen für Variablen, nicht Objekte
- Beispiel:

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

- Zuweisen an covarianten Wildcardtyp \Rightarrow kein schreibender Zugriff über diese Variable:

```
Node<? extends Number> n = i;  
n.setInfo(24); // Fehler
```

- Knoten selbst nicht betroffen:

```
i.setInfo(24); // ok
```

53

Wozu Wildcardtypen?

- Mit Wildcardtypen flexiblerer Code
- Methoden mit Parametern von Wildcardtypen akzeptieren Argumente vieler unterschiedlicher generischer Typen
- Nachteil: Compiler muss Einschränkungen auferlegen, um die statische Typprüfung durchzuhalten
- Ohne lückenlose statische Typprüfung: Fehlerfrei übersetztes Programm kann zur Laufzeit mit Typfehler abstürzen

54

Kontrollfragen

- Wie werden in Java Generics spezifiziert?
- Was ist der Unterschied zwischen Generics und Inheritance? Was passiert zur Laufzeit?
- Was ist der Unterschied zwischen einer Generics und einem Interface?
- Was ist ein Wildcardtyp?
- Was ist eine Varianz und welche 4 Arten werden unterschieden?

55

Zusammenfassung

- *Generische Klasse* = Klasse mit Typvariablen
- *Generischer Typ* = Generische Klasse + Typargument

- 1 generische Klasse \Rightarrow viele generische Typen
- Jeder generische Typ ist eigenständig, gleichrangig zu anderen Javatypen

56
