



Generics

Programmiermethodik 2



Ausblick



- Ich möchte eine Datenstruktur entwickeln, die Objekte eines anderen Typs verwaltet
- Der Typ der internen Objekte soll variable gehalten werden.

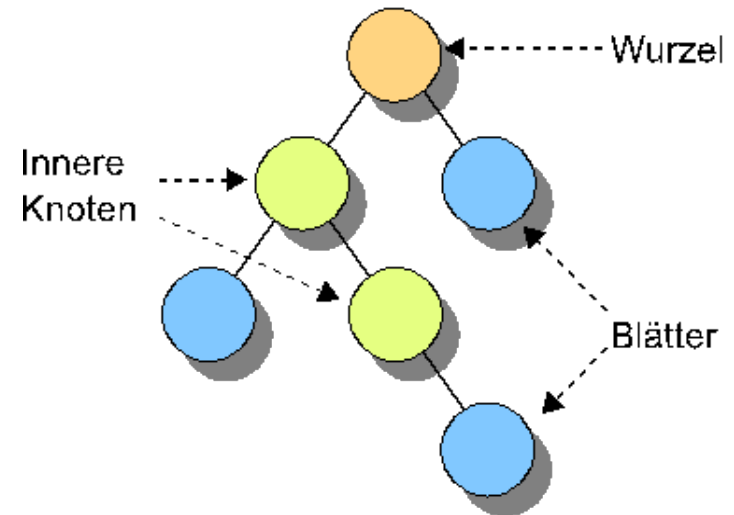
- Einführung
- Typen
- Typebounds
- Kompatibilität
- Generische Methoden



Einführung

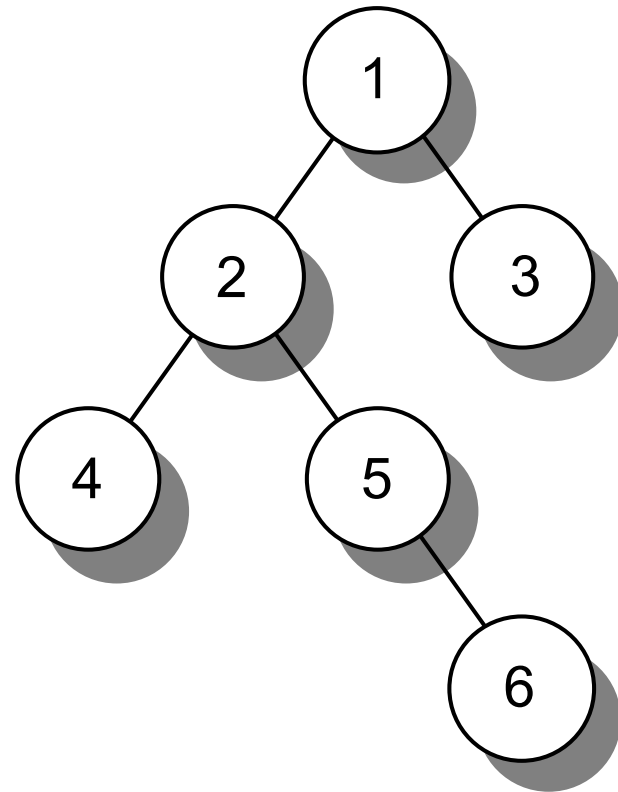
- Container = Klassen zum Speichern anderer Objekte als Elemente
- Beispiele: `Arrays`, `Collections`
- Containerklassen können mit beliebigen Elementtypen arbeiten
 - `collections`: ggf. Wrapperklasse nötig
 - sind ab Java 1.5 als „Generische Klassen“ definiert "Generische Typen" sind schon bekannt:
Beispiele: `LinkedList<Kontakt>`, `HashSet<String>`
- Ziel: Eigene Definition neuer generischer Klassen

- Klasse für Knoten eines binären Baums
- Pro Knoten (maximal) zwei Kindknoten gleichen Typs
- Bezeichnungen:
 - Wurzelknoten = (einziger) Knoten ohne Elternknoten
 - Innere Knoten = Knoten mit einem oder zwei Kindknoten
 - Blattknoten = Knoten ohne Kindknoten
- Technische Umsetzung:
 - Kindknoten **null** = kein Kindknoten



Beispiel: Baum

- Knoten enthalten beliebige Daten (Elemente) als Inhalt (→ Container)



Beispiel: ganzen Zahlen als Knoteninhalt

- Objektvariablen
 - `links`, `rechts` für Kindknoten
 - `element` für Knoteninhalt
 - Beliebiger Elementtyp → Basisklasse `Object` nötig
- Alle drei auswechselbar ⇒ kein `final`
- Java-Code:

```
public class Knoten {  
    private Object element;  
    private Knoten links;  
    private Knoten rechts;  
}
```

- Konstruktor mit Angabe des Inhalts und zwei Kindknoten

```
public Knoten(Object element, Knoten links, Knoten rechts)
{
    this.element = element;
    this.links = links;
    this.rechts = rechts;
}
```

- Getter / Setter für jede Objektvariable, Beispiel:

```
public Object getElement() {
    return element;
}
```

```
KnotenObject wurzelKnoten =  
    new KnotenObject(new Integer(1), new KnotenObject(new Integer(2),  
        new KnotenObject(new Integer(4), null, null), new KnotenObject(  
            new Integer(5), null, new KnotenObject(new Integer(6), null,  
                null))), new KnotenObject(new Integer(3), null, null));  
  
// Access first element  
int ersterWert = (Integer) wurzelKnoten.getElement();  
System.out.println(wurzelKnoten.getLinks());  
System.out.println(wurzelKnoten.getRechts());  
System.out.println(ersterWert);
```

- Finden Sie drei Vor- und drei Nachteile der Verwendung von Object als Datentyp für die Inhalte der Knoten

- Fehleranfällig, da
 - keine Typüberprüfung durch den Compiler möglich ist
 - mehrere verschiedene Elementtypen innerhalb eines Baums möglich sind
 - Gefahr von Laufzeitfehlern!
- Aufwändig, da
 - kein Autoboxing für Wrapperklassen (z.B. `Integer`) verwendet werden kann (keine Typinformation vorhanden)
 - unübersichtlicher Code!
 - jeder Zugriff auf ein Element einen Typcast im Code erfordert
 - hoher Aufwand bei Änderungen!

- Definition einer generischen Klasse:
 - Verwendung einer Typvariablen für den Elementtyp statt eines konkreten Typs
- Automatische Definition eines generischen Typs durch Angabe eines konkreten (Referenz-)Typs für die Typvariable bei der Verwendung

- Typvariable T :

```
public class Knoten<T> {  
    private T element;  
    private Node links;  
    private Node rechts;  
}
```

- Deklaration und Initialisierung

```
Knoten<Integer> knotenInteger = new Knoten<Integer>(1,  
    null, null);  
Knoten<String> knotenString = new Knoten<String>(  
    "Eins", null, null);
```

- Generische Klasse = Klassendefinition mit Typvariablen
- Generischer Typ = Generische Klasse + konkreter Typ
- Eine generische Klasse \Rightarrow viele generische Typen
- Gilt genauso für generische Interfaces
- Typvariablen speichern keine Werte, sondern sind nur Platzhalter für einen konkreten Typ
- Das Ersetzen von Typvariablen durch einen konkreten Typ findet beim Übersetzen durch den Compiler statt \rightarrow zur Laufzeit gibt es keine Typvariablen mehr



Generische Typen

- Syntax
 - Nach dem Klassennamen folgt die Typvariable in spitzen Klammern:
`class Klassenname<Typvariable> { ... }`
- Typvariable: beliebiger Java-Identifizier
- Konvention: einzelner Großbuchstabe in alphabetischer Nähe zum „T“ (type)
- Beispiel
`class Knoten<T> { ... }`
- Im Klassenrumpf
 - Verwendung der Typvariablen wie konkreter Typ
 - kleine Einschränkungen kommen noch

- Generische Klasse + konkreter Typ = generischer Typ
- Jeder generische Typ ist eigenständig, gleichrangig zu anderen Javatypen

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

```
Knoten<Integer> knoten;
```

```
knoten = new Knoten<String>("Hallo",null,null);
```

➔ **Typfehler**

- Es sind daher nur Bäume mit Knoten desselben Typs konstruierbar
 - Typecasts sind nicht mehr nötig:
String str = knoten.getElement();
Knoten<String> links = knoten.getLinks();
 - Compiler sichert korrekte Verwendung ab, keine Tests zur Laufzeit!

- können genauso wie generische Klassen definiert und verwendet werden
 - Ergebnis: generischer Typ
- Beispiel: Generisches Interface für alle Klassen deren Objekte Nachfolger haben

```
/**  
 * Interface für alle Objekte, die Nachfolger haben können.  
 */  
public interface HatNachfolger<T> {  
  
    /**  
     * Liefert den Nachfolger  
     *  
     * @return Nachfolger, falls vorhanden. Ansonsten null.  
     */  
    public T getNachfolger();  
}
```

- Jede Klasse, die das Interface implementiert, muss einen konkreten Typ für das generische Interface angeben:

```
/**
 * Eine fiktive Klasse, um die Implementierung eines generischen Interfaces zu
 * zeigen.
 *
 * @author Philipp Jenke
 */
public class Ding implements HatNachfolger<Ding> {

    /**
     * Jedes Ding hat eine Nachfolge-Ding
     */
    private Ding naechstesDing;

    public Ding(Ding naechstesDing) {
        this.naechstesDing = naechstesDing;
    }

    @Override
    public Ding getNachfolger() {
        return naechstesDing;
    }
}
```

- Auch eine generische Klasse kann ein generisches Interface implementieren
- Nötig: Angabe eines konkreten Typs für das generische Interface
- Möglich: Verwendung der eigenen Typvariablen
 - Typ ist damit ebenfalls eindeutig festgelegt

```
/**  
 * Ein Knoten in einem Binärbaum in dem die Knoten ausgewählt werden können.  
 *  
 * @author Philipp Jenke  
 */  
public class KnotenMitNachfolger<T> extends Knoten<T> implements  
    HatNachfolger<Knoten<T>> {
```

- Es sind beliebig viele verschiedene Typvariablen in einer generischen Klasse definierbar
- Beispiel
 - generische Klasse Paar verknüpft zwei Objekte
 - Typvariablen T, U = unbekannte, unabhängige Typen

- Aufgabe: Schreiben Sie eine Klasse `Paar` zum Verwalten von Paaren von Werten
 - beide Werten sind als generische Typen definiert
 - die Klasse ist unveränderlich (immutable)
 - die Werte werden im Konstruktor gesetzt
 - für beide Werte gibt es eine Getter-Methode

- Beispielanwendung:

```
Paar<String,Integer> paar =  
    new Paar<String,Integer>("Hallo", 42);  
String erstes = paar.getErstes();  
int zweites = paar.getZweites();
```




Typebounds

- Generische Klasse akzeptiert (bisher) beliebige Typen
- Oft sinnvoll: Einschränkung auf bestimmte Typen
- Lösung
 - Typebound bei der Definition einer generischen Klasse als Vorgabe angeben:

```
class Klassenname<Typvariable extends Typebound>
```

- Es werden nur konkrete Typen akzeptiert, die zum Typebound kompatibel sind!
- Typebound kann jeder beliebige Typ sein
- extends steht hier gleichermaßen für Klassen und Interfaces

- Knoten mit ausschließlich numerischem Inhalt:

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

- Integer, Double kompatibel zu Number, aber nicht String, Object →
Compiler erkennt unpassende Typargumente:

```
Knoten<Integer> knotenInteger; // ok
```

```
Knoten<Double> knotenDouble; // ok
```

```
Knoten<Object> knotenObject; // Fehler!
```

```
Knoten<String> knotenString; // Fehler!
```

- Mehrfache Typebounds
- Es kann sogar eine Liste von Typebounds angegeben werden
 - \Rightarrow mehrfache Einschränkungen
- Akzeptierte Typen müssen zu jedem Typebound kompatibel sein
- Syntax:

```
class Klassenname<Typvariable extends Typebound1 &  
    Typebound2 & Typebound3 & ...>
```

- Einschränkung
 - Typebound1 darf eine Klasse sein, der Rest nur Interfaces
- Reihenfolge der Typebounds ab Typebound2 irrelevant

- Typebounds mit Typvariablen
- Eine eigene Typvariable einer generischen Klasse kann auch für einen Typebound verwendet werden!
- Beispiel: Wunsch: Knotenelemente sollen verglichen werden können (um den Baum ggf. neu anzuordnen)
- Umsetzung: Typ `T` eines Knotens muss das Interface `Comparable<T>` implementieren
- `Comparable<T>` ist ein generisches Interface
- Definition einer generischen Knotenklasse für geordnete Bäume:

```
class Knoten<T> extends Comparable<T> { ... }
```
- Compiler überprüft die Einschränkung:

```
Knoten<String> knotenString; // ok, String impl. Interface  
Knoten<Object> knotenObject; // Fehler!
```

- Gesucht ist eine Klasse `Knoten`, mit der Knoten in einem Baum repräsentiert werden können. Jeder Knoten beinhaltet ein Element (Inhalt). Als Elemente sollen Objekte der Klasse `A` und `B` (die Sie bei Bedarf anpassen können) möglich sein.
 - `Knoten<A> knotenA = new Knoten<A>();`
 - `Knoten knotenB = new Knoten();`
- Objekte anderer Klassen sollen nicht erlaubt sein:
 - `Knoten<C> knotenC = new Knoten<C>();`
- Fragen
 - Wie könnte die Signatur der Klasse `Knoten` lauten? Wie stellen Sie die Anforderungen sicher?
 - Wie könnten Sie testen, ob Ihre Umsetzung korrekt ist?



Kompatibilität

- Implizite Typkonversion: für bestimmte primitive Typen

- Beispiel

`int → double:`

```
int intWert = 1;
```

```
double doubleWert = i;
```

- Implementierung: Klasse → Interface

- Beispiel `Deposit → Asset`

- Vererbung: Abgeleitete Klasse → Basisklasse

- Beispiel

`String → Object:`

```
String text = "hallo";
```

```
Object objekt = s;
```


- Autoboxing: Primitiver Typ → Wrapperklasse und umgekehrt
 - Beispiel

```
int → Integer:
int intWert = 1;
Integer integerWert = intWert;
```
- Kompatibilität zwischen Elementtypen überträgt sich auf Array-Typen
 - nur für Referenztypen
 - Beispiel:

```
Object[] objectArray = new String[10]; // ok
```
 - Dagegen keine Kompatibilität bei primitiven Typen und Wrapperklassen:

```
double[] doubleArray = new int[10]; // Fehler
Integer[] intArray = new int[10]; // Fehler
```

- generische Typen der gleichen generischen Klasse (aber unterschiedlicher Typ) nicht kompatibel
 - Kompatibilität zwischen konkreten (Element-)Typen ist irrelevant für generische Typen
 - Integer ist kompatibel zu Number
 - Number n = new Integer(23); // ok
 - aber Knoten<Integer> nicht kompatibel zu Knoten<Number>
Knoten<Number> kn = new Knoten<Integer>(23); → Fehler

- Generischer Typ mit Wildcard-Zeichen ? ("Joker")
- als konkreter Typ = Wildcard-Typ (*"unbekannter Typ"*)
 - Syntax: Generischer-Klassenname<?>
 - Beispiel: Knoten<?> knoten;
- Jeder generische Typ derselben generischen Klasse ist kompatibel zum Wildcardtyp
 - Beispiele:

```
knoten = new Knoten<String>("hallo");    // ok
knoten = new Knoten<Integer>(1);          // ok
```
- Wildcardtypen sind Typen, aber keine Klassen
 - geeignet zur Definition von Variablen
 - aber nicht zur Instanziierung von Objekten (ähnlich wie bei Interfaces)
 - Beispiel:

```
knoten = new Knoten<?>(); // Fehler!
```

- sinnvoll, wenn der konkrete Typ keine Rolle spielt
 - der Code kann für alle generischen Typen einer generischen Klasse identisch sein!
- Beispiel
 - Methode `knotenZaehlen` zum Zählen der Knoten in einem Baum
 - Inhalt der Knoten ist irrelevant, nur die Baumstruktur ist wichtig

```
public static int knotenZaehlen(Knoten<?> knoten) {  
    int result = 0;  
    if (knoten != null) {  
        result =  
            1 + knotenZaehlen(knoten.getLinks())  
            + knotenZaehlen(knoten.getRechts());  
    }  
    return result;  
}
```

Wildcard-Typ



- Compiler kennt konkreten Typ des Wildcard-Argumentes nicht
- daher
 - Kein lesender Zugriff auf Elemente, auch wenn zur Laufzeit ein konkreter Typ vorliegt:

```
Knoten<?> knoten = new Knoten<Integer>(1);  
Integer intWert = knoten.getElement();           // Fehler
```
 - Kein schreibender Zugriff auf Elemente, auch wenn zur Laufzeit ein konkreter Typ vorliegt:

```
Knoten<?> knoten = new Knoten<Integer>(1);  
knoten.setElement(5);    // Fehler
```
- Grund
 - Entscheidung über korrekten generischen Typ trifft der Compiler
 - kein Einfluss der tatsächlichen (dynamischen) Typen!

- Wildcard ? führt zu Kompatibilität mit generischen Typen für alle konkreten Typen
- Einschränkung der Kompatibilität mit Typebound möglich
- Syntax: Generischer-Klassenname<? extends Typebound>
- Es sind nur generische Typen kompatibel, deren konkreter Typ zu Typebound kompatibel ist
- Beispiel:

```
Knoten<? extends Number> knoten = new
    Knoten<Integer>(23); // ok
knoten = new Knoten<Object>(new Object()); // Fehler
```
- Nur lesende Zugriffe sind erlaubt

```
Number num = kn.getElement(); // ok
```

- Umgekehrte Einschränkung:
- Syntax: Generischer-Klassenname <? super Typebound>
- Es sind nur generische Typen kompatibel, bei denen Typebound zum konkreten Typ kompatibel ist
 - d.h. wenn der konkrete Typ eine Basisklasse von Typebound ist
- Beispiel:

```
Knoten<? super Number> knoten;  
knoten = new Knoten<Integer>(23); // Fehler  
knoten = new Knoten<Object>(new Object()); // ok
```
- Nur schreibende Zugriffe sind erlaubt

```
knoten.setElement(1); // ok
```

- 1) Welche(n) der folgenden Typen kann man für die Typvariable bei der Instanziierung eines Knoten-Objektes verwenden, sodass die Instanz der Variablen knotenzugewiesen werden kann?

`Knoten<T super Number> knoten;`

- Object?
 - Number?
 - Integer?
- 2) Geben Sie die Deklaration einer Knoten-Variablen an, in die alle Objekte, die ein Interface A implementieren, als generischer Typparameter gesteckt werden können.



Generische Methoden

- außer generischen Klassen, Interfaces, Typen gibt es auch generische Methoden! Diese können ...
 - in "normalen" Klassen definiert und aufgerufen werden
 - unabhängig von generischen Klassen / generischen Typen verwendet werden
- Methodendefinition mit Typvariablen zwischen Modifiern und Ergebnistyp in spitzen Klammern
- Syntax:
 Modifier <Typvariablen> Ergebnistyp
 Methodenname(Parameterliste)

- Beispiel
 - Methode `ifNull`, liefert zweites Argument, falls erstes `null`:

```
public static <T> T ifNull(T data, T backup) {  
    return (data != null) ? data : backup;  
}
```

- Beispiel: knotenZaehlen-Alternative

```
public static <T> int knotenZaehlenGenerisch(Knoten<T> knoten) {  
    int ergebnis = 0;  
    if (knoten != null) {  
        ergebnis =  
            1 + knotenZaehlenGenerisch(knoten.getLinks())  
              + knotenZaehlenGenerisch(knoten.getRechts());  
    }  
    return ergebnis;  
}
```

- Möglichkeit, den Typ T zu verwenden, wird hier nicht genutzt
 - verhält sich genau wie
int knotenZaehlen(Knoten<?> knoten) { ... }

- Aufruf mit konkreten Typen direkt vor dem Methodennamen
- Syntax:
Zielobjekt.<Typen>Methodenname(Argumentliste)
- Zielobjekt muss immer angegeben werden
 - this bei Aufruf von Methoden der eigenen Klasse
- Beispiele:

```
String text = <String>ifNull("Hallo", "Hi"); // "Hallo"  
int intWert = <Integer>ifNull(null, 4); // 4
```
- Compiler erkennt Typfehler:

```
int intWert = <Integer>ifNull("2", 4); // Fehler
```

- Mechanismus des Compilers, der automatisch fehlende Typinformationen ermittelt
- Konkrete Typen und das Zielobjekt können beim Aufruf einer generischen Methode weggelassen werden, wenn der Compiler diese eindeutig berechnen kann

- Beispiele:

```
String text = <String>ifNull("Hallo","Hi");
```

→ explizites Typargument

```
text = ifNull("Hallo","Hi");
```

→ äquivalent aufgrund Type-Inference

```
text = ifNull("Hallo","Hi");
```

→ äquivalent aufgrund Type-Inference

- Typen der Argumente legen konkreten Typ eindeutig fest

```
String text = ifNull(null, "Hallo");
```

- Rückschluss aus Ergebnistyp legt konkreten Typ **mittelbar** fest:

```
String text = ifNull(null, null);
```

- Kein Rückschluss über zwei oder mehr Stufen

```
String text = ifNull(ifNull(null, null), null);
```

→ wird nicht übersetzt

- Ok mit konkreter Typangabe oder String-Argument

```
String text= ifNull(<String>ifNull(null, null), null);
```

```
String text = ifNull(ifNull(null, "Hallo"), null);
```

- Typebounds können in generischen Methoden wie bei generischen Klassen verwendet werden
- Beispiel
 - Methode `median` liefert mittleres von drei Argumenten (anhand der Größe)
- Forderung: Der konkrete Typ `T` muss einen Größenvergleich zulassen
 - `T` (oder eine Basisklasse von `T`) muss das Interface `Comparable` implementieren!

- Methodensignatur:

```
<T extends Comparable<? super T>> T median(T x, T y, T z)
```

- Compiler sichert Einhaltung des Typebounds:

```
Integer i = median(3, 1, 2); // ok:
```

```
Object c = <Object>median(null,null,null); // Fehler
```


- Schreiben Sie eine generische Methode `print()`, die für eine `ArrayList` von generischen Elementen als Parameter hat. Die Elements sollen in der Methode zeilenweise ausgegeben werden (Verwenden der `toString()`-Methode der Elemente).

- Typen
- Typebounds
- Kompatibilität
- Generische Methoden

- Die Folien basieren teilweise auf Vorlesungsfolien von Prof. Martin Hübner, Hochschule für Angewandte Wissenschaften Hamburg