



coder-friendly's generic mug

Generische Typen

00PM, Ralf Lämmel

Motivation: *Wir wollen einen generischen Keller und nicht einen Keller für Ints.*

```
public class IntStack {  
    private IntListEntry top = null;  
    public void push(int item) {  
        IntListEntry e = new IntListEntry();  
        e.item = item;  
        e.next = top;  
        top = e;  
    }  
    public boolean isEmpty() { return top == null; }  
    public int top() { return top.item; }  
    public void pop() { top = top.next; }  
}
```

Begriffsbestimmung und Motivation

- * Generische Typen (“Generics”):

Typen mit mit einem oder mehreren Typparametern

- * Generische Methoden:

Methoden mit einem oder mehreren Typparametern

Felder?

Hauptanwendung: “Container”

Hauptanwendung von Generics

* Container

- Ein *Bild* besteht aus vielen Elementen.

- Ein *Betrieb* besteht aus vielen Abteilungen, usw.

- Ein *Bank* besteht aus Konten, Kunden, usw.

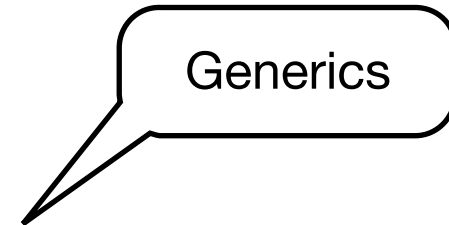
* Annahme: Wir wollen einen Containertyp **wiederverwenden**.

* Wiederverwendung verlangt dass

der ***Containertyp nicht den Elementtyp vorschreibt***.

Generics = Spezialfall von Polymorphismus

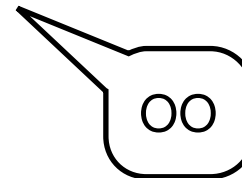
Formen von Polymorphismus



■ **Parametrischer (oder universeller) Polymorphismus**

■ **Beschränkter Polymorphismus (“Subtyping”)**

- Schnittstellenpolymorphismus
- ebenso für Klassentypen



■ **Ad-hoc Polymorphismus (“Überladung”)**

Spezialfall von beschränktem Polymorphismus.

Wiederholung:

Verwendung von *Schnittstellenpolymorphismus* zur Aufsummierung der Kontostände für die Konten einer Bank

```
public class Bank {  
    private Account[] accounts = new Account[42];  
    public float totalBalance() {  
        float result = 0;  
        for (Account a : accounts)  
            result += a.getBalance();  
        return result;  
    }  
    ...  
}
```

Datenkapsel unterschiedlicher Implementationstypen passen in dieses Feld.

Der tatsächliche Typ von *a* entscheidet über die anzuwendende Implementation von *getBalance*.

Programmierung von generischen Typen

Einfach verkettete Listen

```
public class IntListEntry {  
    public int item;  
    public IntListEntry next;  
}
```

Monomorph / nicht generisch: Items können nur vom Typ int sein.

```
public class ListEntry<T> {  
    public T item;  
    public ListEntry<T> next;  
}
```

Polymorph / generisch: Items können nun von einem beliebigen Referenztyp sein.

Methode zum Umkehren

```
public IntListEntry reverse() {
    if (this.next == null)
        return this;
    IntListEntry head = this;
    IntListEntry tail = this.next;
    head.next = null;
    IntListEntry result = tail.reverse();
    result.append(head);
    return result;
}
```

T wird gebunden als formaler Typparameter der Klasse *ListEntry*. Im Typ von *reverse* fungiert *T* als aktueller Typparameter für den Ergebnistyp. *T* wird auch für die Deklaration von lokalen Variablen benötigt.

```
public ListEntry<T> reverse() {
    if (this.next == null)
        return this;
    ListEntry<T> head = this;
    ListEntry<T> tail = this.next;
    head.next = null;
    ListEntry<T> result = tail.reverse();
    result.append(head);
    return result;
}
```

Datentypen mit Typ-Parametern

- * Datentypsdeklarationen die variabel sind in Typen

- **class** *DatentypName*<*Parameter-Typ*> { ... }

- * Datentypsanwendungen die einen Typ wählen

- *DatentypName*<*String*> *x* = ...;

- * Verwendung von formalem *Parameter* in *Datentyp*

- ... als aktueller Typ in generischem Typ

- ... als Typ eines Attributes

- ... als Argumenttyp einer Methode

- ... als Ergebnistyp einer Methode

- ... **aber nicht als Konstruktorname.**

Wir betrachten hier nur uneingeschränkte Parameter; vergl. mit Typschränken und Wildcards für Java Generics.

Methode zum Summieren

```
public int sum() {  
    return this.item + ((this.next==null) ?  
        0 :  
        this.next.sum());  
}
```

Dies ist eine
Instanzmethode von
IntListEntry.

```
public static int sum(ListEntry<Integer> x) {  
    return x.item  
        + ((x.next==null) ?  
            0 :  
            sum(x.next));  
}
```

Dies ist eine *statische* Methode
von ***ListEntry***. Wir müssen den
Typ der zu verarbeitenden
Elemente einschränken. Dies
ist bei einer Instanzmethode
nicht möglich.



Keller mit ints

```
public interface IntStack {  
    void push(int item);  
    boolean isEmpty();  
    int top();  
    void pop();  
}
```



Generische Keller

```
public interface Stack<T> {  
    void push(T item);  
    boolean isEmpty();  
    T top();  
    void pop();  
}
```



Binärbäume mit ints

```
public class BinIntTree {
    private int info;
    private BinIntTree left, right;
    public int sum() {
        return    this.info
                + (this.left != null ? this.left.sum() : 0)
                + (this.right != null ? this.right.sum() : 0);
    }
    ...
}
```

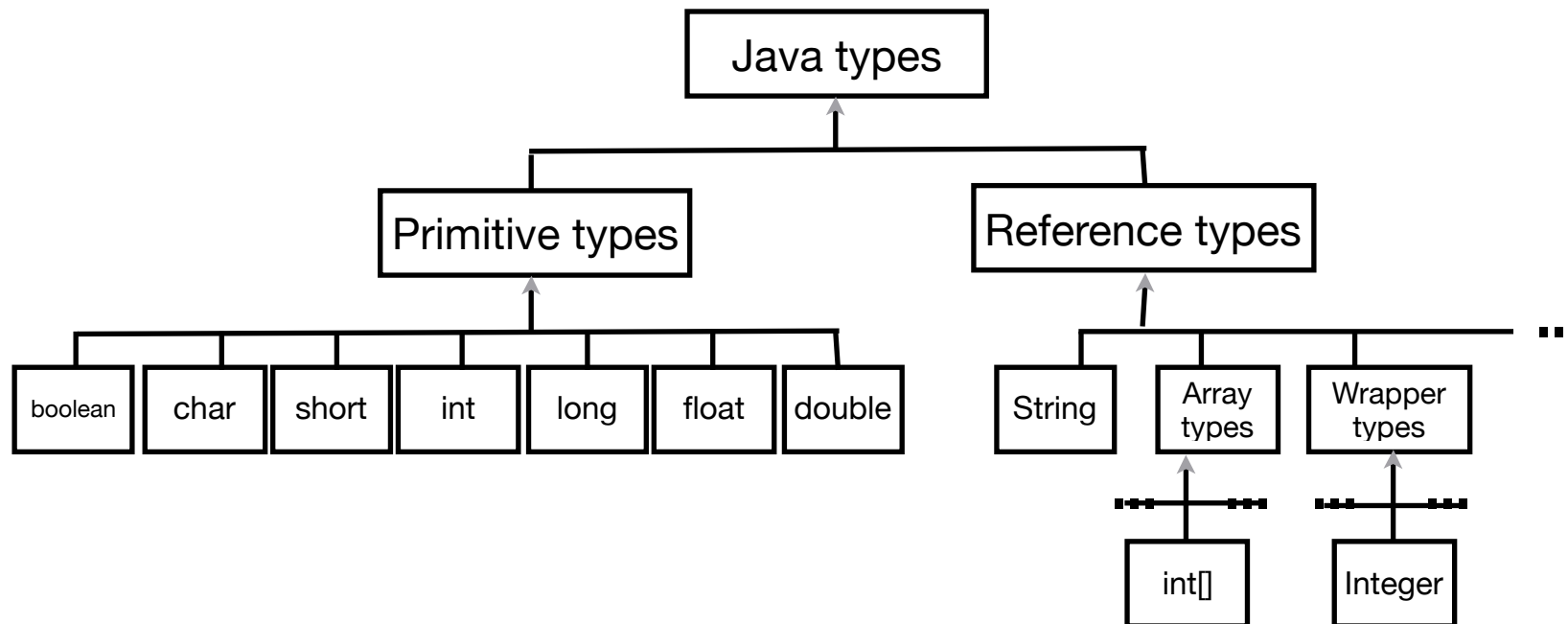
Generische Binärbäume



```
public class BinTree<T> {  
    private T info;  
    private BinTree<T> left, right;  
    public static int sum(BinTree<Integer> x) {  
        return x == null ? 0 : x.info  
            + (x.left != null ? sum(x.left) : 0)  
            + (x.right != null ? sum(x.right) : 0);  
    }  
    ...  
}
```

Wir können `sum` wieder nicht als Instanzmethode von *BinTree* definieren, da diese Klasse generisch ist. Wir können aber sehr wohl eine statische Methode über einem instanziierten Typ *BinTree<Integer>* definieren.

Java's Wrapper-Typen



*Primitive Typen können **nicht direkt** als aktuelle Parameter generischer Typen fungieren.*

Wrapper-Klassen für primitive Typen

- * Wrappen-Klassen:
 - *Integer*: Referenztyp für int
 - analog für anderen primitive Typen
- * Wrappen mit Konstruktor
- * Unwrappen mit Getter
- * Auto-un/boxing: Un-/wrappen implizit durch Typ.

```
Integer ri = null;    // Eine Wrapper-Variable
int pi = 0;          // Eine normale int Variable
ri = new Integer(42); // Wrappen von 42
ri = 42;             // Wrappen mit Autoboxing
pi = ri.intValue();  // Unwrappen von 42
pi = ri;             // Unwrappen mit Auto-Unboxing
```

Listen mit “einfacher” Iteration

```
public interface List<T> {  
    public int getLength();  
    public void add(T item);  
    public void reset();  
    public boolean hasNext();  
    public T next();  
}
```

Es kann nur eine laufende Iteration pro Liste geben.

Listen mit “fortgeschrittener” Iteration

- * Elemente können angehängt werden.
- * Elemente können entfernt werden.
- * Man kann über die Elemente iterieren.

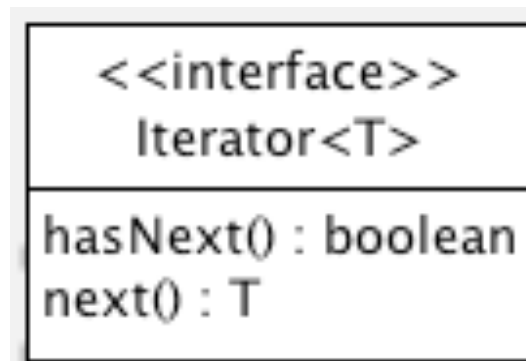
Zur Erinnerung:
solche Iteration wird
auch für Felder
unterstützt.

```
List<Integer> l = new ...;  
l.add(1);  
l.add(2);  
l.add(3);  
int sum = 0;  
for (int i : l)  
    sum += i;  
System.out.println(sum);
```

Diese Erwähnung der Liste
erzeugt einen neuen “Iterator”.

Die Iterator-Schnittstelle

- * Instanzen kapseln Iteration über Objektstruktur.
- * Methoden
 - *hasNext*: Testen ob Iteration noch Elemente liefert.
 - *next*: Liefere nächstes Element und rücke weiter.



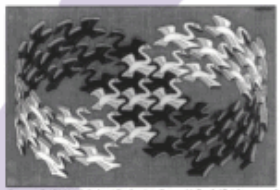
Entwurfsmuster

Ein Entwurfsmuster gibt eine bewährte generische Lösung für ein immer wiederkehrendes Entwurfsproblem, das in bestimmten Situationen auftritt.

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



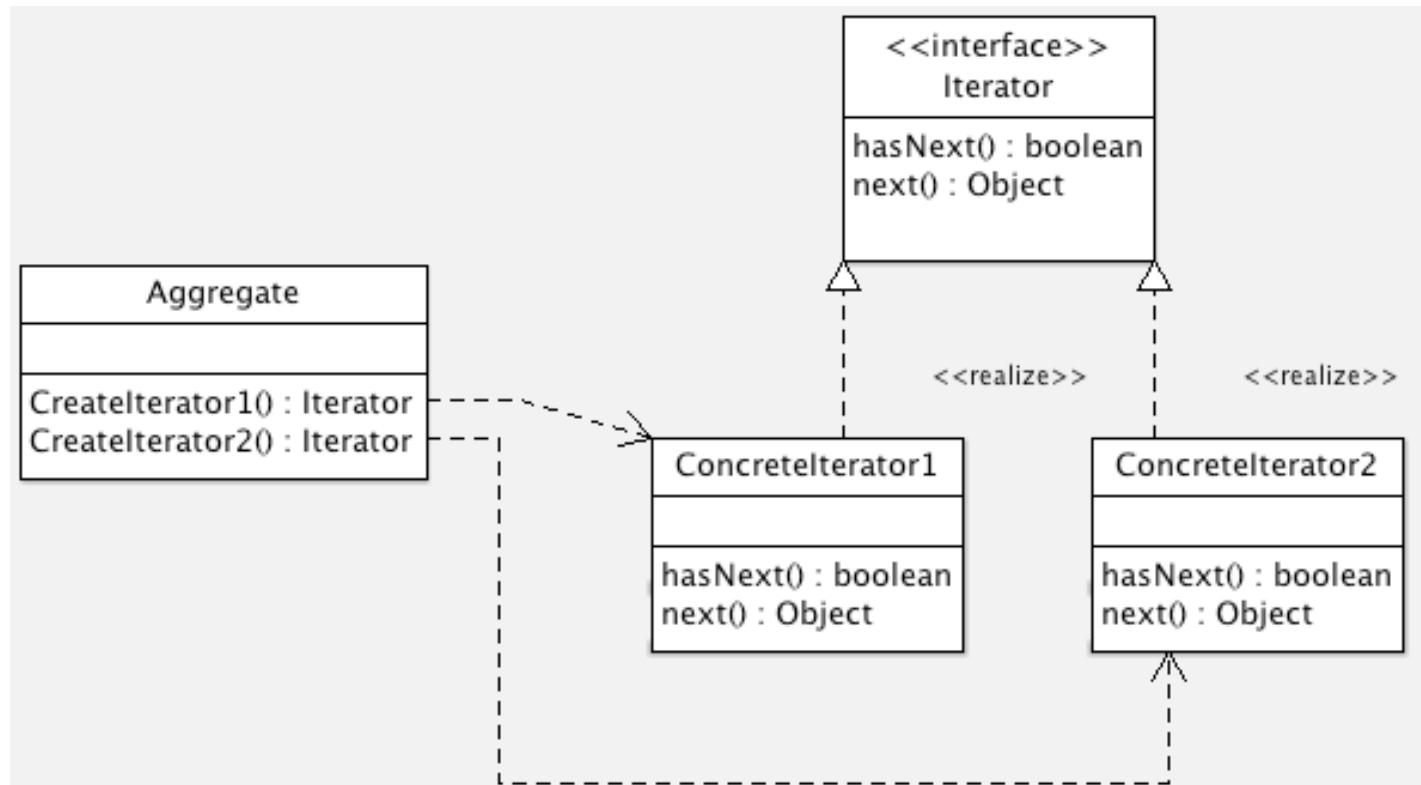
Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Design Patterns. Elements of Reusable
Object-Oriented Software Addison-Wesley
Professional Computing Series, 1995.

Entwurfsmuster *Iterator*

- * **Intention:** Trennung von Objektstruktur und Iteration über “Elemente”. Ermöglichung verschiedener Iterationen über gleiche Objektstruktur. Verzahnung von Iteration und Verhalten per Element.
- * **Teilnehmer:** Aggregate---Klasse für Objektstruktur mit Elementen
- * **Struktur**



Listen mit Iterationen (Schnittstellen)

```
public interface Iterable<T> {  
    Iterator<T> iterator();  
}
```

Ein Datentyp ist "iterierbar" wenn seine Schnittstelle einen "Iterator" zurückgeben kann.

```
public interface Iterator<T>  
    boolean hasNext();  
    T next();  
    ...  
}
```

Ein Iterator ist ein Datentyp, der mittels *next* immer neue Elemente zurückgibt bis das Ende der Iteration erreicht wurde; siehe *hasNext*.

```
public interface List<T> extends Iterable<T> {  
    void add(T item);  
    void remove(T item);  
    Iterator<T> iterator();  
}
```

Der Listen-Datentyp ist "iterierbar". Beachte: diese Vorgehensweise benutzt zwei (abhängige) Datenkapseln: die eigentliche Liste und einen Iterator.

Syntaktischer Zucker für for-each-Schleifen

```
int sum = 0;  
for (int i : l)  
    sum += i;
```

Die for-each-Schleife belegt eine Laufvariable des Elementtyps des Containers.

```
int sum = 0;  
Iterator<Integer> i = l.iterator();  
while (i.hasNext())  
    sum += i.next();
```

Es kann mehrere Iteratoren pro Container-Objekt geben.

Der Compiler erzeugt im Prinzip Code mit einem expliziten Iterator und greift darauf mit *hasNext/next* zu.

Heterogene Container

- * U.U. sollen Container verschiedener Typen beinhalten.

```
l.add(1);
```

```
l.add("2");
```

```
l.add(3);
```

- * Dazu kann "Object" als "Übertyp" verwendet werden.

```
List<Object> l = new LinkedList<Object>();
```

Object als universeller Typ

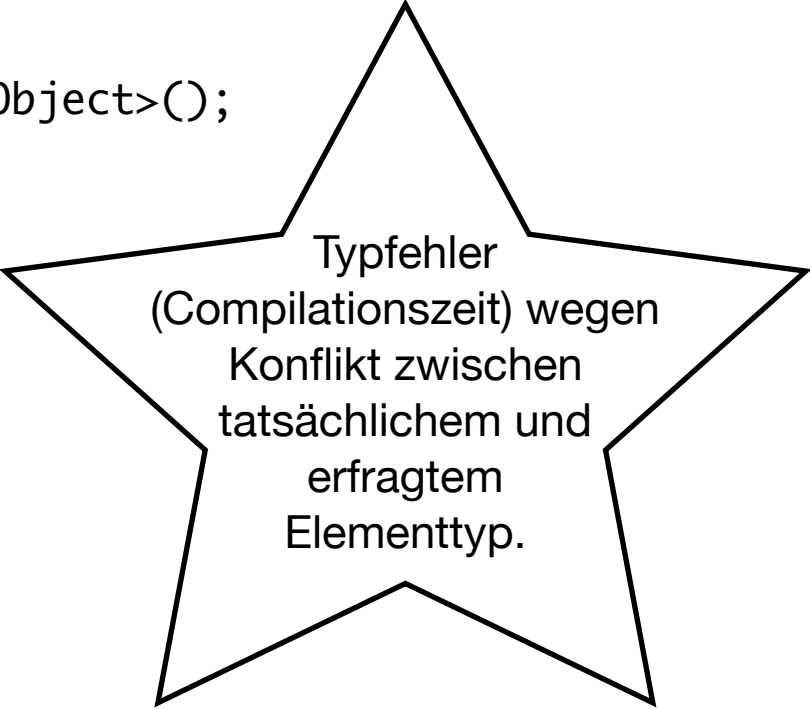
- * Besonderer (universeller) Typ
 - Nicht direkt instanziiierbar
 - Keine spezifischen Operationen
- * Variablen dieses Typs enthalten beliebige Referenzen
 - Zuweisung “vergisst” Typ der rechten Seite
 - Lesender Zugriff benötigt *Cast* zur “Erinnerung”

```
String s = "Ich bin eine Zeichenkette.";
Object o = s; // Nun bin ich nur noch ein Objekt.
// s = o; // Typfehler!
s = (String)o; // Wenn's klappt, ist o eine Zeichenkette.
```

Heterogene Container

- * Wie kann man trotzdem Typannahmen machen?
- * Beispiel Aufsummierung.

```
List<Object> l = new LinkedList<Object>();  
l.add(1);  
l.add("2");  
l.add(3);  
int sum = 0;  
for (int i : l)  
    sum += i;  
System.out.println(sum);
```



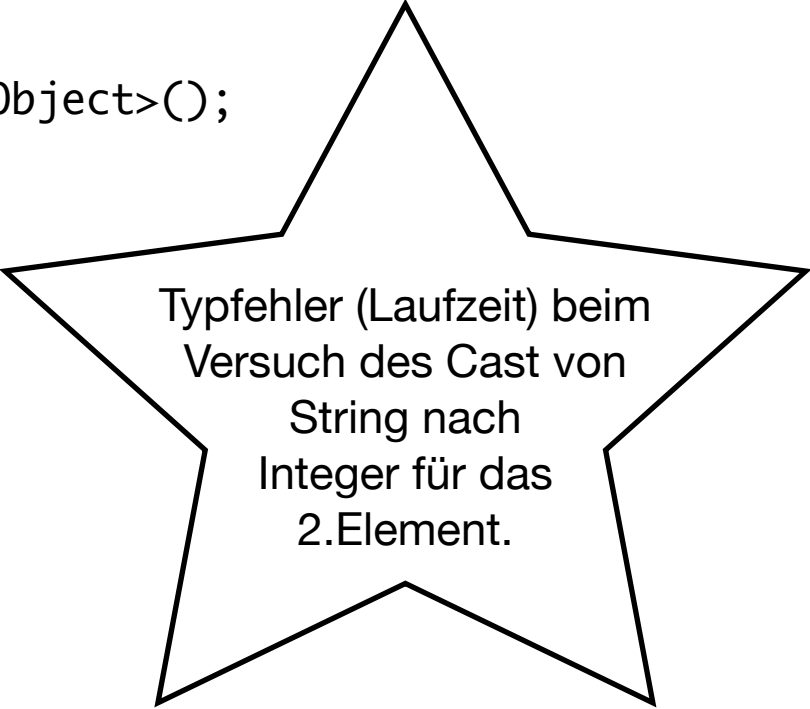
Typfehler
(Compilationszeit) wegen
Konflikt zwischen
tatsächlichem und
erfragtem
Elementtyp.

Heterogene Container

* Wie kann man trotzdem Typannahmen machen?

■ Verwendung von Down-casting: **(Zieltyp) object**

```
List<Object> l = new LinkedList<Object>();  
l.add(1);  
l.add("2");  
l.add(3);  
int sum = 0;  
for (Object o : l)  
    sum += (Integer)o;  
System.out.println(sum);
```



Typfehler (Laufzeit) beim
Versuch des Cast von
String nach
Integer für das
2.Element.

Heterogene Container

- * Wie kann man trotzdem Typannahmen machen?
 - Verwendung von Down-casting **und *Typ*test**.

```
List<Object> l = new LinkedList<Object>();  
l.add(1);  
l.add("2");  
l.add(3);  
int sum = 0;  
for (Object o : l)  
    if (o instanceof Integer)  
        sum += (Integer)o;  
System.out.println(sum);
```



Down-casting in der Bankanwendung

* Szenario:

- Die Kredite von allen Konten sollen aufsummiert werden.
- Nicht alle Konten bieten allerdings das Kredit-Feature.
- Wie selektiert man die geeigneten Konten also?

* Der folgende Versuch ist nicht akzeptabel.

```
public class Bank {  
    private Account[] accounts = new Account[42];  
    public float totalOverdraft() {  
        float result = 0;  
        for (Account a : accounts)  
            result += a.getOverdraft();  
        return result;  
    }  
    ...  
}
```

Die Anwendung
dieser Methode ist
nicht **typkorrekt**.

Aufsummierung der Kredite für die Konten einer Bank

```
public class Bank {  
    private Account[] accounts = new Account[42];  
    public float totalOverdraft() {  
        float result = 0;  
        for (Account a : accounts)  
            if (a instanceof CreditAccount)  
                result += a.getOverdraft(),  
        return result;  
    }  
    ...  
}
```

Test auf Machbarkeit
von Down-Casting

Reicht nicht!

Aufsummierung der Kredite für die Konten einer Bank

```
public class Bank {  
    private Account[] accounts = new Account[42];  
    public float totalOverdraft() {  
        float result = 0;  
        for (Account a : accounts)  
            if (a instanceof CreditAccount)  
                result += ((CreditAccount)a).getOverdraft();  
        return result;  
    }  
    ...  
}
```

Test auf Machbarkeit
von Down-Casting

Down-casting
(Beachte Klammerung!)

Zusammenfassung zu Down-casting und Typtest

- * Gegeben sei eine Variable v vom (statischen) Typ t .
- * t sei weiterhin ein Schnittstellentyp.
- * t' sei ein Klassentyp der t implementiert.
- * Alternativ ist t' ein Schnittstellentyp der t erweitert.
- * Folgende Konstruktionen sind nun erlaubt:

- **v instanceof t'**

- Boolescher Ausdruck für Typtest:
“Ist der aktuelle Typ von v t' ?”

- **$(t') v$**

- Laufzeittypanpassung (ohne Werteanpassung)
 - ▶ Ergebnis ist vom Typ t' , falls definiert.
 - ▶ Ergebnis ist definiert bei erfolgreichem Typtest.
 - ▶ Fehlschlagender Typtest impliziert Ausnahme.

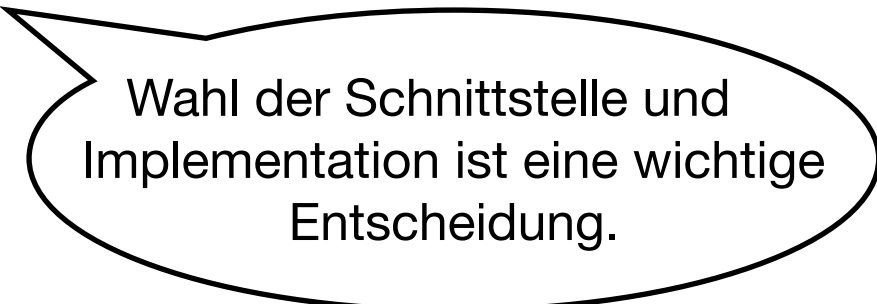
Wir übergehen an dieser Stelle weitere Aspekte, welche “Subtyping” betreffen.

Casting kann auch auf Kombinationen von primitiven Typen angewandt werden.

Die java.util Container-Typen

* Variationen

- Zugriff über Schlüssel, Identität, Iteration, ...
- Erhaltung der Einfügereihenfolge (“Listen”)
- Synchronisierung für parallelen Zugriff
- Unterstützung von Duplikaten
- ...



Wahl der Schnittstelle und Implementation ist eine wichtige Entscheidung.

Die java.util Container-Typen (Auswahl)

* Schnittstellen

■ Collection

- List
- Set
 - SortedSet
- Queue
 - BlockingQueue

■ Map

- SortedMap

* Implementationen

- LinkedList, ArrayList, Stack
- HashSet
- PriorityQueue

■ HashMap

Schnittstellen für Container

* **Collection**

- Hinzufügen von Elementen
- Entfernen von Elementen (Identitätsbasiert)
- Testen auf Leere und Mitgliedschaft
- Bestimmen der Größe
- Konvertieren in ein Feld

* **List**: Zusätzliche Index-basierte Operationen.

* **Set**: Zusätzliche mengentheoretische Operationen.

* **Queue**: Zusätzliche FIFO/LIFO-Operationen.

* **Map**: Funktionen als Schlüssel/Werte-Paare.

java.util.Collection

```
public interface Collection<T> extends Iterable<T>{  
    boolean add(T e);  
    boolean contains(Object o);  
    boolean isEmpty();  
    Iterator<T> iterator();  
    boolean remove(Object o);  
    int size();  
    Object[] toArray();  
    ...  
}
```

Dies ist die grundlegendste Schnittstelle.
Implementierungen müssen nicht eine bestimmte
Reihenfolge der Elemente versprechen.

java.util.List

```
public interface List<T> extends Collection<T> {  
    void add(int index, T element);  
    T get(int index);  
    int indexOf(Object o);  
    T remove(int index);  
    T set(int index, T element);  
    ...  
}
```

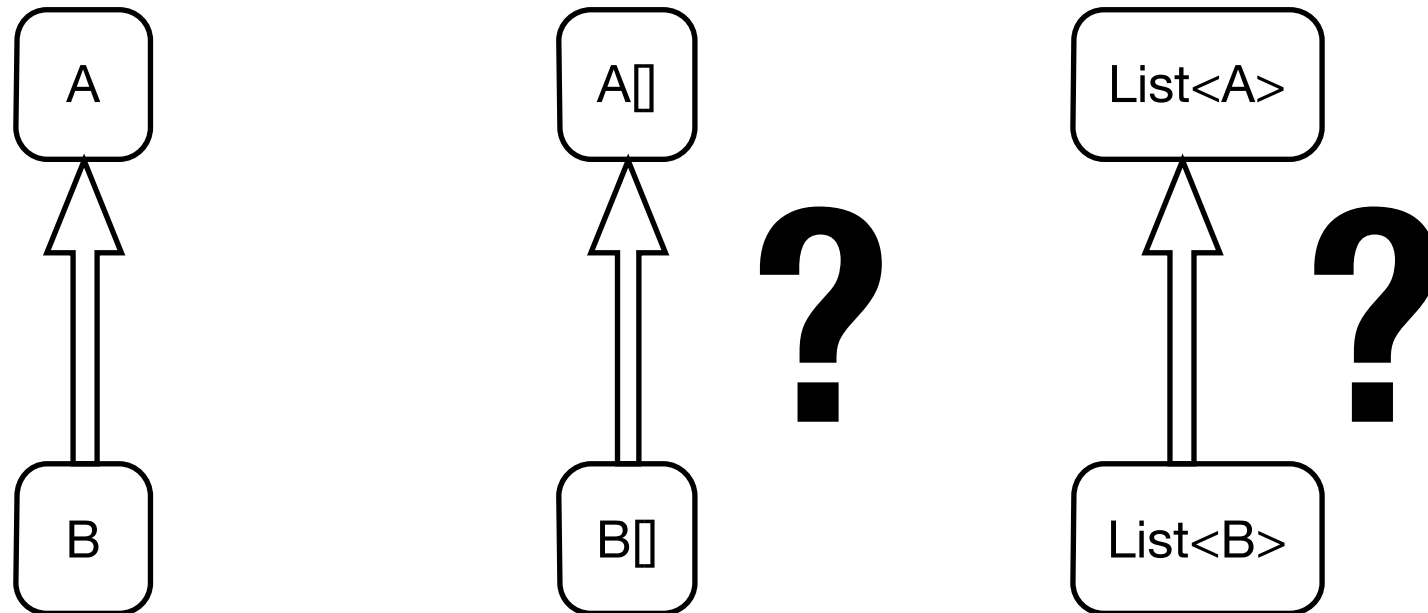
Es hängt von der konkreten Implementation ab, ob z.B. die Index-basierten Operationen effizient unterstützt sind.

Implementierungen für Container

- ArrayList: Erweiterbare Felder
- Vector: wie ArrayList mit Synchronisierung
- LinkedList: Liste mit schnellen Mutationen
- HashSet: Einfache Mengenimplementierung
- ...

Varianz und Invarianz

Was ist In-/varianz?



Wenn die Beziehungen für Feldtypen und Listentypen gelten, dann würden wir von Co-Varianz sprechen!

Covarianz von Feldtypen

* Ein Feldtyp $[B]$ ist ein Untertyp vom Feldtyp $[A]$, wenn B ein Untertyp von A ist.

* Beispiel:

```
public class A {}
```

```
public class B extends A {}
```

```
B[] bs = new B[] { new B() };
```

```
A[] as;
```

```
as = bs;
```

“zuweisungskompatibel”

Laufzeitfehler bei Covarianz

* Beispiel:

```
public class A {}
```

```
public class B extends A {}
```

```
B[] bs = new B[] { new B() };
```

```
A[] as;
```

```
as = bs;
```

```
as[0] = new A();
```



Invarianz von Java “generics”

* Anwendungen von generische Typen auf Typparameter in Untertypsbeziehung stehen in keiner Untertypsbeziehung.

* Beispiel:

```
public class A {}
```

```
public class B extends A {}
```

```
List<B> bs = ...;
```

```
List<a> as;
```

“zuweisungsinkompatibel”

~~as = bs;~~ // nicht kovariant

~~bs = as;~~ // nicht kontravariant

Ko- und Kontravarianz bei Methodentypen

```
public class Plant { ... }  
public class Grass extends Plant { ... }  
public class Herbivore {  
    public List<Plant> diet;  
    public void eat(Plant food) { }  
}  
public class Cow extends Herbivore {  
    public void eat(Grass food) { }  
}
```

Pflanzenfresser fressen
"alle" Pflanzen.

Diese Kühe fressen nur
Gras. Damit ist der Typ des
Arguments **ko-variant**.

Dieser Java-Code überschreibt
nicht wirklich "eat". Was passiert?
Warum ist Java so entworfen?

Im Einklang mit einfacher Typsicherheit:

- Kovariant im Resultatstyp.
- Kontravariant im Argumenttyp.

```
public abstract class A {  
    public A m(B b) { ... }  
}
```

Java hat keine solche Kontravarianz. Es greift Überladung.

Java gestattet tatsächlich solche Kovarianz.

```
}  
public class B extends A {  
    public B m(A a) { ... }  
}
```

* Zusammenfassung

- Datentypen für Container sollen generisch sein.
- Generizität braucht Object oder Typ-Parameter.
- Java bietet alle nötigen Formen von Polymorphismus.
- Java bietet eine reiche Container-Bibliothek.

* Ausblick

- Modellierung mit UML
- Syntax und Semantik von Programmiersprachen
- Weitere Aspekte der OO-Programmierung