Generische Typen in Java

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer und Axel Rauschmayer

Ziele

- Exkurs: Wrapper-Klassen für Grundatentypen kennen lernen
- Verstehen von parametrischer Typ-Polymorphie in Java
- Die Kollektionshierarchie kennen lernen

Exkurs: Grunddatentypen und ihre Klassen

- **Zu jedem Grunddatentyp gibt es eine korrespondierende Klasse:**

Wrapper-Klasse	Primitiver Typ
Integer	int
Double	double
Boolean	boolean
Character	char

- **Autoboxing:**

Java unterstützt die automatische Konversion zwischen prim. Typ und Klasse.

Beispiel:

```
int i     = 1906;
Integer j = i;
int k     = j;
```

```
Integer j1 = new Integer(1906);
System.out.println(j.equals(j1)); //true
System.out.println(j == j1);       //false
int k1 = j1;
System.out.println(k == k1);      //true
```

- **Achtung:**

Durch Klassen werden neue Objekte erzeugt
=> hoher Speicherplatzverbrauch
„==„ auf Klassen vergleicht Objekt-Referenzen; besser „equals“ verwenden!

Realisierung von Typ-Polymorphie in alten Java-Versionen

- Bis Java 1.4 wurde Typ-Polymorphie durch Vererbung von der Klasse `Object` realisiert.
- **Beispiel:** Eine Container-Klasse zur Aufnahme von Objekten verschiedener Typen

```
public class OldBox {

 private Object contents;

 public OldBox(Object cont) {
     contents = cont;}

 public Object getContents() {
     return contents;}

 public void setContents
                 (Object o) {
     contents = o;}

}
```

```
public class TestOldBox {

 public static void main(String[] args){
  OldBox b1 = new OldBox("Apple");
  String s = (String) b1.getContents();
  System.out.println(s);

  OldBox b = new OldBox(new Integer(3));
  int i = (Integer) b.getContents();
  System.out.println(s);
 }
}
```

Typüberprüfung zur Laufzeit

(Einfache) Realisierung mit generischen Typen

- Ab Java 1.5 kann Typ-Polymorphie parametrisch durch (formale) Typparameter realisiert werden (parametrische Polymorphie – eine Art universelle Polymorphie).
- **Beispiel:** Eine Klasse zur Aufnahme von genau einem Objekt

Typvariable

```
public class Box <T> {

 private T contents;

 public Box(T cont) {
    contents = cont;}

 public T getContents() {
    return contents;}

 public void setContents(T o) {
     contents = o;}
```

```
public static void main(String[] args){
 Box<String> b =
        new Box<String>("Apple");
 String s =  b.getContents();
 System.out.println(s);

 Box<Integer> b1 =
        new Box<Integer>(new Integer(3));
 int i = b1.getContents();
 System.out.println(i);
 }
}
```

Aktueller Typparameter: Typüberprüfung zur Übersetzungszeit!

Geschichtliches: Generische Typen in Java

- 1997: Martin Odersky, Philip Wadler: Pizza.
  In: Principles of Programming Languages Conference, POPL 97.
  „Eine funktionale Spracherweiterung von Java mit generischen Typen“
- 1998: Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding Genericity to the Java Programming Language, *OOPSLA 98*, Vancouver, October 1998.
  „Beschreibt Generic Java (GJ) als Weiterentwicklung von Pizza“
- September 2004: Tiger Release Java 1.5 mit generischen Typen basierend auf GJ.

Wadler, Odersky, Bracha, Stoutamire

Und von hinten: *Making Java easier to type, and easier to type*

Generische Typen: Syntax

- Deklaration einer generischen Klasse:

  `class` <<Klassenname>> **`<`** <<Liste von Typvariablen>> **`>`**

- Instantiierung eines generischen Typs:

  <<Klassenname>> **`<`** <<Liste von Typausdrücken>> **`>`**

- **Beispiele für Instantiierungen von Box<T>:**

```
Box<String>
Box<Integer>
Box<Box<String>> //Geschachtelter Typausdruck
Box<int> //unzulässig,Grunddatentypen nicht erlaubt als Typparameter
Box<String, String> //unzulässig, zu viele Parameter
```

- **Beispiel** `Box:` Erweitere z.B. `TestBox1` um:

```
Box<Box<String>> bb = new Box<Box<String>>(b);
Box<String> b1 = bb.getContents();
String s1 =  b1.getContents();
System.out.println(s1);
```

Einfache generische Methoden

- Deklaration einer generischen Methode:

< <<Liste von Typvariablen>> **>** <<Ergebnistyp>> <<Name>> **(** <<Parameterliste>> **)**

- Methodenaufruf einer generischen Methode (wie bisher):

<<Name>> **(** <<Liste von aktuellen Parametern>> **)**

- **Beispiel: Eine Methode, die zufällig ein Ergebnis wählt**

```
public class Random {
  public static <T> T zufall(T m, T n) {
    return Math.random() > 0.5 ? m : n;
  }
}
```

- **Beispiel Methodenaufrufe:**

```
String s = Random.zufall("Essen", "Schlafen");
int i = Random.zufall(100, 50);
```

Übersetzung von generischen Typen

Zwei Möglichkeiten zur Übersetzung von generischen Datentypen:

- **Heterogene Übersetzung.**
  Für jede Instantiierung (etwa `Box<String>, Box<Integer>, Box<Point>`) wird individueller Code erzeugt, also drei Klassen.
- **Homogene Übersetzung.**
  Für jede parametrisierte Klasse (etwa `Box<T>`) wird genau eine Klasse erzeugt, die die generischen Typinformationen löscht („type erasure“) und die Typparameter durch die Klasse `Object` ersetzt. Für jeden konkreten Typ werden zusätzlich Typanpassungen in die Anweisungen eingebaut.

Java nutzt die homogene Übersetzung, (C++ die heterogene):

- Weniger erzeugter Code, Möglichkeit der Übersetzung nach “Old” Java ohne generische Typen,
- Aber geringere Ausdrucksmächtigkeit der Typüberprüfung (zur Compilezeit)

Übersetzung von generischen Typen

Beispiel: Box<T> wird übersetzt in (die von `OldBox` bekannte Version) :

```
public class Box {
 private Object contents;

 public Box(Object cont) {
   contents = cont;
 }

 public Object getContents() {
   return contents;
 }
 . . .

 public static void main(String[] args){
   Box b = new Box("Apple");
   String s = (String) b.getContents();
   System.out.println(s);
 }
}
```

! Das ist eine (gute) Approximation. In Wirklichkeit wird aber direkt in Bytecode übersetzt!

Zusammenhang mit nichtgenerischen Typen

Generische Klassen können auch ohne Typparameter verwendet werden; damit können sie mit „altem“ Programmcode zusammen arbeiten.

- Ein parametrisierter Typ ohne Typangabe heißt ***Raw-Type***. Raw-Types bieten die gleiche Funktionalität wie parametrisierte Typen,
- Allerdings werden die Parametertypen nicht zur Compilezeit überprüft, was zu Problemen führen kann.
- **Beispiel:** Der Raw-Type von Box<T> ist Box.

```
Box<String> sBox = new Box<String>(„apple“);
Box oBox = new Box(); //Jede Parametertypinformation fehlt
oBox = sBox;

oBox.setContents( new Integer(3) ); // Warnung vor Unsafe type operation
//Eclipse schlägt Restrukturierung („Refactoring“) mit
//„infer generic type arguments“ vor.
//Denn:
System.out.println( sBox.getContents() ); // führt zu ClassCastException
```

Vererbung und Generische Typen

- Von generischen Klassen lassen sich in gewohnter Weise Unterklassen ableiten.
- Diese Unterklassen können, aber müssen nicht selbst generische Klassen sein.

- **Beispiel: Generische Klasse als Erbe:**

```
public class Pair<E, S> extends Box<E> {
  private S secondContent;

  public Pair(E first, S second) {
    super(first); this.secondContent = second; }

  public S getSecondContent() { . . . }
  public void setSecondContent(S second) {. . .}
}
```

- **Beispiel Nichtgenerische Klasse als Erbe:**

```
class StringBox extends Box<String> {
     public StringBox(String s) {super(s);}
}
```

Vererbung und Generische Typen

- **Aber: Vererbung bei Parametertypen induziert NICHT Vererbung der generischen Typen!**
- **Beispiel:** Ist eine **Box von Strings** auch eine **Box von Objects?**

```
Box <String> ls = new Box<String>(…);
Box <Object> lo;
lo = ls; //Typfehler!
lo.setContents(new Object());
String s = lo.getContents();  //würde zu ClassCastException führen
```

Es wird versucht, einem `String` ein `Object` zuzuweisen!

- **Regel:**

  Ist `Foo` Subtyp (Subklasse oder Subinterface) von `Bar` und `G` ist eine generische Deklaration,

  so ist `G<Foo>` *kein* Untertyp von `G<Bar>` !

Aus dem Java Tutorial (G. Bracha): „This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.“

Einschränkung von Typvariablen

Mit

<<Typvariable>> `extends` <<Typausdruck>>

ist es möglich nur solche Typparameter zuzulassen, die Erben von <<Typausdruck>> sind.

Beispiel: `FruitBox` enthält nur Früchte (Erben von `Fruit`)

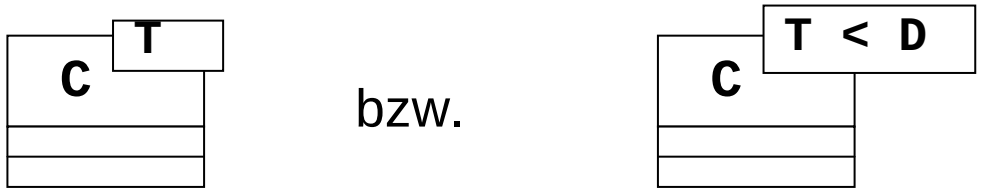
```
public class Apple extends Fruit{}
public class Steak {}
public class FruitBox <T extends Fruit>{
   << analog Box<T> >>
   public static void main(String[] args) {
     Apple a1 = new Apple();
     Steak s1 = new Steak();

     FruitBox<Apple> aBox = new FruitBox<Apple>(a1); //ok
     FruitBox<Steak> sBox =
        new FruitBox<Steak>(s1);//Compilezeitfehler:Bound Mismatch
  }
}
```

Generische Klassen in UML

Eine generische Klasse

`C<T> bzw. C<T extends D>`

Repräsentieren wir in UML durch einen kleinen Kasten in der rechten oberen Ecke der Klasse C, in dem die Typvariable (mit Angabe der Beschränkung) steht.



Platzhalter (Wildcards)

- **Problem:**

  Beim Drucken von `FruitBox` stoßen wir auf ein Problem:

```
public static void fruitBoxPrint( FruitBox<Fruit> f) {
  System.out.println(f); }
```

  druckt nur Objekte der Klasse `FruitBox<Fruit>`,

  da z.B. `FruitBox<Apple>` **keine** Unterklasse von `FruitBox<Fruit>` ist.

- **Lösung:** Der **Platzhalter (Wildcard, Joker) "?"** erlaubt mit

  <<Klassenname>> <?>

  die Verwendung beliebiger (korrekter aktueller) Typen für "?"

- **Beispiel:** Der Platzhalter "?" erlaubt das Drucken beliebiger Früchteschachteln

```
public static void fruitBoxPrint( FruitBox<?> f){
    System.out.println(f);}
public static void main(String[] args) {
   FruitBox<Apple> aBox = new FruitBox<Apple>(new Apple);
   fruitBoxPrint(aBox);
```

Beschränkte Platzhalter (Bounded Wildcards)

- Der **beschränkte Platzhalter (Wildcard, Joker)**

  <<Klassenname>> <? **extends** <<Typausdruck>> >

  beschränkt die Verwendung auf Subtypen von <<Typausdruck>>

- **Beispiel:** Drucken von Früchteschachteln mit Box<T>

```
public static void fruitBoxPrint1(Box<? extends Fruit> f){
     System.out.println(f);}
public static void main(String[] args) {
   Box<Apple> aBox = new Box<Apple>(new Apple);
   fruitBoxPrint1(aBox);
```

- Analog: Der **beschränkte Platzhalter (Wildcard, Joker)**

  <<Klassenname>> <? **super** <<Typausdruck>> >

  beschränkt die Verwendung auf Obertypen von <<Typausdruck>>

Generische Methoden vs. Platzhalter

- **Wann benutzt man generische Methoden, wann Platzhalter?**
  - mit Platzhalter:

```
public static void fruitBoxPrint1(Box<? extends Fruit> f){
  System.out.println(f);}
```

  - als generische Methode:

```
public static <T extends Fruit> void fruitBoxPrint2(Box<T> f){
  System.out.println(f);}
```

- **Benutzung von Platzhaltern, wenn:**
  - der Typparameter T wird nur einmal benutzt wird,
  - der Typ des Rückgabewertes nicht von T abhängt,
  - kein anderes Argument der Methode von T abhängig ist,
  - Das Typ-Argument für Typ-**Polymorphie** verwendet wird, d.h. dass verschiedene Argumenttypen erlaubt sind.

- Platzhalter für Typ-Polymorphie – mit Subtyping.
- Generische Methoden verwenden, um Abhängigkeiten zwischen Argumenttypen und/oder Rückgabetypen einer Methode herzustellen.

Wann braucht man generische Methoden

- **Bei Abhängigkeiten zwischen Argument- und Resultattyp**

Beispiel :

```
public static <T> T trace(T t){
 System.out.println(t);
 return t;
}
```

- **Bei Abhängigkeiten zwischen zwei oder mehr Argumenttypen**

Beispiel :

```
public static <T> void toBox(T t, Box<T> b){
  b.setContents(t);
}
```

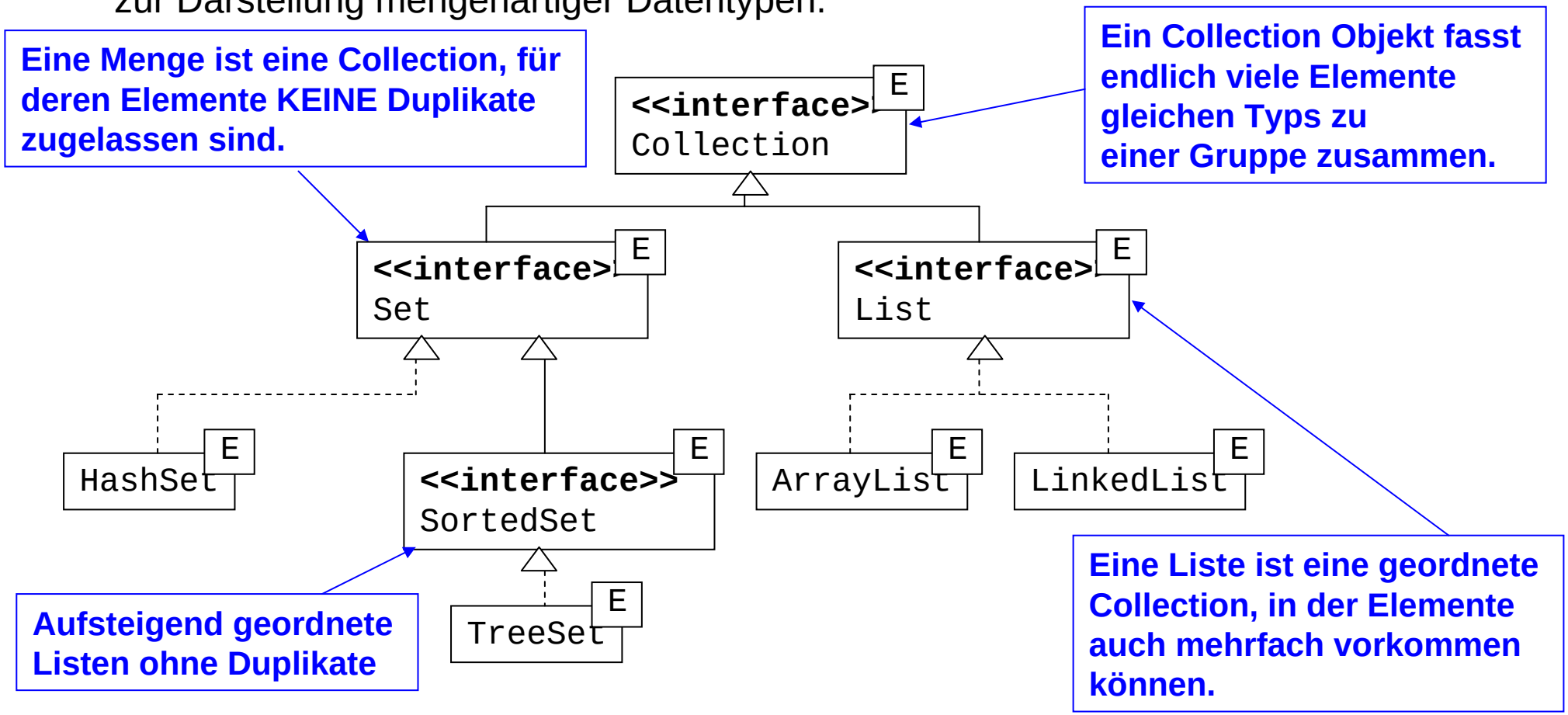
Eine erste Zusammenfassung

- Zu jedem Grunddatentyp gibt es eine korrespondierende **Wrapper Klasse**
- Java unterstützt die automatische Konversion zwischen prim. Typ und Klasse.

- Ab Java 1.5 kann Typ-Polymorphie parametrisch durch (formale) Typparameter realisiert werden (**parametrische Polymorphie**).
- **Generische Klassen und generische Methoden** haben Typvariablen als Parameter. Ihre Instantiierungsmöglichkeiten können durch Subtyp-Anforderungen eingeschränkt sein.
- Vererbung von aktuellen Parametern induziert **NICHT die Vererbung der generischen Typen**.
- Generische Klassen werden **“homogen” übersetzt**:
  Für jede parametrisierte Klasse (etwa Box<T>) wird genau eine Klasse erzeugt, die die generischen Typinformationen löscht („type erasure“) und die Typparameter durch die Klasse Object ersetzt. Für jeden konkreten Typ werden zusätzlich Typanpassungen in die Anweisungen eingebaut.

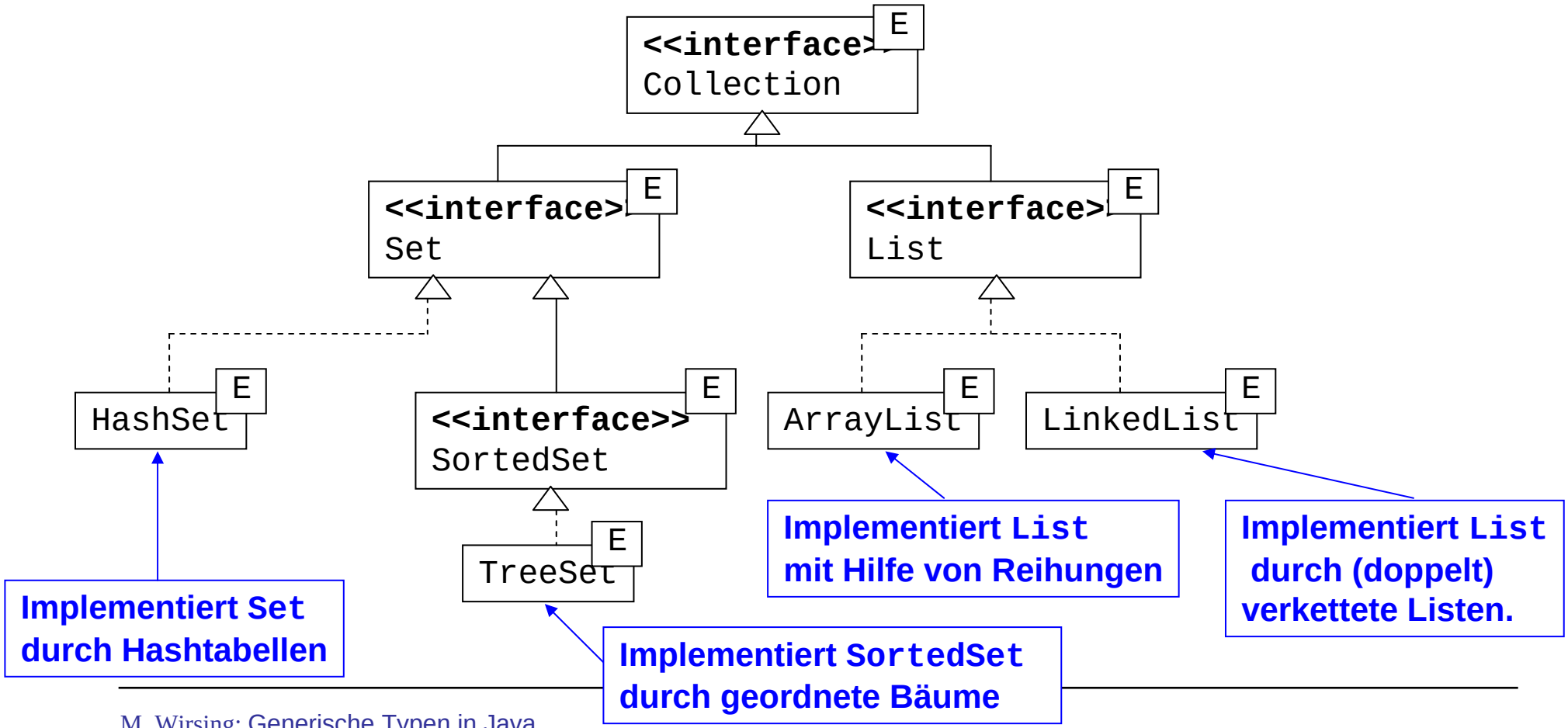
Das Collection Framework

- Das Collection Framework ist ein System von generischen Schnittstellen und Klassen zur Darstellung mengenartiger Datentypen.



Das Collection Framework

- Das Collection Framework ist ein System von generischen Schnittstellen und Klassen zur Darstellung mengenartiger Datentypen.



Die Schnittstelle Collection

Das

```
interface Collection<E> extends Iterable
```

bietet u.a. folgende Methoden an:

- **`boolean add( E o )`**

  Fügt dem Container ein Element hinzu und gibt true zurück, falls sich das Element einfügen lässt. Gibt false zurück, wenn schon ein Objekt gleichen Werts vorhanden ist und doppelte Werte nicht erlaubt sind.

- **`boolean addAll( Collection<? extends E> c )`**

  Fügt alle Elemente der Collection c dem Container hinzu.

- **`boolean contains( Object o )`**

  Liefert true, falls der Container ein inhaltlich gleiches Element enthält.

- **`boolean containsAll( Collection<?> c )`**

  Liefert true, falls der Container alle Elemente der Collection c enthält.

- **`boolean isEmpty()`** Liefert true, falls der Container keine Elemente enthält.
- **`Iterator<E> iterator()`** Liefert ein Iterator-Objekt über alle Elemente des Containers.
- **`boolean remove( Object o )`**

  Entfernt das angegebene Objekt aus dem Container, falls es vorhanden ist.

Die Schnittstelle List und eine Implementierung

- Das Schnittstelle

  `interface List<E> extends` `Collection<E>`

  spezifiziert, dass die geerbten Methoden `add` und `addall` Elemente am Ende der Liste einfügen und dass remove das erste Element entfernt.

  Weitere Operationen sind u.a.:

  - **`void add( int index, E element )`**

    Fügt ein Objekt an der angegebenen Stelle in die Liste ein.

  - **`E get( int index )`** liefert das Element an der angegebenen Stelle der Liste

- Die Klassen

  `class ArrayList<E>`

  `class LinkedList<E>` **(siehe spätere Vorlesung)**

implementieren List<E> .

Eine Anwendung von List

```
import java.util.*;
public class CollectionInfo {

/** Methode zur Ausgabe von Infos über eine Integer Collection */
  public static void printInfo(Collection<Integer> c) {
   System.out.println("Die Menge enthält " + c.size() + " Elemente.");
   System.out.println("Ist 0 in der Menge? " + c.contains(new Integer(0)));
   System.out.println("Ist 17 in der Menge? " + c.contains(17));

   System.out.println("Alle Elemente: ");
   for(Integer x : c)System.out.println(x);
  }
/**bildet Liste aus den Eingaben der Kommandozeile und druckt dafür Infos*/
  public static void main(String[] args) {
   List<Integer> l = new ArrayList<Integer>();
   for(String k : args)
       l.add(Integer.parseInt(k));
   printInfo(l); // Infos ueber die Liste der Eingaben
}
}
```

Die Schnittstelle Set und eine schnelle Implementierung

- Das Schnittstelle

  `interface Set<E> extends` `Collection<E>`

  spezifiziert, dass die geerbten Methoden `add` und `addall` keine Elemente zusätzlich aufnehmen, die schon vorhanden sind.

  Für Set-Objekte u,v entspricht
  - `u.addAll(v)` der Mengenvereinigung,
  - `u.retainAll(v)` dem Mengendurchschnitt,
  - `u.removeAll(v)` der Mengendifferenz `u\v` .

- Die Klasse

  `class HashSet<E>`

  implementiert u.a. Set<E> mit einer schnellen Hash-basierten Datenstruktur.

Konstante Zeitkompl. für add, remove, contains

- **Konstruktoren sind:**
  - **`HashSet()`**

    Erzeugt ein neues HashSet-Objekt mit 16 freien Plätzen und einem Füllfaktor von 0,75.
  - **`HashSet( Collection<? extends E> c )`**

    Erzeugt ein neues HashSet aus der Menge c gegebener Elemente.

Eine Anwendung von Set

Um Mengen (ohne Duplikate) zu betrachten, kann man die HashSet-Implementierung von Set verwenden:

```
...
   Set<Integer> s = new HashSet<Integer>();
   for(String k : args)
       s.add(Integer.parseInt(k));
   printInfo(s); // Infos ueber die Menge ohne Duplikate
```

Nochmals Vererbung und Generische Typen

- **Vererbung bei Parametertypen induziert NICHT**

  Vererbung der generischen Typen!

- **Beispiel:** Ist eine **Liste von Strings** auch eine **Liste von Objects?**

```
List <String> ls= new ArrayList<String>();
List <Object> lo;
lo = ls;
lo.add(new Object());
String s = lo.get(0);
```

Es wird versucht, einem `String` ein `Object` zuzuweisen!

- **Nochmals Regel:**

  Ist `Foo` Subtyp (Subklasse oder Subinterface) von `Bar` und `G` ist eine generische Deklaration,

  so ist `G<Foo>` *kein* Untertyp von `G<Bar>` !

Zusammenfassung

- Das **Collection Framework** ist ein System von generischen Schnittstellen und Klassen zur Darstellung mengenartiger Datentypen.
- **Collection** ist die Basisschnittstelle, sie wird von der Listenschnittstelle **List** und der Mengenschnittstelle **Set** erweitert.
- Listen-Implementierungen sind **ArrayList** und **LinkedList**; Mengen-Implementierungen sind **HashSet** und **TreeSet**.