Karlsruher Institut für Technologie

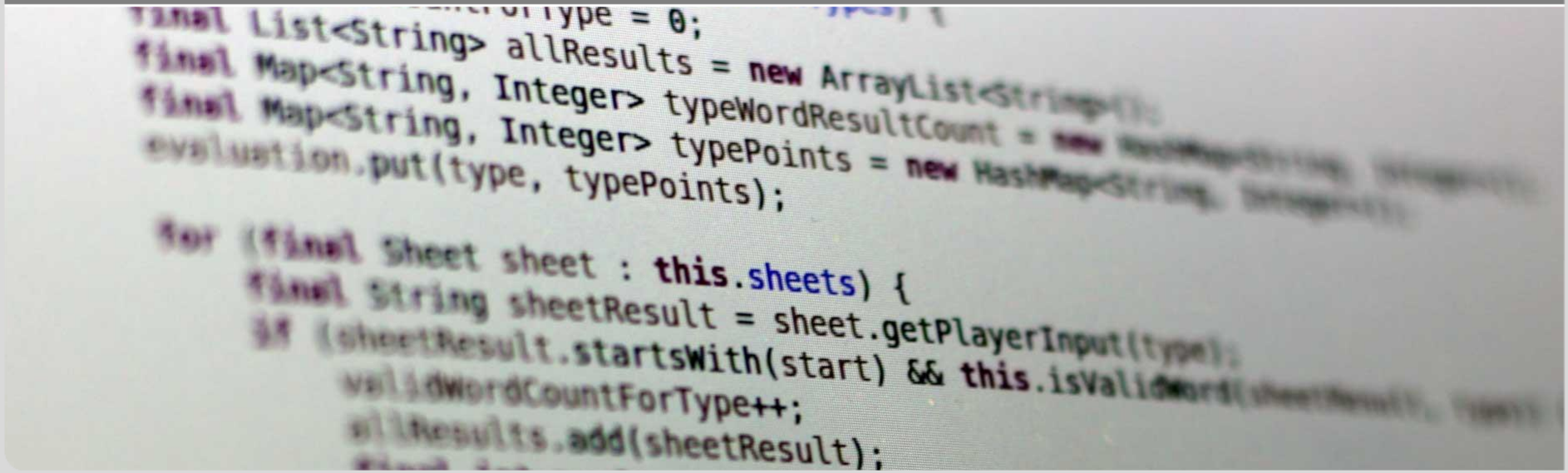
DHBW
Duale Hochschule
Baden-Württemberg

Programmieren II

Datenstrukturen / Das Collection-Framework

Heusch 14
Ratz 12.7

Institut für Automation und angewandte Informatik



Datenstrukturen (1)

- **Datenstrukturen** dienen dazu, **anwendungsspezifische Informationen** im Speicher abzubilden
  - Standardisierte Datenstrukturen helfen beim Software-Erstellungsprozess (Verständnis für „Muster“)
  - Standardisierte Datenstrukturen können durch gängige Operationen (z.B. Sortieren, Maximum/Minimum finden, etc.) unterstützt werden

- Grundlegende Datenstrukturen gab es bereits im JDK 1.0.
  - Das Paket `java.util` enthält hierfür z.B. die Klassen `Hashtable`, `Vector`, `Stack`.
  - Für viele Anwendungen reichten jedoch diese (wenigen) vordefinierten Datenstrukturen nicht aus.

Datenstrukturen (2)

- Aus diesem Grund wird seit JDK 1.2 das **Java Collection-Framework** mitgeliefert.
  Dieses Framework ist eine Sammlung von oft benötigten Containern und Algorithmen in Java.

  - In Java 1.5 wurde es um **generische Datentypen (Generics)** erweitert.

  - Seit Java 1.8 können Collections zusammen mit **Streams** und anonyme Funktionen (**Lambda Expressions)** verwendet werden.

Beispiel: List als „dynamisches“ Array

- **Interface `List`** definiert eine Ansammlung von Objekten
  - Reihenfolge (Position innerhalb der Liste), entsprechend dem Index 0…(n-1) in einem Array.
  - Duplikate sind erlaubt.
  - Das Interface `List` definiert die Schnittstelle (Methoden), die Implementierungen `ArrayList` und `LinkedList` implementieren dieses Interface.
  - In spitzen Klammern (Typparameter) kann der Datentyp angegeben werden, für den die Liste genutzt werden soll.

```
List<String> myList = new ArrayList<String>();
myList.add( "Hello" );
myList.add( "World" );
String myString = myList.get( 0 ); // "Hello"
```

Anwendung generischer Datentypen (1)

- Klassen und Methoden können um **Typparametern** ergänzt werden, um Typsicherheit trotz generischer Programmierung zu ermöglichen, z.B.

```
ArrayList<String> myList = new ArrayList<String>();
myList.add( "Hello" );
```

- In dieser `ArrayList` können nun **ausschließlich Objekte von Typ `String`** gespeichert werden, und die Liste liefert auch Objekte vom Typ `String` zurück.
- `ArrayList` kann jedoch auch mit beliebigen anderen Datentypen (Klassen) genutzt werden – sie ist **generisch**.

Anwendung generischer Datentypen (2)

- Um den **Datentyp einer generischen Klasse** festzulegen, wird dieser dem Klassennamen in Spitzen Klammern <> hinzugefügt. Dies gilt auch für Interfaces.
- Hier werden das Interface `List` und die Klasse `ArrayList` entsprechend für die Verwendung mit `Integer`-Objekten versehen:

```
List<Integer> my = new ArrayList<Integer>();
```

- Seit Java 1.7 kann der Datentyp auch weggelassen werden, wenn er aus dem Kontext heraus bereits feststeht („Diamond Operator“ <>):

```
List<Integer> my = new ArrayList<>();
```

Anwendung generischer Datentypen (3)

- Als Typparameter generischer Interfaces und Klassen können **nur Interfaces, Klassen und komplexe Aufzählungstypen (`enum`)** verwendet werden.
- **Primitive Datentypen** müssen daher in den entsprechenden **Wrapperklassen** gekapselt werden.

```
List<int> my = new ArrayList<>();     // Compiler-Fehler!

List<Integer> my = new ArrayList<>(); // richtig!
my.add( 42 );  // autoboxing
```

- Autoboxing hilft hier ungemein… ☺

Zum Vergleich

- Ohne Generics

```
import java.util.ArrayList;
import java.util.List;
…
List my = new ArrayList();
my.add( 42 ); // autoboxing
my.add( "Hello" );
Object erstes = my.get(0);
Object zweites = my.get(1);
```

➔ Keine Garantie, was in der Liste steckt (`Object`)
➔ Alles kann in der Liste stecken, auch „gemischte“ Typen

- Mit Generics

```
import java.util.ArrayList;
import java.util.List;
…
List<Integer> my = new ArrayList<>();
my.add( new Integer( 4711 ) );
my.add( 42 ); // autoboxing
Integer erstes = my.get(0);
Integer zweites = my.get(1);
```

➔ Compiler kann Datentypen überprüfen
➔ Keine gemischten Einträge
➔ `.get()` liefert korrekten Typ

Die Collection API

- Die Architektur besteht aus drei wichtigen Komponenten:
  - Alle Methoden sind in **Interfaces** im **Paket `java.util`** definiert.
  - Ein **Interface definiert nur die Schnittstelle für den Zugriff**. Die Speicherung der Daten bestimmt die konkrete Implementierung.
    - z.B. `List`, `Set`, `SortedSet`, `Map`
  - **Unterschiedlichen Implementierungen** unterscheiden sich meist darin, wie die Daten verwaltet werden: Eine Liste könnte man z. B. sowohl mit einem Array als auch durch direkte Verkettung der Elemente realisieren.
    - z.B. `ArrayList`, `LinkedList`, `HashSet`, `TreeSet`, `HashMap`, `TreeMap`

Collection Interfaces

- **`Collection`** Eine beliebige Gruppe von Objekten. Die Interfaces **`Set`**, **`SortedSet`** und **`List`** sind von **`Collection`** abgeleitet.
- **`Set`** beschreibt eine Gruppe von Objekten mit Mengencharacter, d.h. Duplikate sind nicht erlaubt.
- **`SortedSet`** ist Analog zu **`Set`** mit dem Unterschied, dass die Elemente sortiert werden.
- **`List`** beschreibt eine Gruppe von Objekten, die über Integer-Werte indiziert werden. Duplikate sind erlaubt. Ein neues Element kann an eine beliebige Stelle positioniert werden.
- **`Map`** definiert eine Zuordnung von Schlüsseln zu Werten. Jedem Schlüssel wird eindeutig ein Wert zugewiesen. Schlüssel können keine Duplikate enthalten, Werte schon.
- **`SortedMap`** ist eine **`Map`**, deren Schlüssel sortiert werden.

Wichtige Methoden des Interfaces `Collection`

- Beispielhaft seien hier einige wichtige Methoden des Interfaces `Collection` aufgeführt:
  - `boolean` **`add`**`(E e)`
    fügt das Element `e` in das `Collection`-Objekt hinzu (falls möglich bzw. nötig). `E`: Typ* der Elemente in dieser `Collection`
  - `boolean` **`addAll`**`(Collection<? extends E> c)`
    fügt alle Element der Collection `c` in das `Collection`-Objekt hinzu (falls möglich bzw. nötig). `E`: Typ* der Elemente in der `Collection`
  - `boolean` **`remove`**`(Object o)`
    Entfernt das Objekt `o` aus dem `Collection`-Objekt, falls es vorhanden ist.
  - `boolean` **`contains`**`(Object o)`
    Liefert `true`, falls das `Collection-Objekt` das Element `o` enthält, sonst `false`. Hinweis: Es gibt auch `containsAll` (…)
  - `int` **`size`**`()`
    Liefert die aktuelle Anzahl der Elemente

Implementierungen von Collections (1)

- Die Tabelle zeigt die seit Java SDK 1.4 verfügbaren Implementierungen der Collection-Interfaces:

Interface / Klasse	List	Set	SortedSet	Map	SortedMap	JDK 1.0
LinkedList	✘					
ArrayList	✘					
Vector	✘					✘
HashSet		✘				
TreeSet		✘	✘			
HashMap				✘		
WeakHashMap				✘		
TreeMap				✘	✘	
Hashtable				✘		✘

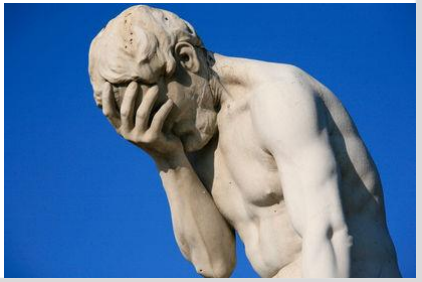
Implementierungen von Collections (2)

- **`HashSet`** implementiert das `Set`-Interface und speichert seine Elemente in einer Hash-Tabelle.
- **`TreeSet`** implementiert das `SortedSet`-Interface und speichert seine Elemente in einem Baum.
- **`ArrayList`** implementiert das `List`-Interface und speichert seine Elemente in einem Array.
- **`LinkedList`** implementiert das `List`-Interface und speichert seine Elemente in einer verketteten Liste. Werden die Elemente häufig durchlaufen, hat `LinkedList` eine bessere Performance als `ArrayList`.
- **`HashMap`** implementiert das `Map`-Interface und speichert seine Elemente in einer Hash-Tabelle.
- **`WeakHashMap`** implementiert das `Map`-Interface und speichert seine Elemente in einer Hash-Tabelle. Wird der Schlüssel außerhalb der `WeakHashMap` nicht mehr referenziert, so wird das entsprechende Schlüssel-Wert-Paar automatisch vom Garbage-Collector entfernt.
- **`TreeMap`** implementiert das `SortedMap`-Interface und speichert seine Elemente in einem Baum.

Implementierungen von Collections (3)

- Auch die beiden Klassen **`Vector`** und **`Hashtable`** sind seit dem JDK 1.2 Implementierungen der Collection-Interfaces (`List` bzw. `Map`).
- Beide Klassen waren seit der Version 1.0 im JDK enthalten und wurden nachträglich an das Collection-Framework angepasst.
- Aus diesem Grund sind sie nicht ganz mit den nun folgenden Implementierungen gleichzustellen:
  - Zum einen besitzen diese Klassen zusätzlich zu den Methoden der Collection-Interfaces weitere Methoden zum Datenzugriff, die bereits aus historischen Gründen vorhanden waren,
  - zum anderen ist der Zugriff sowohl auf `Vector` als auch auf `Hashtable` bereits synchronisiert. Synchronisierte Varianten der oben beschriebenen neuen Implementierungen muss man explizit über Wrapper-Implementierungen anfordern.

➔ **Versions-Kauderwelsch!**



Wichtige Methoden des Interfaces `List`

- Zusätzliche Methoden des Interfaces `List` (erweitern die von `Collection` geerbten Methoden):
  - `E` **`get`**`(int i)`
    Liefert das `i`-te Objekt aus der Liste
  - `int` **`indexOf`**`(Object o)`
    Liefert den Index des ersten Vorkommen von Objekts `o` aus dem `List`-Objekt, falls es vorhanden ist (sonst `-1`).
  - `int` **`lastIndexOf`**`(Object o)`
    Liefert den Index des letzten Vorkommen von Objekts `o` aus dem `List`-Objekt, falls es vorhanden ist (sonst `-1`).
  - `E` **`remove`**`(int i)`
    Entfernt das `i`-te Objekt aus dem `List`-Objekt und gibt es als Ergebnis der Methode zurück

  Anmerkung: Vorsicht ist geboten bei `Integer`-Objekten! Trotz Autoboxing greift hier `boolean remove(Object o)` von `Collection` → es sind Objekte!

Die Klasse `java.util.ArrayList`

- Die Klasse **`ArrayList`** ist eine Implementierung des Interfaces `List`
- Sie repräsentiert quasi ein Array variablen Inhalts, dessen Länge bei Bedarf automatisch angepasst wird.
- Über Indizes kann direkt auf die einzelnen Elemente zugegriffen werden.
- In einer `ArrayList` kann jedes von `Object` abgeleitete Objekt gespeichert werden, ebenso `null`-Werte.
- Hinweis:
  Wenn eine *sichere Nebenläufigkeit* (Thread-Sicherheit) erforderlich ist, muss dies über einen Wrapper angefordert werden (folgt später).

Die Klasse `java.util.LinkedList`

- Die Klasse **`LinkedList`** ist eine Implementierung der Interfaces `List` und `Deque`
- Speichert den Vorgänger und Nachfolger der Elemente (Doppelt verkettete Liste)
- Eignet sich besonders gut beim häufigen Einfügen und Löschen von Elementen, da nur die Nachbarverweise verändert werden müssen
- Zugriff auf bestimmtes Element dadurch jedoch langsam (für `n`-tes Element müssen `n` Verkettungen durchlaufen werden)
- In einer `LinkedList` kann jedes von `Object` abgeleitete Objekt gespeichert werden **und zusätzlich auch** `null`-Werte.

Beispiel: `List` und `ArrayList`

- Die Variable vom Typ des Interfaces `List`,
  die Instanz vom Typ der Implementierung `ArrayList`!

```
import java.util.ArrayList;
import java.util.List;

public class ListExample {

    public static void main(String[] args) {
        List<String> words = new ArrayList<>();
        words.add( "Some" );  // 0
        words.add( "words" ); // 1
        words.add( "in" );    // 2
        words.add( "a" );     // 3
        words.add( "list." ); // 4
        System.out.println( words.get( 2 ) ); // "in"
    }

}
```

Beispiel: Ausgabe einer `List`

```java
import java.util.ArrayList;
import java.util.List;

public class ListExample2 {

    public static void main(String[] args) {
        List<String> words = new ArrayList<>();
        words.add( "This" );
        words.add( "is" );
        words.add( "an" );
        words.add( "example" );

        for(String s : words) {
            System.out.println( s );
        }
    }

}
```

Ausgabe

```
This
is
an
example
```

Methoden des Interfaces `java.util.Set`

- Die Methoden `add(E e)`, `remove(Object o)`, `contains(Object o)`, `isEmpty()`, `size()` werden (wie bei `List`) vom Interface `Collection` geerbt.
- Im praktischen Handling funktionieren `List` und `Set` also sehr ähnlich. Aber
  - `Set` erlaubt keine Duplikate,
  - die Elemente in einem `Set` haben keine feste Position (außer in einem `TreeSet`, da werden sie automatisch sortiert).
    ➔ Man kann daher nicht mit `get(i)` auf das `i`-te Element eines `Set` zugreifen!
- Die Implementierungen **`HashSet`** bzw. **`TreeSet`** speichern die Elemente in einer Hashtabelle bzw. (sortiert) in einer Baumstruktur.

Beispiel: Set mit HashSet

```
import java.util.Set;
import java.util.HashSet;

public class SetExample {

    public static void main(String[] args) {
        Set<String> mySet = new HashSet<>();  // HashSet ➔ Reihenfolge?
        mySet.add("Mia");
        mySet.add("Uli");
        mySet.add("Peter");
        mySet.add("Mia");          // Duplikat wird nicht hinzugefügt!
        for(String s : mySet) {
            System.out.println(s);
        }
        System.out.println("Element count: " + mySet.size());
    }

}
```

Ausgabe REIHENFOLGE WEICHT AB!

```
Uli
Mia
Peter
Element count: 3
```

Beispiel: Set mit TreeSet

```
import java.util.Set;
import java.util.TreeSet;

public class SetExampleSorted {

    public static void main(String[] args) {
        Set<String> mySet = new TreeSet<>();  // TreeSet ➔ sortiere!
        mySet.add("Mia");
        mySet.add("Uli");
        mySet.add("Peter");
        mySet.add("Mia");          // Duplikat wird nicht hinzugefügt!
        for(String s : mySet) {
            System.out.println(s);
        }
        System.out.println("Element count: " + mySet.size());
    }

}
```

Ausgabe SORTIERT!

```
Mia
Peter
Uli
Element count: 3
```

Beispiel: Bestimmte Anzahl von Zufallszahlen

```
import java.util.Random;
import java.util.Set;
import java.util.TreeSet;

public class SetRandomExample {

    public static void main(String[] args) {
        Set<Integer> numbers = new TreeSet<>();  // TreeSet ➔ sortiert!
        Random rnd = new Random();
        // Erzeuge genau 10 verschiedene Zufallszahlen zwischen 1 und 20
        do {
            numbers.add( rnd.nextInt(20)+1 );  // Keine Duplikate!
        } while ( numbers.size() < 10 );       // Stopp, wenn es 10 sind
        // Ausgabe
        for (Integer n : numbers) {
            System.out.print( n + " " );
        }
    }

}
```

Ausgabe

```
1 2 3 8 9 10 11 16 17 18
```

Methoden des Interfaces `java.util.Map`

- Beispielhaft seien hier einige wichtige Methoden des Interfaces `Map` aufgeführt:
  - `V` **`put`**`(K key, V value)`
    Fügt das Objekt `value` unter dem Schlüssel `key` zur `Map` hinzu. Existierte für diesen Schlüssel bereits ein Wert in der `Map`, wird dieser entfernt und als Ergebnis der Methode zurückgegeben.

    `K`: Typ der Schlüssel der `Map` / `V`: Typ der Werte der `Map`
  - `V` **`get`**`(Object key)`
    Liefert den Wert, der in der `Map` unter dem Schlüssel `key` hinterlegt ist (oder `null`, falls kein Wert vorhanden).
  - `int` **`size`**`()`
    Liefert die aktuelle Anzahl der Schlüssel/Wert-Paare, die im `Map`-Objekt gespeichert sind.

Die Klasse `java.util.HashMap` (1)

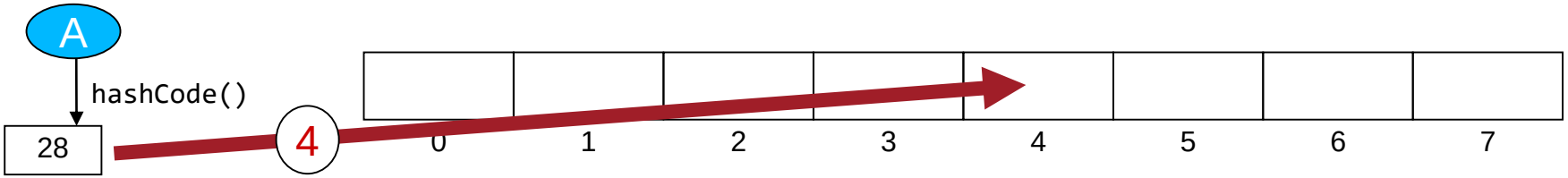
- Die Klasse **`HashMap`** ist eine Implementierung des Interfaces **`Map`**
- `HashMap` ist eine Klasse, die durch Verwendung des Hashing das Speichern und Wiederauffinden von Objekten ermöglicht. Mit Hashing wird ein Kompromiss zwischen verwendetem Speicherplatz und benötigter Zeit zum Wiederauffinden gespeicherter Elemente gemacht. `HashMap` ist von der abstrakten Klasse `AbstractMap` abgeleitet.
- Einer `HashMap` wird ein bestimmtes `Object` zusammen mit einem anderen `Object`, das als Schlüssel dient, hinzugefügt. Mit diesem Schlüssel kann man wieder auf das gespeicherte Objekt zugreifen.

Die Klasse `java.util.HashMap` (2)

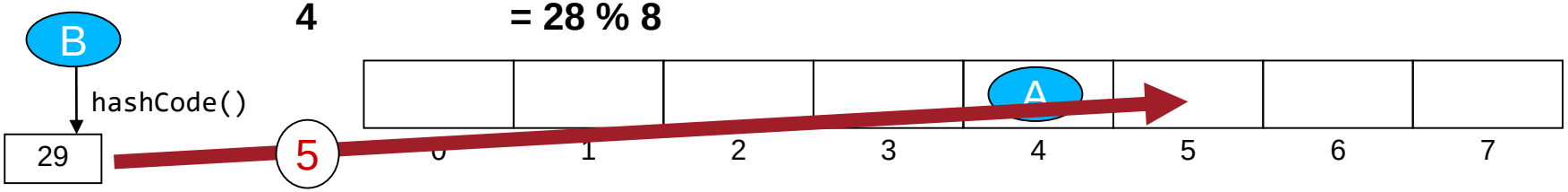
- **`hashCode()`** liefert das Ergebnis der Hash-Funktion eines jeden Objekts (definiert in `java.lang.Object`).
  Die Hash-Funktion bildet die Basis um Schlüssel-Objekte auf die Indexe der Hash-Tabelle (wie die eines Array) ab.
  - Muss innerhalb der Array-Grenzen liegen (z.B.: modulo)
  - ggf. Kollisionsauflösung, z.B. nächste freie Stelle suchen
- Einfügen eines neuen Objekts mit
  `public V put(K key, V value)*`
- Ist ein Schlüssel bereits in der `HashMap` vorhanden (errechneter Hash-Wert gleich), wird diesem Schlüssel der neue Wert zugeordnet. Der alte Wert wird überschrieben und von `put()` als Ergebnis zurückgegeben, ansonsten `null`.

* `K`: Typ der Schlüssel der `Map` / `V`: Typ der Werte der `Map`

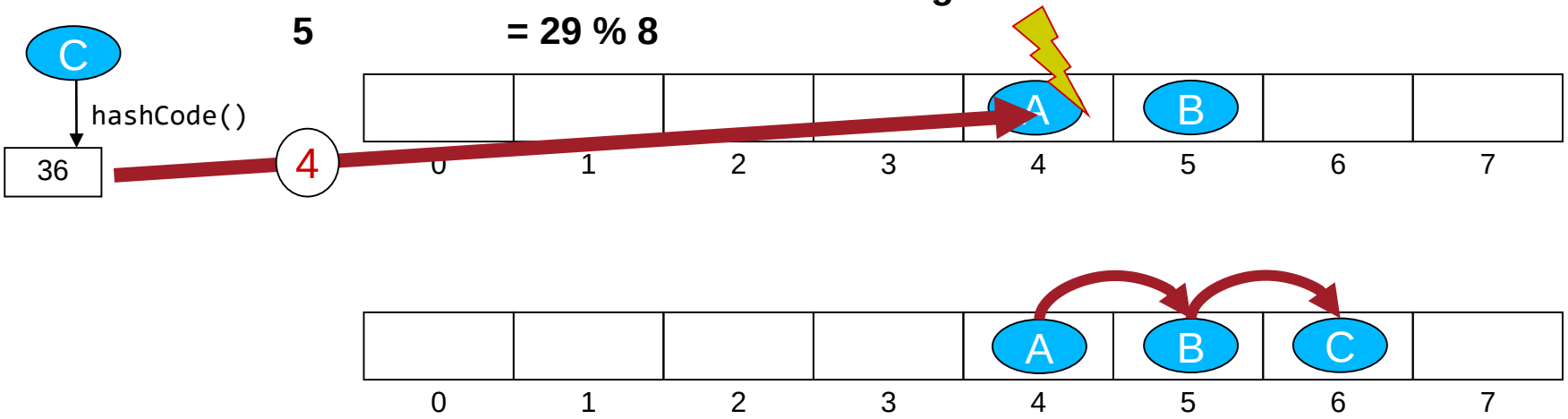
Mögliches Prinzip für Kollisions-Vermeidung



Hash-Index = Hashcode % Tabellengröße
4 = 28 % 8



Hash-Index = Hashcode % Tabellengröße
5 = 29 % 8



Hash-Index = Hashcode % Tabellengröße
4 = 36 % 8
Tatsächlicher Index durch Kollisionsauflösung: 6

Beispiel Map

```
import java.util.HashMap;
import java.util.Map;

public class MapExample {

    public static void main(String[] args) {
        Map<String, String> bdays = new HashMap<>();
        bdays.put( "Mia", "20.04.1992" );
        bdays.put( "Uli", "10.08.1993" );
        bdays.put( "Peter", "19.07.1994" );
        bdays.put( "Mia", "31.12.1991" );   // Duplikat überschreibt!
        for(String name : bdays.keySet()) { // Schlüssel als Set
            System.out.println(name + " born on " +
                               bdays.get(name)); // Schlüssel 'name'
        }
    }

}
```

Ausgabe

```
Uli born on 10.08.1993
Mia born on 31.12.1991
Peter born on 19.07.1994
```

Die Klasse `java.util.TreeMap`

- Die Klasse `TreeMap` ist eine Implementierung des Interfaces `Map` und `SortedMap`
- Die Werte werden bei einer `TreeMap` sortiert nach ihren Schlüsseln abgelegt
- Sämtliche Klassen deren Instanzen als Schlüssel verwendet werden müssen daher die Schnittstelle `Comparable` implementieren – außer es wird eine passende `Comparator`-Instanz beim Erzeugen der `TreeMap` übergeben
  ➔ Vergleichbarkeit der Schlüssel muss gewährleistet sein!

Weitere Methoden von des Interfaces `Map`

- `Object` **`remove`**`(Object key)`
  Entfernt Schlüssel und zugehörigen Wert
- `void` **`clear`**`()`
  Alle Schlüssel und Werte entfernen
- `boolean` **`containsKey`**`(Object key)`
  Prüft: Ist dieser Schlüssel enthalten?
- `boolean` **`containsValue`**`(Object value)`
  Prüft: Ist dieser Wert enthalten?
- `boolean` **`isEmpty`**`()`
  Testet, ob es (gar) keine Schlüssel gibt
- `Set` **`keySet`**`()`
  Alle Schlüssel als `Set`
- `Collection` **`values`**`()`
  Alle Werte als `Collection`

Überschreiben von `hashCode()`

- Die Methode `hashCode()` soll zu jedem Objekt eine möglichst eindeutige Integerzahl (`int`) liefern, die das Objekt identifiziert.
- Die Methode `hashCode()` ist eine Methode von `Object`.
- Sie wird von verschiedenen Klassen überschrieben bzw. soll überschrieben werden.

- Regel: Wenn man die `equals`-Methode überschreibt, wird man aufgefordert, auch die `hashCode`-Methode zu überschreiben (`"Generate missing hashCode()"`).
- Dabei soll gelten:
  Zwei Objekte, die laut `equals`-Methode gleich sind, sollen den gleichen Hashcode zurückliefern.

Beispiel: Überschreiben von hashCode()

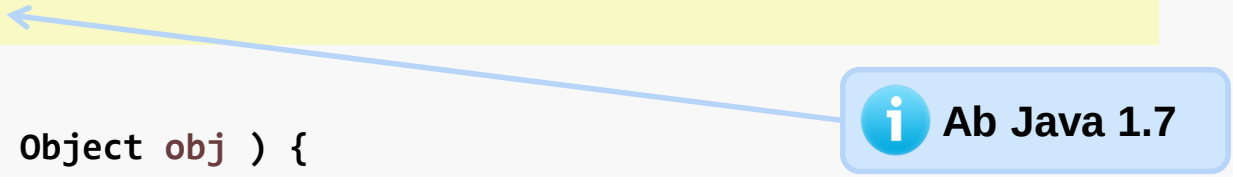
```
import java.util.Objects;
class Person {
   private String surname;
   private String name;
   private int age;

   public Person( String surname, String name, int age ) {
      this.surname = surname;
      this.name = name;
      this.age = age;
   }

   @Override
   public int hashCode() {
      return Objects.hash( this.age, this.name, this.surname );
   }

   @Override
   public boolean equals( Object obj ) {
      // ...
   }

   // Getter und Setter-Methoden
}
```

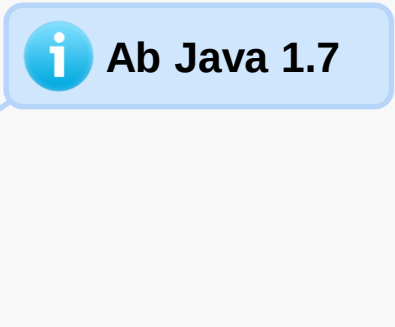


Beispiel: Überschreiben von `equals()`

```java
import java.util.Objects;
class Person {
   // ...

   @Override
   public boolean equals( Object obj ) {
      if ( this == obj ) {
         return true;
      }
      if ( obj == null ) {
         return false;
      }
      if ( this.getClass() != obj.getClass() ) {
         return false;
      }
      Person other = (Person) obj;
      return this.age == other.age && Objects.equals( this.name, other.name )
            && Objects.equals( this.surname, other.surname );
   }

   // ...
}
```



Die Klasse `java.util.Stack`

- Die Klasse `Stack` repräsentiert einen Stapelspeicher:
  - Neue Elemente werden immer „oben drauf“ gepackt
  - Es wird immer das oberste Element zuerst entfernt
- Beim Anlegen eines Stacks wird seine Größe mit „0“ initialisiert:

```
Stack<Integer> myStack = new Stack<>();
```

Methoden von `Stack`

- `boolean` **`empty`**`()`
  Testet, ob der Stack leer ist
- `Object` **`peek`**`()`
  Liefert das oberste Objekt ohne es zu löschen
- `Object` **`pop`**`()`
  Entfernt das oberste Objekt vom Stack und liefert es zurück
- `Object` **`push`**`(Object item)`
  Legt ein Objekt oben auf den Stack
- `int` **`search`**`(Object o)`
  Liefert die Position eines Objekts (von oben) auf dem Stack

Beispiel: Stack mit Zahlen

```
import java.util.Stack;

public class StackExample {

    public static void main(String[] args) {

        Stack<Integer> numbers = new Stack<>();
        numbers.push( 42 );
        numbers.push( 4711 );
        numbers.push( 999 );

        System.out.println("Count: " + numbers.size()); // Count
        System.out.println("Peek: " + numbers.peek());  // Peek Element

        do {
            System.out.println("Pop: " + numbers.pop());
        } while ( !numbers.empty() );

    }

}
```

Ausgabe First-in Last-out

```
Count: 3
Peek: 999
Pop: 999
Pop: 4711
Pop: 42
```

Das Interface `java.util.Enumeration` (1)

- `Enumeration` ist nicht mit obigen Datenstrukturen, die durch Klassen repräsentiert werden, vergleichbar, wird aber trotzdem an dieser Stelle aufgeführt, da sie ebenso Zugriff auf Daten bietet.
- `Enumeration` ist ein **Interface**, das dazu verwendet wird, eine Aufzählung von Elementen anzugeben. Dieses Interface wird z. B. von der Klasse `StringTokenizer` implementiert.

Das Interface `java.util.Enumeration` (2)

- Eine Klasse, die **`Enumeration<T>`** implementiert, muss die folgenden Methoden überschreiben:
  - `boolean hasMoreElements()`
    Liefert `true`, wenn die Aufzählung weitere Elemente enthält.
  - `<T> nextElement()`
    Liefert das nächste Objekt in der Aufzählung. Vorbedingung für diese Methode ist, dass der letzte Aufruf von `hasMoreElements()` das Ergebnis `true` hatte.

- Beispiel (Nutzung in Schleife):

```
while (d.hasMoreElements()) {
    System.out.println(d.nextElement());
}
```

Beispiel: Vorsicht bei remove() (1)

```java
import java.util.ArrayList;
import java.util.List;

public class ListRemove1 {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("This");
        list.add("is");
        list.add("an");
        list.add("example");
        list.remove("is"); // remove() vor der Schleife
        for (String o : list) {
            System.out.println(o);
        }
    }

}
```

Ausgabe

```
This
an
example
```

Beispiel: Vorsicht bei remove() (2)

```java
import java.util.ArrayList;
import java.util.List;

public class ListRemove2 {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("This");
        list.add("is");
        list.add("an");
        list.add("example");
        for (String o : list) {
            if ("is".equals(o)){
                list.remove(o); // Entfernen innerhalb der Schleife
            }
        }
    }

}
```

Ausgabe

```
Exception in thread "main" java.util.ConcurrentModificationException
        at java.util.ArrayList$Itr.checkForComodification(Unknown Source)
        at java.util.ArrayList$Itr.next(Unknown Source)
        at ListRemove2.main(ListRemove2.java:14)
```

Beispiel: Vorsicht bei remove() (3)

```
import java.util.ArrayList;
import java.util.List;

public class ListRemove3 {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("This");
        list.add("is");
        list.add("an");
        list.add("example");
        for (String o : list) {
            if ("is".equals(o)){
                list.remove(o); // Entfernen innerhalb der Schleife
                break; // nach remove!
            }
        }
        System.out.println("Element count: " + list.size());
    }
}
```

Ausgabe

```
Element count: 3
```

Beispiel: Vorsicht bei remove() (4)

```
import java.util.ArrayList;
import java.util.List;

public class ListRemove4 {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("This");
        list.add("is");
        list.add("an");
        list.add("example");
        for(int i=0; i<list.size(); i++) {
            if (list.get(i).equals("is")) {
                list.remove(i);   // i--;
            }
        }
        System.out.println("Element count: " + list.size());
    }

}
```

Ausgabe

```
Element count: 3
```

`Collection` mit `Iterator` „durchlaufen“

- Für Implementierungen des Interface `Collection` gibt es eine Möglichkeit, alle Elemente nacheinander zu durchlaufen.
- Diese Funktion wird vom Interface `Iterator` bereitgestellt. `Iterator` hat prinzipiell dieselbe Aufgabe wie die Klasse `Enumeration`, mit dem Unterschied, dass Elemente beim Durchlaufen auch gelöscht werden können.
- Hierfür stellt `Iterator` die Methode `remove()` zur Verfügung. Über die Methode `iterator()` kann ein `Iterator` für eine Kollektion ermittelt werden. `iterator()` ist im Interface `Collection` definiert.

Iterator benutzen

```
import java.util.Iterator;
import java.util.List;
import java.util.ArrayList;

public class ListExampleWithIterator {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Karl");
        list.add("Otto");
        list.add("Hans");
        Iterator<String> iter = list.iterator();
        for (; iter.hasNext();) {
            System.out.println(iter.next());
        }
    }

}
```

Ausgabe

```
Karl
Otto
Hans
```

`Iterator` benutzen ohne Interface Collection

- Bei Kollektionen, die nicht direkt das Interface `Collection` implementieren, wie z. B. die Klasse `TreeMap`, kann man über die Methode `values()` ein `Collection`-Exemplar ermitteln, das Zugriff auf die Original-Elemente bietet.
- Folgendes Beispiel zeigt, wie ein `Iterator` von der Klasse `TreeMap` ermittelt und bearbeitet wird:

```
TreeMap<String, String> tm = new TreeMap<>();
// Elemente der Kollektion hinzufügen
// ...
Collection<String> col = tm.values();
Iterator<String> iter = col.iterator();
// Elemente durchlaufen
while (iter.hasNext()) {
    String o = iter.next();
    // hier Bearbeiten der Elemente // ...
}
```

`Comparable` und `Comparator`

- Klassen, die **`SortedSet`** und **`SortedMap`** implementieren, bieten eine Sortierung für ihre Elemente an.
- Da prinzipiell beliebige Objekte in einer Kollektion gespeichert werden können, muss ein Sortierungskriterium bereitgestellt werden.
- Zu diesem Zweck werden die Interfaces **`Comparable`** und **`Comparator`** benutzt.
- Objekte, die in einer sortierten Kollektion gespeichert werden, müssen entweder das Interface **`Comparable`** implementieren oder die Kollektion muss einmalig mit einer Implementierung von **`Comparator`** initialisiert werden.

Comparable

- Das Interface **Comparable<T>** definiert eine Methode:

```
public int compareTo(T o)
```

(T durch konkrete Klasse ersetzen!)

- Diese Methode vergleicht das übergebene Exemplar mit dem Exemplar, in dem der Aufruf stattfindet. Der Rückgabewert der Methode sollte folgende Werte annehmen:
  - eine negative Zahl, falls das aufgerufene Objekte kleiner ist
  - 0, falls die Objekte gleich sind
  - eine positive Zahl, falls das aufgerufene Objekte größer ist

Comparator

- Da die Implementierung eines Interface nur möglich ist, wenn man Zugriff auf den Quellcode hat, wird im JDK zusätzlich das Interface **`Comparator`** definiert.
- **`Comparator<T>`** definiert die Methode (T durch konkrete Klasse ersetzen!)

```
public int compare(T o1, T o2)
```

- Diese Methode sollte prinzipiell dasselbe Verhalten wie `compareTo()` bei `Comparable` zeigen:
  - Negativer Rückgabewert, wenn o1 < o2
  - 0 bei Gleichheit
  - positiver Rückgabewert, wenn o1 > o2.
- Ein `Comparator` kann z. B. implementiert werden, wenn man keinen Zugriff auf den Quellcode von Klassen hat, sie aber dennoch in sortierten Kollektionen speichern möchte. Hierzu muss man eine Klasse definieren, die das Interface `Comparator` implementiert und in `compare()` die Vergleichsregeln implementieren.

Comparator in Collection nutzen

- Die Initialisierung einer konkreten Kollektion mit einem Comparator erfolgt bei den vordefinierten Klassen mit dem Konstruktor:

```
SortedMap<String, String> tm = new TreeMap<>(
                                    new StringLengthComparator() );
```

- Hier wird eine TreeMap (eine Implementierung von SortedMap) erzeugt und mit einem Comparator initialisiert.

- Beispiel:

```
import java.util.Comparator;

public class StringLengthComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
}
```

Anonyme Implementierungen (1)

- Über Factory-Methoden der Klasse **`java.util.Collections`** können anonyme Implementierungen von Kollektionen erzeugt werden.
- Das sind Implementierungen eines Collection-Interfaces, deren Klassendefinitionen nicht als `public` deklariert sind.
- Über diese Methoden ist es möglich, von einem Objekt einer beliebigen Klasse, die eines der Collection-Interfaces implementiert, eine Version zu erhalten, die unveränderbar bzw. synchronisiert ist.
- Hierfür stellt die Klasse Collections die Methoden **`synchronizedXXX()`** und **`unmodifiableXXX()`** zur Verfügung, wobei `XXX` für eines der Collection-Interfaces steht.
- Als Parameter bekommen die Methoden jeweils ein Exemplar einer vorhandenen Implementierung übergeben. Als Ergebnis wird die synchronisierte bzw. unveränderbare Implementierung des übergebenen Exemplars zurückgeliefert.

Anonyme Implementierungen (2)

- Um z. B. eine synchronisierte Implementierung einer `TreeMap` zu erhalten, müssen folgende Zeilen Code ausgeführt werden:

```
SortedMap<String, String> sm;
sm = java.util.Collections.synchronizedSortedMap(new TreeMap<>());
// Die Aufrufe an sm sind nun synchronisiert
```

- Die Methoden der Klasse `Collections` sind alle statisch.
- Die erweiterte Funktionalität wird bei dieser Vorgehensweise durch so genannte Wrapper-Implementierungen zur Verfügung gestellt.
  - Beim Aufruf von `synchronizedXXX()` bzw. `unmodifiableXXX()` wird zunächst ein Exemplar der passenden Wrapper-Implementierung erzeugt, das in einem Datenelement einen Verweis auf das übergebene Original-Exemplar speichert (oben ein Exemplar der Klasse `TreeMap`). Beim Aufruf der Methoden stellt nun der Wrapper zunächst die neue Funktionalität zur Verfügung (z. B. Synchronisation) und leitet danach den Aufruf an die Original-Implementierung weiter.

Die Klasse `Collections`: Algorithmen (1)

- In der Klasse `java.util.Collections` (nicht zu verwechseln mit dem Interface `Collection`) sind mehrere Algorithmen implementiert, die auf Kollektionen angewendet werden können.
- Einige dieser Algorithmen sind auf alle Kollektionen anwendbar, andere jedoch nur auf dem Interface `List`.
- Auf *alle* `Collections` anwendbare Methoden:
  - `static Object` **`max`**`(Collection col)`
    liefert das größte Element der `Collection col` zurück.
    Diese Methode kann jedoch nur auf Kollektionen angewendet werden, deren Elemente das *Interface* `Comparable` *implementieren*.
  - `static Object` **`max`**`(Collection col, Comparator comp)`
    Diese zweite Variante verfügt über einen zweiten Parameter vom Typ `Comparator`.
    Bei dieser Methode kann der *Vergleichsoperator explizit angegeben* werden.
    Die Elemente müssen in diesem Fall nicht `Comparable` implementieren.

Die Klasse `Collections`: Algorithmen (2)

- `static Object` **`min`**`(Collection col)`
  arbeitet wie `max()`, liefert jedoch das kleinste Element zurück.
  Auch von `min()` gibt es zwei Varianten.

- Nur auf *Implementierungen von* `List` anwendbare Methoden:
  - `static void` **`sort`**`(List l)`
    sortiert die Elemente in der Liste `l` aufsteigend.
    Diese Variante setzt voraus, dass die Klassen aller enthaltenen Objekte das Interface `Comparable` implementieren.
  - `static void` **`sort`**`(List l, Comparator comp)`
    Dieser zweiten Variante von `sort` wird eine `Comparator`-Instanz als zweites Argument übergeben.
    Sie hat nicht die o.g. Voraussetzung.
  - `static void` **`reverse`**`(List l)`
    ordnet die Elemente der Liste `l` so um, dass sie in umgekehrter Reihenfolge stehen.

Die Klasse `Collections`: Algorithmen (3)

- `static void` **`shuffle`**`(List l)`
  permutiert die Elemente in der Liste `l` zufällig.
- `static int` **`binarySearch`**`(List list, Object key)`
  sucht in der aufsteigend sortierten Liste `list` das Element `key` über die binäre Suche.
  Wird `key` in der Liste gefunden, liefert `binarySearch()` die Index-Position des Elements zurück.
  Wird `key` nicht gefunden, liefert die Methode den negativen Wert der potenziellen Einfügeposition minus 1 (also immer negative Werte, wenn das Objekt nicht gefunden wird).

- Alle Algorithmen in der Klasse `Collections` sind mit statischen Methoden realisiert.

Beispiel: Sortieren einer Liste

```
import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class ListExampleSort {

    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("Karl");
        list.add("Otto");
        list.add("Hans");

        Collections.sort(list);

        for (String name : list) {
           System.out.println(name);
        }
    }

}
```

Ausgabe

```
Hans
Karl
Otto
```

`myClass<T>`

EIGENE GENERISCHE KLASSEN ERSTELLEN

Generics

- Generics bzw. Generizität und generische Elemente sind ein Sprachmittel von Programmiersprachen. Im Falle von Java heißt dies, dass Klassen und Methoden (Methoden auch außerhalb von parametrisierten Klassen), aber auch Schnittstellen mit **Typparametern** parametrisiert werden können.
- Es geht also u. a. um Konstrukte wie z.B. eine „`List` of `String`", also eine Liste, die nur String-Objekte aufnehmen kann.
- Aus anderen Programmiersprachen sind vergleichbare Konzepte bekannt, teilweise unter anderen Namen. In C++ gibt es die Template-Klassen, also Schablonenklassen. Auch wenn diese syntaktisch einige Ähnlichkeit zu den Generics in Java haben, arbeiten diese völlig anders.
- Einer der Hauptgründe für die Einführung von Generics mit der Java-Version 1.5 war der Wunsch, die existierenden Containerklassen wie z. B. `List`, `Map`, `Set` um Typsicherheit zu erweitern.

Unsere erste generische Klasse (1)

```
public class MyLinkedList<T> {
    private T value;
    private MyLinkedList<T> next;

    public MyLinkedList() {
    }

    public boolean isEmpty() {
        return this.value == null;
    }

    public void add(T value) {
        if (this.isEmpty()) {   // letztes Element?
            this.value = value; // neues, leeres Element anhängen
            this.next = new MyLinkedList<T>();
        } else {
            this.next.add(value);
        }
    }
    // ... weiter auf der nächsten Folie
```

Unsere erste generische Klasse (2)

```
    public T get(int index) {
        if (index < 0 || this.isEmpty()) {
            throw new IndexOutOfBoundsException();
        }
        return  index == 0 ? this.value : this.next.get(index-1);
    }

    public int size() {
        return this.isEmpty() ? 0 : (this.next.size()+1);
    }
}

public class MyLinkedListTest {
    public static void main(String[] args) {
        MyLinkedList<Integer> intList = new MyLinkedList<>();
        intList.add(new Integer(17));
        Integer i = intList.get(0);
        System.out.println("i = " + i);
    }
}
```

Invarianz

- Folgende Zuweisung ist nicht erlaubt:

```
MyLinkedList<Integer> intList = new MyLinkedList<Integer>();
MyLinkedList<Number> numList = intList; // error
```

- In Bezug auf den Typparameter verhält sich die Zuweisungskompatibilität bei den Generics in Java also invariant.
- Hierfür gibt es jedoch auch einen guten Grund. Einmal angenommen, die obige Zuweisung wäre doch erlaubt. Dann wäre folgender Code ebenfalls erlaubt:

```
MyLinkedList<Integer> intList = new MyLinkedList<Integer>();
MyLinkedList<Number> numList = intList; // angenommen (!) erlaubt
numList.add(new Double(3.14));          // erlaubt
```

- Der Grund für die Invarianz liegt also hier in der Veränderlichkeit der Containerobjekte.

Wildcards (1)

- Zunächst einmal kann ein Zuweisungsproblem durch Benutzung der so genannten Wildcards umgangen werden.
  Hierbei gibt man explizit an, dass man nicht genau weiß, welcher Typ als Typparameter verwendet wurde:

```
List<Integer> intList = new ArrayList<Integer>();
List<?> list = intList;
list.add(new Double(3.14)); // nicht erlaubt
list.add(new Integer(1));   // nicht erlaubt
list.add(new Object());     // nicht erlaubt
list.add(null);             // erlaubt
```

- Die logische Folge ist, dass nun keine Objekte mehr über die neue Referenz eingefügt werden können. Da für den Compiler nicht ersichtlich ist, mit welchem Typ list parametrisiert wurde, kann er für die `add()`-Methode nicht den Typ des nötigen Parameters bestimmen. Das einzige, was er zulassen kann, ist der Wert `null`.

Wildcards (2)

- Ähnlich verhält es sich beim Auslesen von Objekten per `get()`-Methode:

```
List<Integer> intList = new ArrayList<Integer>();
intList.add(new Integer(42));
List<?> list = intList;
Integer i = list.get(0);      // nicht erlaubt
Object o = list.get(0);       // erlaubt
```

- Der Compiler kann nichts über die konkreten Objekte, die von `get()` geliefert werden, garantieren, außer dass sie zuweisungskompatibel zu `Object` sind (dies gilt für jedes konkrete Objekt).

Kovarianz und Kontravarianz (1)

- Die durch die Verwendung von Wildcards gewonnene Flexibilität ist natürlich nicht sehr befriedigend. Daher kann man die Wildcards im Klassenbaum nach unten (Kovarianz) oder nach oben (Kontravarianz) einschränken. Die Schlüsselwörter hierfür sind `extends` bzw. `super`:

```
List<Integer> intList = new ArrayList<Integer>();
intList.add(new Integer(42));
List<? extends Number> list = intList;

Integer i = list.get(0); // nicht erlaubt
Number n = list.get(0);  // erlaubt
list.add(n);             // nicht erlaubt
list.add(i);             // nicht erlaubt
list.add(null);          // erlaubt
```

- In Zeile 3 wird die Variable `list` definiert und für den Typparameter festgelegt, dass dieser `Number` oder aber eine Unterklasse von `Number` sein muss. Diese Bedingung erfüllt das Objekt, das von `intList` referenziert wird, so dass die Zuweisung hier erlaubt ist.

Kovarianz und Kontravarianz (2)

- Das Schlüsselwort `super` hat umgekehrte Auswirkungen:

```
List<Number> numList = new ArrayList<Number>();
numList.add(new Integer(42));
List<? super Integer> list = numList;

Integer i = list.get(0);   // nicht erlaubt
Number num = list.get(0);  // nicht erlaubt
Object obj = list.get(0);  // erlaubt

list.add(obj);             // nicht erlaubt
list.add(num);             // nicht erlaubt
list.add(i);               // erlaubt
```

- In Zeile 1 wird dieses Mal eine Liste erzeugt, die mit `Number` parametrisiert ist und somit neben `Integer`-Objekten auch `Double`-, `Long`- und andere Objekte akzeptiert. Beispielhaft wird in Zeile 2 ein `Integer`-Objekt eingefügt.

Kovarianz und Kontravarianz (3)

- In Zeile 3 wird nun eine Variable definiert, die auf eine Liste referenziert, die mit dem Typ `Integer` oder einer Oberklasse davon (also bis einschließlich `Object`) parametrisiert wurde. Diese Bedingung erfüllt das Objekt, das von `numList` referenziert wird, so dass die Zuweisung erlaubt ist.
- Nun kann beim Auslesen von Objekten über deren konkreten Typ natürlich nichts zugesichert werden, außer dass sie zuweisungskompatibel zu `Object` sind (dies wäre ja der "schlimmste" Fall, den man annehmen könnte). Daher liefert die `get()`-Methode auch nur eine `Object`-Referenz (Zeile 7).
- Beim Hinzufügen von Elementen verhält es sich umgekehrt: Es werden nur noch `Integer`-Referenzen akzeptiert (theoretisch auch Unterklassen von `Integer`), da dies wiederum der speziellste Typ ist, der hätte verwendet werden können. Ließe man allgemeinere Typen zu, bestünde die Möglichkeit, dass ja doch `Integer` als Typparameter verwendet wurde, und somit das Typsystem umgangen wäre.