Skript
Informatik B

Objektorientierte Programmierung in Java

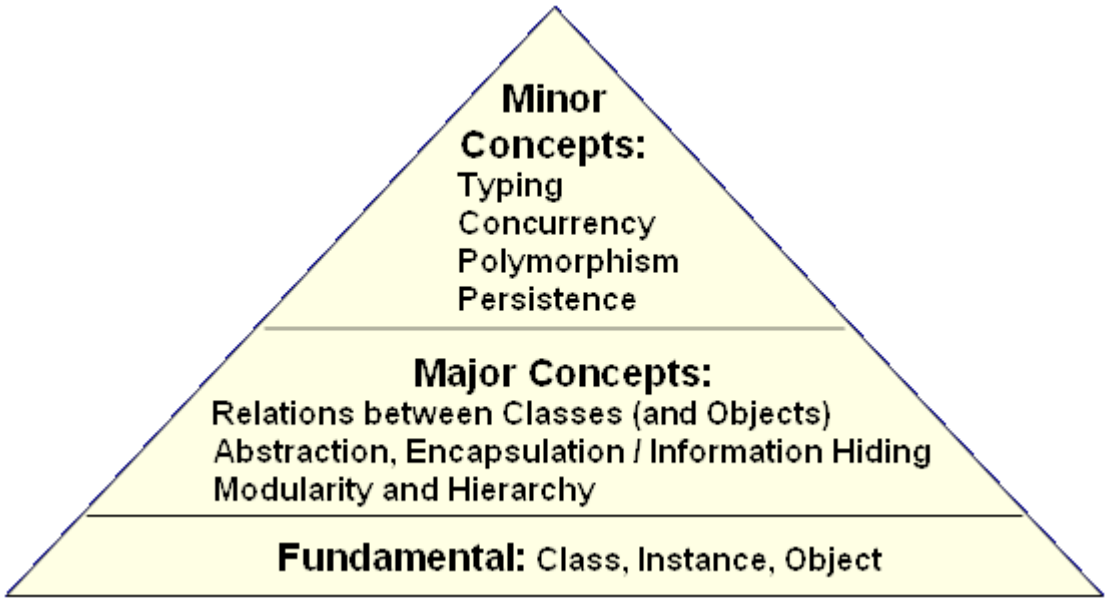
Sommersemester 2011

- Teil 5 -

Inhalt

… wird schrittweise erweitert

Kapitel 5:
Generizität (Generics)



[Boo94]

Vorbemerkung:
Generizität kann als Konzept sowohl unter dem Thema *Abstraktion* wie auch unter dem Thema *Typsysteme* eingeordnet werden.

5.1 Nutzen und mögliche Alternativen

Beispiel in Java: Wir wollen eine Box, in die verschiedenartigste Objekte abgelegt (und verwaltet) werden können. Die Implementierung(slogik) ist weitgehend unabhängig vom Typ des verwalteten Objekts.
Folgende Realisierung in Java führt zu drei sehr ähnlichen Klassen:

```
public class StudentBox {
 private Student obj;

 public void add(Student o) {
        this.obj = o;
 }
 public Student get() {
        return obj;
 }
}
```

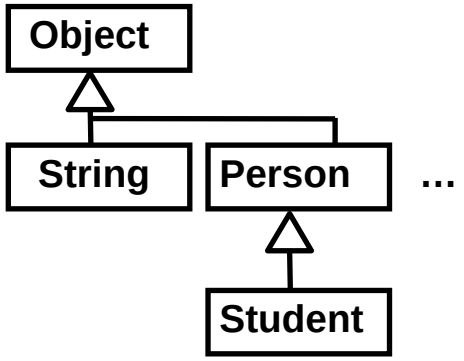
```
public class StringBox {
 private String obj;

 public void add(String o) {
        this.obj = o;
 }
 public String get() {
        return obj;
 }
}
```

```
public class IntegerBox {
 private Integer obj;

 public void add(Integer o) {
        this.obj = o;
 }
 public Integer get() {
        return obj;
 }
}
```

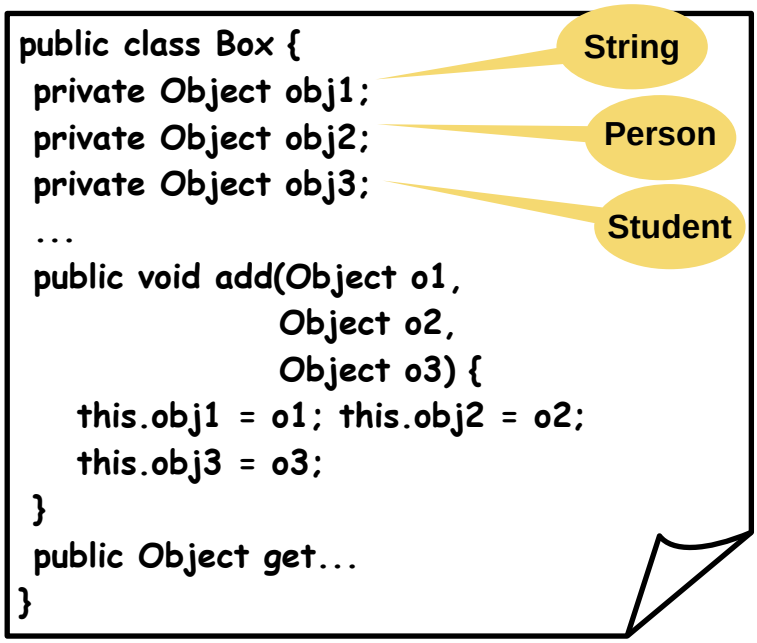
Erster Versuch zur Verbesserung: Nutzung des Substitutionsprinzips

Wir wollen Redundanz bei der Box vermeiden: Wir bilden eine Klasse, die beliebige Objekte verwaltet $\Rightarrow$ Wiederverwendung
Wir erweitern das Beispiel gleichzeitig: Es sollen ein paar mehr Objekte in die Box.

```
public class Box {
 private Object obj1;
 private Object obj2;
 private Object obj3;
 ...
 public void add(Object o1,
                 Object o2,
                 Object o3) {
    this.obj1 = o1;
    this.obj2 = o2;
    this.obj3 = o3;
 }

 public Object get...
}
```



Problem mit dem ersten Verbesserungsversuch:



Wir erhalten eine „Mischwaren"-Box: Wir wissen nicht was wirklich enthalten ist
⇒ zur Verwendung sind Typecasts notwendig

Beispiel für den Zugriff auf die Elemente in diesem Versuch:

```
Object p = aBox.getObj1();
p.getMatNr();
```

⇒ Compilerfehler (die Typprüfung ergibt, dass die Klasse `Object` keine Methode `getMatNr()` hat)

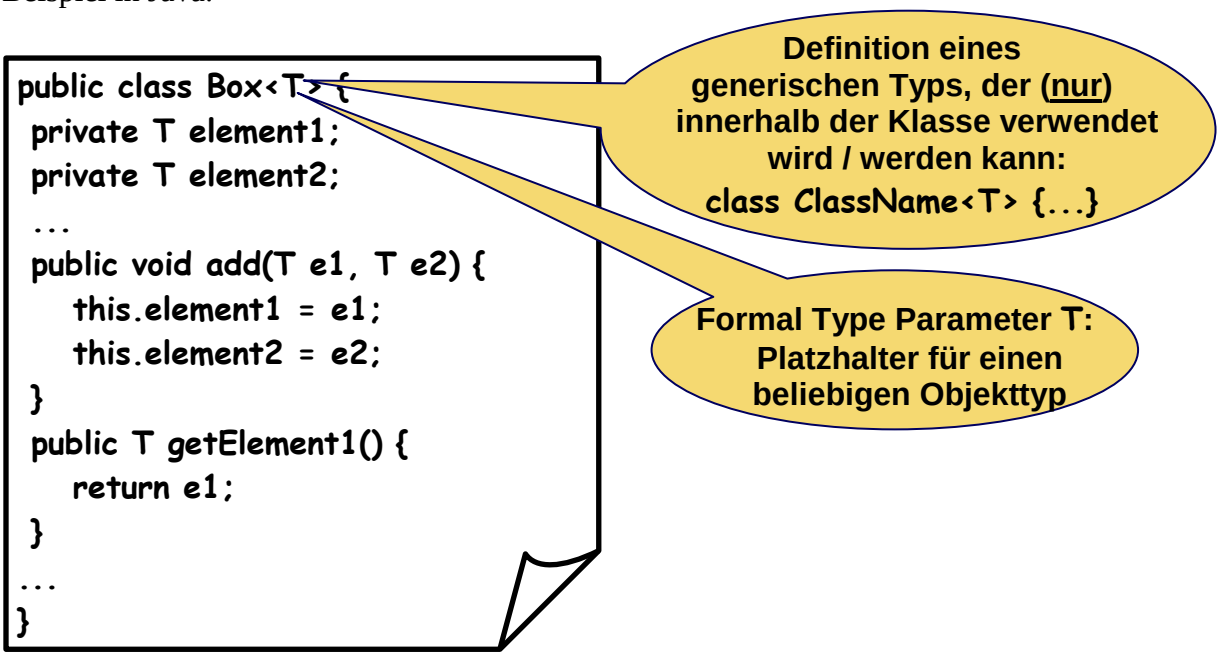
```
Student p = (Student)aBox.getObj1();
p.getMatNr();
```

⇒ ClassCastException

Unser Versuch hat zu unschönen Casts und zu keiner typsicheren Lösung geführt. Wir wollen zusätzlich mehr Typkontrolle bzw. Typsicherheit.

Zweiter Versuch zur Verbesserung: Einsatz von Generizität mit mehr Typsicherheit

Beispiel in Java:



Auch Interfaces können mit Generizität erstellt werden:

```
interface InterfaceName<T> {...}
```

Typsichere Erzeugung von Box-Instanzen zu unserer Box<T>-Klasse:

```
Box<String> b = new Box<String>();
```

Es gilt:

- Eine parametrisierte/parametrisierbare Klasse ist eine Klasse mit einem oder mehreren generischen formalen Parametern (nicht leere Parameterliste).
- Achtung variierende Begriffsverwendung: Parametrisierbare, parametrisierte Klasse, Bound Element, Template-Klasse, Template
- Eine parametrisierbare Klasse definiert eine Familie von Klassen: Sie ist eine mit generischen formalen Parametern versehene Schablone, mit der gewöhnliche (d.h. nicht-generische) Klassen erzeugt werden können.
- Interna der Klasse (z.B. Attribute) können von den Parametern abhängig definiert sein.
- Generische Typen gibt es auch in Java (seit Java Version 1.5).
  Generizität in Java heißt Generics.
  Achtung: In Java nicht für primitive Datentypen!
- Generische Typen werden auch in C++ und Eiffel unterstützt.

Zusammenfassend:

- Generizität bedeutet: Typunabhängigkeit.
- Es spielt keine Rolle, welche Arten von Objekten bearbeitet werden, solange das passende Interface implementiert ist.
- Parametrisierte Typen: Platzhalter, mit denen erst zu einem späteren Zeitpunkt festgelegt wird, für welchen Typ sie stehen.

5.2 Umsetzung

Hat der Entwickler generische Typen verwendet, müssen Compiler und Laufzeitumgebung damit umgehen. Generische Typen können für die Abarbeitung grundsätzlich auf zwei Arten umgesetzt werden:

(1) Heterogene Realisierung:
Für jeden aktuellen Typparameter wird individueller Code erzeugt (z.B. je eine eigene Klasse für `StringBox`, `StudentBox`, `IntegerBox`).

(2) Homogene Realisierung:
Für jede parametrisierte Klasse wird genau eine Klasse erzeugt.
Statt des generischen Typs wird der Typ `Object` eingesetzt (Löschen der Typinformationen: *type erasure*).
Für jeden konkreten Typ werden Typanpassungen in die Anweisungen eingesetzt.

In Java: Homogene Realisierung (der Compiler löscht bei generischen Typen alle Typinformationen).

Die technische Umsetzung geschieht meist mit einer Makrotechnik (Textersetzung).

5.3 Erweiterungen

5.3.1 Mehrere Typparameter

Verwendung mehrerer Typparameter (am Beispiel der Programmiersprache Java):

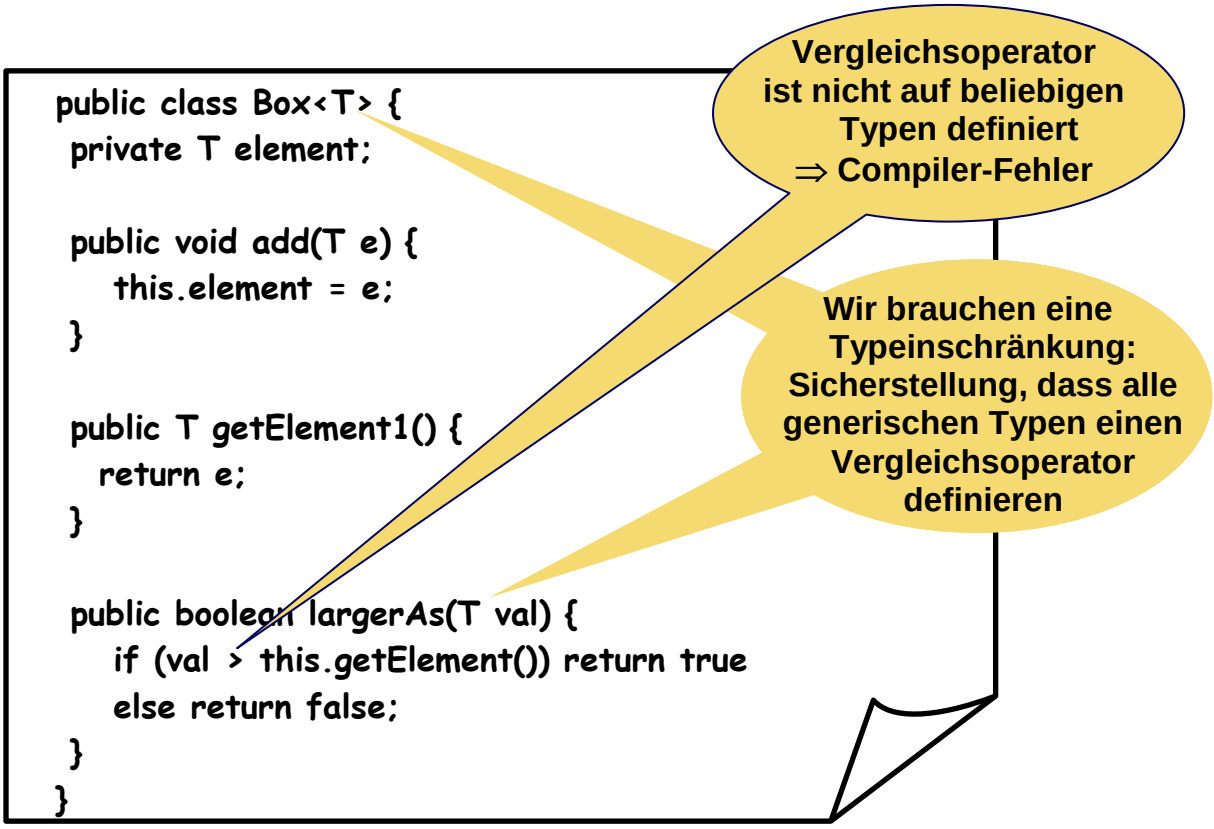
```
public class Box<T1,T2> {
 private T1 element1;
 private T2 element2;
 ...
 public void add(T1 e1, T2 e2) {
    this.element1 = e1;
    this.element2 = e2;
 }
 public T1 getElement1() {
    return element1;
 }
...
}
```

Erzeugung von Instanzen von Klassen, die mit mehreren Typparametern definiert sind:

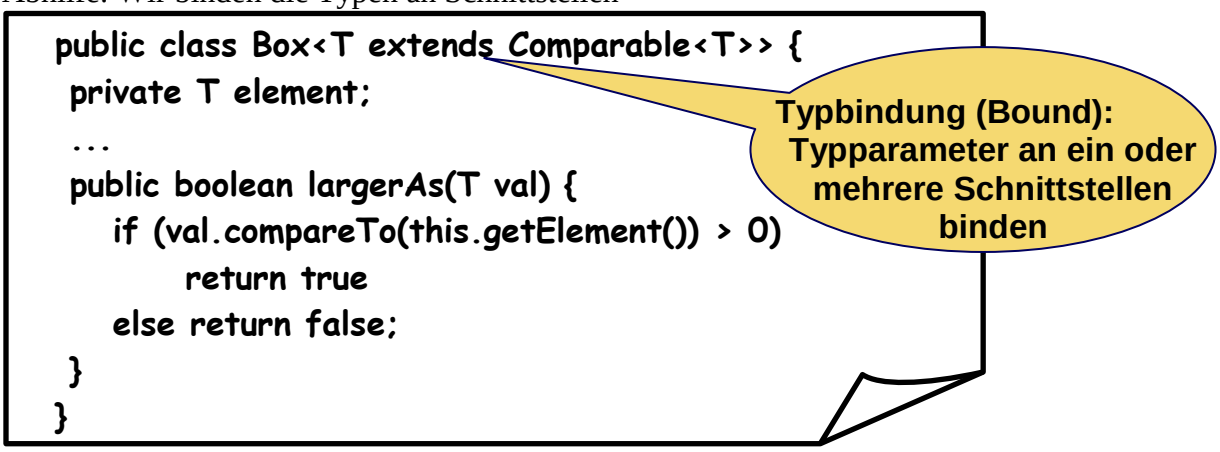
```
Box<String,Student> b = new Box<String,Student>();
```

5.3.2 Typeinschränkungen: Bounds

Problem am Beispiel in Java: Arbeiten mit generischen Typen und Typsicherheit



Abhilfe: Wir binden die Typen an Schnittstellen



Erzeugung: `Box<Student> b = new Box<Student>();`

Voraussetzung: Die Klasse `Student` muss das Interface `Comparable` implementieren (und damit die Methode `compareTo(T o)`).

Allgemeine Form in Java: Mehrfache Einschränkung des Typparameters durch Klassen und/oder Interfaces:

Voraussetzung:

- Der unbekannte Typ muss von der *Bound* (im Beispiel oben z.B. von `T1`, `I2`, `I3`, …) abgeleitet sein und/oder diese implementieren.
- Der Compiler prüft dann, ob der konkrete Typ, der für den Typparameter später eingesetzt wird, zu den angegebenen *Bounds* passt.

Beispiel:

```
public class Box<T extends Comparable<T> & Serializable>
                 implements Serializable
{   ...
}
```

Es können zwei Arten von Typeinschränkungen (Bounds) unterschieden werden:

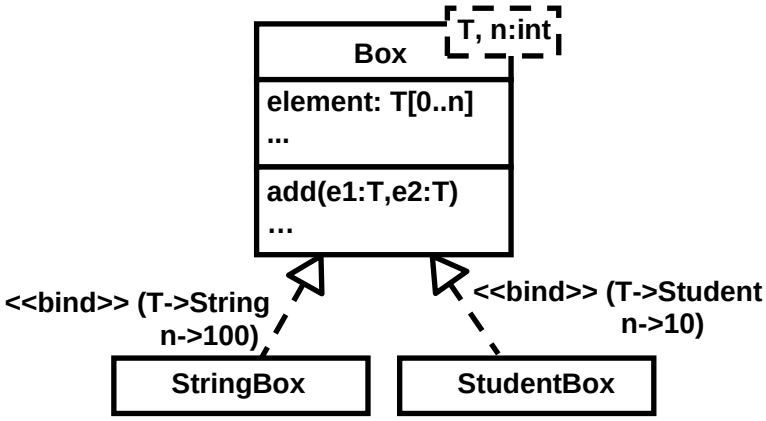
1. Upper Bounds (in Java durch Verwendung von „`extends`“): Der Typ muss der angegebenen *Bound* entsprechen oder eine Subklasse davon sein. Die *Bound* ist also in der Klassenhierarchie eine Beschränkung nach oben.
2. Lower Bounds: Der Typ muss der angegebenen *Bound* entsprechen oder eine Superklasse dazu sein.

   Beispiel in Java: `Box<? super Person>`

   In diesem Java-Beispiel wäre an der Stelle des `?` der Typ `Person` oder die Superklasse `Object` möglich.

5.4 Modellierung in UML

Beispiel: Darstellung der generischen Box in UML

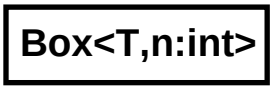


Parametrisierte Klasse
(*Parameterized Class*, *Template*)
mit zwei formalen Parametern
(*Template Parameter*)

```
public class Box<T,int> {
  // ...
}
```

Schablonenbindungsbeziehung

Schablonenklasse in Kurzform: (Notationsvariante)



Mit Bindung:

Anonym gebundene Klassen:
In der Notationsvariante haben wir im Gegensatz zur ersten Darstellungsvariante eine anonym gebundene Klasse: Die neue `Box`-Klasse, die nach der Parametrisierung entsteht, hat keinen Namen. In der ersten Darstellung hatten wir eigene Namen für die Klassen (z.B. `StringBox`).

Nutzung von generischen Typen in unserem Beispiel: Bindung aktueller Typparameter an formale Parameter:

```
Box<String,100> stringBox;
Box<Student,10> studentBox;
```

Das Ergebnis der Bindung ist ein *Bound Element* bzw. eine *Parameterized Class*.

Variable `stringBox` und `studentBox` haben passend gewählte Variablennamen. Der Typ mit dem sie deklariert sind, ist jeweils aber ohne eigenen Namen.

Bemerkung:
- In Java gibt es nur anonym gebundene Klassen.
- Java unterstützt keine primitiven Datentypen als Typparameter bei Generics.
  Der Typparameter `int`, der im Beispiel oben verwendet wird, funktioniert daher in Java nicht!

.

5.5 Generizität bei Objekt- und Klassenmethoden

Bei der Generizität an Objekt- und Klassenmethoden wird der generische Typ an der Methode (oder am Konstruktor) anstelle an der Klassendefinition verwendet.

Beispiel in Java:

```
public class MyUtil {

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

}
```

Klassenmethode auf beliebigem Typ

Eine Nutzung ohne Typangabe ist möglich.
Vorteil einer parametrisierten Methode: Typprüfung.

Nutzung im Beispiel:

```
String s1 = MyUtil.random("Essen","Schlafen");
String s2 = MyUtil.<String>random("Essen","Schalfen");

String s3 = MyUtil.random(1,"Schlafen");
```

korrekt

falsch
(statisch erkannt)

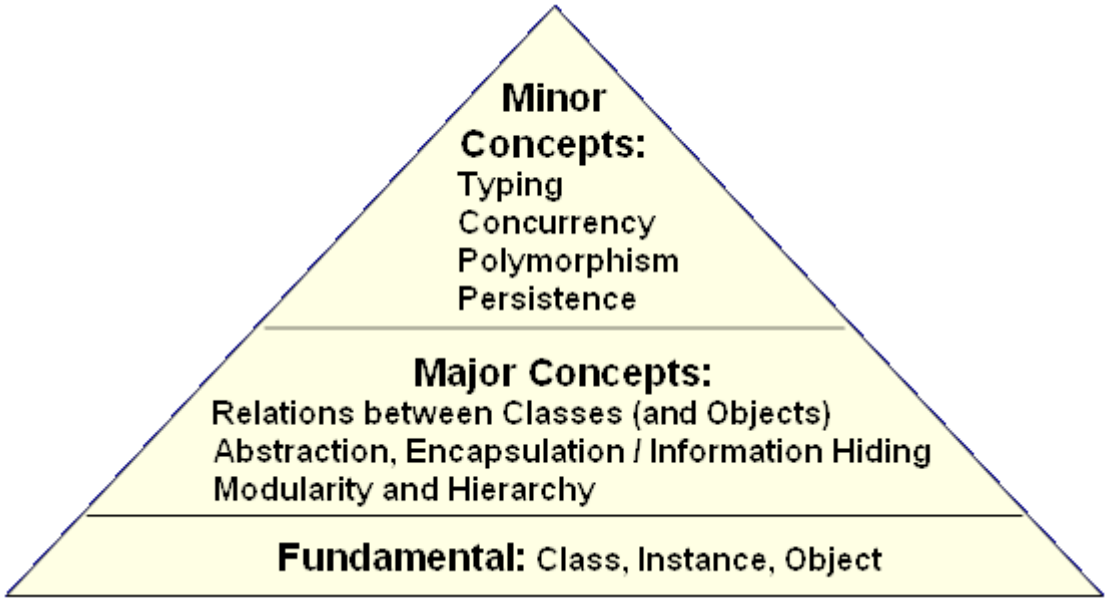
Geschachtelte generische Typen:

```
public static <U> void fillBoxes(U u, List<Box<U>> boxes) {
  for (Box<U> box : boxes) {
      box.add(u);
  }
}
```

Nutzung ohne konkrete Typangabe, aber mit Typprüfung (Konsistenz der Parameter)

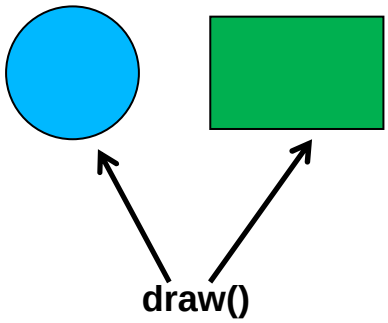
```
Crayon red = ...;
List<Box<Crayon>> crayonBoxes = ...;
Box.<Crayon>fillBoxes(red,crayonBoxes);
```

Kapitel 6:
Polymorphie / Polymorphismus



[Boo94]

- Polymorph = Vielgestaltigkeit (griech.):
  Eigenschaft [eines Bezeichners], sich in Abhängigkeit von der Umgebung, in der er verwendet wird, unterschiedlich darzustellen (Gegenteil: Monomorphie).
- In der Programmierung z.B.:
  - Eine Variable bzw. eine Methode kann gleichzeitig mehrere Typen haben.
  - Verschiedene Objekte können auf die gleiche Nachricht unterschiedlich reagieren.

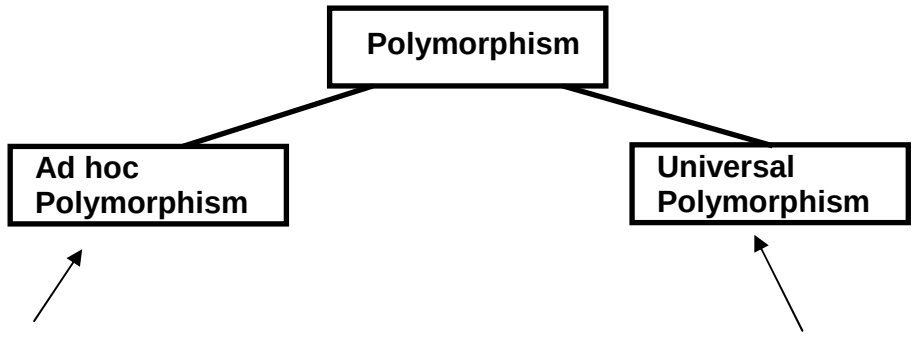


Beispiel:
Wir können die Nachricht *draw* an die jeweilige Form schicken (d.h. die gleiche Nachricht).
Das jeweilige Form-Objekt reagiert dabei unterschiedlich (zeichnet sich verschieden).

Ausgehend von dieser grundsätzlichen Bedeutung von Polymorphie, können verschiedene konkrete Umsetzungen in der Programmwelt klassifiziert werden.

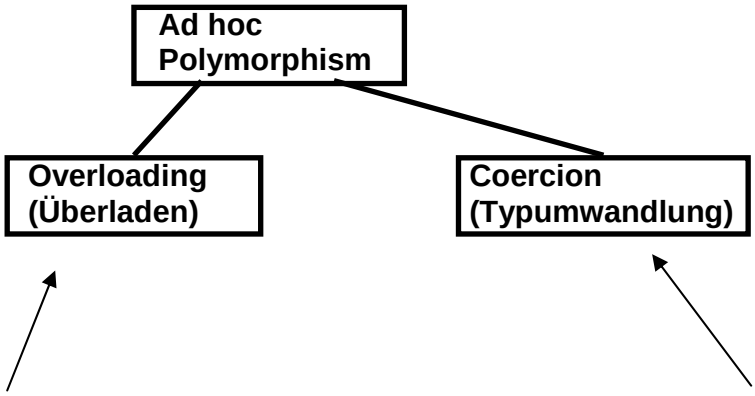
Arten von Polymorphie:

Polymorphie taucht in der Programmierung an verschiedenen Stellen auf.
Eine Orientierung zur Unterscheidung der Polymorphie-Arten gibt Strachey 1967 und Cardelli / Wegener, 1985.



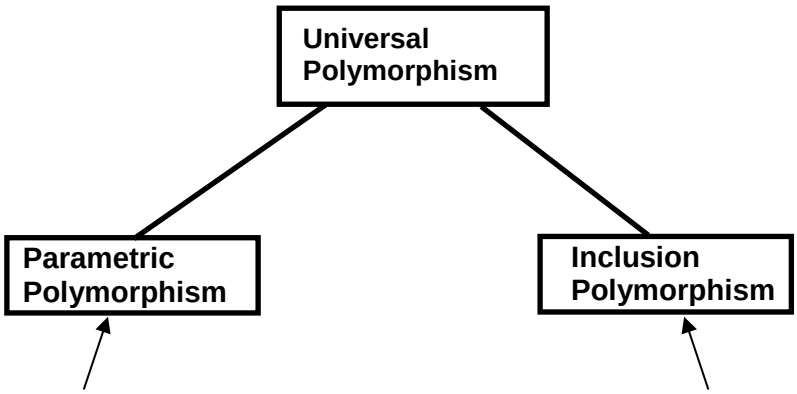
Ad hoc Polymorphie: Prinzipiell nur scheinbar polymorph.
Die Einheit hat tatsächlich nur einen Typ.
Die Syntax spielt polymorphes Verhalten vor.

Universelle Polymorphie: „Echte“ Polymorphie
Die gleiche Arbeit wird auf vielen verschiedenen Typen ausgeführt.



Verschiedene Methoden mit gleichem Namen, die auf verschiedenen Mengen von Parametertypen operieren.

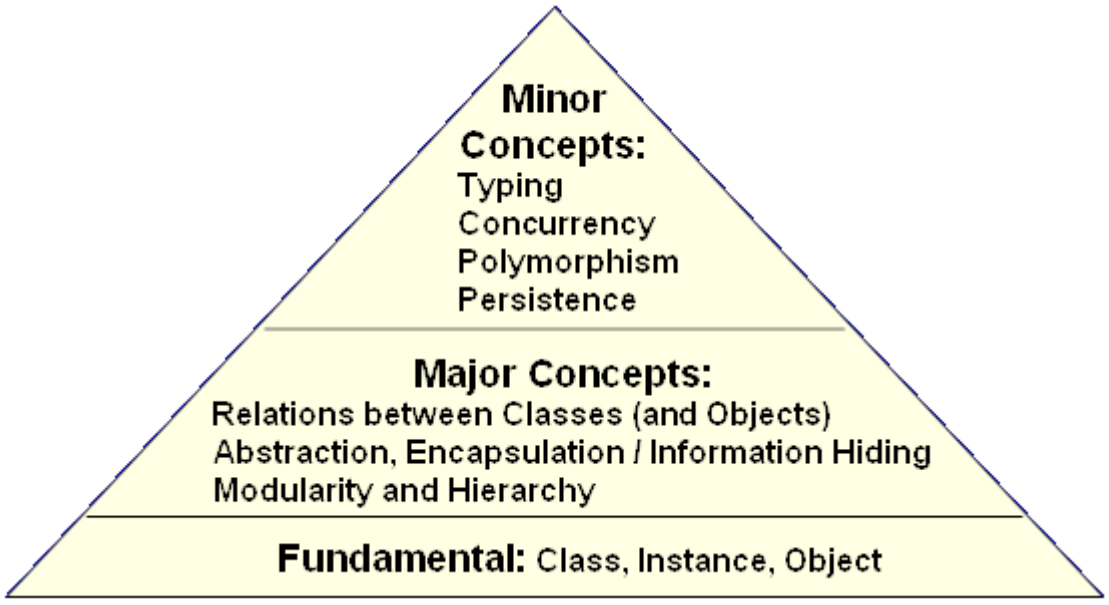
Implizite Typumwandlung von Werten; An die Operation (z.B. +) übergebene Typen werden in den passenden Typ (z.B. int) umgewandelt.



Der Parameter für den tatsächlich verwendeten Datentyp wird übergeben (generische Programmierung, z.B. Generics in Java).

1. Einer Variablen können Objekte unterschiedlichen Typs zugewiesen werden (der Typ beschreibt nur die Schnittstelle): Substitutionsprinzip.
2. Beim Aufruf einer Operation wird abhängig vom Objekt die entsprechende Methode ausgewählt (Überschreiben mit dynamischem Binden).

Kapitel 7: Klassenbibliothek (Java Collection Framework)



[Boo94]

Vorbemerkung:
Klassenbibliotheken setzen wie z.B. Pakete die Konzepte Abstraktion, Kapselung und Modularisierung um.
Sie gliedern sich demnach am besten in die wichtigen objektorientierten Konzepte ein.
Da wir für Klassenbibliotheken Generizität einsetzen, behandeln wir die Bibliotheken erst jetzt in einem eigenen Kapitel.

7.1 Klassenbibliotheken allgemein

- **Situation:** Immer wieder auftretende gleiche / ähnliche Problemstellungen und Aufgaben.
- **Ziel:** Wiederverwendung statt Neuimplementierung.

⇒ Spart Erarbeitungszeit.
⇒ Erleichtert die Benutzung durch Vereinheitlichung. (Die Abstraktion von der Plattform bzw. von der konkreten Implementierung verhindert plattformspezifische oder implementierungsspezifische Benutzungsvarianzen.)
⇒ Verbessert die Qualität und Performance durch vielfache Bewährung und Verbesserung.

Es gibt eine Vielzahl von Bibliotheken. Jede wichtigere Programmiersprache besitzt sie in der Regel in mehr oder weniger umfangreicher Ausführung.
Sie sind entweder in die Programmiersprache bzw. die Programmierumgebung integriert oder aber werden von Dritten als Zusatz zur Nutzung bereitgestellt.

Java bietet mit seinen Paketen eine Standardbibliothek (Funktionsbibliothek) an (z.B. java.io).
Umfang der Java-Bibliothek ([JavaInsel09] zu Java Version 6):
Mehr als 200 Pakete.
Im Vergleich: Java 1.0: 8 Paketen.

Die wichtigsten Java-Pakete sind:

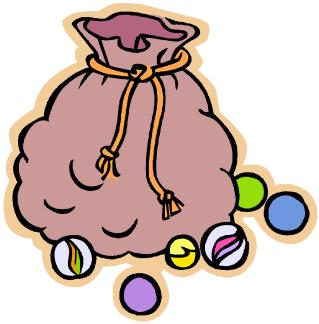
- Sprachbasis: java.lang (wird automatisch eingebunden, unverzichtbar)
- Hilfspakete: java.text, java.util
- Ein-/Ausgabe: java.io
- GUI Programmierung (Grafik): java.awt, java.swing
- Netzwerkprogrammierung: java.net

Hinzu kommen die Klassen für die Laufzeitumgebung (über 3000 Klassendateien für die Oracle/Sun Laufzeitumgebung in den Paketen sun und sunw).

Einige Pakete (z.B. javax.xml.parsers) liegen im übergeordneten Paket javax.
Ursprünglich war das Paket javax für Kern-Klassen ergänzende Erweiterungspakete (extensions) angelegt. Viele davon sind im Laufe der Zeit in die Standard-Distribution gewandert. Eine „Extension“ ist es heute also nicht mehr wirklich.

7.2 Java Collection Framework (JCF)

Das Java Collection Framework (JCF) ist die Klassenbibliothek in Java.



- Ziel: Objekte sammeln, speichern, verwalten
- Verschiedene Algorithmen brauchen eine unterschiedliche, passende Datenorganisation (→ Effizienz).
- Wir können jedesmal eigene Verwaltungsstrukturen bauen
  für z.B.: - Liste
  - Keller
  - Baum
  - Hash-Tabelle

  ... oder aber Vorhandenes und Bewährtes wiederverwenden, z.B. JCF mit `java.util`.

Collection Framework:
- Collection (engl.) = Sammlung, Behälter, Container
- Framework = Rahmenwerk
- Java Collection Framework:
  Wichtige Bibliothek (mit einer Menge von Klassen und Interfaces), die viele häufig benötigte Datenstrukturen mit nützlichen Methoden zur Verwaltung und Manipulation von Sammlungen (Collections) von Objekten bereitstellt.
- Die Verwaltung ist unabhängig von den enthaltenen Objekten.
- Das Java Collection Framework befindet sich im Package: `java.util`.
- Vergleichbar mit der *Standard Template Library* (STL) in C++.

Die Entwicklung des Collection Frameworks in Java:
Manche "seltsame Dinge" im Collection Framework sind in seiner Historie begründet (z.B. Klassen mit ähnlichen Namen und ähnlicher Funktionalität aber Unterschieden im Detail).

JDK 1.0: "traditionelle" Collection-Klassen: `Vector`, `Stack`, `HashTable`, `Properties`, `BitSet`

JDK 1.2: grundlegende Umarbeitung und Erweiterung → `Collection` Interface

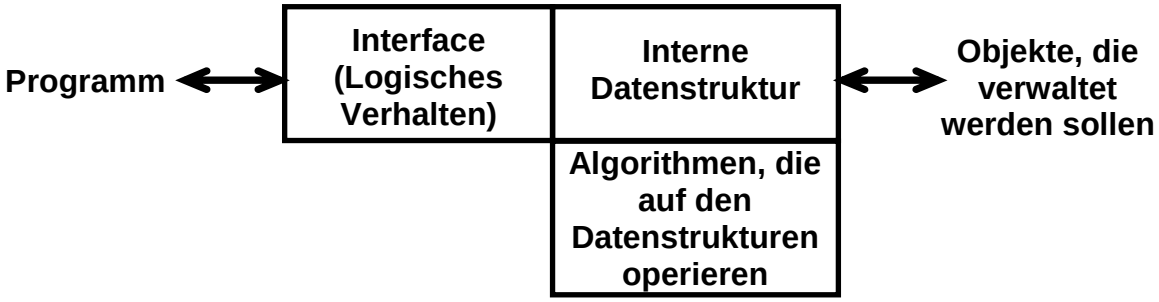
Erweiterungen und Verfeinerungen, z.B. Generizität (ab Java 5)

Ziele bzw. Grundsatz für die Entwicklung des Collection Frameworks: Kleine API (nur mit Basisoperationen), wenig konzeptioneller Balast.

7.3 JCF Organisation und Aufbau

Kernbausteine:

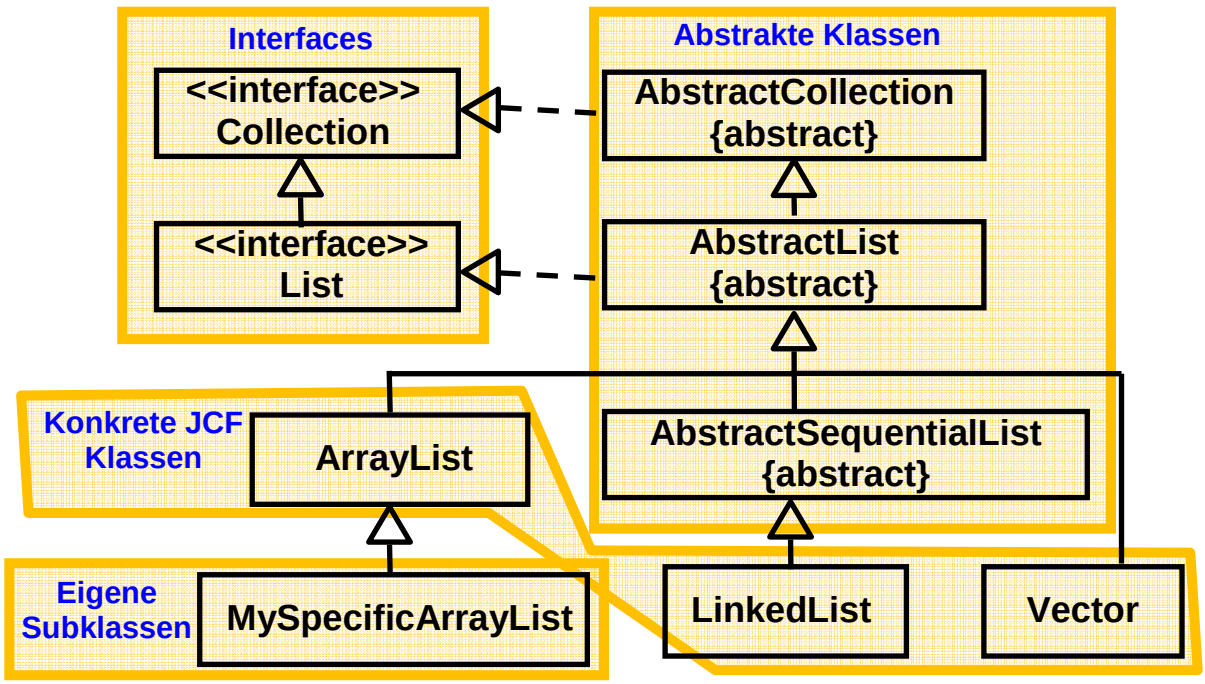
- Abstraktion durch Trennung von Schnittstelle, Datenstruktur (und Algorithmus).



Optionale Elemente in Schnittstellen:

- Viele Methoden in den Collection Interfaces sind als optional gekennzeichnet. Diese sind zwar im Interface vorgeschrieben, müssen aber trotzdem nicht zwingend implementiert werden!
- Für Implementierungen, die optionale Schnittstellenmethoden nicht unterstützen gilt:
  ⇒ Dokumentation der Implementierung!
  ⇒ Bei Zugriff auf nicht implementierte Methode: `UnsupportedOperationException`.

Aufbau und Zusammenhänge im JCF:



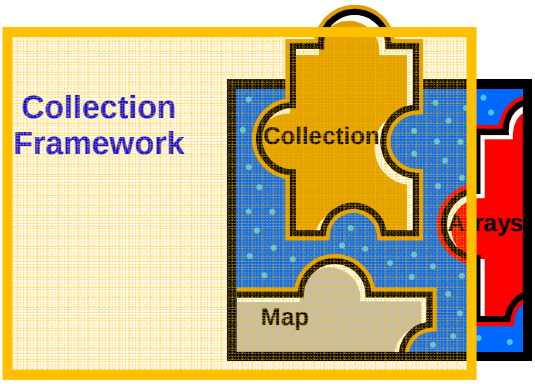
- Zu jedem Interface hat das JCF eine oder mehrere abstrakte Klasse(n) (z.B. `AbstractList`) als Grundgerüst, die das Erstellen eigener Collection-Klassen erleichtern.
- Zu jedem Interface gibt es im JCF eine oder mehrere konkrete Implementierung(en) (z.B. `ArrayList`), die sich in den verwendeten Datenstrukturen und Algorithmen unterscheiden.

Collection Framework: Grundlegende Varianzen der Datenstrukturen
Die im JCF unterstützten Datenstrukturen unterscheiden sich insbesondere in den folgenden Punkten:

- Modifiable versus unmodifiable: Modifikationsoperationen (add, remove, clear, ...) werden unterstützt oder nicht.
- Mutable versus immutable: Die enthaltenen Objekte können verändert werden oder nicht (z.B. bei Schlüsselelementen).
- Fixed-size versus variable size: Die Größe (Anzahl der Elementplätze) ist konstant oder nicht.
- Random-access versus sequential access: Ein schneller wahlfreier Zugriff auf der Basis eines Index ist möglich oder nur ein sequentieller unter Einhaltung einer fixen Reihenfolge.
- Ordnung über den Elementen (oder ungeordnet): Die enthaltenen Elemente sind geordnet oder nicht.
- Einschränkung, welche Elemente gespeichert bzw. verwaltet werden können: Die Datenstruktur kann beispielsweise folgende Einschränkungen festlegen:
  - Nur Elemente mit bestimmter Typeinschränkung können verwaltet werden.
  - Null-Werte sind nicht möglich.
  - Die Elemente müssen ein bestimmtes Prädikat erfüllen.

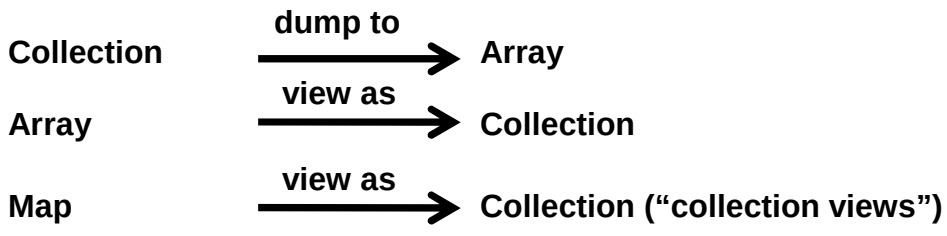
  Verletzender Zugriff ⇒ `ClassCastException`, `IllegalArgumentException`, `NullPointerException`
- Duplikate: Eine Instanz eines Elements kann nur einmal oder mehrfach (Duplikate) enthalten sein.
- Homogen versus inhomogen: Objekte unterschiedlicher Klassen können gleichzeitig verwaltet werden oder nicht.
- Elementanzahl: Es können endlich oder unendlich viele Elemente enhalten sein.
- Zugriffsform: Direktzugriff oder über ein assoziiertes Element.

Verschiedene Arten von Sammlungen von Objekten:



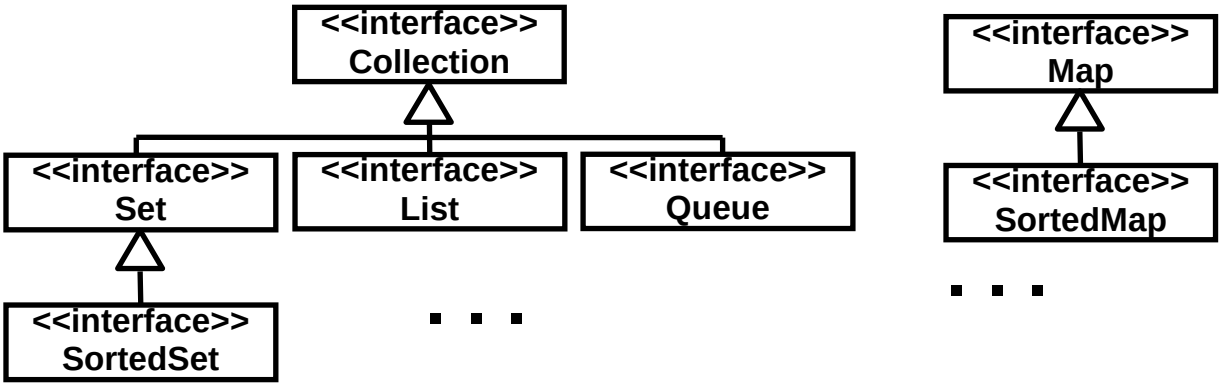
Arrays sind Teil der Sprache. Zusätzlich gibt es für sie Hilfsmethoden im JCF (*Array Utilities*).

Die verschiedenartigen Sammlungen von Objekten sollen auch interoperieren.
Es gibt daher spezielle JCF Methoden zur "Transformation" zwischen den verschiedenen Arten von Objektsammlungen:



7.4 JCF Schnittstellen

In JDK6 gibt es ca. 14 zentrale Interfaces.



- Das Basis-Interface Collection wird erweitert von: Set, List, SortedSet, NavigableSet, Queue, Deque, BlockingQueue, BlockingDeque
- Das zusätzliche Interface Map wird erweitert von: SortedMap, NavigableMap, ConcurrentMap, ConcurrentNavigableMap
- Grund der Trennung Collection und Map: Grundsätzlich unterschiedliche Funktionalität.

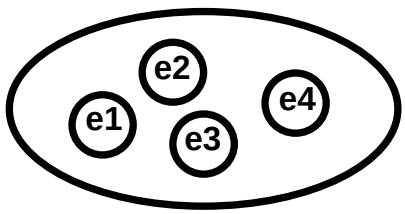
Grundkonzepte und Funktionalitäten der wichtigsten Interfaces

Interface Collection:
Das Basis-Interface Collection fasst die wesentlichen Operationen der Sammlungen vom Typ Set und List (und Queue) zusammen:

- Einfügen von Elementen (z.B. add)
- Entfernen von Elementen (ein Objekt: remove, alle Objekte: removeAll, ...)
- Abfragen (Anzahl enthaltener Objekte: size, ob ein bestimmtes Objekt enthalten ist: contains)
- Umwandlung (Speicherung der Objekte in einem Array, ...), vgl. oben („view as“, „dump to“)
- Zugriff auf Elemente (per Index: get, sequentiell mit Iteratoren, ...)

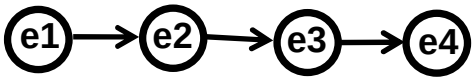
Es gibt keine direkte Implementierung dieses Interface!

Interface Set:



- Eine ungeordnete Menge von Elementen.
- Keine doppelten Elemente, d.h.
  x.equals(y) ergibt false für alle x, y ∈ Set
- Konsequenzen:
  - Wert null ebenfalls nur einmal enthalten.
  - Absicherung in den Konstruktoren, dass keine doppelten Einträge passieren.
  - Vorsicht bei *mutable* Objekten (durch Objektänderung können zwei unterschiedliche Objekte plötzlich mit equals als gleich gewertet werden).

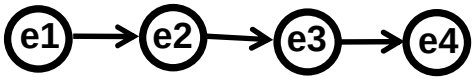
Interface List:



- Eine Menge mit einer Ordnung über den enthaltenen Elementen.
- Doppelte Elemente und null-Werte sind erlaubt.
- Speicherung und Zugriff auf eine spezifische Position über einen Index
  z.B. mit add(int index, Object element), get(int index) ,
  wobei die erste Position bei Index 0 liegt.
- Einfüge- und Löschoperationen können je nach spezifischer Implementierung unterschiedlich lange dauern.
- Die Suche ist oft aufwendig, da sie meist linear durchgeführt wird
  z.B. mit indexOf(Object o), lastIndexOf(Object o)

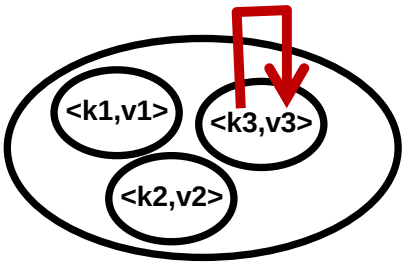
Bemerkung:
In anderen Bibliotheken findet sich die List-Funktionalität gelegentlich auch unter dem Namen „Sequence“.

Interface Queue:



- Verwaltung von Warteschlangen mit verschiedener (Warte-)Ordnung.
- FIFO (First-In-First-Out)
  oder: Prioritätswarteschlangen, LIFO (Last-In-First-Out).
- Es finden sich Operationen wie „Element (typischerweise am Ende) einhängen“ (z.B. Operation offer) und Kopfelement auf verschiedene Arten abfragen bzw. entnehmen (z.B. Operationen element, poll).

Interface Map:



- Verwaltung von Schlüssel/Wert-Paaren (assoziativer Speicher).
- Vergleichbar mit einem Wörterbuch.
- Die Schlüssel sind eindeutig:
  - Jeder Schlüssel kann nur einmal in einer Map enthalten sein.
    Vorsicht bei *mutable*, d.h. veränderbaren Schlüsselobjekten!
  - Jeder Schlüssel kann nur zu einem Wert gehören.
- Es ist keine bestimmte Reihenfolge der Elemente definiert. (Bemerkung: Manche Map Implementationen wie z.B. TreeMap können aber eine Reihenfolge garantieren.)
- Die verwalteten Elemente haben den Elementtyp: Map.Entry<K,V>.
- Das Interface Map fasst die wesentlichen Operationen einer assoziativen Verwaltung zusammen (z.B. put, get, containsKey, keySet).
- Das Interface Map ist nicht von Interface Collection abgeleitet (da Schlüssel/Werte-Paare andere Methoden brauchen).
- Map steht in keiner Beziehung mit dem Interface Iterable (d.h. eine Iteration über den Elementen mit Iteratoren aus der Bibliothek ist nicht direkt möglich)!
  ⇒ Abhilfe, um doch die vorhandenen Iteratoren über der Map nutzen zu können:
  Bildung einer Collection-View einer Map (Transformation, siehe oben).
  Erzeugung von Collection-Objekten aus der Map (z.B. ein Set aus der Menge der Schlüssel mit Operation keySet auf die Map).

7.5 JCF Implementierungen

- Die im JCF vorhandenen general-purpose Implementierungen implementieren auch alle optionalen Operationen der Interfaces, haben keine Elementeinschränkungen und sind unsynchronisiert.
  Zum Thema „Synchronisation“ sehen wir in einem späteren Kapitel (Thema „Nebenläufigkeit“) mehr. Eine unsynchronisierte Datenstruktur kann in einen fehlerhaften Zustand kommen, falls gleichzeitig von verschiedenen Seiten auf sie zugegriffen wird.
- Typischer Aufbau der Klassennamen: *<ImplementationStyle><Interface>*

Die folgende Tabelle zeigt einige wichtige Implementierungen in ihrer Einordnung:

einige wichtige „general-purpose“ Implementierungen

Interfaces	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue				LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Wichtige JCF Implementierungen (Klassen) mit ihren besonderen Eigenschaften

ArrayList:

- Implementiert Listen-Funktionalität mit einem `Array`.
- Schneller Zugriff auf einzelne Elemente.
- Dynamische Anpassung beim Einfügen und Löschen; es werden ggf. die Werte umkopiert (bei Platzmangel).
- Überschreiten der Feldgröße erfordert *Resizing* der Listengröße (standardmäßig mit Faktor 1,5).
  Passen die neuen Elemente nicht mehr in den `Array`
  ⇒ Anlegen eines ganz neuen `Array` und Umkopieren der Werte.

LinkedList:

- Klasse `LinkedList` implementiert eine doppelt verkettete Liste.
- Die Einträge besitzen eine Referenz auf den jeweiligen Nachfolger und Vorgänger.
- Schnelles Einfügen oder Löschen (auch mitten in der Liste).
- Langsamer Zugriff auf bestimmte Elemente anhand eines Indexwertes.

HashSet:

- Die Elemente werden anhand des Hashwertes (`hashCode()`) eingefügt.
- Schnelles Auffinden und Einfügen von Elementen.

TreeSet:
- Elemente werden sortiert in einem Baum vorgehalten.
- Relativ langsames Einfügen und Löschen.
- Sortierte Ausgaben sind schnell möglich.

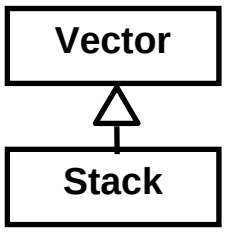
LinkedHashSet :
- Eine schnelle Mengen-Implementierung, die sich zusätzlich auch die Reihenfolge der eingefügten Elemente merkt.
- Schnelles Einfügen/Löschen.

HashMap:
- Implementiert einen assoziativen Speicher durch ein Hashverfahren.
- Unsortierte Speicherung von key/value-Paaren.
- Schneller Zugriff und schnelles Einfügen über Schüssel.
- Achtung bei veränderlichen (*mutable*) Schlüsseln
  (der Hashwert des Schlüssels bestimmt die Position in der `HashMap` ⇒ Änderung des Schlüssels ohne Umpositionierung führt dazu, dass das Objekt danach nicht mehr wiedergefunden wird.)

TreeMap:
- Exemplare dieser Klasse halten ihre Elemente in einem Binärbaum sortiert vor (die Schlüssel sind in einem Baum sortiert abgespeichert).
- Relativ langsames Einfügen/Löschen.
- Sortierte Ausgabe.

LinkedHashMap:
- Ein schneller Assoziativspeicher, der sich zusätzlich auch die Reihenfolge der eingefügten Elemente merkt.
- Schnelles Einfügen/Löschen.

Vector und Stack:
- `Vector` und `Stack` (Keller) als Subklasse zu `Vector`
  sind *Legacy*-Implementierungen (d.h. aus Kompatibilitätsgründen zu den alten Bibliotheken noch vorhanden).
- `Stack`: LIFO-Prinzip (Last-In-First-Out), Methoden wie, z.B. `empty`, `push`, `pop`
- Vererbungsproblematik:
  `Stack` erbt von `Vector` Fähigkeiten wie z.B. Aufzählung und wahlfreier Zugriff auf die Elemente (z.B. `insertElementAt`).
  ⇒ Beispiel für einen schlechten Einsatz von Vererbung: `Stack` erbt Methoden, die im Gegensatz zu den Kellereigenschaften stehen.



Bemerkung zum Begriff „Legacy“:
Legacy-Code ist die Bezeichnung für schon vorhandenen, meist von der zeitlichen Entwicklung überholten Altbestand an Code. Dieser kann nicht einfach weggeworfen werden, da manche Programme darauf aufbauen. Im Bereich der Software-Wartung spielt Legacy-Code eine große Rolle. Ein Entwickler wird in der Regel häufiger mit Legacy-Code zu tun haben, als mit Neuentwicklungen.

7.6 JCF Algorithmen

- Das Collection Framework enthält neben vordefinierten Klassen für die Datenstrukturen auch Algorithmen für die Verarbeitung von und über diesen Datenstrukturen und deren Inhalten.
- Klasse `Collections`: Stellt eine Sammlung von Algorithmen bereit.
- `Collections` ist eine Utility-Klasse mit zusätzlichen Algorithmen für die Verarbeitung von Datenstrukturen. Als Utility-Klasse werden keine Instanzen von ihr verwendet, sondern die Algorithmen sind in statischen Methoden (`Collections` besitzt daher auch nur einen privaten Konstruktor).
- Beispiele enthaltener Algorithmen: `sort` (Sortieren einer Liste nach aufsteigender Größe), `binarySearch` (Elementsuche über einer sortierten Liste), `max` (Rückgabe des größten Elements einer Liste), `shuffle` (Listenelemente zufällig mischen)
- Algorithmen sind nicht orthogonal auf konkrete `Collection`-Objekte anwendbar (sondern meist nur für `List`-Objekte).
  Beispiel: Sortieren ist nur auf `Collection`-Implementierungen möglich, die eine Ordnung definieren.

Größenvergleich in den Algorithmen:
Viele Algorithmen vergleichen Elemente der Größe nach.
Für beliebige Elementtypen ist nicht automatisch klar, welches Element das größere ist.
Abhilfen:

1. Klassen implementieren das Interface `Comparable<T>` (und definieren damit zwingend die Methode `compareTo`)
2. Verwendung eines Comparator (die noch leistungsfähigere Variante):
   - Viele Methoden in Java sind mit einem zusätzlichen Parameter überladen: einem Comparator.
   - Eine Comparator Implementierung implementiert das Interface `Comparator<T>`. Das Interface definiert die Methode `compare` zum Größenvergleich von zwei Objekten.
   - Mit `Comparator` können Objekte nach verschiedenen Kriterien verglichen werden.

Bemerkung zum Begriff „Orthogonalität“:
Orthogonal sind Elemente zueinander dann, wenn sie beliebig und ohne Einschränkungen kombiniert werden können. Der Vorteil ist, dass die Benutzung vereinfacht wird. Es müssen keine Regeln für die Kombination und keine Ausnahmen gelernt werden.

7.7 JCF Typsicherheit

- Seit Java 5: Einsatz von Generics in der Collection-API
- Vorher: Collections haben mit dem Typ `Object`, manuellem Casting und manueller Typprüfung (`instanceof`) zur Laufzeit gearbeitet.
  Der Zugriff ist dabei allerdings nicht typsicher (vgl. Kapitel zum Thema *Generizität*).
- Mit Generizität: Einschränkung der Typen, die in einer Collection verwaltet werden können und Überprüfung der Typen durch den Compiler.
- Die Nutzung von `Collection`-Klassen ohne Generics ist möglich. Der Compiler warnt aber mit einer *type safety* Warnung.
- Alle Collections sollten nur typsicher verwendet werden

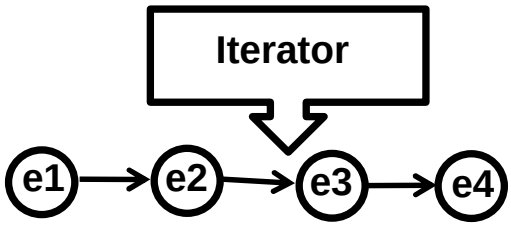
Beispiele:

- Einsatz von generischen Typen (im folgenden Beispiel mit formalem Typparameter `E`) im Interface (analog in einer Klasse): `interface Collection<E>`
- Einsatz von generischen Typen mit Methoden:
  `boolean add(E o), Iterator<E> iterator()`
- Nutzung durch den Programmierer: Erzeugung eines `Collection`-Objekts vom Typ `List` (implementiert `Collection<E>`), wobei nur Objekte vom Typ `MyType` verwaltet werden dürfen:

```
List<MyType> mylist = new ArrayList<MyType>();
```


7.8 Iteration

Ein Iterator-Objekt wird eingesetzt, um eine Collection zu durchlaufen: Mit dem Iterator kann auf die Elemente aus der Collection der Reihe nach zugegriffen werden.



7.8.1 Das Entwurfsmuster Iterator

Wir haben nun also verschiedenartige Implementierungen zur Sammlung von Objekten (z.B. Listen, Keller, Hash-Tabelle).
Um auf die Objekte in diesen Datenstrukturen sinnvoll zugreifen zu können, müssen wir über die enthaltenen Objekte iterieren können.
Trotzdem möchten wir aber auf jeden Fall verhindern, dass der iterierende Zugreifer die Implementierung sieht und womöglich wissen muss, wie die Objekte intern genau gespeichert sind.

Das Problem an einem Beispiel:
Wir haben eine Menge von Modulen, die für den Studiengang angeboten werden, in einer Collection gesammelt. Nun möchten wir über den Modulen darin iterieren, d.h. wir möchten beispielsweise alle Module auf dem Bildschirm ausgeben. Das könnte folgendermaßen aussehen (wir nehmen an, dass die verwendeten Klassen alle schon vorliegen):

Zunächst legen wir ein Modulangebot als Ausgangsbasis an:

```
Modulangebot modulangebot = new Modulangebot();
```

Nun greifen wir auf die Modulsammlung zu:

```
ArrayList module = modulangebot.getItems();
for (int i = 0; i < module.size(); i++) {
  Modul modul = (Modul)module.get(i);
  System.out.println(modul);
}
```

Implementieren wir die Modulangebot-Collection dagegen mit einem `Array` anstelle einer `ArrayList`, ändert sich der Zugriff:

```
Module[] module = modulangebot.getItems();
for (int i = 0; i < module.length; i++) {
  Modul modul = module[i];
  System.out.println(modul);
}
```

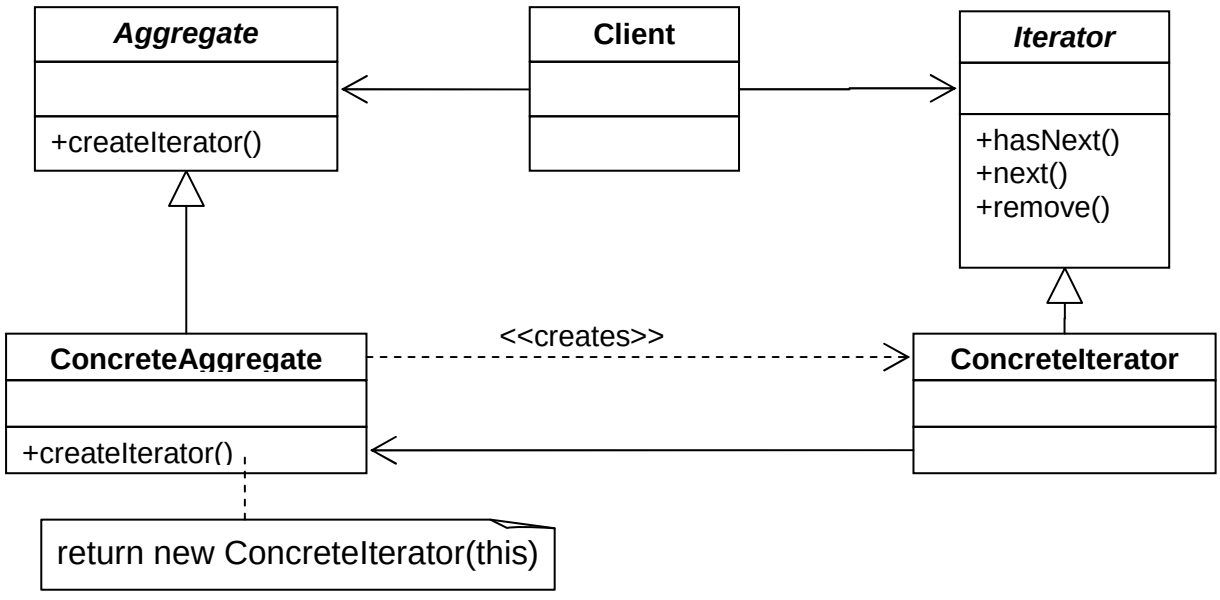
Die genaue Nutzung der Collection hängt also von der internen Implementierung ab. Die interne Implementierung soll aber für einen Nutzer der Collection versteckt sein (Kapselung und Geheimnisprinzip).
Unsere Nutzung codiert gegen die Implementierung anstelle gegen eine Schnittstelle.
Eine Änderung der internen Implementierung würde eine Änderung im Nutzer notwendig machen.

Nehmen wir weiter an, dass wir neben der Sammlung mit angebotenen Modulen eine weitere Sammlung mit gewählten Modulen verwalten möchten. Eine Iteration würde ähnlich aussehen und zu redundantem Code führen.

Was können wir verbessern?

- Statt einer Wiederholung des leicht variierenden Schleifen-Codes für die Iteration in verschiedenen Implementierungen möchten wir diese offenbar häufiger benötigte Funktionalität kapseln und separat bereithalten. Die Collection kann sich damit auf ihre eigentliche Aufgabe zur Verwaltung der Datenstruktur fokussieren und nicht noch zusätzlich auf die Iteration. Jede Aufgabe bzw. Verantwortlichkeit ist eine mögliche Ursache für spätere Änderungen. Es gilt aber das Entwurfsprinzip: Eine Klasse sollte nur einen einzigen Grund zur Änderung haben. Auch hier gilt wieder das Verantwortlichkeitsprinzip und Kohäsionsprinzip.
- Wir vermeiden die Nutzung der internen Implementierung und setzen stattdessen ein Iterator-Interface ein. Dadurch ist die Implementierung leicht austauschbar (z.B. der Umbau von einer `ArrayList` zu einer `HashTable`). Die Benutzung wird dagegen einheitlich und bleibt bei einer Änderung der Implementierung stabil (bzw. unsichtbar für den Nutzer).

Lösung:. Entwurfsmuster Iterator (Iterator Pattern)
Mit dem schon aus den vorderen Teilen bekannten *Singleton* haben wir bereits eine erste Bekanntschaft mit Entwurfsmustern gemacht.
Ein Muster ist eine wiederverwendbare Lösung für häufig wiederkehrende Probleme bzw. eine Vorlage, wie ein bestimmtes Problem gelöst werden kann.
Auch für die beschriebene Problemstellung der Iteration gibt es ein bekanntes Entwurfsmuster unter dem Namen *Iterator*.



Besonderheiten des Iterator Pattern:

- Das Muster enthält eine Schnittstelle Iterator, die für alle Nutzer die Methoden `hasNext()`, `next()` einheitlich festlegt. Die Schnittstelle kann in Java z.B. als Interface oder abstrakte Klasse realisiert sein. Ein Nutzer (Client) ruft die Methoden der Schnittstelle, um über der Datenstruktur (*Aggregate*) zu iterieren.
- Die Schnittstelle Iterator sollte auch eine Methode `remove()` enthalten. Mit ihr können Elemente in der Datenstruktur entfernt werden und der Iterator kann in seinem aktuellen Durchlaufzustand gleichzeitig entsprechend angepasst werden. Wird `remove()` nicht unterstützt, sollte eine Exception geworfen werden (in Java z.B. `java.lang.UnsupportedOperationException`). Die Methode `remove()` muss beispielsweise sicherstellen, dass auch verschiedene Iteratoren, die nebenläufig die Collection durchlaufen korrekt weiterfunktionieren, wenn sich die Datenstruktur ändert.
- Das Muster umfasst spezifische Iterator-Implementierungen (*ConcreteIterator*), die die Schnittstelle Iterator implementieren und die Implementierung für eine spezifische Datenstruktur (z.B. `ArrayList` oder `Array`) umsetzen.
- Eine Methode `createIterator()` in der Datenstruktur selbst liefert einen für die Datenstruktur passenden Iterator zurück, mit dem die Datenstruktur durchlaufen werden kann. Eine Datenstruktur mit einer Menge von Objekten wie z.B. Sammlungen (Collection) wird gelegentlich auch *Aggregate* genannt.
  Die konkreten Datenstrukturen sind wiederum Implementierungen der Aggregate-Schnittstellenvorgabe.
  Beispiel für die Umsetzung des Musters in Java: Die Methode `iterator()` von `ArrayList` liefert einen passenden Iterator zurück.
- Die Objekte, die in der Datenstruktur verwaltet werden, werden nur indirekt über den Iterator adressiert. Die Repräsentation und Struktur der Objekte und der Objektsammlung ist für den Zugreifer (Client) versteckt.

Bemerkungen:

- Der Iterator für eine bestimmte Collection kann gut als innere Klasse realisiert werden, da dadurch einerseits eine gute Kapselung und Modularisierung erreicht werden kann, andererseits aber der engen und starken Kooperation und Kommunikation zwischen Iterator und Collection Rechnung getragen wird. Zum Thema „innere Klassen“ folgt unten mehr.
- In früheren Iterator-Mustern finden sich die Methoden `first()`, `next()`, `isDone()`, `currentItem()` im Iterator-Interface. Grundsätzlich spricht nichts dagegen, diese oder andere Methoden im Iterator-Interface einzusetzen. Durch Interface-Vererbung können darüber hinaus weitere spezifischere Iterator-Methoden im Interface vorgeschrieben werden.
- Im Allgemeinen sollte bei der Verwendung von Iteratoren keine Annahme über die Ordnung der Elemente innerhalb der Datenstruktur gemacht werden (sofern die Datenstruktur dies nicht entsprechend dokumentiert).

Zusammenfassend: Ein Iterator liefert einen Weg, auf die Elemente einer Datenstruktur (im Muster als Aggregate Object bezeichnet) sequentiell zuzugreifen, ohne dass ein Zugriff auf die darunterliegende Implementierung oder die Kenntnis dieser Implementierung notwendig ist.

7.8.2 *Null Iterator* und „Entwurfsmuster“ *Null Object*

Angenommen eine besondere, konkrete Collection-Implementierung hat nichts, über das sinnvoll iteriert werden kann bzw. soll.
Das Interface Iterator schreibt allerdings die Methode `createIterator()` vor.

Lösung 1: Null-Rückgabe

Unsere Methode `createIterator()` schreiben wir für diese spezielle Collection so, dass sie `null` zurückgibt.
Nachteil: Wir haben einen Sonderfall im Code und wir müssen im Nutzer eine Abfrage einbauen, die diesen Spezialfall `null` behandelt. Der Nutzer muss sich also um etwas kümmern, was nicht in seinen Aufgabenbereich fällt.

Lösung 2: Null Iterator

Unsere Methode liefert ein „Dummy-Objekt“ zurück. In diesem Fall ist es ein *Null Iterator*, das zwar vom Typ `Iterator` ist, aber immer `false` zurückliefert, wenn `hasNext()` gerufen wird und auch sonst nichts macht.

Beispiel für einen Null Iterator in Java:

```
import java.util.Iterator;
public class NullIterator implements Iterator {
  public Object next() {
    return null;
  }

  public boolean hasNext() {
    return false;
  }

  public void remove() {
    throw new UnsupportedOperationException();
  }
}
```

Ein Null Iterator ist ein Beispiel für das sogenannte „Entwurfsmuster“ *Null Object*:
Ein Null Object ist ein Stellvertreterobjekt, wenn kein sinnvolles Objekt zurückgegeben werden kann. Damit nicht der Nutzer (Client) die Verantwortung trägt, `null`-Abfragen in seinen Code einzufügen, bekommt er stattdessen ein Objekt vom passenden Typ als Rückgabewert. Dieses versteht damit alle Nachrichten, macht aber jeweils nichts.

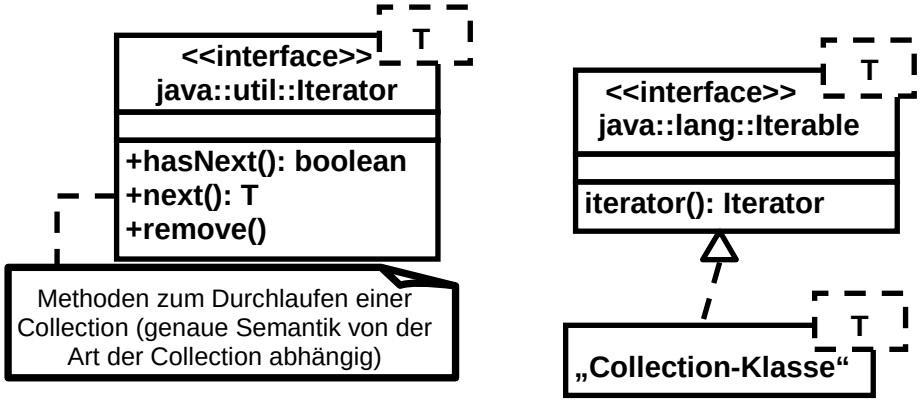
Bemerkung:
Ob das „triviale“ Null Object wirklich als eigenes Entwurfsmuster bezeichnet werden kann, ist umstritten.

7.8.3 Iteration in Java

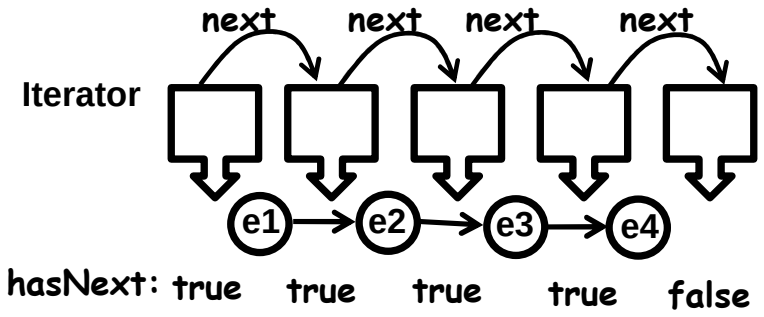
Viele JCF-Klassen in Java unterstützen Iteratoren, d.h. sie bieten mindestens eine Methode an, mit dem ein passender Iterator von der Datenstruktur bezogen werden kann.
Für Objektsammlungen in Java, die nicht schon von Haus aus einen Iterator anbieten, kann, dem gezeigten Entwurfsmuster folgend, ein eigener Iterator aufgebaut werden.

`Iterator` und `Iterable` Interface in Java

Ein Blick in das Interface `Collection` zeigt, dass dort bereits eine Methode `iterator()` vorgeschrieben ist. Das Interface erbt diese Methodenvorschrift von Interface `Iterable`.
Eine Collection-Klasse implementiert demnach neben dem Interface `Collection` auch das Interface `Iterable`. Sie zeigt damit an, dass ihre Elemente mit einem Iterator durchlaufen werden können. Die Elemente in der Collection sind dabei generisch und damit auch das Interface `Iterable`. Die Methoden, die das Interface vorschreibt, müssen für verschiedene Elementtypen ausgelegt sein.



Grundlegende Eigenschaften eines Iterators:
- Ein Iterator läuft von Beginn an Element für Element durch eine Collection (keine Sprünge, kein Start an beliebiger Stelle).
- Ein Iterator steht immer zwischen zwei Elementen.



Ist das Ende erreicht ⇒ Der Iterator ist verbraucht (keine Wiederverwendung möglich!).

Es gibt weitere, spezialisierte Iteratoren wie z.B. `ListIterator<T>` mit besonderen Eigenschaften:
- Kann sich vorwärts und auch rückwärts bewegen (bidirektional).
- Ist nicht verbraucht, wenn er am Ende der Liste angekommen ist.

Fail Fast Iterator

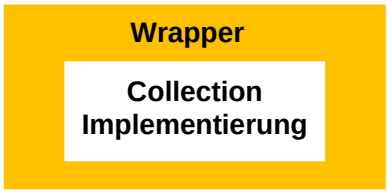
- Wird eine Collection mit einem Iterator durchlaufen, sollte diese nicht verändert werden, da sich sonst das Iterator-Objekt in einem inkonsistenten Zustand befindet.
- Daher können die Iteratoren eine Veränderung der Collection erkennen.
- Wird eine Collection während des Durchlaufens strukturell verändert, wirft der Iterator beim nächsten Methodenaufruf eine `ConcurrentModificationException` (der Iterator wird also unbrauchbar).
- In früheren Java-Versionen wurde keine Exception geworfen und der Iterator befand sich in einem nicht definierten Zustand, der eine weitere korrekte Verwendung des Iterators nicht garantierte.
- Ein sicheres Löschen ist nur mit den (optionalen) Iterator-Methoden `next()` und `remove()` möglich (die Funktionsfähigkeit des Iterators bleibt auf diese Weise erhalten).

foreach-Schleife statt explizite Iterator-Erzeugung

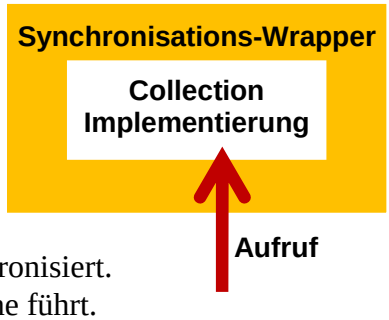
- Nutzen einer foreach-Schleife: Vereinfachte Schreibweise zur Iteration für den Spezialfall:
  - wenn alle Elemente einer Collection komplett von vorn nach hinten abgelaufen bzw. abgefragt werden sollen,
  - der Index nicht sichtbar ist (bzw. sein muss),
  - nur Abfragen und kein Setzen durchgeführt werden sollen (oder müssen).
- Syntax in Java: `for(ElementeTyp Variablenbezeichner : Collection-Variable)`
  Diese foreach-Schleife durchläuft die Collection, die in der `Collection-Variable` gespeichert ist, und holt Element für Element in die Variable mit Namen `Variablenbezeichner`.
- Die foreach- und die explizite Iterator-Lösung arbeiten im Prinzip gleich und bilden sich auf den gleichen Java Bytecode ab.

7.9 JCF Wrapper Implementierung

Beispiele für Wrapper Implementierungen:



- *Unmodifiable-Wrapper*:
  Erzeugung unveränderlicher Sichten auf vorhandene Datenstrukturen, d.h. die Datenstruktur wird nicht verändert. Eine Verwendung aller schreibenden Methoden wirft stattdessen eine `UnsupportedOperationException`.

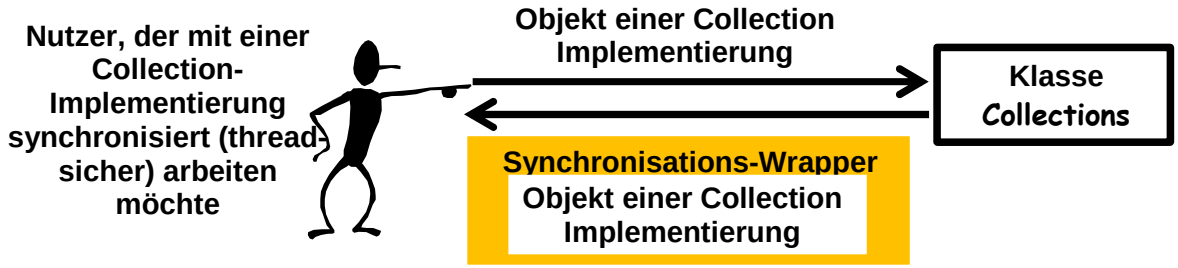


- *Synchronisations-Wrapper*:
  Dieser Wrapper liefert zu einer Collection eine thread-sichere Collection zurück (alle Zugriffe über diese Wrapper-Collection sind thread-sicher).

Zum Thema Collections und Synchronisation (Thread-Sicherheit von Collections):
Traditionelle Collections (z.B. `Vector`) sind weitgehend synchronisiert.
Das bedeutet, dass ein nebenläufiger Zugriff zu keinen Probleme führt.
Das übrige Collection Framework (d.h. die neueren JCF Klassen) ist ansonsten durchweg unsynchronisiert.
Die Idee: Statt einer großen Collection-Klasse mit umfassender Schnittstelle und Funktionalität sollen Collection-Klassen angeboten werden, die sich auf die Kernfunktionalität (die eigentliche Verwaltung der Datenstruktur) konzentrieren. Sie sollen nicht durch alle möglichen Zusatzfunktionen (wie z.B. Synchronisation) aufgebläht werden. Ist eine Synchronisation z.B. einer Liste oder eines Assoziativspeichers doch notwendig, kann diese Funktionalität nachträglich durch einen Wrapper ergänzt werden. Der Wrapper legt eine synchronisierende Hülle um alle Methodenaufrufe. In diesem Entwurfsprinzip bildet sich wiederum das Verantwortlichkeits- und Modularisierungsprinzip ab. Die Gesamtfunktionalität wird in die klar abgegrenzten Aufgabenbereiche unterteilt und je nach Bedarf wieder flexibel zusammengesetzt.

Beispiel: Einsatz eines Synchronisations-Wrapper in Java

- Statische Methoden in der Klasse `Collections` liefern für verschiedene Collection Implementierungen entsprechende synchronisierte Varianten zurück.
- Kein Nutzer darf unsynchronisiert zugreifen.

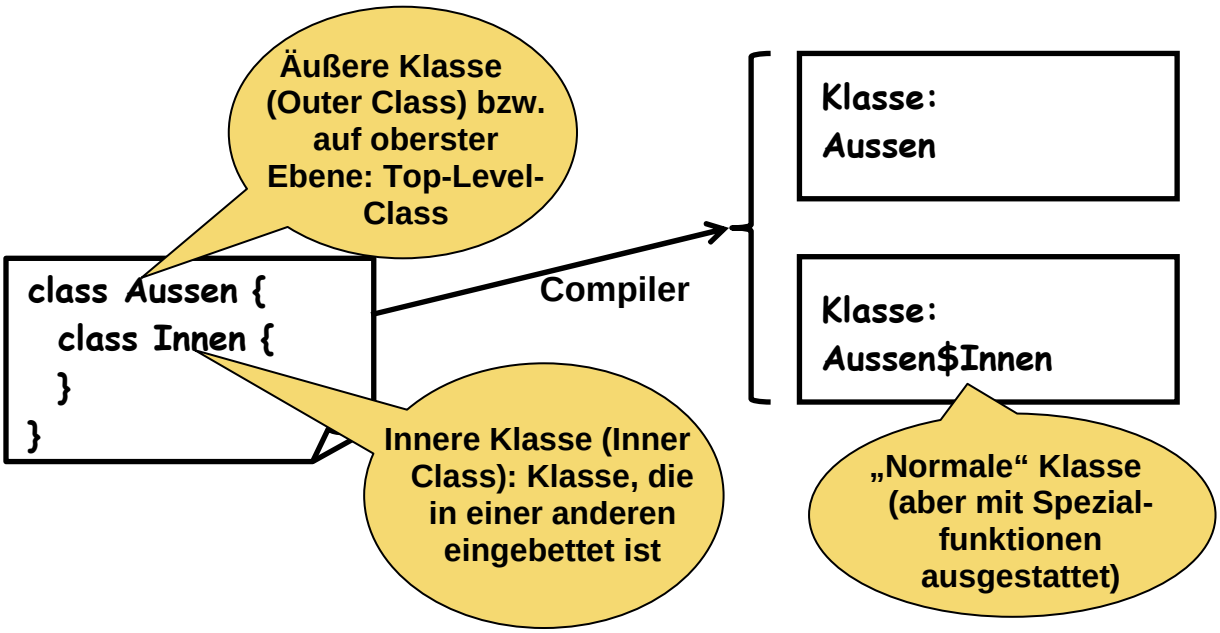


Beispiel in Java: `Set s = Collections.synchronizedSet(new HashSet());`
Für die neu erzeugte `HashSet` wird über `Collections` eine synchronisierte Variante zurückgeliefert.
Im Kapitel zum Thema „Nebenläufigkeit“ werden wir die *Thread-Sicherheit* und *Synchronisation* noch etwas genauer betrachten.

7.10 Exkurs: Innere Klassen

Ein Iterator läuft die Elemente einer Collection der Reihe nach ab.
Dazu müssen dem Iterator alle Elemente bekannt sein.
Da eine Collection die innere Struktur nicht preisgeben soll (Kapselung), stellt sich die Frage, wie auf die enthaltenen Objekte zugegriffen werden kann.
Abhilfe: Innere Klassen

Einbettung von Klassen in andere Klassen ⇒ sehr enge Bindung
In Java sind innere Klassen eine Erweiterung der Sprache seit Java 1.1. Die Erweiterung ist keine JVM Erweiterung.



Nutzen innerer Klassen:
- Hilfsklassen
- Strukturierung, Modularisierung und Kapselung: klarer Code, weniger Quell-Dateien
- Einsatzbeispiel: Mitgliedsklassen `Entry` und `Iterator` der `Collection`-Klassen

In Java können vier Arten von inneren Klassen unterschieden werden:

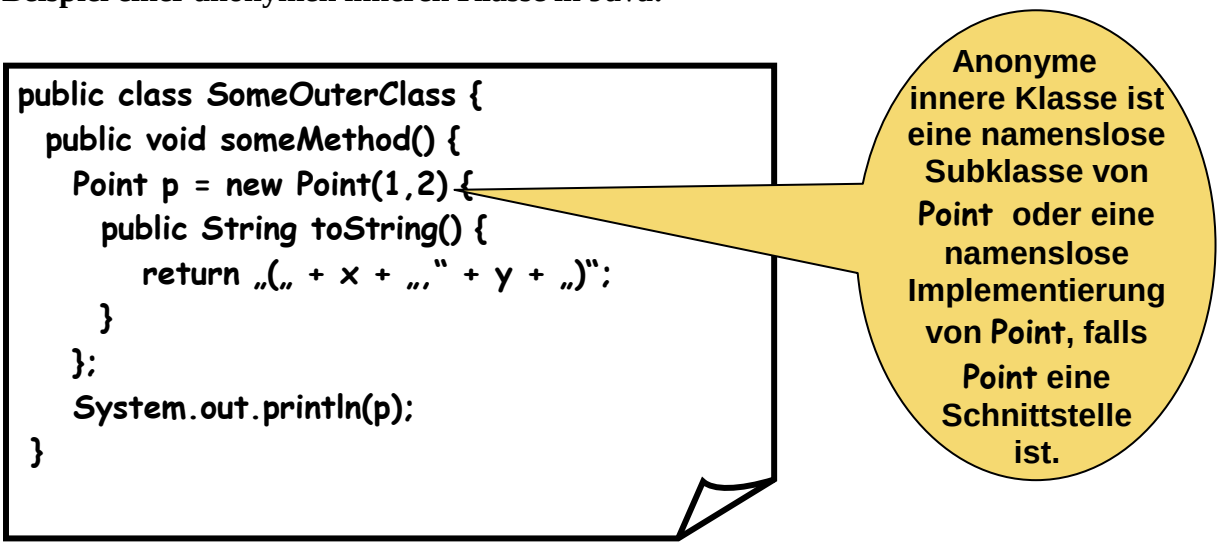
1) **Member Class (Mitgliedsklasse, Elementklasse):** Zur Erzeugung muss ein Objekt der äußeren Klasse existieren; die innere Klasse kann auf alle Attribute der äußeren zugreifen (auch private); innen sind keine statischen Eigenschaften erlaubt.

2) **Nested top-level Class (statische innere Klasse, statische Mitgliedsklassen), static:** Sie können das Gleiche wie “normale” Klassen nur bilden sie quasi ein kleines Unterpaket mit eigenem Namensraum; zur Erzeugung sind keine Objekte der umgebenden Klasse notwendig; Zugriff nur auf statische Attribute der umgebenden Klasse

3) **Lokale Klassen:** In Anweisungsblöcken von Methoden oder Initialisierungsblöcken definiert (nur im Block sichtbar).

4) **Anonyme innere Klassen:** Ohne Namen; sie erzeugen immer automatisch ein Objekt; zusätzliche extends/implements-Angaben sind innen nicht möglich; in der inneren Klasse sind nur Instanzmethoden (keine Konstruktoren) und finale statische Variablen erlaubt; ein Zugriff ist auf eigene Eigenschaften und nur auf die lokalen Konstanten (final) des umschließenden Blocks möglich.

Beispiel einer anonymen inneren Klasse in Java:



Die Klassendeklaration und Objekterzeugung sind zu einem Sprachkonstrukt verbunden.

Erzeugung einer anonymen inneren Klasse allgemein in Java:
new KlasseOderInterface() { /* Eigenschaften der inneren Klasse */}

