# Informatik B

Vorlesung 12

Collections, Queue, Stack,

Innere Klassen

# Rückblick

- Interface Collection
- **`Set`**
  - **`HashSet`**
  - **`TreeSet`**
- **`List`**
  - **`ArrayList`**
  - **`LinkedList`**
  - **`ListIterator`**
- **`Map`**

# Algorithmen der Klasse Collections

- Um Probleme in der Informatik zu lösen, ist die Wahl einer geeigneten Datenstruktur nur der erste Schritt
- Im zweiten Schritt müssen Algorithmen implementiert werden
- Da viele Algorithmen immer wiederkehrende (Teil-)Probleme lösen, hilft auch hier die Java-Bibliothek mit einigen Standardalgorithmen weiter
- Dazu zählen Methoden zum Sortieren und Suchen in Collections und das Füllen von Collections
- Um die Methoden möglichst flexibel einzusetzen, implementierten die Bibliotheksentwickler die Klasse **`Collections`** (nicht mit dem Interface **`Collection`** verwechseln!!)
- **`Collections`** bietet notwendige Algorithmen als statische Methoden an

# Algorithmen der Klasse Collections

- Viele Algorithmen sind nur für **`List`**-Objekte deklariert, da sie auf allgemeinen Collections nicht funktionieren können
- Das ist nicht erstaunlich, denn wenn z.B. eine Collection keine Ordnung definiert, kann sie nicht sortiert werden
- Auch die binäre Suche erfordert Collections mit einer impliziten Reihenfolge der Elemente
- Nur Min- und Max-Methoden arbeiten auf allgemeinen **`Collection`**-Objekten
- Alle Methoden sind statisch, sodass **`Collections`** eine Utility-Klasse wie **`Math`** ist
- Ein Objekt von **`Collections`** lässt sich nicht anlegen da der Konstruktor **`private`** ist

# Methodenüberblick

- **`shuffle()`:** Eine Liste durcheinander wirbeln
- **`reverse()`:** Reihenfolge in der Liste umdrehen
- **`copy()`:** Eine Liste in eine andere kopieren
- **`binarySearch()`:** Binäre Suche nach einem Element
- **`frequency()`:** Häufigkeit eines Elementes einer Collection
- **`max()/min()`:** Liefert das Maximum/Minimum der Collection
- **`sort()`:** Sortiert eine Liste
- Usw.

# Synchronisation einer Collection

- Ein großer Unterschied zwischen den klassischen Datenstrukturen wie **`Vector`** oder **`HashTable`** und den neueren besteht darin, dass alle Methoden durch synchronisierte Blöcke vor parallelen Änderungen geschützt waren
- Bei den neuen Klassen wie **`ArrayList`** und **`HashMap`** sind Einfüge- und Löschoperationen nicht mehr automatisch **`synchronized`**
- Sollen Listen, Mengen oder Assoziativspeicher vor nebenläufigen Änderungen sicher sein, gibt es Synchronisierende Wrapper

# Synchronisierende Wrapper

- Nichtsynchronisierte Collections können nachträglich synchronisiert werden
- Dazu gibt es in der Klasse Collections Methoden, die einen Wrapper liefern
- Diese Methoden liefern eine neue Collection, die sich wie eine Hülle um die existierende Datenstruktur legt (Wrapper) und alle Methodenaufrufe synchronisiert

# Methoden für nachträgliche Synchronisation

- `static <T> Collection<T> synchronizedCollection(Collection<T> c)`
- `static <T> List<T> synchronizedList(List<T> list)`
- `static <K,V> Map<K,V> synchronizedMap(Map<K,V> m)`
- `static <T> Set<T> synchronizedSet(Set<T> s)`
- `static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m)`
- `static <T> SortedSet<T> synchronizedSortedSet(SortedSet<T> s)`

# Weitere Collections

- Neben den drei Standardcollections gibt es noch weitere
  - **`Queue`**
  - **`Stack`**

# Queue

- In der Klassenbibliothek von Java gibt es das Interface **`java.util.Queue`** für Datenstrukturen, die nach dem FIFO-Prinzip (*First In First Out*) arbeiten
- Das Interface **`Queue`** erweitert **`Collection`** und ist somit auch vom Typ **`Iterable`**
- Zu den Klassen, die **`Queue`** implementieren, gehört unter anderem **`LinkedList`**

# Queue

- In dem Interface **`Queue`** sind folgende Methoden definiert:
  - **`E element()`** : Kopf liefern (falls leer wird eine Exception geworfen)
  - **`boolean offer(E o)`** Element einhängen, liefert keine Exception, wenn es nicht klappt
  - **`E peek()`** Kopf liefern (falls leer, keine Exception)
  - **`E poll()`** Kopf liefern und gleich entfernen (falls leer, keine Exception)
  - **`E remove()`** Kopf liefern und entfernen (falls leer wird eine Exception geworfen)

# Verschiedene Queues

- **`ConcurrentLinkedQueue`**
  - Thread-sichere Queue durch verkettete Listen implementiert
- **`DelayQueue`**
  - Queue, aus der die Elemente erst nach einer gewissen Zeit entnommen werden können
- **`ArrayBlockingQueue`**
  - Queue mit einer maximalen Kapazität, abgebildet auf ein Feld
- **`LinkedBlockingQueue`**
  - Queue beschränkt oder mit maximaler Kapazität, abgebildet durch eine verkettete Liste

# Verschiedene Queues

- **`PriorityQueue`**
  - Hält Elemente sortiert und liefert bei Anfragen das jeweils kleinste Element
  - Wie beim **`TreeSet`** müssen die Elemente entweder **`Comparable`** implementieren, oder es muss ein **`Comparator`** angegeben werden
  - Die FIFO Reihenfolge gilt hier nicht!
- **`PriorityBlockingQueue`**
  - Wie PriorityQueue, nur blockierend
- **`SynchronousQueue`**
  - Eine blockierende Queue zum Austausch von genau einem Element

# Consumer/Producer mit Queues

- Mit einer **`BlockingQueue`** ist es sehr einfach das Producer/Consumer Problem zu implementieren
- Dazu bietet eine **`BlockingQueue`** die Methoden **`take()`** und **`put()`** an, die ggf. ein **`wait()`** initiieren, falls der Puffer leer oder voll ist und etwas abgeholt oder eingestellt werden soll
- Da innerhalb der Methoden gewartet wird, kann es zu einer **`InterruptedException`** kommen, die abgefangen werden muss
- Beispiel: **`producerconsumer1`**

# `Stack`

- Die Klasse **`Stack`** repräsentiert einen Keller
- Ein **`Stack`** funktioniert nach dem LIFO-Prinzip (*Last-In-First-Out*)
- Beim Hinzufügen von Elementen wächst die Datenstruktur dynamisch
- Die Klasse **`Stack`** ist eine Erweiterung der Klasse **`Vector`** womit die Klasse zusätzliche Funktionalität besitzt, beispielsweise die Fähigkeit der Aufzählung und des wahlfreien Zugriffs auf Kellerelemente

# Stack

- **`Stack`** besitzt nur wenige zusätzliche Methoden, verglichen mit **`Vector`**
  - **`boolean empty()`** Keller leer?
  - **`E push(E item)`** **`item`** auf den Keller legen
  - **`E pop()`** Das letzte Element vom Keller holen. **`EmptyStackException`** signalisiert einen leeren Stapel.
  - **`E peek()`** Das oberste Element liefern, aber nicht entfernen. Bei leerem Stapel wird eine **`EmptyStackException`** ausgelöst.
  - **`int search(Object o)`** Sucht im Stapel nach dem obersten Eintrag, der mit dem Objekt **`o`** übereinstimmt. Gibt den Index zurück oder **`-1`**, falls das Objekt nicht im Stapel ist

# `Stack`

- Eine genaue Betrachtung der Klasse **`Stack`** zeigt den unsinnigen und falschen Einsatz der Vererbung
- **`Stack`** erbt alle Methoden von **`Vector`** und damit viele Methoden, die im krassen Gegensatz zu den charakteristischen Eigenschaften eines Kellers stehen
- Dazu zählen unter anderem die Methoden **`elementAt()`, `indexOf()`, `insertElementAt()`, `removeElementAt()`, `setElementAt()`** und weitere
- Wenn eine Unterklasse Eigenschaften und Methoden erbt, die im Widerspruch zur eigentlichen Funktionalität stehen, ist die Vererbung falsch angewendet

# `Iterator`

- 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, stellt sich die Frage, wie auf die enthaltenen Objekte zugegriffen werden kann
- Eine Lösung stellen die inneren Klassen dar

# Innere Klassen

- Klassen waren bisher in Paketen organisiert und wurden in eine Datei geschrieben
- Diese Form von Klassen heißen Top-Level-Klassen
- Es gibt zusätzlich die Möglichkeit, eine Klasse in eine andere Klasse einzubetten und sie damit noch enger aneinander zu binden
- Eine Klasse, die so eingebunden wird, heißt „innere Klasse“ (*inner class*):

```
class Aussen {
    class Innen {
    }
}
```

# Innere Klassen

- Der Compiler generiert aus den inneren Klassen normale Klassen, die jedoch mit einigen Spezialfunktionen ausgestattet sind
- Für die geschachtelten inneren Klassen generiert der Compiler neue Namen nach dem Muster: **`ÄußereKlasse$InnereKlasse`**
- Ein Dollar-Zeichen trennt die Namen von äußerer und innerer Klasse
- In Java gibt es vier verschiedene Arten von Inneren Klassen

# *nested top-level class*

- Die einfachste Variante einer inneren Klasse oder Schnittstelle wird wie eine statische Eigenschaft in die Klasse eingesetzt und heißt statische innere Klasse
- Wegen der Schachtelung wird dieser Typ im Englischen *nested top-level class* genannt
- Die Namensgebung betont mit dem Begriff *top-level*, dass die Klassen das Gleiche können wie normale Klassen oder Schnittstellen nur bilden sie quasi ein kleines Unterpaket mit eigenem Namensraum
- Insbesondere sind zur Erzeugung von Exemplaren von inneren Klassen keine Objekte der äußeren Klasse nötig

# *nested top-level class*

- Eine statische innere Klasse besitzt Zugriff auf alle anderen statischen Eigenschaften der äußeren Klasse
- Ein Zugriff auf Objektvariablen ist aus der statischen inneren Klasse nicht möglich, da sie als extra Klasse gezählt wird, die im gleichen Paket liegt
- Der Zugriff von außen auf innere Klassen gelingt mit der Schreibweise **`ÄußereKlasse.InnereKlasse;`**
- Der Punkt wird also so verwendet, wie wir es von den Paketen als Namensraum gewöhnt sind
- Die innere Klasse muss einen anderen Namen als die äußere haben
- Es sind die Modifizierer **`abstract`**, **`final`** und Sichtbarkeitsmodifizierer erlaubt
- Beispiel: **`nestedtoplevel1`**

# *nested top-level class*

- Statische innere Klassen können von außen sichtbar sein
- Sind sie sichtbar können sie wie jede andere Klasse auch verwendet werden
- Es muss lediglich die Art des Zugriffs beachtet werden:
  **`Aussen.Innen var = new Aussen.Innen(…);`**
- Alternativ kann die Klasse **`Innen`** auch über einen **`import`** bekannt gemacht werden: **`import package.Aussen.Innen;`**
- Der Sichtbarkeitsbereich kann jedoch eingeschränkt werden
- Es ist z.B. möglich eine
  **`static private class Innen {…}`**
  zu implementieren
- Es stellt sich die Frage, wozu eine solche Klasse genutzt werden kann

# *nested top-level class*

- Wird eine *nested top-level class* als **`private`** deklariert, kann sie dennoch von der äußeren umgebenden Klasse erreicht werden
- Auch wenn die Instanzvariablen **`private`** sind können diese direkt von Methoden der äußeren Klasse manipuliert werden
- Auf diese Art und Weise können innere Klassen definiert werden, die nur von der äußeren Klasse verwendet werden können
- So kann Programmcode weiter modularisiert werden, ohne nach außen hin neue Klassen zu schaffen
- Beispiel: **`nestedtoplevel2, nestedtoplevel3`**

# *member class*

- Eine *nested top-level class* hat keinen Zugriff auf die Instanzvariablen der äußeren Klasse, wie denn auch, sie ist ja an kein konkretes Objekt gebunden und wüsste nicht, auf welche Instanzvariablen sie zugreifen soll
- Eine Mitgliedsklasse (engl. *member class*), auch Elementklasse genannt, kann auf alle Attribute der äußeren Klasse zugreifen
- Dazu zählen auch die privaten Eigenschaften, eine Designentscheidung, die sehr umstritten ist und kontrovers diskutiert wird
- Um innerhalb der äußeren Klasse ein Objekt der inneren *member class* zu erzeugen, muss ein Objekt der äußeren Klasse existieren, im Gegensatz zu einer *nested top-level class* die auch ohne Objekt der äußeren Klasse existieren kann
- Eine zweite wichtige Eigenschaft ist, dass eine *member class* selbst keine statischen Eigenschaften deklarieren darf

# *member class* (Objekterzeugung)

- Innerhalb der äußeren Klasse kann einfach mit dem **`new`**-Operator ein Exemplar der inneren Klasse erzeugt werden
- Soll von außerhalb ein Objekt erzeugt werden, muss bei *member classes* sichergestellt werden, dass es ein Exemplar der äußeren Klasse gibt
- Die Sprache schreibt eine neue Form für die Erzeugung mit **`new`** vor: **`referenz.new InnereKlasse(...)`**
- Dabei ist **`referenz`** eine Referenz der Äußeren Klasse
- Durch diese schreibweise werden der *member class* sofern erforderlich Instanzvariablen verfügbar gemacht
- Werden Werte am Objekt der äußeren Klasse nachträglich geändert werden diese Änderungen für das mit diesem Objekt verbundene Objekt der Inneren Klasse sichtbar
- Beispiel: **`memberclass1`**

# *member class* (**`this`**)

- Möchte eine innere Klasse **`In`** auf die **`this`**-Referenz der umgebenden Klasse **`Out`** zugreifen, schreiben man **`Out.this`**
- Wenn sich Variablen überdecken, so schreibt man **`Out.this.eigenschaft`**, um an die Eigenschaften der äußeren Klasse zu gelangen
- *Member classes* können beliebig geschachtelt sein
- Da der Name eindeutig ist, gelangen man mit **`Klassenname.this`** immer an die jeweilige Eigenschaft
- Eine *member class* ist gut geeignet, um einen **`Iterator`** zu implementieren, da sie Zugriff auf die Attribute der äußeren Klasse hat
- Beispiel: **`memberclass2`**

# Lokale Klassen

- Lokale Klassen sind auch innere Klassen, die jedoch nicht als Eigenschaft direkt in einer Klasse eingesetzt werden
- Diese Form der inneren Klasse befindet sich in Anweisungsblöcken von Methoden oder Initialisierungsblöcken
- Lokale Interfaces sind nicht möglich
- Die Deklaration der inneren Klasse ist wie eine Anweisung eingesetzt
- Ein Sichtbarkeitsmodifzierer ist ungültig und die Klasse darf keine Klassenmethoden und allgemeine statische Variablen (Konstanten schon) deklarieren
- Jede lokale Klasse kann auf Methoden der äußeren Klasse zugreifen und zusätzlich auf die lokalen Variablen und Parameter, die mit dem Modifizierer **`final`** als unveränderlich ausgezeichnet sind
- Liegt die innere Klasse in einer statischen Methode, kann sie jedoch keine Objektmethode aufrufen

# Lokale Klassen

- Durch lokale Klassen können schnell mehrmals spezifische Objekte hergestellt werden
- Die Implementation einer eigenen Klasse ist nicht notwendig
- Die Klasse ist nur innerhalb eines Blockes sichtbar
- Beispiel: lokaleklasse1

# Anonyme Innere Klassen

- Anonyme Klassen gehen noch einen Schritt weiter als lokale Klassen
- Sie haben keinen Namen und erzeugen immer automatisch ein Objekt
- Klassendeklaration und Objekterzeugung sind zu einem Sprachkonstrukt verbunden
- Die allgemeine Notation ist folgende:

```
new KlasseOderSchnittstelle() {
    /* Eigenschaften der inneren Klasse */
}
```

# Anonyme Innere Klassen

- In dem Block geschweifter Klammern lassen sich Methoden und Attribute deklarieren oder Methoden überschreiben
- Hinter **`new`** steht der Name einer Klasse oder Schnittstelle
- **`new Klassenname(Argumente) { … }`**
  - Steht hinter **`new`** ein Klassentyp, dann ist die anonyme Klasse eine Unterklasse von Klassenname
  - Es lassen sich mögliche Argumente für den Konstruktor der Basisklasse angeben, den die anonyme automatisch mit super() aufruft
- **`new Schnittstellenname() { … }`**
  - Steht hinter **`new`** der Name eines Interface, erbt die anonyme Klasse von **`Object`** und implementiert das Interface
  - Implementiert sie nicht die Operationen des Interface ist das ein Fehler; man hätte nichts davon, denn dann hätten man eine abstrakte innere Klasse, von der kein Objekt erzeugt werden könnte
- Für anonyme innere Klassen gilt die Einschränkung, dass keine zusätzlichen **`extends`**- oder **`implements`**-Angaben möglich sind. Ebenso sind keine eigenen Konstruktoren möglich

# Anonyme Innere Klassen

- Auch für innere anonyme Klassen erzeugt der Compiler eine normale Klassendatei
- Im Fall einer normalen inneren Klasse wird die Notation **`ÄußereKlasse$InnereKlasse`** gewählt
- Das klappt bei anonymen inneren Klassen nicht mehr, da der Name der inneren Klasse fehlt
- Der Compiler wählt daher folgende Notation für Klassennamen: **`InnerClassName$1`**
- Falls es mehr als eine innere Klasse gibt, werden diese Klassen durchnummeriert

# Anonyme Klassen für Threadimplementationen

- Das Interface **`Runnable`** schreibt die Methode **`run()`** vor
- Will man einen Thread starten, kann die Implementation einer eigenen **`Runnable`** Klasse anonym als innere Klasse erfolgen:

```
new Thread(new Runnable() {
    public void run() {
        for (int i = 0; i < 10; i++)
            System.out.println(i);
    };}).start();
```

- Beispiel: **`producerconsumer2, sleepingbarber`**

# Visitor-Pattern

- Eine Collection kann direkt (oder bei **`Map`** mittels einer Collection-View) mit einem **`Iterator`** durchlaufen werden
- Mittels des Iterators in einer foreach-Schleife durchlaufen wir jedes Element
- Alternativ kann der Programmierer entscheiden wie viele Elemente er durchlaufen möchte
- Eine andere Sichtweise ist der **`Visitor`**
- Einem Besucher der Collection sollen alle Elemente einmal vorgestellt werden

# Visitor-Pattern

- Interface **`Visitable`**:
  - Enthält Methode zum Anliefern eines **`Visitor`**-Objektes bei einer Collection
- Interface **`Visitor`**:
  - Enthält Methode mit der die Collection dem **`Visitor`** ihre Insassen vorstellt
- Eine Collection ist **`Visitable`**, wenn sie Besucher empfangen kann
- Ein Objekt ist ein **`Visitor`**, wenn ihm Objekte vorgeführt werden können

# Visitor-Pattern

- ```
  public interface Visitable {
      /** Erhaelt einen Visitor.
       *  @return true wenn der Visitor immer true
       *          liefert
       */
      boolean visit(Visitor v);
  }
  ```

- ```
  public interface Visitor {
      /** Besucht ein Objekt
       *  Liefert true wenn der Besuch weiter gehen
       *  soll
       */
      boolean visit(Object o);
  }
  ```

# Visitor-Pattern

- Das Interessante am Visitor-Pattern ist, dass der **`Visitor`** selber entscheidet, ob der Besuch weitergehen soll oder nicht
- Ein **`Iterator`** hingegen kann dies nicht entscheiden
- Ein weiterer Vorteil ist, dass der **`Visitor`** austauschbar ist
- Je nach Zweck können eigene **`Visitor`**-Objekte mit unterschiedlicher Funktionalität implementiert werden
- Beispiel: **`visitor1`**

# Zusammenfassung

- **`Collections`**
- **`Queue`**
- **`Stack`**
- Innere Klasen
  - *nested top-level class*
  - *member class*
  - Lokale Klasse
  - Anonyme Klasse
- Implementierung eines **`Iterator`** als *member class*
- Visitor-Pattern

# Ausblick

- Generics
- Innere Klassen