Informatik B

Vorlesung 13

Innere Klassen, Visitor-Pattern, Generics

Rückblick

- **`Collections`**
- **`Queue`**
- **`Stack`**
- Innere Klasen
- Implementierung eines **`Iterator`** als *member class*

Innere Klassen (Wiederholung)

- *nested top-level class*
  - Eine innere Klasse, die mit dem Schlüsselwort **`static`** versehen ist
- *member class*
  - Eine innere Klasse ohne Schlüsselwort **`static`**
- Lokale Klassen
  - Eine Klasse mit Namen, die innerhalb eines Blockes definiert wird
- Anonyme Klassen
  - Eine Klasse ohne Namen, die innerhalb eines Blockes definiert und von der sofort ein Objekt erzeugt wird

Innere Klassen

- Generell führt die Schachtelung von Klassen nicht zu einem Aufbau einer Vererbungshierarchie!!
- Das Schachteln bedeutet lediglich, dass eine Klasse nur in Kombination mit einer anderen (äusseren) Klasse Sinn macht oder das eine Klasse nur einmalig benötigt wird
- Eine Vererbungshierarchie wird weiterhin ganz normal mit dem Schlüsselwort **`extends`** (oder ggf. **`implements`**) aufgebaut

nested top-level class

- Von außen zugängliche *nested top-level classes* verhalten sich genau wie nicht geschachtelte eigenständige Klassen und Interfaces
- Diese Klassen machen dann Sinn, wenn diese nur innerhalb der äußeren Klasse oder nur in Verbindung mit der äußeren Klasse eingesetzt werden sollen
- Obwohl möglich, ist deshalb der Import einer *nested top-level class* ohne ihre äußere Klasse in der Regel nicht sinnvoll

nested top-level class

- Statische innere Klassen und Interfaces können ineinander geschachtelt werden
- Alle statischen inneren Klassen sowie die äußere Klasse können untereinander auf alle statischen Attribute und Methoden zugreifen auch wenn diese **`private`** sein sollten
- Statische Klassen können Subklassen von äußeren sein
- Statische Klassen können parallel liegende innere Interfaces implementieren
- Beispiel: **`nestedtoplevel1`**

member classes

- Eine Instanz einer Member-Klasse ist mit genau einer vorher erschaffenen Instanz der äußeren Klasse verbunden
- Member-Klassen können bis auf Klassenkonstanten (**`static final`** erklärte Attribute) keine statischen Felder oder Methoden besitzen
- Member-Klassen haben Zugriff auf alle Attribute und Methoden der äußeren Klasse, inklusive der *private* deklarierten
- Member-Klassen können ineinander geschachtelt werden
- Member-Klassen können Subklassen von äußeren Klassen sein

member classes

- Member-Klassen sind besonders sinnvoll, wenn Objekte erzeugt werden, die über die Innere Struktur eines anderen Objektes Kenntnisse haben müssen, obwohl alle Datenfelder **`private`** sind
- Ein gutes Beispiel stellt der **`Iterator`** bei den Collections dar

Lokale Klassen

- Eine lokale Klasse ist nur in dem Block gültig, in dem sie deklariert ist und kann weder **`private`**, **`protected`**, **`public`** noch **`static`** sein (macht ja auch keinen Sinn)
- Lokale Klassen haben Zugriff auf **`final`** deklarierte lokale Variablen oder Parameter, die in ihrem Gültigkeitsbereich liegen
- Lokale Klassen können bis auf Konstanten (**`static final`**) keine statischen Felder oder Methoden besitzen
- Lokale Klassen können natürlich Instanzvariablen deklarieren
- Ist eine lokale Klasse in einer statischen Methode deklariert, hat sie Zugriff auf alle statischen Felder und Methoden der Klasse
- Beispiel: **`lokaleklasse1`**
- Ist sie in einer Instanz-Methode deklariert, gelten zusätzlich die Regeln zu Member-Klassen
- Beispiel: **`lokaleklasse2`**

Lokale Klassen

- Lokale Klassen sind einfacher zu verwenden als Member-Klassen, da sie außerhalb des Blocks und damit auch außerhalb der Klasse nicht direkt benutzbar sind
- Das Einsatzgebiet von lokalen Klassen ist weit gesteckt:
  - Lokale Klassen sind ideal für Instanzen geeignet, die nur innerhalb einer Methode benötigt werden
  - Instanzen von lokalen Klassen können außerhalb der Methode, in der die lokale Klasse deklariert ist, verwendet werden

Lebensdauer

- Alle lokalen Variablen werden außerhalb des Blocks, in dem sie deklariert sind, ungültig, d.h. stehen nicht mehr im Zugriff
- Mit dem Verlust der Gültigkeit (*scope*) verlieren lokale primitive Variablen auch ihre Existenz
- Im Allgemeinen stimmen also Gültigkeit und Lebensdauer überein
- Dies gilt aber nicht unbedingt für Instanzen von lokalen Klassen, denn sie können außerhalb des Blocks als Objekte eines anderen Typs überleben
- In diesem Fall ist also die Lebensdauer von lokalen Objekten größer als die Gültigkeit ihrer Klasse
- Beispiel: **`lokaleklasse3`**

Anonyme Klassen

- Anonyme Klassen sind lokale Klassen ohne Namen
- Die Definition einer Anonymen Klasse ist direkt mit der Erzeugung eines Objektes verbunden (*One-Shot*-Klasse)
- Einen idealen Einsatz finden sie bei der Implementierung von Interfaces mit nur einer, höchstens zwei Methoden (z.B. **`Runnable`**)
- Sie unterliegen denselben Regeln wie lokale Klassen, sind aber aufgrund ihrer Namenlosigkeit weiter eingeschränkt
- Eine anonyme Klasse kann keine eigene Konstruktoren definieren
- Zusätzliche Methoden, die nicht in der Superklasse bzw. im Interface deklariert sind, können nicht von außen aufgerufen werden

Anonyme Klassen

- Die Deklaration erlaubt kein **`extends`** oder **`implements`** und erfolgt alternativ durch:
  - **`new SuperClassName ([arguments]) { ... }`**
    wobei die Superklasse einen Konstruktor haben muss, der eine zu den Argumenten passende Signatur hat
  - **`new InterfaceName() { ... }`**
    wobei die anonyme Klasse das Interface vollständig implementieren und direkte Subklasse von Object sein muss
  - Konstruktor-Argumente sind also nicht erlaubt

Anonyme Klassen

- Anonyme Klassen sind gut geeignet Interfaces mit wenigen Methoden zu implementieren
- Voraussetzung sollte sein, dass lediglich ein Objekt benötigt wird
- Beispiel: **`anonymeklasse1`**
- Eine anonyme Klasse kann auch sehr einfach dazu verwendet werden eine Methode zu überschreiben
- Beispiel: **`anonymeklasse2`**

Visitor-Pattern

- Eine Collection kann direkt (bzw. bei **`Map`** mittels einer *Collection-View*) mit einem **`Iterator`** durchlaufen werden
- Mittels eines **`Iterator`** in einer foreach-Schleife durchläuft man jedes Element
- Alternativ kann der Programmierer entscheiden wie viele Elemente er durchlaufen möchte (**`hasNext(), next()`**)
- Eine andere Sichtweise ist der **`Visitor`**
- Einem Besucher der Collection sollen alle Elemente einmal vorgestellt werden

Visitor-Pattern

- Eine Collection soll befähigt werden einen Besucher anzunehmen
- Die Collection selbst bestimmt die Reihenfolge, in der dem Besucher die Insassen der Collection vorgestellt werden
- Der Besucher kann selber mitteilen, dass er seinen Besuch beenden möchte
- Diese Konzept ist ein sehr gängiges Design Pattern
- Dazu gibt es in Java keine vorgefertigten Interfaces, es muss also komplett selbst implementiert 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`**

`Visitor` vs. `Iterator`

- Prinzipiell bieten die beiden Konzepte die gleiche Funktionalität: Durchlaufen einer Collection
- Bei beiden Konzepten entscheidet die Collection selbst, wie sie durchlaufen werden soll
- Der **`Iterator`** muss vom Programmierer aktiv befragt und angewiesen werden
- Der **`Visitor`** entscheidet selber wie lange er die Collection durchlaufen möchte

Generics

- Viele Programmiersprachen bieten die Unterstützung von generischen Typen
- In Java werden generische Typen seit der Version 1.5 (Java5) unterstützt
- Es handelt sich dabei um parametrisierte Typen
- Es existieren also Platzhalter mit denen erst zu einem späteren Zeitpunkt festgelegt wird, für welchen Typ sie stehen

Nutzen von Generics

- Generische Typen werden benötigt, um Datenstrukturen effizient implementieren zu können
- Will man eine Collection implementieren, stellt man fest, dass die Implementierung weitgehend unabhängig vom Typ der Elemente ist, die in der Collection abgelegt werden
- Man möchte deshalb idealerweise nur eine einzige Implementierung der Collection zur Verfügung stellen, die dann für das Ablegen beliebiger Elementtypen verwendet werden kann
- Ohne die generischen Typen erreicht man das, indem die Collection-Klassen Referenzen vom Typ **`Object`** verwalten
- Da **`Object`** die Superklasse aller Java-Klassen ist, kann eine Collection dann Objekte beliebigen Typs enthalten

Nutzen von Generics

- Eine Eigenschaft der so implementierten Collections ist, dass sie nicht notwendig Elemente desselben Typs enthalten
- Eine Collection kann also eine Mischung von Objekten unterschiedlichen Typs verwalten
- Beim Herausholen eines Objektes aus einer Collection weiß man daher zunächst nicht von welchem Typ das betreffende Element ist
- Um das Element verwenden zu können, muss erst ein Cast durchgeführt werden, was zu einer **`ClassCastException`** führen kann
- Bei diesen Problemen helfen generische Typen

Definition eines generischen Typs

- Soll innerhalb einer Klasse ein generischer Typ verwendet werden, so muss dieser definiert werden, dies geschieht bei der Klassendefinition in spitzen Klammern: **`class Klassenname<T> { … }`**
- Der Typ **`T`** kann nun innerhalb der Klasse verwendet werden
- Ebenso kann ein Typ für ein Interface definiert werden: **`interface Interfacename<T> { … }`**
- **`T`** stellt einen Platzhalter für einen beliebigen Referenztyp dar
- **`T`** wird auch als *formal type parameter* bezeichnet

Beispiel

- Folgende nicht typsichere Klasse sei gegeben:

```
public class Box {
    private Object object;

    public void add(Object object) {
        this.object = object;
    }
    public Object get() {
        return object;
    }
}
```

- Objekte dieser Klasse können beliebige Objekte verwalten

Beispiel

- Die Klasse kann einfach mit einem generischen Typ versehen werden:

```
public class Box<T> {
    private T t;
    public void add(T t) {
          this.t = t;
    }
    public T get() {
          return t;
    }
}
```

- Eine Instanz dieser Klasse kann nun typsicher erzeugt werden mit:
  `Box<String> b = new Box<String>();`
- Beispiel: **`generics1`**

Namenskonvention

- Im allgemeinen sollten die folgenden Buchstaben für generische Typen verwendet werden:
  - **`E`** - Element (ausgiebig im Collection Framework genutzt)
  - **`K`** – Key
  - **`N`** - Number
  - **`T`** - Type
  - **`V`** - Value
  - **`S`**, **`U`**, **`V`** etc. - 2nd, 3rd, 4th types

Wildcards

- Bei der Definition eines generischen Typs kann ein Wildcard angegeben werden
- Als Wildcard ist nur das **?** zulässig
- Es steht für einen beliebigen Typ
- Mit Hilfe des **?** Können weitere Einschränkungen der Typen z.B. innerhalb einer Collection vorgenommen werden

Bounds

- Typeinschränkungen können bei der Implementierung einer Klasse (Definition eines generischen Typs) oder bei der Deklaration einer Referenz erforderlich sein
- Nehmen wir an, wir implementieren ein **`SortedSet`**
- In diese Klasse dürfen wir nur Objekte einfügen, die sich sortieren lassen
- Wie kann man diesen Fall mit einem generischen Typ erfassen?

Bounds

- Für diesen Fall gibt es die *Bounds*
- Ein Typparameter kann einen oder mehrere so genannte *Bounds* haben
- *Bounds* sind Klassen oder Interfaces, von denen der unbekannte Typ abgeleitet sein muss
- Der Compiler prüft dann, ob der konkrete Typ, der für den Typparameter später eingesetzt wird, zu den angegebenen *Bounds* passt
- Unterschieden werden:
  - *upper bound* (**`extends`**)
  - *lower bound* (**`super`**)

Upper Bound

- Ein *upper bound* gibt eine obere Grenze innerhalb der Vererbungshierarchie an
- Sollen für einen generischen Typ nur eine bestimmte Klasse (oder ein Interface) und deren Unterklassen zugelassen werden, kann folgendes deklariert werden:
  `T extends Oberklasse`
  `T extends Comparable<T>`
- Es können auch mehrere Bounds angegeben werden
- Diese werden mit einem **`&`** miteinander Verknüpft:
  `T extends Oberklasse & Comparable<T>`
- Da es in Java keine Mehrfachvererbung gibt, macht die Verknüpfung von *Bounds* nur bei Interface-Typen Sinn
- Beispiel: **`bounds1`**

Lower Bound

- Neben einer Obergrenze gibt es auch eine Untergrenze
- Diese Untergrenze kann bei der Deklaration einer Referenz angegeben werden
- Bei der Definition eines generischen Typs ist ein *Lower Bound* nicht erlaubt
- Eine Referenz vom Typ **`List<? super Long>`** darf also nur auf Objekte vom Typ **`List<Long>`** oder **`List<Number>`** oder **`List<Comparable>`** usw. verweisen

Generische Methoden

- Ebenso wie bei der Definition einer Klasse kann auch bei einer Methode (oder einem Konstruktor) ein generischer Typ definiert werden

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

- Hierbei definiert `<U>` einen neuen generischen Typ, der im Beispiel dazu verwendet wird, um die beiden Parameter konsistent zu halten
- Der Aufruf dieser Methode kann wie folgt aussehen:

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

Zusammenfassung

- Innere Klassen
  - nested top-level class
  - member class
  - Lokale Klasse
  - Anonyme Klasse
- Visitor-Pattern
- Generics

Ausblick

- Reflections
- Serialisierung
- Autoboxing
- Netzwerkprogrammierung