STRUKTURIERTE UND OBJEKTORIENTIERTE PROGRAMMIERUNG

HINTERGRUNDINFORMATIONEN

Simon Gebert – E-Mail: s.gebert@kant.de. – Januar 2021

Inhaltsverzeichnis

Einleitung

Bildungsplanbezug

Das vorliegende Material soll exemplarisch Ideen zur Umsetzung der Bildungsstandards Informatik im Leistungsfach der Kursstufe zum Thema: „Strukturierte und objektorientierte Programmierung“ skizzieren. Der Fokus liegt dabei auf folgenden Kompetenzen:

(14) generische Datentypen bei der Instanziierung vorgegebener *Klassen* verwenden

(15) sprachliche Mittel zur Sammlung gleichartiger Objekte (Collection) und zum Durchlaufen aller Elemente der Sammlung[1] (zum Beispiel Iteration, foreach, Lambdaausdrücke etc.) nutzen

Von den Lernenden wird in beiden Kompetenzen lediglich die Nutzung verlangt, ohne näheres über die Implementierung der generischen Datentypen und der ebenfalls generischen Sammlungen zu kennen. Dies ist nur dann möglich, wenn eine Beschreibung dieser vorliegt. Anstatt diese den Schülerinnen und Schülern vorzugeben, bietet sich die offizielle Dokumentation der grundlegenden APIs der Java-Plattform an. Dies ist ganz im Sinne folgender Kompetenz im selben Abschnitt des Bildungsplans:

(22) Dokumentationen zu gegebenem Quellcode und Bibliotheken nutzen

Entsprechend befähigt die Kompetenz (22) gerade dazu (14) und (15) zu erlernen.

1 Es gibt zwei Arten zusammengesetzter Datentypen zur Speicherung gleichartiger Objekte. *Arrays* und *Collections* wie Listen, Bäume, Queues, Stacks, etc. Da der Bildungsplan bei Arrays von „Speicherung und Verarbeitung gleichartiger Daten“ (Brückenkurs 3.1.2 Algorithmen) spricht und bei „sprachliche Mittel zur Sammlung gleichartiger Objekte“ (15) explizit den Begriff *Collection* angibt, werden Arrays hier nicht zu den Sammlungen gezählt.

Einbindung in den Unterricht

Dieses Dokument liefert nun die Hintergründe zu den oben genannten Kompetenzen und basiert größtenteils auf der offiziellen Sprachspezifikation[2] und der Dokumentation der grundlegenden APIs[3] insbesondere `java.util` und `java.util.function`. Erstere enthält das Collections Framework, letztere funktionale Interfaces, welche für Lambdaausdrücke benötigt werden.

Es gibt zu dem Thema viele offizielle wie inoffizielle Erklärungen und Anleitungen in englisch wie auch auf deutsch. Allerdings werden Fachbegriffe oft unterschiedlich und teils auch ungenau bzw. falsch ins deutsche übersetzt. Bei der Verwendung dieser Quellen und Begriffe im Unterricht können daher Fehlvorstellungen entstehen.

So wird beispielsweise im Zusammenhang mit Methoden der Begriff „actual parameter“ teils mit „aktueller Parameter“ übersetzt. „Aktuell“ suggeriert, dass sich der Parameter mit der Zeit ändern kann. Genau genommen kann jedoch nur ein konkreter Wert beim Methodenaufruf übergeben werden, ohne dass sich der Parameter (benannter Wertespeicher) selbst ändert. Im Methodenkontext ist diese Ungenauigkeit nicht weiter schlimm, da der Wert von der Methode selbst manipuliert werden darf. Wird allerdings im Kontext generischer Typen „actual type parameter“ mit „aktueller Typparameter“ übersetzt, könnte der Eindruck entstehen, dass es sich hier ebenfalls um einen Wert handelt, welcher nach Instanziierung einer generisch definierten Klasse von dieser verändert werden kann. Dies ist aber nicht der Fall. Zur klaren Abgrenzung verwendet dieses Dokument daher den Begriff „Argument“ für manipulierbare Werte oder Objekte und „Parameter“ für benannte Wertespeicher. Bei einer Abstraktion wird ‚formal‘ vorangestellt, bei einer Konkretisierung ‚konkret‘. Die genauen Begriffsdefinitionen finden sich im entsprechenden Abschnitt (Seiten 7,12).

Das Hintergrunddokument ist für die Lehrkraft gedacht. Es behandelt weit aus mehr, als im Unterricht mit den Schülerinnen und Schülern behandelt werden kann und muss. Vielmehr soll es das Wichtigste für die Lehrkraft präzise und knapp auf deutsch zusammenfassen und sicherstellen, dass eine korrekte und einheitliche Fachsprache verwendet wird.

In den Dokumenten „02_Stoffverteilungsplan.odt“ und „03_Unterrichtsverlauf.odt“ wird ein möglicher Unterrichtsgang mit Aufgaben skizziert. Dieser stellt klar, welche Inhalte tatsächlich im Unterricht behandelt werden sollen und verweist auf entsprechende Stellen des Hintergrunddokuments, die für eigenes Unterrichtsmaterial geeignet sind.

Der Unterrichtsgang setzt voraus, dass die Grundlagen der Objektorientierung (Klasse, Attribute, Methoden mit Parametern und Rückgabewerten, Datentypen und Typumwandlung, Methodensignatur, Zugriffsmodifikation insbesondere Setter/Getter, Kapselung, Konstruktoren und Instanziierung) bereits bekannt und eingeübt sind.

Es bietet sich an, die Einheit vor der Unterrichtseinheit zu ADTs einzuschieben, da dort die Implementierung einiger ADTs generisch erfolgen kann. Im Graphentester der Unterrichtseinheit „Graphen und ihre Algorithmen“ können Lambdaausdrücke verwendet werden. Sollen Lambdaausdrücke unterrichtet werden, dann am besten vor dieser Einheit.

2 *https://docs.oracle.com/javase/specs/jls/se11/html/index.html* (ausgewertet am 22.12.2020)

3 *https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html* (ausgewertet am 22.12.2020)

Technische Hintergründe zur Programmiersprache Java

Zur Umsetzung der Bildungsstandards wird in diesem Material ausschließlich die Programmiersprache Java verwendet. Für das Verständnis generischer Programmierung und Sammlungen in Java wird zunächst auf Programmierparadigmen im allgemeinen und anschließend auf die Umsetzung in Java eingegangen. Um die Funktionsweise generischer Datentypen und Lambdaausdrücke nachzuvollziehen, wird auf die Typisierung in Java und Schnittstellen näher eingegangen.

Programmierparadigmen

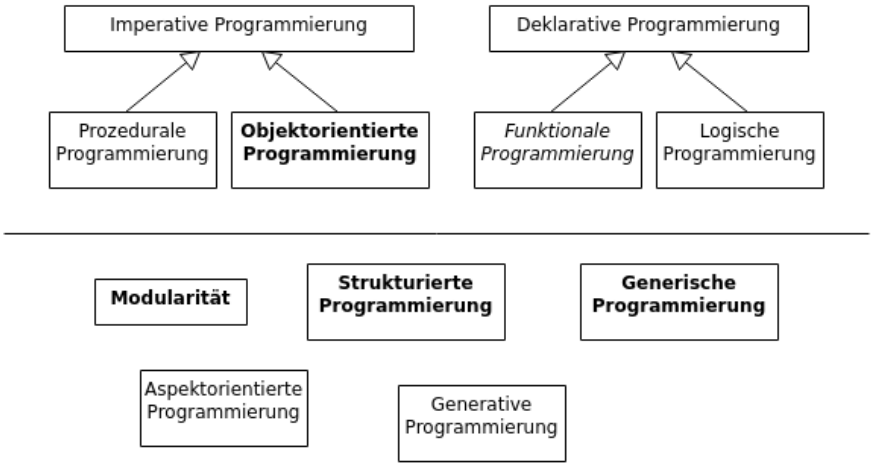

Abbildung 1: Programmierparadigmen

Es gibt verschiedene grundlegende Prinzipien der Programmierung, die als Programmierparadigmen bezeichnet werden[4]. Hierzu gehören auch strukturierte und objektorientierte Programmierung. Jedes Paradigma fordert bestimmte sprachliche Mittel und stellt Algorithmen durch verschiedene Notationsformen dar. Durch die in einer Programmiersprache zur Verfügung stehenden sprachlichen Mittel, wird das Programmieren nach einem oder mehreren Paradigmen ermöglicht oder ausgeschlossen.

Java ist eine strukturierte, objektorientierte, modulare Programmiersprache, die zudem generische Programmierung ermöglicht und Elemente funktionaler Programmierung bereit stellt.

Tabelle 1: Auswahl an Programmierparadigmen und ihrer sprachlichen Mittel/Notationsformen; Themen aus dem Bildungsplan 2016 sind fett hervorgehoben

Paradigma	Sprachliche Mittel	Notationsformen / *Beispiele*
Imperative Programmierung	Zuweisungen (Speichervariablen), Kontrollstrukturen: Schleifen	**Nassi-Shneiderman-Diagramme/Struktogramme** ***Assembler***
Prozedurale Programmierung	Lesbar benannte Teilprogramme/Prozeduren, Daten und Objekte verarbeitende Prozeduren getrennt;	*C, Pascal*
Objektorientierte Programmierung	Objekte mit Daten und darauf arbeitenden Methoden als eine Einheit.	UML Diagramme: z.B. **Klassendiagramme**, Objektdiagramme *Java, Python,Snap!, Scratch*

4 *https://www.itwissen.info/Programmierparadigma-programming-paradigm.html* (ausgewertet am 22.12.2020)

Paradigma	Sprachliche Mittel	Notationsformen / *Beispiele*
Deklarative Programmierung	Kontrollstrukturen: **Rekursion und Substitution („Aufruf“)**	*SQL*
Funktionale Programmierung	Beschreibung durch Funktionen (mit Funktionsvariablen) die auf Daten arbeiten. **Listen** (map-filter-reduce)**, Lambda,** Closures Objekte werden nicht verändert, sondern neu erstellt. Begünstigt Parallelisierung	Teile von UML: Aktivitätsdiagramme, Zustandsdiagramme *LISP-basierte Sprachen,* ECMAScript (Java-Script) *teils in Java: Funktionale Interfaces,* ***Lambdaausdrücke***
Modulare Programmierung	Module Bei OOP: **ADTs** (Daten und Funktionen zur Behandlung dieser Daten als Einheit)	
Strukturierte Programmierung	Lesbarer Code durch Kontrollstrukturen: Sequenz, Verzweigung, Schleifen Gültigkeitsbereich von Variablen (Scope)	
Generische Programmierung	Algorithmen für mehrere Datentypen verwendbar machen. **Generische Datentypen**	*In Java:* ***Collections Framework***

Typisierung

Jeder Ausdruck in einem Java Programm ergibt entweder kein Ergebnis (void) oder einen Wert eines Typs, der zur Laufzeit abgeleitet werden kann. Ein Ausdruck muss mit dem Typ des Kontexts, in dem er erscheint (**Zieltyp**), kompatibel sein. Ansonsten kommt es zu einem **Kompilierungsfehler**. Der Typ eines Ausdrucks kann abhängig vom Zieltyp sein, das heißt je nach Kontext, in dem der selbe Ausdruck vorkommt, wird vom Compiler ein anderer Typ ermittelt („abgeleitet“). Das „+“ kann zum Beispiel je nach Kontext die Addition von z.B. Integer-Werten oder die Konkatenation von String-Werten bedeuten. Man spricht von:

- **Typinferenz**: Typen werden entsprechend ihrem Zieltyp automatisch abgeleitet.
- **Typumwandlung**: Stimmt der Typ des Ausdrucks nicht mit dem Zieltyp überein, kann manchmal ohne Kompilierungsfehler eine implizite Umwandlung in den Zieltyp statt finden.
- **Typkonvertierung**: Der Zieltyp kann vom Programmierer z.B. mit dem „type cast“-Operator explizit vorgegeben werden. Bei Abweichung vom Zieltyp findet eine explizite Umwandlung in den Zieltyp statt.
- **Typdeklaration:** Der Kontext mancher Ausdrücke kann definiert werden. So wird bei der Deklaration einer Variable oder eines Parameters der Zieltyp für jedes weitere Vorkommen des Ausdrucks festgelegt und dann durch Typinferenz abgeleitet.

Bei manchen Typumwandlungen finden Typüberprüfungen oder Typübersetzungen erst zur Laufzeit statt. Kommt es dabei zu einem **Laufzeitfehler**, wird eine Ausnahme (engl. exception) geworfen.

Eine genaue Beschreibung möglicher Ausdrücke, Kontexte, Typen und Umwandlungen bietet die Java® Language Specification[5].

Eine Typumwandlung kann erweiternd (engl. widening) oder einschränkend (engl. narrowing) sein. Bei einer erweiternden Umwandlung geht sicher keine Information verloren, die Umwandlung kann implizit statt finden. Bei einer einschränkenden Umwandlung kann Information verloren gehen, hier muss eine explizite Konvertierung vorgenommen werden.

Beispiel:

```
int i = 12.5f;  // einschränkend, weil 12.5 nicht verlustfrei als
                // int dargestellt werden kann → Kompilierungsfehler
int i = (int)12.5f;   // explizite einschränkende Typumwandlung;
                // 12.5 wird auf 12 abgeschnitten.
float f = i; // implizite erweiternde Typumwandlung, weil jeder int-
                // Wert verlustfrei als float dargestellt werden kann.
```

Bei primitiven Datentypen findet eine Typumwandlung nach in der Sprachspezifikation definierten Regeln statt. Bei der Umwandlung von Referenz-Typen muss eine Verwandtschaft gegeben sein. Als Ausnahme sind hier in Java lediglich die Umwandlung primitiver Typen in einen String-Typ möglich sowie das so genannte Autoboxing, also die Umwandlung primitiven Zahldatentypen (int, short usw) in eine Instanz der jeweiligen Wrapper-Klasse (bspw. int zu Integer).

Beispiel:

```
String s = "Die Zahl lautet " + i; // Umwandlung int nach String
Integer intObjekt = i; // Autoboxing: Umwandlung int nach Integer
```

Bei Referenztypen ist eine Umwandlung erweiternd, wenn die Umwandlung von einem Subtyp in die Superklasse geschieht. Die Umwandlung von Referenztypen in ihren Basistyp `Object` ist immer möglich. Eine Umwandlung ist einschränkend, wenn sie aus Sicht einer Superklasse in die erweiterte Klasse erfolgen soll. Vor allem die einschränkende Konvertierung von Referenztypen birgt die Gefahr von Laufzeitfehlern. Es ist dann oft eine Ausnahmebehandlung im Code notwendig.

Im Kontext von Methoden wird der Zieltyp für formale Parameter bei der Deklaration angegeben. Beim Methodenaufruf ist der Zieltyp dann durch Typinferenz festgelegt. In Tabelle 2 sind einige Fachbegriffe im Methodenkontext festgelegt, wie sie in diesem Dokument verwendet werden.

Wichtig zu erwähnen ist, dass in Java Argumente immer nach dem Prinzip Pass-By-Value übergeben werden, dass heißt es wird immer eine Kopie des konkreten Argumentes an die Methode übergeben. Das Original im aufrufenden Kontext kann nicht durch die Methode geändert werden. Dies gilt auch für Referenztypen. Hier wird allerdings eine Kopie der Referenz übergeben[6]. Man könnte das Prinzip daher auch Pass-Reference-By-Value nennen.

5 *https://docs.oracle.com/javase/specs/jls/se11/html/jls-15.html* (ausgewertet am 22.12.2020)

6 Eine gute Erklärung für Pass-Reference-By-Value: *https://www.codejava.net/java-core/the-java-language/java-variables-passing-examples-pass-by-value-or-pass-by-reference* (ausgewertet 3.2.2021)

Tabelle 2: Fachbegriffe zu Wertparametern in Methoden

Beispiel:

```
public void printN( int n, String s){
    for( int i = 0; i < n; i++){
        System.out.println( s );
    }
}

printN( 5, "konkret" );
```

Begriff	Beispiel
Typ (engl. type)	`int` `String`
formale (Wert)Parameter (engl. formal value parameter)	`n` `s`
konkrete Argumente (engl. actual argument)	`5` Referenz auf das String-Objekt `"konkret"`

Schnittstellen – Interfaces

Im Allgemeinen stellt eine Schnittstelle eine nach außen bekannte Beschreibung einer Blackbox dar, über die mit der Blackbox kommuniziert werden kann. Es ist dabei unerheblich, wie diese Blackbox intern Informationen verarbeitet. So definiert die Java-Spezifikation in Module und Pakete untergliederte Programmierschnittstellen (API: engl. application programming interface). Das Collections-Framework ist zum Beispiel Teil des Paktes `java.util`, des Moduls `java.base` und beschreibt neben konkreten, wie abstrakte Klassen auch Algorithmen.[7] In der objektorientierten Programmierung gibt eine Schnittstelle an, welche Methoden in einer Klasse vorhanden sind, bzw. bei der Implementierung vorhanden sein müssen. Knapp gesagt eine Schnittstelle ist der public-Teil einer Klasse.

Die Programmiersprache Java bietet mit dem sprachlichen Mittel eines `Interface` eine Möglichkeit eine solche Schnittstelle zu definieren. Im Englischen ist der Begriff für eine allgemeine Schnittstelle (im Sinne des public-Teils einer Klasse) der selbe wie für das sprachliche Mittel. Im Deutschen lässt sich dies jedoch differenzieren. In diesem Dokument wird für das sprachliche Mittel der Begriff *Interface* verwendet, ansonsten der Begriff *Schnittstelle*.

Interfaces ermöglichen es dem Programmierer, so wie er es mit Klassen tut, selbst Typen zu definieren. Interfaces geben wie abstrakte Klassen die minimale Typsignatur für alle Subtypen vor, mit dem Unterschied selbst keine Implementierung (Ausnahme: `static` und `default` Methoden)[8] zu besitzen. Von einem Interface kann wie von einer abstrakte Klassen keine Instanz erzeugt werden.

Ein Interface wird wie folgt deklariert:

```
Modifikator interface Schnittstelle
{
    final Typ konstante = Wert;
    Modifikator Rückgabetyp methode( Parameterliste );
}
```

7 *https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/doc-files/coll-reference.html* (ausgewertet 3.2.2021)

8 Default-Methoden ermöglichen Rückwärtskompatibilität, Static-Methoden werden wie statische Methoden in regulären Klassen mit dem Interface anstatt einem Objekt assoziiert und daher von allen Objekten die das Interface implementieren geteilt. Auf beide wird hier nicht weiter eingegangen. Für eine genauere Beschreibung mit Beispielen siehe *https://docs.oracle.com/javase/tutorial/java/landl/defaultmethods.html* (ausgewertet am 3.2.2012)

Das Interface besitzt nur Methodensignaturen und Konstanten. Bei den Methoden kann der Modifikator **`static`** oder **`default`** sein. Die Methoden sind automatisch **`public`**, **`private`** ist nicht erlaubt.

Um einen Subtyp zu deklarieren geht man wie folgt vor:

```
Modifikator class Klasse extends Superklasse implements Schnittstelle
{
    public Rückgabetyp methode( Parameterliste )
    {
        // Anweisungen
    }
}
```

Mit dem Schlüsselwort **`implements`** wird angegeben, welche Interfaces die Klasse implementiert. Dies können anders als beim Erweitern einer Klasse, auch mehrere sein (mit Komma getrennt hintereinander aufgelistet). Implementiert eine Klasse ein Interface, muss sie alle Methoden, die von der Schnittstelle vorgegeben sind, implementieren. Diese sind per Definition **`public`**. Es kann hierbei nicht zu Konflikten kommen, da jedes Interface nur Signaturen und keine Implementierungen vorgibt.

Interfaces können auch, wie Klassen, eine Vererbungshierarchie besitzen. Dort gelten die selben Regeln wie bei Klassen.

Der große Vorteil von Interfaces liegt darin, dass sie flexibler („Mehrfachimplementierung“) als abstrakte Klassen („keine Mehrfachvererbung“) sind und damit bestens geeignet einen Zieltyp mit klarer Minimalsignatur vorzugeben. Allerdings muss die komplette Signatur im Gegensatz zur Erweiterung abstrakter Klassen bei der Implementierung eines Interfaces erneut aufgeschrieben werden. Als Ersatz für abstrakte Klassen sind Interfaces daher nicht gedacht.

Für das Verständnis generischer Sammlungen und Lambdaausdrücke ist es unverzichtbar, im Unterricht Interfaces zu thematisieren, auch wenn diese nicht explizit im Bildungsplan erwähnt werden. Für die Schülerinnen und Schüler ist es aber kein weiter Weg, Interfaces zu verstehen. Im Unterrichtsverlauf müssen sie lediglich Interfaces als Zieltyp identifizieren und selbst keine definieren oder implementieren.

Generische Datentypen

Notwendigkeit für generische Datentypen anhand eines Beispielprojekts

Eine der fundamentalen Idee der Informatik ist die Abstraktion zur Vermeidung von Duplikaten. Entsprechend gilt das Mantra:

> „Wenn Du zwei Programmteile siehst, die sich nur an wenigen Stellen unterscheiden und die inhaltlich verwandt sind, abstrahiere![9]“

Werkzeuge der strukturieren, objektorientierten Programmierung sind hierfür beispielsweise Klassen, Methoden und Schleifen. Aber selbst unter weitestgehender Nutzung dieser Mittel kann es dazu kommen, dass es Methoden bzw. Klassen gibt, die sich nur in ihrer Signatur und der darin verwendeten Typen unterscheiden. Häufig findet man solche Doppelungen bei Sammlungen gleichartiger Objekte.

9 Michael Sperber, Herbert Klaeren; Schreibe Dein Programm! – Einführung in die Programmierung, 24.Juli.2020, *https://www.deinprogramm.de/sdp/*

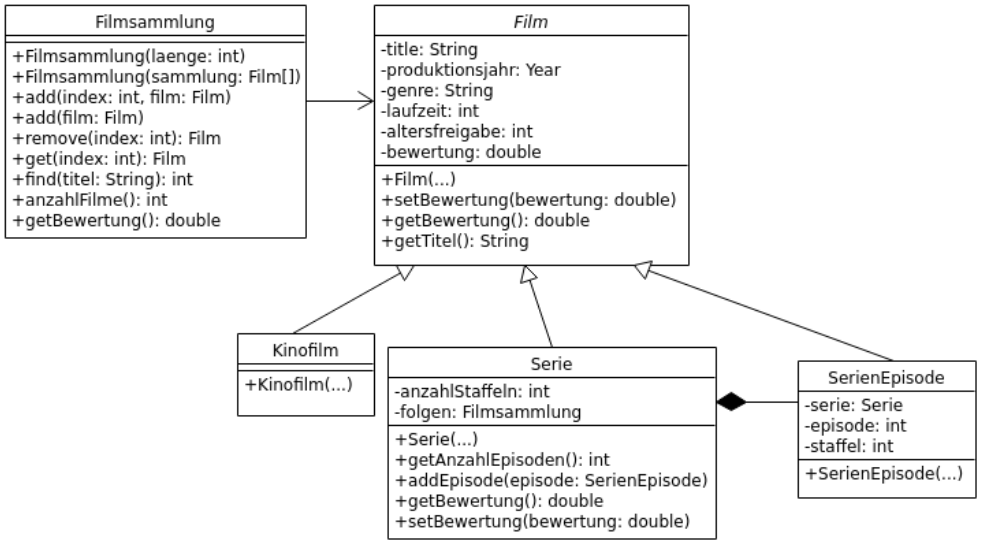

Abbildung 2: Klassendiagramm des Projektes „Filmsammlung“

Als Beispiel dient uns eine `Filmsammlung` (betrachte das zugehörige Klassendiagramm in Abbildung 2), die sowohl `Kinofilme` als auch `Serien` erfasst. Eine `Serie` ist dabei wiederum eine Sammlung von `SerienEpisoden`, deren Anzahl bestimmt werden kann. Sowohl `Filmen` als auch `SerienEpisoden` können eine Bewertung bekommen. Die Bewertung einer `Serie` bestimmt sich aber aus den Bewertungen der enthaltenen `SerienEpisoden`.

Die Filmsammlung kann nun auf einfachste Weise als eine Reihung implementiert werden, in die Filme eingetragen werden können. Durch Polymorphie ist sowohl ein `Kinofilm`, eine `Serie`, als auch eine `SerienEpisode` ein `Film`. Instanzen dieser Klassen können demnach problemlos einer Filmsammlung hinzugefügt werden. Im Begleitmaterial ist eine solche einfache Implementierung mitgegeben.

Für die folgenden Erläuterungen werden als Beispielobjekte die Serien `s1` und `s2` mit Episoden `e1` bis `e4` genutzt. In Abbildung 3 sind diese in einem Objektdiagramm dargestellt.

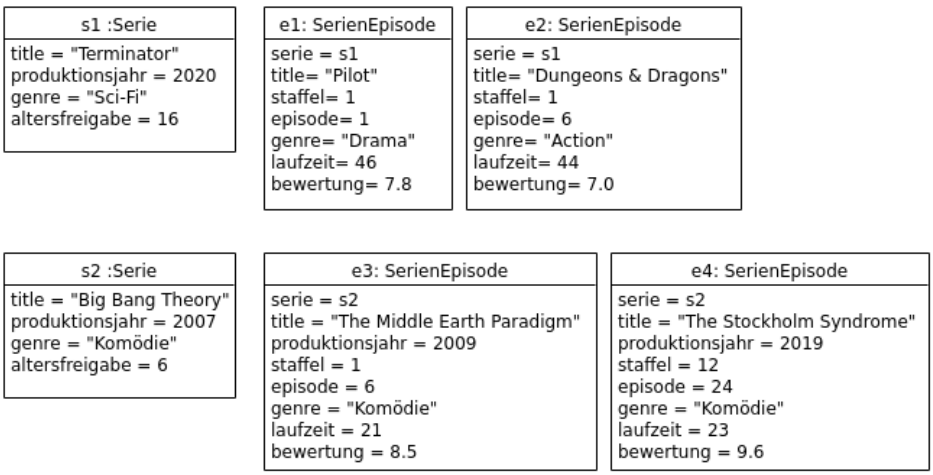

Abbildung 3: Beispielobjekte in einer Filmsammlung

Betrachte nun folgenden gekürzten Ausschnitt eines Programms

```
1|  Filmsammlung fs = new Filmsammlung( new Film[]{s1, s2, e1} );
2|  fs.get(1).setBewertung( 0.0 );
3|  ...println( fs.get(0).getTitel() );
4|  ...println( fs.get(0).getAnzahlEpisoden() );
5|  ...println( ( (Serie) fs.get(0) ).getAnzahlEpisoden() );
6|  ...println( ( (Serie) fs.get(2) ).getAnzahlEpisoden() );
```

- In Zeile 2 wird durch Polymorphie die überschriebene Methode der Serie aufgerufen. Dort kann sichergestellt werden, dass die Bewertung nicht verändert wird.
- In Zeile 3 findet eine implizite erweiternde Typumwandlung (`Serie` → `Film`) statt. Es wird der Titel der Serie ausgegeben.
- Zeile 4 führt zu einem Kompilierungsfehler, da eine einschränkende Umwandlung (`Film` → `Serie`) durchgeführt werden müsste, die in Java eine explizite Typkonvertierung erfordert.
- Damit die Anzahl der Episoden ausgegeben werden kann, muss in Zeile 5 die Typkonvertierung (`Film` → `Serie`) angegeben werden. Es findet dann eine explizite einschränkende Typumwandlung statt. Da es sich bei `s1` tatsächlich um eine `Serie` handelt, wird die Anzahl Episoden wie gewünscht ausgegeben.
- Wenn allerdings wie in Zeile 6 eine Typkonvertierung (`Film` → `Serie`) auf einen falschen Subtyp angewandt wird, kommt es zu einem Laufzeitfehler: Es wird eine ClassCastException geworfen. Das Objekt `e1` ist eine `SerienEpisode`, welche nicht in direkter Verwandtschaft zu einer `Serie` steht.

Der Laufzeitfehler bei der Typkonvertierung kann vermieden werden, indem man für Serienepisoden eine eigene neue Klasse `Episodensammlung` schreibt. Diese unterscheidet sich dann von der `Filmsammlung` nur in der Signatur (siehe Abbildung 4). Die Implementierung wäre nahezu identisch.

Filmsammlung
+Filmsammlung(laenge: int) +Filmsammlung(sammlung: Film[]) +add(index: int, film: Film) +add(film: Film) +remove(index: int): Film +get(index: int): Film +find(titel: String): int +anzahlFilme(): int +getBewertung(): double

Episodensammlung
+Episodensammlung(laenge: int) +Episodensammlung(sammlung: SerienEpisode[]) +add(index: int, film: SerienEpisode) +add(film: SerienEpisode) +remove(index: int): SerienEpisode +get(index: int): SerienEpisode +find(titel: String): int +anzahlEpisoden(): int +getBewertung(): double

Abbildung 4: Filmsammlung und Episodensammlung: beinahe identisch

Mit Version 5 der JDK wurde Java um generische Datentypen erweitert. Diese ermöglichen es, bei der Programmierung einer Klasse von den enthaltenen Datentypen zu abstrahieren, sich bei der Nutzung aber trotzdem auf einen Typ zu beschränken.

Einfache generische Typen deklarieren

Zum Verständnis der Syntax folgt ein kleiner Auszug aus den Definitionen der Schnittstellen `List` und `Iterator` im Paket `java.util`:

```
public interface List <E> {
    void add(E x);
    Iterator<E> iterator();
}

public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Neu in diesem Code sind die Angaben in den spitzen Klammern. Damit werden die **formalen Typparameter** der beiden Schnittstellen `List` und `Iterator` deklariert.

Formale Typparameter können, bis auf wenige Ausnahmen, überall dort verwendet werden, wo man normale Typen verwenden würde.

Wird die `Filmsammlung` auf diese Weise parametrisiert, sieht das Klassendiagramm aus wie Abbildung 5. Vergleicht man mit der Signatur des generischen Typs `ArrayList<E>` aus dem Paket `java.util`, fällt auf, dass `FilmsammlungGeneric<E>` lediglich andere Konstruktoren und wenige zusätzliche Methoden hat, ansonsten aber die selbe Funktionalität fordert. Bei der Implementierung erweitert man daher am besten einfach den generischen Typ.

```
public class FilmsammlungGeneric<E> extends ArrayList<E>
```

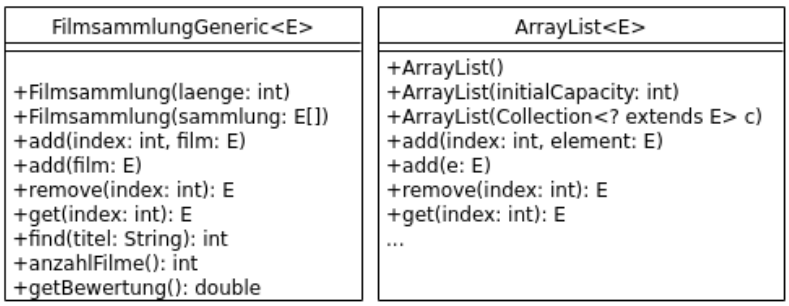

Abbildung 5: Klassendiagramm der generischen Typen `FilmsammlungGeneric<E>` und `ArrayList<E>`

Es ist auch möglich mehrere formale Typparameter zu deklarieren, diese werden dann innerhalb der spitzen Klammern Komma-getrennt aufgelistet.

```
public interface Map<K,V>{…
```

Einfache generische Typen nutzen

Mit Hilfe selbst deklarierter oder bereits in Bibliotheken vorhandener generischer Typen kann bereits im Programmcode vorgegeben werden, welche konkreten Typparameter erlaubt sind. Fehler werden dann bereits bei der Kompilierung erkannt und es kommt nicht zu einem Laufzeitfehler, wie zuvor im Beispiel erläutert. Dadurch wird Typsicherheit gewährleistet.

Bei der Nutzung generischer Typen wird analog der Nutzung normaler Typen vorgegangen.

```
1|  ArrayList<Integer> l1 = new ArrayList<Integer>();
2|  ArrayList<Integer> l2 = new ArrayList<>(); // Typinferenz
3|  ArrayList<Integer> l3 = new ArrayList(); // Laufzeitfehler möglich
4|  ArrayList l4 = new ArrayList<Integer>(); // Laufzeitfehler möglich
5|  ArrayList<int> l5 = new ArrayList<int>(); //Kompilierungsfehler
```

Dabei sollte immer die Syntax wie in Zeile 1 oder 2 verwendet werden. Wird wie in Zeile 3 und 4 nur der **Originaltyp** (ohne spitze Klammern) notiert, dann wird Typsicherheit nicht bei der Kompilierung überprüft sondern zur Laufzeit verlagert. Das Verhalten ist dann identisch zum nicht parametrisierten Originaltyp. Es können Laufzeitzeitfehler an Stellen im Programmcode auftreten, an denen das jeweilige Objekt verwendet wird.

Zeile 2 ist erlaubt, da hier durch Typinferenz der **konkrete Typparameter** (in den spitzen Klammern) abgeleitet wird. Man nennt die zwei spitzen Klammern ohne konkreten Typparameter auch den **Diamant-Operator <>**. Dieser kann immer dann verwendet werden, wenn Typinferenz möglich ist.

Zeile 5 schlägt fehl, da in generischen Typen nur Referenztypen als konkrete Typparameter erlaubt sind. Primitive Datentypen wie `int` müssen zunächst durch Boxing in einen Referenztyp gepackt werden.

Betrachte nun den kurzen Programmausschnitt, diesmal mit generischem Datentyp

```
1|  FilmsammlungGeneric<Serie> serien =
     new FilmsammlungGeneric<>( new Serie[]{s1, s2} );
2|  serien.get(1).setBewertung( 0.0 );
3|  ...println( serien.get(0).getTitel() );
4|  ...println( serien.get(0).getAnzahlEpisoden() );
```

Zeile 2 funktioniert nach wie vor mit dem selben Ergebnis.

Auch Zeile 3 liefert das gleiche Ergebnis, allerdings ist keine Typumwandlung notwendig. Die von der Superklasse geerbte Methode wird aufgerufen.

In Zeile 4 ist weder eine explizite Typkonvertierung, noch eine implizite Typumwandlung nötig, da sicher gestellt ist, dass es sich um ein Objekt des Typs `Serie` handelt.

Für das Verständnis der generischen Typen und der Fehlermeldungen, die beim Programmieren auftreten können, ist es im Unterricht notwendig, Fachbegriffe klar zu definieren und durchgängig richtig anzuwenden. Leider wurden in der deutschen Literatur die englischen Begriffe vielfach unterschiedlich und teils falsch bzw. missverständlich übersetzt. Tabelle 3 fasst die in diesem Dokument verwendeten Begriffe zusammen. Ich empfehle ausschließlich diese deutschen Begriffe oder die englischen Originalbegriffe zu verwenden.

Tabelle 3: Fachbegriffe zu generischen Typen

Beispiel:

```
public interface List <E> {
    void add(E x);
    Iterator<E> iterator();
}

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

Begriff	Beispiel
generischer Typ (engl. generic type)	`List<E>`
formaler Typparameter (engl. formal type parameter)	`E`
parametrisierter Typ (engl. parameterized type)	`List<Integer>`
konkreter Typparameter (engl. actual type parameter)	`Integer`
Originaltyp (engl. raw type)	`List`
konkretes Typargument (engl. actual type argument)	Referenz auf ein `ArrayList` Objekt, dass nur `Integer` Werte enthält

Generische Typen und Vererbung[10]

Innerhalb einer Vererbungshierarchie kann ein Objekt aus unterschiedlichen Sichten betrachtet werden. So kann ein Objekt eines Typs dann einem Objekt eines anderen Typs zugewiesen werden, wenn diese kompatibel sind, also eine Typumwandlung möglich ist. Für Referenztypen ist dies i.A. dann der Fall, wenn eine Vererbungsbeziehung besteht.

10 *https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html* (ausgewertet am 22.12.2020)

Generische Typen können genauso wie normale Typen erweitert werden. Falls bei der Nutzung die konkreten Typparameter nicht variiert werden, bleibt die Subtyp-Beziehung bestehen.

Gehe von folgender Situation aus:

```
// Klassen Signaturen
final class Integer extends Number
final class Double extends Number
class BigBox<T> extends Box<T>

// Methoden Signaturen
public void numberTest( Number n );
public void boxTest( Box<Number> b );
```

Die folgenden Zeilen Code kompilieren bis auf die letzte ohne Fehler.

```
Object einObjekt;
Integer einInteger = new Integer(10);
einObjekt = einInteger; // OK: Integer ist Subtyp von Object

numberTest( new Integer(10));  // OK

Box<Number> box = new Box<Number>();
box.add( new Integer(10) );  // OK
Box<Integer> bbox = new BigBox<Integer>; // OK
boxTest( new Box<Number>() ); // OK
boxTest( new BigBox<Number>() ); //OK

boxTest( new Box<Integer>() ); // FEHLER
```

Die letzte Zeile führt zu einem Kompilierungsfehler, da `Box<Integer>` kein Subtyp von `Box<Number>` ist, obwohl `Integer` ein Subtyp von `Number` ist (siehe Abbildung 6).

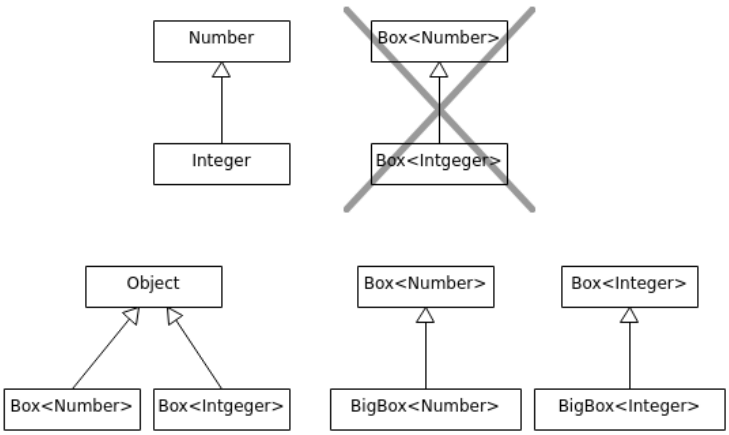

Abbildung 6: Beispiel möglicher Vererbungshierarchien bei generischen Typen

Dies liegt daran, wie generische Typen vom Compiler verarbeitet werden (Type Erasure[11]). Der generisch deklarierte Typ wird in eine einzige class-Datei kompiliert, so wie jede andere Klasse oder Interface. Es gibt nicht mehrere Versionen des Codes für verschiedene konkrete Typparameter: weder im Code, noch zur Laufzeit. Jede Instanz einer generischen Klasse teilt sich die selbe Klasse unabhängig vom im Code verwendeten konkreten Typparameter.

Typparameter sind analog zu den normalen Parametern, die in Methoden oder Konstruktoren verwendet werden. Ähnlich wie eine Methode formale Wertparameter hat, die die Art von Werten beschreiben, mit denen sie arbeitet, hat eine generische Deklaration formale Typparameter.

11 *https://docs.oracle.com/javase/tutorial/java/generics/erasure.html* (ausgewertet am 22.12.2020)

Wenn eine Methode aufgerufen wird, werden die formalen Parameter durch konkrete Argumente ersetzt, und der Methodenkörper wird ausgewertet. Wenn eine generische Deklaration aufgerufen wird, werden die formalen Typparameter durch konkrete Argumente ersetzt. Formale Typparameter werden nach Aufruf weder bei der Kompilierung, noch zur Laufzeit durch konkrete Typparameter ersetzt, sondern durch die konkreten Typargumente.[12]

Einschränken der Typparameter über Bounds

Im Beispielprojekt ermöglicht der generische Typ `FilmsammlungGeneric<E>` Typsicherheit und vermeidet doppelten Code. Die Methode `getBewertung()` einer Filmsammlung soll den Durchschnitt der Bewertungen aller Elemente in der Sammlung zurück geben. Die Objekte in der Sammlung – deren Typ durch den konkreten Typparameter vorgegeben ist – müssen demnach selbst eine Methode `getBewertung()` haben.

Allerdings kann bei der Instanziierung jeder beliebige konkrete Typparameter verwendet werden. Der Compiler kann beim Parsen der generischen Klasse deshalb nicht wissen, ob zur Laufzeit nur solche konkreten Typargumente vorliegen, die die Methode `getBewertung()` haben. Damit das Programm ohne Fehler compiliert, muss über einen sogenannten Bound sichergestellt werden, dass nur solche konkreten Typparameter erlaubt sind, die die Methode `getBewertung()` bereitstellen. Im Beispiel sind das all diejenigen Typen, die Subtyp von `Film` sind. `Film` stellt in der Vererbungshierarchie eine obere Schranke, den sogenannten **upper Bound**, dar. Im Programmcode wird dies durch das Schlüsselwort `extends` erreicht, wobei dies sowohl erweitern (wie bei Klassen) oder implementieren (wie bei Schnittstellen) bedeutet. Der Typparameter mit Bound lautet dann:

```
FilmsammlungGeneric<E extends Film>
```

Nun sind für `E` noch die Typen `Film`, `Kinofilm`, `Serie` und `Serienepisode` erlaubt.

Als Bound dürfen sowohl Klassen als auch Schnittstellen verwendet werden. Bei der Verwendung von Schnittstellen ist es sogar möglich, mehrere Bounds anzugeben. Ist einer der Bounds eine Klasse, muss diese als erstes angegeben werden. Zum Beispiel:[13]

```
class A { /* ... */ }
interface B { /* ... */ }
interface C { /* ... */ }

class D <T extends A & B & C>{ /* ... */ }
```

Generische Methoden

Auch Methoden können generisch definiert werden. Die Syntax lautet:

```
public <K, V> boolean compare( Pair<K, V> p1, Pair<K, V> p2 );
```

Dabei steht eine Liste der formalen Typparameter in spitzen Klammern vor dem Rückgabewert der Methode. Beim Aufruf der Methode müssen die konkreten Typparameter nicht angegeben werden, sondern der Compiler leitet sie automatisch ab (Typinferenz).

Nicht mit generischen Methoden zu verwechseln sind Methoden, die Typvariablen aus einer sie umgebenden generischen Klasse in ihrer Definition haben:

```
public E set (int index, E element)
```

Sie sind nicht generisch, weil der Typparameter E an anderer Stelle festgelegt wird und hier gar nicht mehr frei („generisch“) ist. Im Unterricht müssen generische Methoden nicht behandelt werden. Die Schülerinnen und Schüler sollten lediglich die Syntax der Signatur einer generischen Methode kennen. Dieser begegnen sie bei der Arbeit mit der offiziellen Java-Dokumentation.

12 *https://docs.oracle.com/javase/tutorial/extra/generics/intro.html* (Oracle, *Gilad Bracha*, The Java™ Tutorials Lesson: Generics, ausgewertet am 22.12.2020)

13 *https://docs.oracle.com/javase/tutorial/java/generics/bounded.html* (ausgewertet am 22.12.2020)

Bei generischen Methoden spielen neben upper Bounds auch lower Bounds und Wildcards eine Rolle, auf diese wird hier aber nicht näher eingegangen. Es empfiehlt sich das Studium der entsprechenden Quellen.[14]

Das Collections Framework

Das Paket `java.util` bietet mit dem Collections Framework generische Datentypen für Sammlungen[15] gleichartiger Objekte an. Für die meisten Anwendungen finden sich hier Schnittstellen und Implementierungen. Es ist daher selten notwendig, selbst komplexe generische Datentypen zu deklarieren. Als Ausgangspunkt für die Suche nach dem „richtigen" Datentyp bietet sich folgende Seite in der Java-Dokumentation an:

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/doc-files/coll-reference.html

Ebenso gibt es ein offizielles Tutorial zum Framework:

https://docs.oracle.com/javase/tutorial/collections/index.html

Die Dokumentation der jeweiligen Schnittstellen ist ausführlich genug, dass Schülerinnen und Schüler der Kursstufe nach Einführung der generischen Datentypen und einer Erklärung zu Interfaces, diese ohne weitere Hilfsmittel nutzen können.

In Abbildung 7 sind die wichtigsten Interfaces und Klassen dargestellt. Dieses Klassendiagramm kann im Unterricht zur Verfügung gestellt werden. Wenn genügend Zeit ist, kann der Kurs einen Teil davon selbst durch Recherche in der Dokumentation entwerfen.

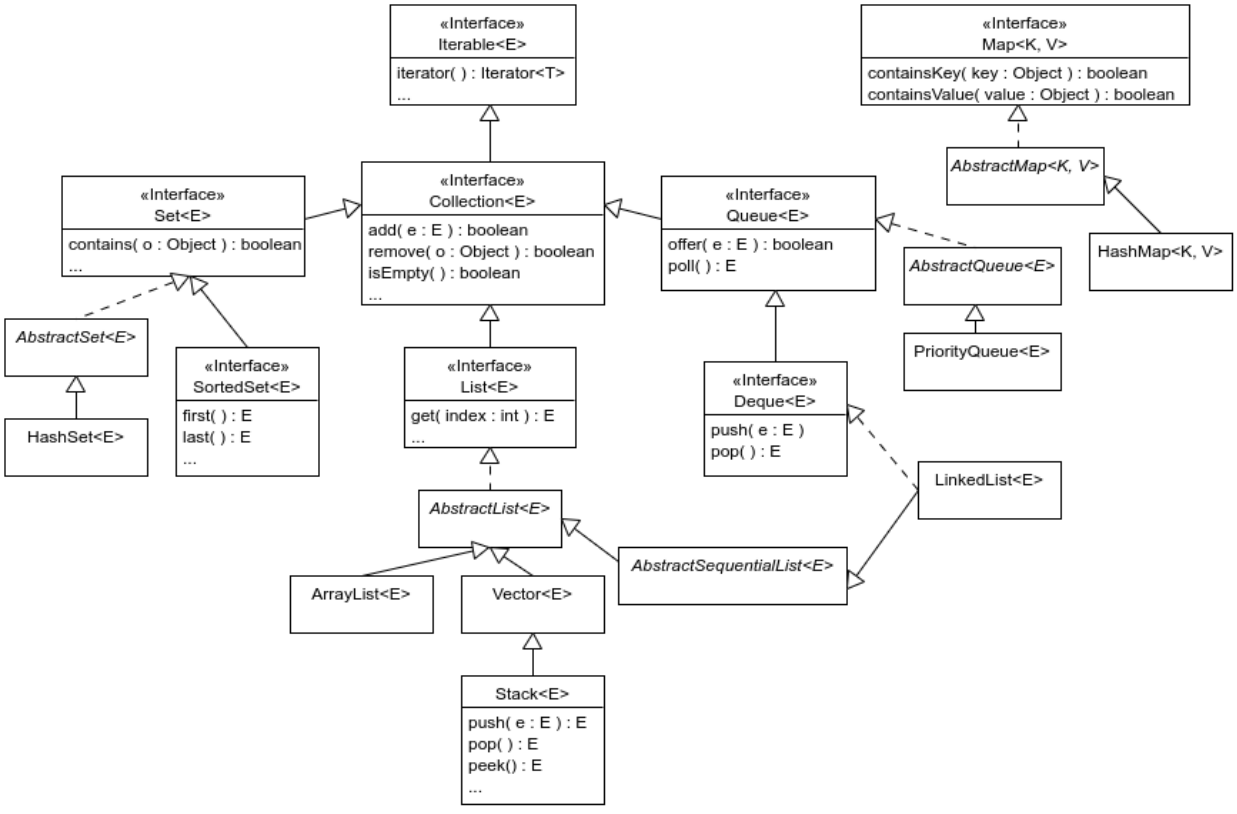

Abbildung 7: Klassendiagramm des Collections Framework

„Collection" gilt als Wurzel-Interface der Collection Hierarchie. Allerdings ist es für den Unterricht wichtig zuerst das Iterable Interface genauer zu betrachten. Dabei kann auch die Arbeit mit der Dokumentation exemplarisch vorgeführt werden.

14 *https://docs.oracle.com/javase/tutorial/java/generics/methods.html* (ausgewertet am 22.12.2020)

15 Siehe auch Fußnote 1 auf Seite 2.

Iteration und erweiterte for-Schleife

Nahezu alle Klassen des Collection Framework implementieren die `Iterable` Schnittstelle. Diese bildet die Grundlage für das Durchlaufen aller Elemente einer Sammlung.

Diese Schnittstelle ist recht einfach und schreibt lediglich drei Methoden vor:

```
void forEach(Consumer<? Super T> action)
Iterator<T> iterator()      // Iteration
Spliterator<T> spliterator()      // Traversierung und
Partitionierung
```

Die erste wird zusammen mit Lambdaausdrücken genutzt. Die zweite liefert einen Iterator, der mit Schleifen (while, for und erweitertes for) genutzt wird. Die letzte wird für den Unterricht nicht benötigt.

Im Unterricht sollte zunächst auf den Iterator eingegangen werden. Ein gemeinsamer Blick in die Dokumentation zeigt, dass diese Schnittstelle vier Methoden vorschreibt. Zwei werden für den weiteren Unterrichtsgang benötigt:

```
boolean hasNext() //Gibt true zurück, falls noch unbesuchte Elemente
E next() //Besucht das nächste Element und gibt es zurück
```

Es wird klar, dass eine Sammlung durchlaufen werden kann, indem man sich zunächst den Iterator zurückgeben lässt und mit diesem Schritt für Schritt alle Elemente besucht bzw. durchläuft. Für folgende Beispiele sei eine Sammlung `Collection<String> c` beliebiger Zeichenketten gegeben, welche der Reihe nach ausgegeben werden sollen.

Iteration mit while-Schleife:

```
Iterator<String> iter = c.iterator();

while( iter.hasNext() ){
    System.out.println( "Wert= " + iter.next() );
}
```

Iteration mit for-Schleife:

```
for( Iterator<String> it = c.iter(); iter.hasNext(); ){
    System.out.println( "Wert= " + iter.next() );
}
```

Vereinfachte Notation mit der erweiterten for-Schleife:

```
for( String s : c ){ //Sprich: für jeden String s in c
    System.out.println( "Wert= " + s );
}
```

Da im Beispielprojekt `FilmsammlungGeneric<E>` ein Subtyp von `ArrayList<E>` ist, implementiert diese auch die nötige Iterable Schnittstelle. Sollen beispielsweise für eine `FilmsammlungGeneric<Film> fsG` die Titel aller enthaltenen Filme ausgegeben werden, geht dies mit der erweiterten for-Schleife wie folgt:

```
for( Film f : fsG ){// Für jeden Film f in der Filmsammlung fsG
    System.out.println( f.getTitel() );
}
```

Lambdaausdrücke

Java ist keine funktionale Programmiersprache, unterstützt aber seit Version 8 einige Aspekte funktionaler Programmierung. In Java sind Methoden immer an ein Objekt gebunden, auf das über eine Referenz zugegriffen wird. In einer funktionalen Sprache können Funktionen als Argumente übergeben werden. Dies ist in Java nicht möglich, da es keine Referenz auf Methoden gibt.

Allerdings entspricht ein Objekt ohne Attribute mit nur einer einzigen Methode im Wesentlichen einer Funktion. Die Referenz auf dieses Objekt kann als Referenz auf die enthaltene Methode aufgefasst werden. Deklariert man einen Typ durch ein Interface oder eine abstrakte Klasse mit nur einer einzigen Methode, nennt diesen auch SAM-Typ (engl. single abstract method). Ist ein formaler Wertparameter von einem solchen SAM-Typ, kann ihm als konkretes Argument jedes Objekt übergeben werden, das den SAM-Typ erweitert. Das Argument kann aus einer benannten Klasse, aber auch über eine anonyme Klasse instanziiert werden. In beiden Fällen entsteht Boilerplate[16]. Um die damit verbundene Schreibarbeit zu vermeiden und den Quellcode lesbarer zu gestalten, stellt Java sogenannte **funktionale Interfaces** und **Lambdaausdrücke** zur Verfügung.

Ein **Funktionales Interface** ist mit der Annotation `@FunctionalInterface` versehen und besitzt nur eine einzige abstrakte Methode, die **funktionale Methode** genannt wird. Ein solche funktionale Schnittstelle liefert den Zieltyp für einen Lambdaausdruck.

Ein Lambdaausdruck ist eine Notation für anonyme (namenlose) Objekte, die ein funktionales Interface implementieren. Sie lesen sich somit wie die direkte Deklaration einer Funktionen.

Die allgemeine Syntax für einen Lambdaausdruck lautet:

```
( Parameterliste ) -> { Ausdruck oder Anweisungen }
```

Der Lambdaausdruck wird als konkretes Argument für einen formalen Parameter vom Typ eines funktionalen Interfaces übergeben. Sowohl der Rückgabewert des Ausdrucks, als auch die Typangaben in der Parameterliste werden durch Typinferenz automatisch abgeleitet und können daher weggelassen werden.

Für die Zieltypen der Lambdaausdrücke stellt das Paket `java.util.function`[17] genügend generische funktionale Interfaces zur Verfügung. Auch hier sollten die Schülerinnen und Schüler direkt mit der Dokumentation arbeiten, um passende Interfaces zu einer Aufgabe zu finden.

Beispiele für Lambdaausdrücke:

Tabelle 4: Beispiele für Lambdaausdrücke[18]

Code Schnipsel	Erläuterung	Interface mit funktionaler Methode Typ des formalen Parameters

`( File f ) -> {`
`  return f.isFile();}`
	Bei Anweisungen müssen geschweifte Klammern gesetzt werden.	`Predicate<T>{`
		`  boolean test(T t)}`
		`Predicate<File>`

`double x -> 2 * x`	Bei Ausdrücken entfallen die rechten geschweiften Klammern. Bei nur einem Argument zudem die runden Klammern.	`ToDoubleFunction<T>{`
		`  double applyAsDouble(`
		`  T value)}`
		`ToDoubleFunction<double>`

16 Unter Boilerplate werden hier Codeschnipsel ohne eigene Funktionalität verstanden, welche beinahe unverändert an vielen Stellen im Code eingefügt werden müssen.

17 *https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/function/package-summary.html* (ausgewertet am 22.12.20)

18 *https://www.torsten-horn.de/techdocs/java-lambdas.htm* (ausgewertet am 22.12.2020) ergänzt um rechte Spalte.

Code Schnipsel	Erläuterung	Interface mit funktionaler Methode Typ des formalen Parameters
`() -> "blupp"`	Eine leere Parameterliste ist erlaubt.	`Supplier<T>{ T get()}` `Supplier<String>`
`( int x, int y ) -> x + y`	Es können mehrere Parameter deklariert werden.	`IntBinarayOperator{ int applyAsInt(int left, int right)}` `IntBinaryOperator`
`( x, y ) -> x + y`	Auch bei mehreren Parametern können die Typangaben entfallen.	
`( x, y ) -> ( x > y ) ? x : y`	Der ?:-Operator gilt als ein einziger Ausdruck und die rechten geschweiften Klammern entfallen.	
`( x, y ) -> { if( x > y )return x; if( y > x ) return y; return 0;}`	Anweisungssequenzen sind möglich.	

Lambdaausdrücke können in mehreren Kontexten verwendet werden, z.b. in Zuweisungen, Methodenaufrufen, und Typkonvertierungen:

```
// Zuweisungs-Kontext
Predicate<String> p = String::isEmpty;

// Methodenaufruf-Kontext
stream.filter(e -> e.getSize() > 10)...

// Cast-Kontext (Typkonvertierung)
stream.map((ToIntFunction) e -> e.getSize())…
```

Iteration mit Lambda-Ausdrücken

Die `Iterable` Schnittstelle schreibt folgende Methode vor, welche als Wertparameter das funktionale Interface `Consumer<T>` deklariert:

```
void forEach(Consumer<? super T> action)
```

Diese Methode kann demnach mit einem Lambdaausdruck aufgerufen werden. Dabei wird der Lambdaausdruck auf jedes Element der Sammlung (welche die Schnittstelle implementiert) angewandt.

Für folgendes Beispiel sei eine Sammlung `Collection<String> c;` beliebiger Zeichenketten gegeben, welche der Reihe nach ausgegeben werden sollen.

```
c.forEach( s → { System.out.println( "Wert= " + s ) } );
```

Mit dem Lambdaausdruck wird der Code für die Iteration über die Sammlung sehr einfach und zudem für einen Menschen gut lesbar.

Sollen beispielsweise für eine `FilmsammlungGeneric<Film> fsG` die Titel aller enthaltenen Filme ausgegeben werden geht das mit einer einzigen Zeile Code:

```
fsG.forEach(f -> { System.out.println( f.getTitel() ); });
```

Sollte beim Entwickeln der Filmdatenbank `FilmsammlungGeneric` auf eine andere Collection (z.B. `LinkedList` statt `ArrayList`) umgestellt werden, dann funktioniert das Beispiel immer noch und zwar ohne Änderung der Syntax. Sollen statt dem Titel die Bewertungen aller Filme ausgegeben werden, muss nicht etwa `FilmsammlungGeneric` um eine Methode hierfür ergänzt, sondern lediglich der Lambdaausdruck angepasst werden. Dies erhöht die Wiederverwendbarkeit und Flexibilität von Code drastisch.

Streams

Neben den Lambdaausdrücken wurden mit Java Version 8 mit dem Paket java.util.stream mächtige Schnittstellen für Operationen auf Reihungen und Sammlungen eingeführt. Da eine genaue Behandlung den Rahmen des Bildungsplans überschreitet, wird hier auf die offizielle Dokumentation verwiesen, die „Streams“ recht gut erklärt.

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/stream/package-summary.html

Das `Collection<E>` Interface definiert die Methode `Stream<E> stream()`, welche die Elemente der Sammlung als Strom von Referenzen darstellt. Dieser erlaubt es, verkettete Operationen auf diesen Referenzen nacheinander oder parallel auszuführen. Wie bei funktionaler Programmierung werden die Daten, die durch die Referenzen repräsentiert werden, durch den Stream selbst nicht verändert.[19]

Von der Schnittstelle `Stream<E>` wird keine Implementierung mitgeliefert. Dies braucht es auch nicht, da die von ihr definierten Methoden meist Lambdaausdrücke übergeben bekommen. Die Methoden werden in zwei Hauptkategorien eingeteilt:

- intermediäre Operationen (intermediate operations) liefern wiederum einen `Stream`, der weiterverarbeitet werden kann (z.B. `filter()`, `map()`, `distinct()`, `sorted()`, etc.).
- terminale Operationen (terminal operations) führen ihrerseits Operationen auf den Referenzen des Streams aus (`forEach()`, `reduce()`, `toArray()`, etc.). Sie können einen Wert liefern und schließen den Strom. Danach können keine weiteren Operationen auf ihm ausgeführt werden.

Nehmen wir beispielsweise eine `FilmsammlungGeneric<Film> fsG`, kann der Filmtitel mit der schlechtesten Bewertung in einer Zeile Code bestimmt werden:

```
fsG.stream().reduce(fsG.get(0),
    (min, t) -> (t.getBewertung() < min.getBewertung()) ? t : min)
  ).getTitel();
```

Oft werden die Methoden nach dem Muster Filter-Map-Reduce auf einen Stream angewandt.

- Filter: Zunächst werden gewünschte Elemente aus dem Stream ausgewählt. Zum Beispiel mit `filter()`
- Map: Die Elemente des Streams werden transformiert. Zum Beispiel mit `map()`, `mapToInt()`
- Reduce: Der Stream wird auf ein Endergebnis reduziert. Zum Beispiel mit `reduce()`, `sum()`

Die Anzahl aller Filme der Sammlung mit einer Bewertung besser als 7.0 erhält man mit:

```
fsG.stream().filter( f -> f.getBewertung()>7.0 ).mapToInt(f -> 1).sum() );
```

19 *https://javabeginners.de/Arrays_und_Verwandtes/Streams.php* (ausgewertet am 22.12.2020)

Quellen zum Nachlesen

The Java™ Tutorials, Raymond Gallardo, Scott Hommel, Sowmya Kannan, Joni Gordon, Sharon Biocca Zakhour – Sixth edition, 2015 Oracle *https://docs.oracle.com/javase/tutorial/*
Trails zu: Java>Generics, Collections, Generics
→ Java Tutorial auf Englisch. Liefert neben Beispielen auch präzise Erklärungen. Erfordert allerdings ein Grundwissen über Sprachdetails von Java. Für Schüler eher ungeeignet.

The Java® Language Specification, Java SE 11 Edition, 2018-08-21 Oracle
https://docs.oracle.com/javase/specs/jls/se11/html/index.html
→ Für diejenigen die es ganz genau wissen wollen. Die vollständige Java Spezifikation.

Java® Platform, Standard Edition & Java Development Kit, Version 11 API Specification – Module java.base, Oracle
https://docs.oracle.com/en/java/javase/11/docs/api/java.base/module-summary.html
→ Die Java-Dokumentation, zur Nutzung als Nachschlagewerk während des Programmierens.

Java ist auch eine Insel, Christian Ullenboom –12. Auflage, Rheinwerk-Verlag
http://openbook.rheinwerk-verlag.de/javainsel/
→ Frei verfügbares Online-Lehrwerk auf Deutsch. Für Anfänger geeignet mit vielen Beispielen und Übungen, allerdings begrifflich nicht an allen Stellen präzise.

Funktionale Programmierung, Lambda-Ausdrücke, Stream-API, Bulk Operations on Collections, Filter-Map-Reduce, Torsten Horn, 2014 *https://www.torsten-horn.de/techdocs/java-lambdas.htm* (ausgewertet im Dez.2020)
→ Übersicht / Zusammenfassung zu den Ansätzen der funktionalen Programmierung in Java auf Deutsch.

Effective Java - A column by Angelika Langer & Klaus Kreft, January 2002 - September 2017)
http://www.angelikalanger.com/Articles/EffectiveJava.html
→ Eine Sammlung von Artikeln zu den Grundlagen von Java auf Deutsch. Darunter auch Artikel zu generischen Typen, Lambdaausdrücken und Streams.

6.005: Software Construction - Fall 2015, Max Goldman, Rob Miller, 2015 MIT
https://web.mit.edu/6.005/www/fa15/
→ Eine Vorlesung zu Softwareentwicklung auf Englisch. Schöne Beispiele insbesondere zu Streams und dem Filter-Map-Reduce-Muster. Nicht für Schüler geeignet.