
# Kontrollfragen und Übungen

zu

# Programmierung 2

Michael Lang

# Inhaltsverzeichnis

# 1. Exception-Handling

## 1.1. Kontrollfragen

4. Frage

Welche Unterschiedlichen Fehler kennen Sie im Java-Umfeld? Welche Fehler sollten nicht, können oder müssen behandelt werden?

https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/3

5. Frage

Welche Arten von Ausnahmen sind Ihnen im Java-Umfeld bekannt?

Exceptions & RuntimeExceptions

6. Frage

Erläutern Sie das Grundprinzip der Ausnahmebehandlung!

https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/3/1

7. Frage https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/3/2

Was verstehen Sie unter „eine Ausnahme“ werfen, fangen und weitergeben)

8. Frage

Welches sind die wesentlichen Methoden der Klasse Throwable?

https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/5

9. Frage

Worin unterscheiden sich checked und unchecked Exceptions?

https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/6

10. Frage

Wozu dient der Finally()-Block bei einer Try-Catch-Anweisung?

Finally wird unabhängig vom auftreten einer Ausnahme IMMER aufgerufen
—> Wird für „Aufräum-Arbeiten“ genutzt (zum Beispiel für Ressourcen)

11. Frage

Welche Möglichkeiten sind Ihnen zum Erzeugen eigener Ausnahmeklassen bekannt?

Implizites oder explizites Ableiten der Exception Klasse

12. Frage

Was würde passieren, wenn eine Ausnahme nicht abgefangen, sondern immer weitergegeben wird?

Letztendlich für eine nicht verarbeitete Ausnahme zu einem Programmabbruch

## 1.2. Übungen

1. Übung

Das folgende Programm soll die Zahlen von 1 bis 100 in eine Textdatei mit dem Namen ausgabe.txt schreiben. Leider kann das Programm in dieser Form nicht ausgeführt werden. Was ist der Grund dafür und wie können Sie es korrigieren?[1] (Anmerkung: Auch wenn der Input- & Output-Stream noch nicht behandelt wurde, können Sie die Frage schon jetzt beantworten.)

```
import java.io.FileWriter;

public class Uebung1 {

      public static void main(String[] args) {

            FileWriter datei;
```

1 vgl. Niemann, Alexander, Der bhv Co@ch Übungsbuch Java, bhv-Verlag, 3. überarbeitete Auflage 2006, Seite 152

```
            String text;

            datei = new FileWriter("ausgabe.txt");
            text = "1\n";

            for(int i = 2; i <=100; i++) {
                  text += i;
                  text += "\n";
            }
            datei.write(text, 0, text.length());
            datei.flush();
      }

}
```

2. Übung

In Ihrem Unternehmen beziehen Sie Autositze. Für das Beziehen stehen Ihnen die Materialien *Leder* und *Stoff* zur Verfügung. Den *Stoff* können Sie in jeder beliebigen Farbe liefern. *Leder* jedoch ist nur in den Farben *Schwarz* und *Weiß* lieferbar.

Implementieren Sie eine Klasse *AutoSitze* gemäß der Vorgaben des UML-Diagramms. Der Konstruktor soll eine Fehlermeldung der von Ihnen zu implementierenden Fehlerklasse *FalscheParameter* erzeugen, sobald er ungültige Parameterkombinationen erhält. Der Meldungstext der Fehlerklasse soll die fehlerhafte Parameterkombination ausgeben.

Testen Sie Ihre Klasse *AutoSitze* mit einem kleinen Testprogramm *TestAutoSitzeException*. In Ihrem Testprogramm soll nach erfolgreichem Durchlauf des Konstruktors eine Meldung ausgegeben werden, in welcher Farbe und in welchem Material der Sitz bezogen wurde. Sollte während dem Durchlauf des Konstruktors eine Ausnahme ausgelöst werden, geben Sie bitte in Ihrem Testprogramm den Meldungstext der Ausnahme sowie einen kleinen Hinweis, dass das Beziehen fehlgeschlagen ist, aus.

| **AutoSitze** |
| --- |
| - ledersitze: boolean<br>- farbe: String |
| + AutoSitze(bezug: boolean, color: String)<br>+ getFarbe(): String<br>+ isLedersitze(): boolean |

3. Übung

Im folgenden Quellcode befindet sich ein logischer Fehler. Worin besteht er und wie kann er behoben werden?

```
public class TankLeerDemo {

      public static void main(String[] args) {

            Auto bmw = new Auto(0, 35487);

            bmw.tanken();

            try {
                  bmw.fahren();
```

```
            } catch (TankLeer e) {
                  System.out.println(e.getMessage());
            }

            bmw.tanken();

            try {
                  bmw.fahren();
            } catch (Exception e) {
                  e.printStackTrace();
            } catch (TankLeer e) {
                  System.out.println(e.getMessage());
                  System.out.println(e.toString());
                  e.printStackTrace();
            } finally {
                  System.out.println("Der neue Kilometerstand: " +
                                          bmw.getKmCount());
            }
      }
}
```

# 2. Collection Framework

## 2.1. Kontrollfragen

4. Frage
Was können Datencontainer des Collection Framework enthalten?

Jegliche Art von Objekten

5. Frage
Welche drei Arten von Containern kennen Sie?

https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/10/2

6. Frage
Worin unterscheiden sich die drei Containerarten?

https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/10/2

7. Frage
Was ist der Unterschied zwischen Arrays und Containern des Collection Framework?

Container haben eine dynamische Größe - Arrays sind statisch nach der Initialisierung (semi-dynamisch)

8. Frage
Sie möchten ein Schachfeld oder auch den Spielplan von „Schiffe versenken“ in Java implementieren. Dazu müssen Sie einen Repräsentanten des Spielfeldes im Speicher erzeugen und verwalten. Welche Datenstrukturen würden Sie dabei verwenden? Wie begründen Sie Ihre Entscheidung?[2]

Arrays aufgrund der statischen Größe der Anforderungen und der besseren Zugrissperformance über den Index

9. Frage
Wozu werden Iteratoren benötigt?

Iteratoren werden zum durchlaufen von Containertypen verwendet mit der Möglichkeit diese (bedingt) während des Durchlaufens auch zu modifizieren.

10. Frage
Was müssen Sie bei Containern beachten, in denen die Objekte sortiert abgelegt werden?

Implementieren des Comparable Interfaces oder angeben eines Comparator Objektes für die Sortierung

11. Frage
Wozu brauchen Sie die Interfaces Comparable und Comparator?

Zum sortieren von Objekte (zum Beispiel in SortedSets oder „manuelles“ Sortieren einer Liste

12. Frage
Können Sie Listen-Container sortieren? Wenn ja, wie gehen Sie bei der Realisierung vor?

Ja, über die Sortmethode an der Liste bzw. Collections.sort()

13. Frage
Welche Möglichkeiten haben Sie, um Objekte miteinander zu vergleichen? Worin liegt der Unterschied der Vergleichsmöglichkeiten?

== –> Referenz Vergleich, equals() –> Inhaltsvergleich, compareTo() –> inhaltsvergleich mit relationaler Aussage

14. Frage
Was müssen Sie beim Überschreiben der equals()-Methode für direkte und indirekte Sub-Klassen der Klasse Object beachten?

equals Contract: https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/14/1
Implementierung: https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/14/2

15. Frage
Worin besteht der Zusammenhang zwischen der hashCode()- und der equals()-Methode?

https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/14/5

---

[2] vgl. Niemann, Alexander, Der bhv Co@ch Übungsbuch Java, bhv-Verlag, 3. überarbeitete Auflage 2006, Seite 75

16. Frage
Beschreiben Sie zwei einfache Möglichkeiten, um eine hashCode()-Methode zu überschreiben!

https://matthiasbergneels.github.io/md-scripts/01_Programmieren_Vorlesung/Programmierenskript_2.Semester.html#/14/6

siehe GitRepo

17. Frage
Was sind Wrapper-Klassen und wozu werden sie benötigt?

Wrapper-Klassen packen den Wert eines Primitiven Datentyps als Objekt ein um diese Werte in Datencontainern und anderen Strukturen, welche Objekte erwarten/benötigen, nutzen zu können.

18. Frage
Sie legen Objekte der Wrapper-Klasse Double in einem Datencontainer der Klasse TreeSet ab. Warum ist dies problemlos möglich?

Die Wrapper-Klasse zu Double implementiert das Comparable-Interface (und damit eine natürliche Ordnung). Hiermit kann ein Objekt der Klasse Double eine SortedList (bsp TreeSet eingefügt werden.

19. Frage
Worin liegen die wesentlichen Unterschiede zwischen Maps und Lists?

Liste: wahlfreier und sequentieller Zugriff, Reihenfolge des Einfügens bleibt erhalten, Dublikate erlaubt
Maps: Speichern von Werte mit einem Schlüssel, Zugriff über Schlüssel (welcher eindeutig sein muss), Werte können Dublikate enthalten, Ordnung der Schlüssel hängt an der Implementierung der Map

## 2.2. Übungen

1. Übung
Implementieren Sie zunächst eine Klasse *Kunde* gemäß der Vorgaben des UML-Diagramms.

| **Kunde** |
|---|
| - name: String<br>- vorname: String<br>- kundenNummer: int |
| + Kunde(name: String, vorname: String, nummer: int)<br>+ setName(name: String): void<br>+ getName(): String<br>+ setVorname(vorname: String): void<br>+ getVorname(): String<br>+ setKundeNummer(nummer: int): void<br>+ getKundenNummer(): int |

Schreiben Sie sich ein Testprogramm *TestKunde*. In diesem Testprogramm erzeugen Sie sich zunächst fünf Objekte der Klasse *Kunde*. Namen, Vornamen und Kundennummern finden Sie in dieser Tabelle:

| **Name** | **Vorname** | **Kundennummer** |
|---|---|---|
| Mustermann | Klaus | 4711 |
| Beispiel | Hans | 5180 |
| Mustermann | Hilde | 4712 |
| Vorbild | Theodor | 8278 |
| Dummy | Jimmy | 1111 |

Nachdem Sie die Objekte erzeugt haben, möchten Sie die Objekte in einem Datencontainer der Klasse *TreeSet* ablegen. Erweitern Sie entsprechend Ihre Klasse *Kunde*, so dass die Objekte der Klasse *Kunde* aufsteigend nach der Kundennummer sortiert werden.

Geben Sie abschließend die Elemente aus dem *TreeSet* über einen *Iterator* auf der Konsole aus, um den Erfolg der Sortierung zu überprüfen.

2. Übung
Erweitern Sie die Klassen *Kunde* und *TestKunde* aus der vorangegangenen Übung so, dass Sie die Objekte der Klasse *Kunde* in einem Datencontainer der Klasse *Vector* nach dem Namen und bei Namensgleichheit nach dem Vornamen aufsteigend sortiert ablegen können.

Geben Sie abschließend auch die Elemente aus dem *Vector* über einen *Iterator* ebenfalls auf der Konsole aus, um auch den Erfolg der zweiten Sortierung zu überprüfen.

3. Übung
Ermöglichen Sie es, dass Objekte der Klasse *Kunde* aus den vorangegangenen Übungen miteinander verglichen werden können. Implementieren Sie dazu alle erforderlichen Methoden in der Klasse *Kunde*.

Testen Sie die Vergleichsmethode(n), indem Sie in einem Testprogramm *TestVergleichKunde* zwei gleiche Objekte der Klasse *Kunde* erzeugen und diese miteinander vergleichen. Geben Sie das Ergebnis des Vergleichs auf der Konsole aus.

# 3. Swing

## 3.1. Kontrollfragen

1. Frage
Erläutern Sie die fünf wesentlichen Unterschiede zwischen AWT- und Swing-Komponenten!

2. Frage
Beschreiben Sie die wesentlichen Komponenten eines JFrame-Fensters!

3. Frage
Welche Bedeutung kommt den Layout-Managern bei der Gestaltung von Benutzeroberflächen zu?

4. Frage
Erläutern Sie den Zusammenhang zwischen Containern der Klasse JPanel und den Layout-Managern!

5. Frage
Welche unterschiedlichen Rahmenarten sind Ihnen bekannt?

6. Frage
Nennen Sie fünf Swing-Komponenten und deren Funktion zur Gestaltung von Benutzerdialogen!

7. Frage
Welche unterschiedlichen Arten von Textfeldern sind Ihnen bekannt?

8. Frage
Beschreiben Sie die Vorgehensweise bei der Erstellung von validierenden Textfeldern!

9. Frage
Beschreiben Sie die Aufgaben und Anwendungsgebiete des ItemListener!

10. Frage
Wozu wird der ActionListener benötigt? Wie funktioniert er?

11. Frage
Beschreiben Sie die wesentlichen Unterschiede zwischen den Objekten der Klassen JCheckBox und JRadioButton!

12. Frage
Welche unterschiedlichen Klassen und Interfaces benötigen Sie zum Erzeugen von Benutzermenüs? Beschreiben Sie jeweils mit einem Satz die entsprechenden Klassen!

13. Frage
Was sind Tooltips und wie werden diese in Java realisiert?

## 3.2. Übungen

14. Übung

1. Entwickeln Sie einen Benutzerdialog zum Erfassen von Einwohnern in der Klasse StammdatenErfassen analog dem Screenshot.


   Stellen Sie dabei zunächst nur die Eingabefelder für die Kontaktdaten, die Radiobuttons für den Familienstand und die Buttons zur Verfügung. Dabei ist zu beachten, dass die Auswahlliste für die Anrede die Werte Herr, Frau und --- zur Verfügung stellen soll. Die Buttons sollen noch keine Funktionalität beinhalten, sondern nur angezeigt werden. Der Nachrichtentext (im Screenshot „Einwohner hinzugefügt“) wird erst zu einem späteren Zeitpunkt implementiert.
   Beim Schließen des Fensters soll die Anwendung beendet werden.

2. Entwickeln Sie die Klasse Einwohner. Die Klasse Einwohner soll das Interface Comparable implementieren. Stellen Sie einen Konstruktor und die entsprechenden Getter-Methoden für die privaten Attribute zur Verfügung. Die compareTo-Methode soll Objekte nach dem Namen und bei gleichen Namen nach dem Vornamen aufsteigend sortieren.

| **Einwohner** |
| --- |
| - name: String<br>- vorname: String<br>- gebName: String<br>- eMail: String<br>- anrede: String<br>- familienstand: String |
| + Einwohner(name: String, vorname: String, gebName: String, eMail: String, anrede: String, familienstand: String)<br>+ compareTo(einw: Einwohner): int<br>+ Getter-Methoden für die privaten Attribute |

Über die Klasse Einwohner sollen später Einwohner-Objekte auf Basis der im Benutzerdialog gemachten Eingaben erzeugt und in einem TreeSet gespeichert werden.

3. Stellen Sie für die Radiobuttons und die Buttons je einen eigenen ActionListener zur Verfügung.

    a. Der ActionListener für die Radiobuttons soll folgende Funktionalität erfüllen. Wählt der Benutzer den Familienstand „ledig“ aus, so wird der Beschreibungstext für den Geburtsnamen ausgeblendet und das zugehörige Eingabefeld initialisiert und ausgeblendet. Wird ein anderer Familienstand angeklickt, so sollen die beiden Felder wieder eingeblendet werden.
    b. Die Methoden für die Umsetzung der Funktionalität sollen als Klassenmethoden in einer eigenen Klasse Listenoperationen zur Verfügung gestellt werden (s. UML-Diagramm).

| **Listenoperationen** |
| --- |
| - liste: TreeSet |
| - ermittleObjekt(name: String, vorname: String): Einwohner<br>+ hinzufuegen(anrede: String, name: String, vorname: String, gebName: String, eMail: String, familienstand: String): boolean<br>+ anzeigen(name: String, vorname: String): boolean<br>+ suchen(name: String, vorname: String): boolean<br>+ loeschen(name: String, vorname: String): boolean<br>+ listeAusgeben(): void |

Der für die Buttons zuständige ActionListener übernimmt lediglich die Steuerung, welche Methode aufgerufen wird und er reagiert auf die Rückgabewerte der bool’schen Methoden, indem er den Nachrichtentext entsprechend aktualisiert. Die einzelnen Buttons bekommen folgende Funktionen zugewiesen

- die Methode *ermittleObjekt* der Klasse Listenoperationen durchsucht das TreeSet nach Einwohnerobjekten. Suchkriterien sind dabei der Name und Vorname des gesuchten Einwohners. Wird ein Objekt im TreeSet gefunden bei dem Name und Vorname mit den Suchkriterien übereinstimmen, wird das Objekt als Rückgabewert zurückgegeben, wird kein passendes Objekt gefunden, so gibt die Methode *null* zurück.
- **Hinzufügen** – ruft die Methode *hinzufuegen* der Klasse Listenoperationen auf, die wiederum auf Basis der übergebenen Parameter ein Einwohnerobjekt erzeugt und in das TreeSet einfügt. Der Erfolg oder Misserfolg des Einfügens soll im Nachrichtentext angezeigt werden (z.B. „Einwohner hinzugefügt“ oder „Einwohner existiert schon“)
- **Anzeigen** – ruft die Methode *anzeigen* der Klasse Listenoperationen auf. Die Methode ermittelt zunächst das anzuzeigende Objekt mithilfe der Methode *ermittleObjekt*. Existiert ein entsprechendes Objekt im TreeSet, werden dessen Attribute auf der Konsole sowie ein entsprechender Nachrichtentext im Benutzerdialog ausgegeben. Wird kein passendes Objekt im

TreeSet gefunden, so wird nur der Nachrichtentext im Benutzerdialog ausgegeben.

- **Suchen** – ruft die Methode *suchen* der Klasse Listenoperationen auf. Die Methode durchsucht das TreeSet mithilfe der Methode *ermittleObjekt* und meldet im Nachrichtentext zurück, ob der Einwohner gefunden wurde oder nicht.
- **Löschen** – ruft die Methode *loeschen* der Klasse Listenoperationen auf. Mithilfe der Methode *ermittleObjekt* wird zunächst geprüft, ob der zu löschende Einwohner im TreeSet vorhanden ist. Wird der zu löschende Einwohner im TreeSet gefunden, wird er aus dem TreeSet gelöscht. Der Erfolg bzw. Misserfolg des Löschens wird im Nachrichtentext des Benutzerdialogs angezeigt.
- **Liste anzeigen** – ruft die Methode *listeAusgeben* der Klasse Listenoperationen auf, die wiederum alle Objekte (z.B. in Form der wesentlichen Attribute) aus dem TreeSet auf der Konsole ausgibt.

4. Übung (optional)

Erweitern Sie Ihr Programm aus Übung 1 um ein geeignetes Menü (gleiche Funktionalitäten, wie die Buttons) und entsprechende Tooltips.

# 4. Input- & Output-Stream

## 4.1. Kontrollfragen

5. Frage
Welche Klasse verwenden Sie, wenn Sie auf Objekte des File-Systems zugreifen möchten?

6. Frage
Nennen Sie die wesentlichen Methoden, um Informationen über Verzeichnisse und Dateien des File-Systems zu beschaffen!

7. Frage
Mit welchen Methoden der Klasse File ist es Ihnen möglich Verzeichnisse bzw. Dateien anzulegen?

8. Frage
Warum macht es Sinn, dass die Methoden zum Anlegen, Umbenennen und Löschen von Dateien bzw. Verzeichnissen vom Typ boolean implementiert sind?

9. Frage
Prinzipiell unterscheidet man zwei grundlegende Arten von in- und Output-Streams. Worin liegt der Unterschied?

10. Frage
Nennen Sie jeweils 3 Beispiele für Klassen des Input- und Output-Streams auf Zeichen- und Byte-Basis!

11. Frage
Warum gibt es für den InputStreamReader keine vergleichbare Klasse auf Byte-Ebene?

12. Frage
Was bewirkt die Methode toString() und wie wird sie normalerweise aufgerufen?

13. Frage
Was ist der wesentliche Unterschied zwischen den beiden nachstehenden Anweisungen?

**Coding 1**

```
import java.io.IOException;

public class EingabeTastatur {

	public static void main(String[] args) {

		byte[] eingabe = new byte[255];

		System.out.println("Geben Sie einen Text ein:");
		try {
			System.in.read(eingabe, 0, 255);
		} catch (IOException e) {
			e.printStackTrace();
```

```
            }
            System.out.println(eingabe);
            System.out.println(new String(eingabe));
      }
}
```

**Coding 2**

```
import java.io.*;

public class EingabeTastaturString {

      public static void main(String[] args) {

            InputStreamReader strRead = new InputStreamReader(System.in);
            BufferedReader bufString = new BufferedReader(strRead);
            String eingabe = "";

            System.out.println("Geben Sie Ihren Text ein: ");
            try {
                  eingabe = bufString.readLine();
            } catch (IOException e) {
                  e.printStackTrace();
            }
            System.out.println(eingabe);
      }
}
```

14. Frage
Welche Klassen sollten Sie zum Lesen und Schreiben von Textdateien anwenden?

15. Frage
Warum ist es bei Dateien, die nicht im Textformat gespeichert sind, sinnvoll über Byte-basierte Klassen zu Lesen und zu Schreiben?

16. Frage
Was sind die wesentlichen Vorteile von Properties-Dateien?

17. Frage
Wozu werden Objekte der Klasse ResourceBundle eingesetzt?

18. Frage
Wie gestaltet sich die Namensgebung von ResourceBundle-Dateien?

## 4.2. Übungen

1. Übung
Verschaffen Sie sich eine Überblick über alle bekannten Laufwerke und geben Sie an, ob auf die Laufwerke zugegriffen werden kann.

2. Übung
Erstellen Sie mit dem Notepad-Editor ein einfaches Text-File. In diesem Text-File sollten Sie Ihren Namen sowie die Namen Ihrer benachbarten Kommilitonen eintragen und dieses speichern.
Erstellen Sie nun ein einfaches Programm, dass es Ihnen ermöglicht über die Konsole sowohl den Pfad als auch den Dateinamen dieser Textdatei anzugeben.

Lesen Sie basierend auf Ihrer Eingabe über einen geeigneten Input-Stream die Datei in Ihr Programm ein und geben Sie den Inhalt dieser Datei aus.

3. Übung

Nutzen Sie Ihr Coding zur Eingabe über die Konsole aus der Übung zuvor. Dieses Mal sollen Sie sowohl Pfad als auch Name von je einer Quell- und Zieldatei über die Konsole angeben. Kopieren Sie den Inhalt der Quelldatei in die Zieldatei. Beachten Sie dabei, dass Sie mit Ihrem Kopier-Programm jedes beliebige Dateiformat unabhängig von der Dateigröße kopieren möchten.

4. Übung

Implementieren Sie ein Programm, dass Ihnen die System-Properties in einem XML-File auslagert. Sollten Sie eine Java-Version 1.4 oder älter verwenden, speichern Sie die System-Properties bitte entsprechend in einem Text-File.

5. Übung

Erweitern Sie Ihr Programm aus Übung 3 so, dass die Eingabefelder abhängig von den lokalen Spracheinstellungen beschriftet werden. Legen Sie die sprachabhängigen Texte in Deutscher und englischer Sprache ab, wobei Englisch die Standardsprache der Anwendungstexte sein soll.

# 5. Threads

## 5.1. Kontrollfragen

1. Frage
Erläutern Sie den wesentlichen Unterschied zwischen Multi-Tasking und Multi-Threading!

2. Frage
Definieren Sie den Begriff Thread!

3. Frage
In welchen Bereichen finden Threads Anwendung? Wodurch ergeben sich daraus Vorteile?

4. Frage
Welche Möglichkeiten haben Sie, um Threads zu erzeugen? Wie können Threads gestartet werden?

5. Frage
Nennen und beschreiben Sie kurz vier unterschiedliche Zustände von Threads!

6. Frage
Welche Möglichkeiten haben Sie, um Threads zu beenden? Beschreiben Sie eine Möglichkeit in wenigen Sätzen!

7. Frage
Was würde passieren, wenn Sie einen Thread über die run()- statt über start()-Methode starten?

8. Frage
Eigene Thread-Klassen sollten immer das Interface Runnable implementieren und nicht von der Klasse Thread erben. Nehmen Sie zu dieser These kritisch Stellung!

9. Frage
Warum ist eine zeitliche Synchronisation von Threads erforderlich, wenn diese gemeinsamen Daten nutzen? Welche Möglichkeiten haben Sie, um Threads zu synchronisieren?

10. Frage
Beschreiben Sie das Consumer-Producer-Problem und skizzieren Sie einen möglichen Lösungsansatz! Wo findet sich dieses Problem in der Praxis wieder?

11. Frage
Was ist ein Dämonen-Thread und wie wird dieser erzeugt?

12. Frage
Was ist das besondere an einem Dämonen-Thread im Vergleich zu einem normalen Anwendungsthread?

## 5.2. Übungen

1. Übung

Ein *Ticketshop* verkauft eine bestimmte Menge WM-Tickets an die angereisten *Fußballfans*. Implementieren Sie die Klassen *Ticketshop und Fan* analog der nachstehenden UML-Diagramme, wobei die Klasse *Fan* eine Subklasse der Klasse *Thread* darstellt.

| **Ticketshop** |
|---|
| - tickets: int |
| + Ticketshop(tickets: int)<br>+ kaufen(name: String): void<br>+ getTickets(): int |

| **Fan** |
|---|
| - shop: Ticketshop |
| - name: String |
| + Fan(name: String, shop: Ticketshop)<br>+ run(): void |

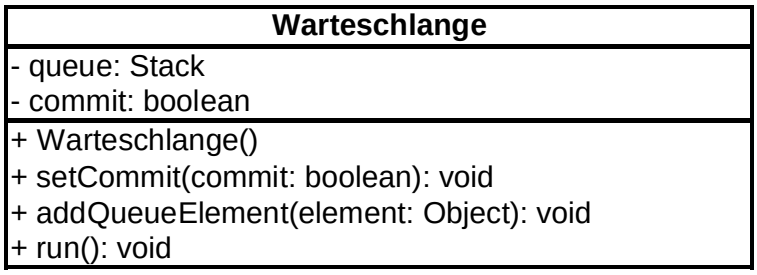
Geben Sie dem Konstruktor der Klasse *Ticketshop* die Anzahl der zu verkaufenden Tickets mit. Erzeugen Sie innerhalb der main()-Methode der Klasse *Vorverkauf* drei Instanzen der Klasse *Fan* und starten Sie die entsprechenden *Threads*. Die Fans kaufen Tickets bis der Vorrat im Ticketshop aufgebraucht ist. Sobald die Tickets ausverkauft sind, sollen die *Threads* über den Aufruf der interrupt()-Methode beendet werden. Achten Sie darauf, dass ggf. Methoden synchronisiert werden müssen.

2. Übung

Erweitern Sie Ihr Programm aus Übung 1. Die Tickets sind noch nicht gedruckt, sondern sollen während dem Programm von einer *Druckerei* (Producer) hergestellt werden. Die maximal produzierbare Ticketanzahl beläuft sich auf 100 Stück und es sollen maximal 25 Tickets vorrätig sein. Die Tickets sollen weiterhin über den *Ticketshop* (Consumer) an die Fans verkauft werden. Setzen Sie das Consumer-Producer-Modell entsprechend zur Lösung der Problematik ein.
Vor der Implementierung sollten Sie zunächst die UML-Klassendiagramme für die Klassen *Druckerei* und den *Ticketshop* notieren.

3. Übung

Implementieren Sie die Klasse *Warteschlange* als Dämonen-Thread gemäß der Vorgaben des UML-Diagramms. In einer Testklasse *TestWarteschlange* sollen über eine for-Schleife 15 String-Objekte über die addQueueElement-Methode in die Warteschlange aufgenommen werden.
Sobald zehn Elemente in der Warteschlange sind oder das Attribut commit explizit auf true gesetzt wird, soll der Dämonen-Thread eigenständig den Stack über einen Iterator ausgeben und danach den Stack initialisieren. Nach jeder Ausgabe des Stacks soll der Anwender mit einem kurzen Text informiert werden, dass die Warteschlange abgearbeitet wurde.
Ein Beispiel für die Ausgabe finden Sie neben dem UML-Diagramm.




```
Druckauftrag 0
Druckauftrag 1
Druckauftrag 2
Druckauftrag 3
Druckauftrag 4
Druckauftrag 5
Druckauftrag 6
Druckauftrag 7
Druckauftrag 8
Druckauftrag 9
Warteschlange abgearbeitet
Druckauftrag 10
Druckauftrag 11
Druckauftrag 12
Druckauftrag 13
```