# Inhaltsverzeichnis

# Antworten zur Altklausur

## *Aufgabe (1) - Programmstruktur*

## a) Bytecode, JavaVM

(a) Erläutern Sie das Konzept des Java Bytecodes: Wie entsteht er, wie wird er ausgeführt und wieso werden Java-Programme durch ihn plattformunabhängig? (3)

Der Java-Bytecode entsteht bei der Kompilierung durch den Java-Compiler. Dieser Bytecode kann unabhängig von der Rechner-Plattform von einer virtuellen Maschine ausgeführt werden. Hierdurch können Java-Programme unabhängig vom jeweiligen Betriebssystem und der zugehörigen Rechner-Hardware plattformunabhängig ausgeführt werden.

## b) Pakete

(b) In der Datei C.java haben Sie mittels

```
package p.q;
class C {...}
```

eine Klasse definiert. Jetzt wollen Sie der Klasse den neuen Namen `D` geben. Außerdem soll sie nach der Umbenennung zum Paket `r.s` gehören. Was müssen Sie tun? In welchem Verzeichnis muss die Datei stehen?

Bisher : src/p/q/C.java Neu : src/r/s/D.java

## c) ActionListener

(c) Erläutern Sie genau, was der Ausdruck

```
new ActionListener(void actionPerformed(ActionEvent ae){...});
```

bewirkt!

Klicken wir z.B. auf eine Schaltfläche, welcher der ActionListener angefügt wird, so wird die Aktion als ActionEvent-Objekt ae gemeldet. Dieses Objekt wird an den ActionListener gesendet. Ein ActionListener wird üblicherweise mit der Methode addActionListener() an ein Objekte (z.B. Schaltfläche) angeheftet, welche nun Aktionen auslösen können. ActionListener ist eine Schnittstelle (Interface) mit der Methode actionPerformed(). Das Interface ActionListener ist eine erweiterung des Interfaces EventListener, welches von allen Listener-Interfaces implementiert werden muss.

## d) Klassen, Methoden

(d) Dürfen zwei Attribute einer Klasse den gleichen Namen besitzen? Dürfen zwei Methoden einer Klasse den gleichen Namen besitzen? Begründen Sie Ihre Antwort!

Attribute beschreiben eine ganz bestimmte Eigenschaft der jeweiligen Klasse. Theoretisch kann eine Klasse 2 oder mehr gleichnamige Attribute besitzen, solange die Attribute immer in gleicher Reihenfolge verarbeitet werden. Jedoch ist dann die eindeutige Unterscheidbarkeit nicht mehr vorhanden und kann sehr leicht zu Verwechslung beim entwickeln eines Programms führen (unnötige Fehlerquelle).

Zwei oder mehr Methoden dürfen in Java denselben vollständigen Namen haben, das heißt, sie dürfen :

- zur selben Klasse gehören und zugleich
- mit demselben Identifier als Namen der Methode bezeichnet sein, wenn :
  - sie sich entweder in der Anzahl der Parameter unterscheiden
  - oder (falls die Anzahl gleich ist) wenigstens die Liste der Typen der Parameter sich unterscheidet.

Eine derart duplizierte Methode heißt **überladen**.

## e) Datenfeld

(e) Das Datenfeld x sei innerhalb der Klasse C im Paket p ohne eines der Sichtbarkeitsattribute (**public**, **private** oder **protected**) definiert:

```
package p;
class C{int x;}
```

Erläutern Sie, an welchen Stellen eines Programmes der Name x sichtbar ist und wo nicht!

Die Variable x ohne Sichtbarkeitsattribute ist an die Lebensdauer der Klasse C gebunden, die Variable existieren nur solange die Klasse aktiv ist. Die Variable ist auch nur innerhalb der Klasse C sichtbar.

## *Aufgabe (2) – Initialisierung, Typen, Speicherverwaltung*

## a) Typen

(a) Zählen Sie alle Arten der Initialisierung von Datenfeldern einer Klasse auf, in der Reihenfolge ihrer Ausführung!

**1.Primitive Typen**

Zahlen, Unicode-Zeichen, Boolesche Werte und Wahrheitswerte.

Variablen mit primitivem Typ enthalten ihren Wert immer direkt . Initialisierung bei Deklaration.

**2.Referenztypen/Klassentypen**

Zeichenketten, Dateien, Felder, Arrays oder Datenstrukturen.

Variablen mit Klassentyp enthalten immer Referenzen auf Objekte . Also Initialisierung erst nach der Erstellung der Objekte.

## b) Zeitmessung

(b)Ist es möglich, die genaue Ausführungsdauer einer Anweisung anzugeben? Begründen Sie!

Es wäre möglich, anhand der Taktzahl des Computers und den Ausführungsschritten einer Anweisung, eine Ausführungsdauer zu bestimmen. Da jedoch auf dem Computer noch verschiedene andere Prozesse laufen (round robin) kann es gerade bei kurzen Messungen zu großen Ungenauigkeiten kommen. Selbst bei der Reduzierung aller Prozesse bis  auf ausschließlich System relevanten-Prozesse, kann eine genaue Messung nicht durchgeführt werden.

## c) Hashcode

(c)Ist für ein Array, wie z.B. mit `int[] ia = new int [8];` eingeführt, der Ausdruck `ia.toString();` zulässig? Begründen Sie!

ia.toString() ist zulässig. Der Ganzzahlwert eines Arrays heißt Hashcode, dieser soll zu jedem Objekt eine möglichst eindeutige Integer zahl (sowohl positiv als auch negativ) liefern, die das Objekt identifiziert. Mit toString() wird dieser Wert als String ausgegeben.

## d) finalize()

(d) Unter welchen Umständen wird die Methode `finalize()` der Klasse `Object` ausgeführt? Ist garantiert, dass diese Methode für *jedes* Objekt irgendwann einmal aufgerufen wird?

Einen Destruktor, der wie in C/C++ am Ende eines Gültigkeitsbereiches einer Variablen aufgerufen wird, gibt es in Java nicht. Wohl ist es möglich, eine Methode finalize() für solche Aufräumarbeiten zu überschreiben, die Finalizer genannt wird. Der Garbage-Collector ruft die Methode finalize() immer dann auf, wenn er ein Objekt entfernen möchte.

Objekte von Klassen, die eine finalize()-Methode besitzen, kann Suns JVM nicht so schnell erzeugen und entfernen wie Klassen ohne finalize(). Das liegt auch daran, dass der Garbage-Collector vielleicht mehrmals laufen muss, um das Objekt zu löschen.

## e) Garbage-Collector

(e) Wie kann man in einem Java-Programm den nicht mehr benötigten Speicherplatz eines Objektes explizit wieder freigeben?

Wird ein Objekt nicht mehr referenziert, findet der Garbage-Collector dieses Objekt und gibt den nicht mehr verwendeten Speicherplatz frei.

# *Aufgabe (3) - Nebenläufigkeit*

## a) Kritischer Abschnitt

(a) Was versteht man unter einem kritischen Abschnitt?

Kritische Abschnitten müssen mit Schutzmaßnahmen abgesichert werden, damit der konkurrierende Zugriffe durch mehrere Threads auf gemeinsam genutzte Daten gewährleistet werden kann.

## b) Wechselseitiger Ausschluss

(b) Wie läßt sich in Java wechselseitiger Ausschluss garantiereren? Welche Objekte können als Sperre dienen?

In einem kritischen Abschnitt gilt, jedes Objekt kann als Monitor dienen. Ein kritischer Abschnitt ist synchronized und wird durch ein Sperrobjekt, gegen ungewollten eintritt anderer Threads gesichert.

## c) Deprecated Methoden

(c) In der API-Dokumentation werden die Methoden `suspend()`, `resume()` und `stop()` der Klasse `Thread` als *deprecated* gekennzeichnet und werden möglicherweise in zukünftigen Versionen der Klassenbibliothek nicht mehr vorhanden sein. Davon abgesehen, warum wird dringend davon abgeraten, diese Methoden zu verwenden? (3)

Thread.**suspend**() (warten) und Thread.**resume**() (rückkehren) gibt die Möglichkeit einen Thread zu pausieren. Diese Methoden sind aber aus gutem Grund veraltet (deprecated). Intensive Nutzung kann zu einem Deadlock führen.

Bei Thread.**stop**() wird der Thread beendet, egal was er zuvor unternommen hat. Es wird ihm keine Chance mehr gegeben, seinen Zustand konsistent zu verlassen.

## d) String und StringBuffer

(d) Auf ein Objekt der Klasse `java.lang.String` sollen mehrere Threads zugreifen können. Was ist zu tun? Begründen Sie! Gilt Ihre Antwort auch für ein `StringBuffer`-Objekt? (4)

## e) wait()

(e) Innerhalb eines Methodenrumpfes wird die Methode `wait()` aufgerufen. Bei Ausführung des Programmes wird an dieser Stelle aber eine `IllegalMonitorStateException` ausgelöst. Woran mag das liegen?

Der aktuelle Thread wartet an dem aufrufenden Objekt darauf, dass er nach einem *notify()/notifyAll()* weiterarbeiten kann. Der aktive Thread muss natürlich den Monitor des Objekts belegt haben. Andernfalls kommt es zu einer IllegalMonitorStateException.

# *Aufgabe (4) - Vererbung*

## a) extends

(a) Die Methode `A.meth(X,Y)` wird in der Klasse `B` überschrieben (s.u.). In welchem Zusammenhang müssen die Klassen `AException` und `BException` stehen? Begründen Sie!

```
class A {...
 void meth(X x, Y y)throws AException{..}
}
class B extends A{...
 void meth(X x, Y y)throws BException{...}
}
```

Die Methode meth(X,Y) in B kann meth(X,Y) in A nicht überschreiben.

void meth(X x, Y y) in A wirft eine BException aber die überschriebene Methode kann keine BException werfen. Wird eine Methode überschrieben, darf die überschreibende keine neue checked Exception erzeugen!

Die checked Exception ist Bestandteil des Methodenkontraktes und muss daher in der Signatur mittels throws-Klausel angegeben werden .

## b) Vor- / Nachbedinungen

(b) Eine Methode `B.meth(...)` werde von einer Methode `A.meth(...)` überschrieben. Wie verhält es sich mit den Vor- und Nachbedingungen der beiden Methoden, dürfen sie abgeschwächt oder verschärft werden? Begründen Sie Ihre Antwort mit dem Ersetzungsprinzip!

Die überschreibende Methode (A.meth) darf weder die Vorbedingung verschärfen noch die Nachbedingung abschwächen . Die überschreibende Methode muss den Vertrag der überschriebenen auch erfüllen!

**Vorbedingung:**

A.meth() muss in allen Fällen an die Stelle von B.meth() treten können.

**Nachbedingung:**

A.meth() muss B.meth() mit Garantien übernehmen können.

## c) Unterklassen

(c) Es sei `B` eine Unterklasse von `A`. Wie wird gewährleistet, dass bei Erzeugung eines `B`-Objektes auch ein Konstruktor von `A` aufgerufen wird. Wie lautet die Regel, wenn `B` überhaupt keinen Konstruktor definiert?

Jedes Objekt durchläuft, für jeden Klassentyp zu dem es gehört , eine Konstruktorinitialisierung um die Klasseninvariante erstmalig zu etablieren .

Reihenfolge:

- zuerst Konstruktor der Basisklasse,
- dann der der abgeleiteten Klasse

Nur die Wahl des Konstruktors der Basisklasse ist mittels super(...) beeinflussbar.

## d) Konsruktoren

(d) Warum werden Konstruktoren nicht an abgeleitete Klassen vererbt?

Durch Vererbung an abgeleitete Klassen, kann eine Abschwächung der Invariante der Oberklasse entstehen. Die Invarianten beinhaltet die Zusicherung über den Zustand aller Objekte der Klasse während ihrer gesamten Lebensdauer . Diese Zusicherung kann bei einer Vererbung nicht mehr gegeben werden.

# *Aufgabe (5) - Schnittstellen*

## a) Runnable

(a)Im Paket `java.lang` ist die Schnittstelle `Runnable` deklariert. Kann man mit

```
Runnable r = new Runnable();
```

ein `Runnable` Objekt erzeugen? Begründen Sie!

Die Runnable Schnittstelle ist eine **abstrakte** Klasse (**abstract class**). Von dieser Klasse können keine Exemplare gebildet werden, und der Versuch einer Objekterzeugung führt zu einem Compilerfehler.

Eine Oberklasse besitzt dabei Vorgaben für die Unterklasse, das heißt, alle Unterklassen erben die Methoden. Ein Exemplar der Oberklasse selbst muss nicht existieren.

Um dies in Java auszudrücken, wird der Modifizierer **abstract** an die Typdeklaration der Oberklasse gehängt .

Ansonsten verhalten sich die **abstrakten** Klassen wie normale, enthalten die gleichen Eigenschaften und können auch selbst von anderen Klassen erben.

**Abstrakte** Klassen sind das Gegenteil von **konkreten** Klassen.

## b) Schnittstellenattribute

(b)Können in einer Schnittstellendefinition Attribute eines Objektes definiert werden?

Eine Schnittstelle enthält keine Implementierungen, sondern deklariert nur den Kopf einer Methode :
Modifizierer, den Rückgabetyp und die Signatur – ohne Rumpf.

Ausnahme sind Static final-Variablen (benannte Konstanten) diese sind in einer Schnittstelle erlaubt, statische Methoden jedoch nicht.

In einer Schnittstelle werden keine Methoden aus programmiert und keine Objekt-variablen deklariert.

## c) Schnittstelle vs. abstrakte Klasse

(c)Erläutern Sie den Unterschied zwischen einer Schnittstelle und einer abstrakten Klasse!

**Abstrakte Klassen** könne Attribute und implementierte Methoden enthalten . Eine Klasse kann nur eine abstrakte Klasse erweitern (Vererbungsjoker weg) . Sie besitzen einen Konstruktor und sind von Object abgeleitet, sind also Bestandteil des Klassenbaums . Eine abstrakte Klasse gibt die Möglichkeit von einer Oberklasse lediglich Methoden für die Unterklassen vorzugeben ohne wissen zu müssen, wie sie diese zu implementieren sind.

**Schnittstellen** enthalten nur Konstanten und abstrakte Methoden . Eine Klasse kann viele Schnittstellen implementieren . In einer Schnittstelle muss nichts Initialisiert werden. Jede Schnittstelle kann Wurzel einer Schnittstellenhierarchie sein .

## d) Runnable Thread

(d)Die Klasse `Thread` besitzt u.a. einen Konstruktor `Thread(Runnable)`, dessen Parameter den Schnittstellentyp `Runnable` besitzen muss. Was bewirkt dieser Konstruktor? An welchen Stellen kann bei Verwendung dieses Konstruktors die Methode `run()` implementiert werden? Vergleichen Sie mit der Schnittstelle `ActionListener` und ihrer Verwendung bei der Registrierung an der Ereignisquelle mittels `addActionListener(..)`.

Das **Runnable Interface** sollte von jeder Klasse implementiert werden, in deren Instanz ein Thread ausführt wird.

Nun reicht es nicht aus, einfach die run()-Methode einer Klasse direkt aufzurufen, denn dann wäre nichts nebenläufig, sondern wir würden einfach eine Methode sequenziell ausführen.

Damit der Programmcode parallel zur Applikation läuft, müssen wir ein Thread-Objekt mit dem Runnable verbinden und dann den Thread explizit starten.

Dazu übergeben wir dem Konstruktor der Klasse Thread eine Referenz auf das Runnable-Objekt und rufen start() auf.

Nachdem start() für den Thread eine Ablaufumgebung geschaffen hat, ruft es intern selbstständig die Methode run() genau einmal auf.

Läuft der Thread schon, so löst ein zweiter Aufruf der start()-Methode eine IllegalThreadStateException aus.

**Das ActionListener-Interface** empfängt Action *events*. Klassen welche dieses Interface implementieren müssen ein Objekt besitzen, welches die *addActionListener* Methode verwendet. Tritt nun ein *action event* auf, wird die Methode *actionPerformed* aufgerufen.

Der **ActionListener** hat eine abstrakte Methode, d.h. Eine Methode mit Vertrag aber ohne Implementierung :

```
interface ActionListener{
        public abstract void
                actionPerformed(ActionEvent);
}
```

## *Aufgabe (6) – Implementierung eines Beispiels*

## a) Stapel implements Stapel1000

Gegegeben sei die folgende Schnittstelle für einen Stapel (Stack) beschränkter Größe:

```
interface Stapel1000{
// Maximale Größe des Stapels
public static final int MAXSIZE = 1000;

//legt elem auf dem Stapel ab
public void push(Object elem)throws StapelVollException;

//liefert das oberste Element des Stapels und entfernt es
public Object pop() throws StapelLeerException;

//versetzt den Stapel in den Anfangszustand 'leer'
public void reset();

//liefert das oberste Element des Stapels, ohne es zu entfernen
public Object peek();

//liefert die momentane Größe des Stapels
public int getSize();
}
```

(a) Definieren Sie eine Klasse `Stapel`, die diese Schnittstelle implementiert und zwei Konstruktoren besitzt:
1) einen parameterlosen Konstruktor, der den Stapel in den Zustand „leer“ versetzt, und
2) einen Konstruktor `Stapel(Object[] inhalt)`, der den Stapel mit den Elementen von `inhalt` füllt und möglicherweise eine Ausnahme wirft.
Die Elemente des Stapels sollen intern in einem Feld (Array) abgelegt werden. Auf dieses Feld sol außerhalb dieser Klasse nicht zugegriffen werden können. Die Klassen `StapelVollExceptior` und `StapelLeerException` werden als bereits definiert angesehen.
**Warnung**: Eine Implementierung als verkettete Liste oder mit Hilfe von `java.util.Vector` wird nicht akzeptiert!

## b) Klasseninvarianten, Vor-/Nachbedingungen

(b) Formulieren Sie die Klasseninvariante der Implementierung in (a) sowie die Vor- und Nachbedingungen aller Methoden.

## c) Methode test()

(c) Schreiben Sie eine Methode `test()`, das die Zeichenketten `"unten"`, `"mitte"`, `"oben"` ablegt und danch wieder entfernt und ausgibt. Geben Sie die ausgegebenen Zeichenketten in der richtigen Reihenfolge an.
Weiterhin soll die Methode prüfen, ob die Ausnahmen `StapelVollException` und `StapelLeerException` tatsächlich auftreten!

```
public static void main(String[] args) throws StapelVollException, StapelLeerException {
        Object[] testArr = {"unten", "mitte", "oben"};
        Stapel stapel = new Stapel(testArr);
        System.out.println("Size : " + stapel.getSize());


        while (stapel.getSize() >= 0) {
            System.out.println("Peek[" + stapel.getSize() + "] : " + stapel.peek());
            System.out.println("Element : " + stapel.pop().toString());
        }

        int i = 0;
        for (i = 0; i < testArr.length; i++) {
            stapel.push(testArr[i]);
            System.out.println("Peek[" + stapel.getSize() + "] : " + stapel.peek() + "\n");
        }
    }
```

## d) Stapel mit nebenläufigem Zugriff

(d) Wie können Sie Ihre Implementierung "thread-sicher" machen, d.h. den nebenläufigen Zugriff auf einen Stapel durch mehrere Threads so einschränken, dass keine Wettbewerbssituationen (race conditions) eintreten können?

Der kritische Abschnitt sollte eine Folge von Befehlen beinhalten, die ein Thread vollständig abarbeiten muss,
ohne dass ein anderer Thread währenddessen den kritischen Abtritt betritt. Umgibt man den kritischen Bereich mit Semaphoren, kann nun ein Thread in einer atomaren Methode "ungestört" arbeiten. So werden Race Conditions verhindert.