Name:	**Studiengang: ☐ B.A. | ☐ M.A.**
Vorname:	**Matrikelnummer:**
Studienfächer:	**Fachsemester:**

Allgemeine Hinweise:

1. Überprüfen Sie bitte, ob Sie alle Seiten der Klausurangabe vollständig erhalten haben (Gesamtzahl: **10**)

2. **Bearbeitungszeit: 90 Minuten**, maximal erreichbare **Punktzahl: 48**. Die jeweils erreichbare Punktzahl ist bei jeder Frage angegeben. Bitte teilen Sie Ihre Arbeitszeit entsprechend ein.

3. Denken Sie daran, die Daten oben einzutragen, **bevor** Sie mit der Bearbeitung beginnen.

4. Verwenden Sie für die Beantwortung aller Fragen diese Klausurangabe. Sie können jederzeit auch die Rückseiten beschreiben, falls der Platz auf der Vorderseite nicht ausreichen sollte. Bitte geben Sie in jedem Fall an, auf welche Frage sich die Lösung jeweils bezieht. Bei Multiple-Choice-Fragen treffen Sie bitte die Auswahl Ihrer Antworten ebenfalls auf der Klausurangabe.

5. Benutzen Sie keine Bleistifte, keine rot schreibenden Stifte und kein TippEx, o.ä.

6. Zugelassene Hilfsmittel: **keine**

7. Mobiltelefone sowie Computer am Arbeitsplatz - auch ausgeschaltet - sind **nicht zugelassen**.

8. Bitte legen Sie Lichtbildausweis und Studierendenausweis gut sichtbar vor sich, damit Ihre Identität möglichst störungsfrei überprüft werden kann.

9. Geben Sie keine mehrdeutigen (oder mehrere) Lösungen an. In solchen Fällen wird stets die Lösung mit der geringeren Punktzahl gewertet. Eine richtige und eine falsche Lösung ergeben also 0 Punkte.

10. Wenden Sie sich bei Unklarheiten in den Aufgabenstellungen immer an die Aufsichtsführenden. Hinweise und Hilfestellungen werden dann, falls erforderlich, offiziell für alle Teilnehmer durchgegeben.

Aufgabe 1 - Google Java Style Guide (3 Punkte)

Markieren Sie in folgender Tabelle, welche Schreibweise für welche Art von Name gemäß dem Style Guide erlaubt (✓) bzw. nicht erlaubt (✗) ist. Nach vollständiger Bearbeitung ist kein Feld mehr leer.

	Klasse	Methode	Konstante	Feld (nicht konstant)	Parameter	lokale Variable	Typ-Variable
UpperCamelCase							
lowerCamelCase							
UPPER_CASE							

Aufgabe 2 - Idiome (3 Punkte)

Nennen sie drei positive Eigenschaften einer Klasse, die das Immutable-Idiom erfüllt.

- kein ungültiger Zustand möglich
- kein veränderlicher Zustand
    1) -> geringe Komplexität
    2) -> automatisch thread-safe
- kann ohne defensive Kopie weitergegeben werden; braucht kein clone()
- guter Schlüssel für Maps/Sets (und oft ein guter Baustein für komplexere Klassen)

Aufgabe 3 - Code Smells / Refactoring (2+2+2 Punkte)

Führen Sie am folgenden Code drei von der Art her unterschiedliche Verbesserungen hinsichtlich Verständlichkeit bzw. Lesbarkeit durch. Die Änderungen dürfen nötigenfalls aufeinander aufbauen. Begründen Sie jeweils kurz, warum es sich um eine Verbesserung handelt.

```
1  public static boolean doStuff (int i, int j) {
2            int s1 = 0;  int a = 0;
3            int a1 = Math.abs(i);
4                    int a2 = Math.abs(j);
5            String tempI = Integer.toString(a1);
6            int[] arr1 = new int[tempI.length()];
7            for (int k = 0; k < tempI.length(); k++) {
8                    arr1[k] = tempI.charAt(k) - '0';
9                    }
10                   String tempJ = Integer.toString(a2);
11                   int[] array2 = new int[tempJ.length()];
12                   for (int k = 0; k < tempJ.length(); k++) {
13                           array2[k] = tempI.charAt(k) - '0';
14           }
15           for (int k = 0; k < arr1.length; k++) {
16                   s1 = s1 + arr1[k];
17           }
18           for (int k = 0; k <= array2.length; k++) {
19
20                   a = a + array2[k];
21           }
22           if (a == s1) {
23                   return true;
24                   } else {
25                   return false;
26           }
27 }
```

Aufgabe 4 - Java Collections Framework (1+1+1+1+1 Punkte)

Container

a) Was ist der Unterschied zwischen FIFO- und LIFO-Verhalten?

Reihenfolge, in der eingefügte Elemente wieder entfernt werden: First-In-First-Out vs. Last-In-First-Out

b) Beschreiben sie den wichtigsten konzeptuellen Unterschied zwischen List und Collection.

Sortiert vs. unsortiert; definierte vs. undefinierte Reihenfolge der Elemente

c) Beschreiben sie den wichtigsten konzeptuellen Unterschied zwischen Set und Collection.

Ein Set enthält gleiche Elemente nur höchstens einmal (Gleichheit definiert durch equals()).

equals, hashCode, compareTo

d) Was muss gelten, damit die von einem Comparator c erzeugte Ordnung auf einer Menge M von Elementen als "konsistent zu equals" gilt?

$\forall$ e1, e2 $\in$ M: (c.compare(e1, e2) == 0) $\Leftrightarrow$ e1.equals(e2)
oder:
Der Comparator bezeichnet zwei Elemente genau dann als äquivalent, wenn sie auch von equals() für äquivalent gehalten werden.
(Schreibweise "e1.compareTo(e2)" statt "c.compare(e1, e2)" wird auch noch akzeptiert)

e) Was sollte jede Implementierung der Methode hashCode bezüglich der Methode equals gewährleisten?

$\forall$ Objekte x, y: x.equals(y)==true $\Rightarrow$ (x.hashCode() == y.hashCode())
oder:
Wenn zwei Objekte gleich sind, dann müssen sie den gleichen Hash-Code haben.
Wenn zwei Objekte verschieden sind, dann müssen sie deshalb keine unterschiedlichen Hash-Codes haben.

Aufgabe 5 - Komplexitätsanalyse (2+3+3+7 Punkte)

a) Sortieren Sie die folgenden neun Komplexitätsgrade nach aufsteigender Wachstumsgeschwindigkeit. Dabei ist c eine Konstante > 1 und n der wachsende Parameter.

$O(n^{1/c})$	$O(n^3)$	O(n*log(n))
O(n)	O(1)	$O(c^n)$
O(log(log(n)))	O(log(n))	$O(n^2)$

O(1) O(log(log(n))) O(log(n)) O(n^(1/c)) O(n) O(n*log(n)) O(n^2) O(n^3) O(c^n)

Gegeben sei für die restlichen Teilaufgaben der Komplexitätsanalyse folgender Code:

```
1  void quicksort(int[] a) {
2   quicksort(a, 0, a.length - 1);
3  }
4
5  void quicksort(int[] a, int lo, int hi) {
6   if (lo < hi) {
7    int p = partition(a, lo, hi);
8    quicksort(a, lo, p - 1);
9    quicksort(a, p + 1, hi);
10  }
11 }
12
13 int partition(int[] a, int lo, int hi) {
14  int pivot = a[hi];
15  int i = lo - 1;
16  for (int j = lo; j < hi; j++) {
17   if (a[j] < pivot) {
18    i = i + 1;
19    swap(a, i, j);
20   }
21  }
22
23  swap(a, i + 1, hi);
24  return i + 1;
25 }
26
27 void swap(int[] a, int x, int y) {
28  int temp = a[x];
29  a[x] = a[y];
30  a[y] = temp;
31 }
```

Gehen Sie bei Ihren Komplexitätsanalysen - soweit relevant - vom "average case" aus, nicht vom "worst case".

b) Welchen Komplexitätsgrad O(?) bezüglich des **Speicherbedarfs** weist die Funktion **`quicksort(int[] a)`** insgesamt auf? Begründen Sie Ihre Antwort.

O(log(n))
+ Konstanter Speicherbedarf pro Stackframe / Aufruf
+ baumartige, rekursive Aufrufe; n wird pro Aufruf halbiert -> Rekursionstiefe log(n)

c) Handelt es sich um einen In-Place-Algorithmus? Begründen Sie Ihre Antwort.

Ja
+ Eingabe-Array wird modifiziert / überschrieben
+ zusätzlicher Speicherbedarf ist nur sublinear: O(log(n)), siehe Teilaufgabe b)

d) Welchen Komplexitätsgrad O(?) bezüglich des **Laufzeitverhaltens** weist die Funktion **`quicksort(int[] a)`** insgesamt auf? Begründen Sie Ihre Antwort ausführlich.

O(n*log(n))
(wurde u.A. in Vorlesung diskutiert)

Aufgabe 6 - Planning Poker (2 Punkte)

Planning Poker ist eine Heuristik, um den Aufwand von Aufgaben im Bereich weniger Personentage genauer abzuschätzen, kurz bevor deren Bearbeitung beginnt. Beschreiben sie kurz das Vorgehen, nennen sie dabei einzuhaltende Grundregeln und was mit deren Einhaltung erreicht werden soll.

+ höhere Wahrscheinlichkeit, alle relevanten Informationen zu berücksichtigen
+ geringere Varianz durch Abstimmung und Mittelwert-/Medianbildung
+ Vermeiden von fachfremden Fehlern wie Anchoring oder zu präzisen Zahlen

Aufgabe 7 - Semantic Versioning (3 Punkte)

Welche Information kann der Benutzer einer Software-Bibliothek, die sich an "Semantic Versioning 2.0.0" hält, aus der Veränderung der Versionsnummer gewinnen? Beantworten Sie die Frage, indem Sie die Interpretation der drei Haupt-Versionsnummern (Major, Minor, Patch) und wie diese bei einer Code-Änderung zu verändern sind, beschreiben.

- vor 1.0.0 gibt es keine Auflagen
- Veränderungen bzgl. der nach außen sichtbaren API / öffentlichen API:
    - keine Veränderung: Patch-Version inkrementieren
    - abwärtskompatible Veränderung: Minor-Version inkrementieren
    - nicht abwärtskompatible Veränderung: Major-Version inkrementieren
- Verändern einer höheren Stelle setzt alle niedrigeren Stellen auf 0 zurück

Aufgabe 8 - Inversion of Control (2+3 Punkte)

a) Beschreiben Sie die zentrale Idee des Paradigmas "Inversion of Control".

Die Steuerung des Kontrollflusses wird teilweise abgegeben: Stichwort "Don't call us, we'll call you". Eigener Code wird nicht selbst gestartet, sondern lediglich bei einer Bibliothek / Framework / Engine registriert. Diese ruft ihn dann nach eigenem Ermessen auf. Entkopplung von Implementierung und Ausführung.

b) Nennen Sie drei Beispiele, wo dieses Paradigma typischerweise Anwendung findet.

+ Game Engines (Unity, Unreal)
+ Game Frameworks (LibGDX)
+ Application Server (Tomcat)
+ Application Frameworks (Spring)
+ Dependency Injection Bibliotheken (Guice, Dagger (2), Spring DI)
+ jegliche Plugin-Schnittstelle
+ Actor Model (Akka)
+ Shadercode

Aufgabe 9 - Dependency Injection (1+2+3 Punkte)

a) Wo am eigenen Code tritt bei Verwendung eines Dependency Injection Frameworks "Inversion of Control" auf ?

Initialisierungscode (Konstruktoren / Factory-Methoden / Setter / Schreibzugriffe auf Felder) wird nicht direkt, sondern durch das DI-Framework aufgerufen.

b) Welches Java-Sprachfeature wird typischerweise benutzt, um Dependencies manuell zu konfigurieren? Wofür wird dieses Feature außerhalb des Kontexts Dependency Injection noch verwendet?

+ Annotations bzw. Annotation Processing
+ Mehr Informationen in den Code annotieren, für Compiler und Menschen

c) Nennen Sie zwei Vorteile und einen Nachteil, die die Verwendung eines Dependency Injection Frameworks typischerweise mit sich bringt.

+ leichtere Testbarkeit
+ Betonung der Trennung von Interface und Implementierung: Software-Design-Prinzipien Lockere Kopplung und Seperation of Concerns werden gefördert
+ bessere Übersichtlichkeit und Unabhängigkeit bei grossen Projekten, da nur Interfaces bekannt sein müssen, aber nicht die Implementierung(en) dahinter
+ Austausch einzelner Komponenten durch Konfigurationsänderung; ohne Code-Änderung
+ Komplexität innerhalb der einzelnen Klassen wird verringert, da sich so wenig wie möglich um Dependencies gekümmert wird
- erhöhte Grundkomplexität: das Framework muss beherrscht werden; es entstehen typischerweise mehr Klassen und Interfaces
- Fehler in der Konfiguration bzw. Relation der Dependencies untereinander werden nicht mehr zur Compilezeit sichtbar, sondern erst zur Laufzeit.