Übungsaufgabe

Java-Einführung: Syntax, Typen, Generics, Speicherverhalten und Unit-Tests

Universität:	Technische Universität Berlin
Kurs/Modul:	Algorithmen und Datenstrukturen
Erstellungsdatum:	September 6, 2025

Aufgabe 1: Java-Syntax und Typen

In dieser Aufgabe arbeiten Sie mit Grundzügen der Java-Syntax, primitive Typen versus Referenztypen, Generics, Speicherverhalten sowie Unit-Tests. Beantworten Sie die folgenden Unteraufgaben sorgfältig.

a) Betrachten Sie folgenden Java-Code:

```
public class Beispiel {
  public static void main(String[] args) {
    int a = 5;
    Integer b = 10;
    double d = 2.5;
    boolean flag = true;
    char ch = 'A';
    String s = "Hallo";
  }
}
```

Welche Typen sind primitv (nennen Sie die Namen) und welche sind Referenztypen? Geben Sie außerdem für jeden primitiven Typen einen passenden Wrapper-Typen an (falls vorhanden) und erläutern Sie kurz die Zuordnung zwischen Wert und Objekt.

b) Betrachten Sie folgenden Code-Ausschnitt:

```
import java.util.*;
List<Integer> list = new ArrayList<>();
list.add(1);
list.add(2);
```

Erläutern Sie anhand dieses Beispiels, was der Typparameter bedeutet, ob zur Laufzeit eine Unterscheidung der generischen Typen existiert und was unter Type Erasure zu verstehen ist. Welche Auswirkungen hat dies auf Typprüfungen zur Laufzeit?

c) Speicherverhalten. Diskutieren Sie, wie die oben deklarierten Variablen im Speicher abgelegt werden (Stack vs. Heap), welche Rolle Autoboxing/Unboxing spielt und welche Auswirkungen dies auf Garbage Collection haben kann. Gehen Sie auf Unterschiede zwischen primitiven Typen und Referenztypen ein.

d) Unit-Tests. Skizzieren Sie, wie Sie in JUnit 5 einen Unit-Test entwerfen würden, der eine einfache Java-Funktion testet (z. B. eine Methode, die zwei Werte addiert). Beschreiben Sie Aufbau, Annotations (z. B. @Test), Assertions und typisches Testverhalten.

Aufgabe 2: Generics, Speicherverhalten und Unit-Tests in Java

Diese Aufgabe vertieft das Thema Generics, Speicherverhalten in Java-Sammlungen sowie das Design von Unit-Tests. Verfassen Sie zu den Unteraufgaben ausführliche, konzeptionelle Antworten bzw. Skizzen, ohne fertigen Code bereitzustellen.

a) Generische Sammlungen. Beschreiben Sie, wie generische Typen in Java Collections genutzt werden, z. B. eine Liste von Strings oder eine Map von String zu Integer. Gehen Sie dabei auf Typargumente, Typ-Sicherheit zur Compile-Zeit und die Rolle von Typ-Erasure zur Laufzeit ein. Erörtern Sie auch, welche Vorteile Generics gegenüber rohen Typen haben.

b) Speicherverhalten bei Generics. Diskutieren Sie, wie Generics das Speichermanagement beeinflussen (z. B. Bezug auf Autoboxing, Instanziierung von Objekten, Wrapper-Typen) und welche typischen Fallstricke bei Speicherverbrauch und GC auftreten können.

c) Unit-Tests für generische Datenstrukturen. Entwerfen Sie einen Testplan bzw. eine Testklasse (ohne konkreten Code) für eine generische Datenstruktur (z. B. eine Box<T> oder eine einfache Stack<T>). Beschreiben Sie Testziele, Testfälle, notwendige Setup-Schritte, Randfälle und wie Sie Typ-Sicherheit in den Tests prüfen würden (z. B. verschiedene Typen T wie Integer, String, Benutzerdefinierte Typen).

d) Ausblick: Test-Strategien. Nennen Sie kurze Leitlinien zu Unit-Tests in Java (JUnit 5) im Kontext von Speicherkonsistenz, Generics und Verhalten unter Multithreading. Welche Arten von Tests würden Sie ergänzend empfehlen (z. B. Property-Based-Tests, Integrationstests) und warum?

Lösungen

Lösung zu Aufgabe 1: Java-Syntax und Typen

In den folgenden Antworten werden die Kernpunkte kompakt zusammengefasst. Anmerkungen zu Typen beziehen sich auf die in der Aufgabe gezeigten Deklarationen.

a) Typen in dem gezeigten Code

- Primitive Typen (im gezeigten Ausschnitt): - int a = 5; - double d = 2.5; - boolean flag = true; - char ch = 'A';

- Referenztypen (im gezeigten Ausschnitt): - Integer b = 10; - String s = "Hallo";

- Zuordnung Wrapper-Typen (falls vorhanden) und Bedeutung der Zuordnung: - int -> Integer - double -> Double - boolean -> Boolean - char -> Character

- Erläuterungen zur Zuordnung Wert <-> Objekt: - Primitive Typen speichern Werte direkt in der Variable (meist im Stack-Frame des Aufrufkontexts). Sie sind keine Objekte. - Wrapper-Typen sind Objekte auf dem Heap. Sie kapseln den jeweiligen primitiven Wert in einem unveränderlichen Objekt (z. B. Integer enthält ein int-Feld). - Autoboxing/Unboxing: Der Compiler wandelt bei Bedarf automatisch primitive Werte in Wrapper-Objekte bzw. Wrapper-Werte zurück (z. B. int zu Integer bzw. Integer zu int). Typische Beispiele: - Integer x = 5; // Autoboxing - int y = x; // Unboxing - Besonderheiten: Wrapper-Objekte können null sein (bei Referenzvariablen), Primitivwerte jedoch nicht. Integer.valueOf(..) verwendet Caching (z. B. -128..127), wodurch Wiederverwendung von Objekten möglich ist; ansonsten wird ein neues Wrapper-Objekt erzeugt.

- Weitere Anmerkung zur String-Referenz: - s ist eine Referenz auf ein String-Objekt auf dem Heap. Literale wie "Hallo" werden in der String-Pool (intern) verwaltet und können intern geteilt werden.

b) Typparameter, Type Erasure und Laufzeit

- Typparameter bedeutet, dass die Liste so deklariert ist: List<Integer> list = new ArrayList<>(); - Der Typparameter erzwingt zur Compile-Zeit, dass nur Integer-Objekte in die Liste eingelegt werden dürfen, und dass aus der Liste beim Zugriff Integer-Objekte zurückgegeben werden. - Laufzeit (Type Erasure): - Generics werden zur Laufzeit „ausgeerntet“; der Typparameter verschwindet in der Bytecode-Generierung. Die JVM kennt also zur Laufzeit nur List bzw. ArrayList, nicht List<Integer>. - Als Folge: - Zur Laufzeit existieren keine Informationen, ob es sich um eine List<Integer>, List<String> o. ä. handelt. - Methoden wie get() liefern zur Compile-Zeit den erwarteten Typ (Integer), zur Laufzeit bleibt es allerdings eine allgemeine Referenz (und ggf. wird ein cast durch den Compiler eingefügt). - Auswirkungen auf Typprüfungen: - Generische Typen bieten Compile-Zeit-Sicherheit, nicht jedoch Standard-Laufzeit-Typsicherheit. Raw-Typen oder Mischformen (z. B. List raw = new ArrayList(); List<String> s = raw) verursachen Warnungen bzw. ClassCastException zur Laufzeit, wenn Elemente typfremd verwendet werden. - Praktische Folge: - Die Zeile list.add(1); benutzt Autoboxing, um den primitiven Wert 1 in ein Integer-Objekt umzuwandeln, das in der Liste gespeichert wird. - list.get(0) liefert vom Compiler her einen Integer; zur Laufzeit handelt es sich um ein Objekt vom Typ Integer (aufgrund der Objekthülle), aber die Typinformation der generischen Grenze existiert nicht mehr.

c) Speicherverhalten (Stack vs. Heap, Autoboxing/Unboxing, GC)

- Speicherorte: - Lokale Primitive und Referenzen (wie a, d, flag, ch, list) befinden sich typischerweise im Stack/MM-Feld des aufrufenden Threads (bzw. in der lokalen Variablenliste der Methode). Die JVM kann diese Werte auch in CPU-Register legen, abhängig vom Optimierer. - Objekte wie Integer, Strings (und deren Inhalte) liegen auf dem Heap. Die Referenzen (z. B. b, s) liegen ebenfalls im Stack/Heap-Verbund. - Autoboxing/Unboxing: - Beim Zuweisen eines primi-

tiven Werts an ein Wrapper-Objekt (z. B. Integer b = 10) entsteht im Heap ein Integer-Objekt (bzw. ggf. aus dem Cachebereich von Integer.valueOf), und die Referenz wird auf dem Stack gespeichert. - Bei Berechnungen, die primitive Werte erfordern (z. B. b + 5), wird b automatisch zu int ausgepackt (Unboxing), die Operation durchgeführt und ggf. das Ergebnis wieder verpackt (Autoboxing) bei Zuweisung in ein Integer-Objekt. - Garbage Collection: - Primitivwerte an lokalen Variablen werden nicht direkt GC-verwaltet; wenn sie jedoch in Feldern von Objekten gespeichert sind, deren Objekte GC-kollabieren, könnten auch diese Werte „mitgezogen“ werden. - Wrapper-Objekte (z. B. Integer) entstehen durch Autoboxing und verbleiben im Heap, solange Referenzen existieren. Nicht mehr referenzierte Wrapper-Objekte werden durch den GC gesammelt. - Strings: Literale wie "Hallo" werden in einem String-Pool verwaltet; unveränderliche Strings können mehrfach geteilt bzw. referenziert werden, wodurch einige Speicheroptimierungen möglich sind. - Unterschiede primitive vs Referenz: - Primitives speichern Werte direkt; Referenztypen speichern Verweise auf Objekte; Primitives haben keine Identität (Equals vergleicht Werte), Referenzobjekte haben Identität über Referenzen. - Default-Werte für Felder (nicht lokal) sind 0, 0.0, false bzw. ’0̆000’ für primitive Typen; Referenzen default auf null.

d) Unit-Tests. Skizzieren Sie, wie in JUnit 5 ein Unit-Test entworfen würde (Beispiel: Methode addiere zwei Werte)

- Aufbau: - Testklasse z. B. CalculatorTest (oder MathOpsTest), wobei die zu testende Methode z. B. public static int add(int a, int b) heißt. - Struktur: Eine oder mehrere Testmethoden, jede mit der Annotation @Test. - Optionaler Aufbau: @BeforeEach/ @AfterEach für Setup bzw. Cleanup, sofern benötigt.

- Beispielhafte Testfälle (skizzenhaft, ohne fertigen Code): - Testfall: Addition zweier positiver Werte - Erwartetes Ergebnis: add(2, 3) == 5 - Testfall: Addition mit Null - Erwartetes Ergebnis: add(0, 7) == 7 - Testfall: Addition mit negativen Werten - Erwartetes Ergebnis: add(-4, 6) == 2 - Testfall: Randfall Overflow (optional, je nach Spezifikation) - Erwartetes Verhalten: z. B. Integer overflow gemäß Java-Semantik (Integer.MAX$_V ALUE + 1 == Integer.MIN_V ALUE$)

- Assertions-API: - Verwendung von Assertions.assertEquals(expected, actual) bzw. Assertions.assertEquals(actual, message). - Typische Assertions in JUnit 5: assertEquals, assertNotEquals, assertTrue, assertFalse, assertThrows (bei Ausnahmen). - Typischer Testverlauf: - Tests sollten deterministisch, schnell und isoliert sein; keine Abhängigkeiten von Systemzeit, Dateisystem oder Netzwerkanbindungen. - Tests nutzen feste, deterministische Eingaben; Ergebnisse sind deterministisch reproduzierbar.

Lösung zu Aufgabe 2: Generics, Speicherverhalten und Unit-Tests in Java

Zur Vertiefung der im Kurs behandelten Themen werden hier konzeptionelle Antworten und Skizzen gegeben. Fertigen Code liefern wir nicht, sondern geben Orientierungen für das Design von Lösungen.

a) Generische Sammlungen

- Nutzung generischer Typen in Java Collections: - Typische Muster: List<String> strings = new ArrayList<>(); Map<String, Integer> map = new HashMap<>(); - Typargumente ermöglichen Typ-Sicherheit zur Compile-Zeit: Nur Objekte vom Typ String dürfen in strings eingefügt werden; Nur Schlüssel vom Typ String und Werte vom Typ Integer dürfen in map eingefügt werden. - Rolle der Type-Erasure zur Laufzeit: Die Typinformation geht zur Laufzeit verloren; der generische Typ wird durch seinen Raw-Typ ersetzt (z. B. List statt List<String>). Die Laufzeit kennt typischerweise nur die erweiterte Basisklasse. - Vorteile von Generics gegenüber rohen Typen: Unmittelbare Compile-Zeit-Checks, Eliminierung von (unsicheren) Casts beim Zugriff auf Elemente, weniger Laufzeitfehler wie ClassCastException, bessere Lesbarkeit des Codes.

- Weitere Punkte: - Durch Type-Erasure bleiben Keine zusätzlichen Typinformationen zur Laufzeit vorhanden; daher existieren in der JVM nur die konkreten Implementierungen (z. B. ArrayList), nicht die spezifischen Typ-Parameter. - Warnings bei nahezu rohen Typen: Bei der Mischung roher Typen mit parameterisierten Typen kann der Compiler Warnungen erzeugen (unchecked operations).

b) Speicherverhalten bei Generics

- Generics selbst verursachen keinen zusätzlichen Speicheraufwand zur Laufzeit: Sie sind eine Sprach- und Compiler-Einfügung, die zur Laufzeit durch Type-Erasure umgesetzt wird. - Einfluss von Autoboxing und Wrapper-Typen: - Wenn Generics z. B. List<Integer> verwenden, speichern die List-Elemente Referenzen zu Integer-Objekten auf dem Heap; primitive Werte in den List-Objekten werden durch Autoboxing zu Integer-Objekten verpackt. - Typische Fallstricke: Häufige Autoboxing-/Unboxing-Operationen erhöhen Garbage-Collection-Arbeit; viele kleine Wrapper-Objekte können Speicherverbrauch und GC-Überhead erhöhen. - Typische Probleme: - Kein Laufzeit-Typ-Check der generischen Typparameter (aufgrund von Type-Erasure); Rohtypen können zu Sicherheitsproblemen führen, wenn sie nicht sorgfältig verwendet werden. - Arrays und Generics: Die direkte Erstellung von Arrays von generischen Typen (z. B. new T[n]) ist nicht erlaubt; stattdessen werden häufig Collections (z. B. List<T>) verwendet. - Null-Toleranz: Generische Typen können T-Objekte enthalten, die null sein können; entsprechende Checks sind sinnvoll. - Praktische Lehre: - Wenn möglich, konservativ auf Primitive im inneren Datenträger setzen oder spezialisierte Sammlungen nutzen (z. B. IntStream/primitive Collections), um Autoboxing zu vermeiden. - Bei Großprojekten bewusstes Ressourcen- und Speicher-Management (Profiling) betreiben.

c) Unit-Tests für generische Datenstrukturen

- Ziel eines Testplans: - Validierung der generischen Struktur (z. B. Box<T> oder Stack<T>) über verschiedene Typen hinweg (Integer, String, benutzerdefinierte Typen). - Prüfung der typischen Operationen (setzen, holen, größenabhängige Eigenschaften), Randfälle (null-Werte), Typ-Sicherheit (Kompilierbarkeit mit unterschiedlichen Typen). - Testziele und -fälle (ohne konkreten Code): - Typ-Sicherheit: Verwendete Typen (z. B. Box<Integer>, Box<String>) kompilieren und funktionieren; Verhindern von unsicheren Typumwandlungen. - Grundoperationen: Set-

zen und Abrufen von Werten, Größenänderung, Leeren/Initialisieren. - Randfälle: null-Werte, Leere Strukturen, maximale Kapazität (falls relevant). - Typwechsel: Gleiche Tests mit unterschiedlichen Typen T (Integer, String, eigener Typ). - Vorgehen: - Verwenden Sie JUnit 5-Parameterisierte Tests, um T mit mehreren Typen abzudecken. - Strukturieren Sie Tests so, dass sie isoliert bleiben (keine Seiteneffekte, deterministische Eingaben). - Prüfen Sie sowohl korrekte Typen als auch potentielle Fehlerfälle, die durch falsche Nutzung (z. B. unsichere Raw-Types) entstehen könnten. - Hinweise: - Falls die Struktur über eine Instanzmethode verfügt, testen Sie sowohl zustandsabhängige als auch zustandsunabhängige Operationen. - Nutzen Sie Assertions, um Randfälle und Invarianzen zu validieren (Größe, Inhalt, Konsistenz).

d) Ausblick: Test-Strategien

- Leitlinien zu Unit-Tests in Java (JUnit 5) im Kontext von Speicherkonsistenz, Generics und Multithreading: - Memory- bzw. Sichtbarkeits-Tests: Verwenden Sie Synchronisation (volatile Felder, Atomic-Klassen) oder Concurrency-Utilities (CountDownLatch, CyclicBarrier), um sichtbares Verhalten über Threads sicher zu testen. - Generics: Nutzen Sie Parameterized Tests, um verschiedene Typen systematisch abzudecken; vermeiden Sie Raw-Typen in Tests, um die Compile-Zeit-Sicherheit zu nutzen. - Multithreading: Schreiben Sie Tests, die Race-Conditions reproduzierbar machen (z. B. durch gezieltes Scheduling, Latches). Verfolgen Sie deterministische Resultate trotz Parallelität. - Neben Tests: Property-Based-Tests (z. B. jqwik) zur Erzeugung von Zufallsdaten mit invarianten Eigenschaften; Integrationstests, um Verhalten zwischen Modulen/Subsystemen zu prüfen; System- bzw. End-to-End-Tests für echte Nutzungsszenarien. - Test-Strategie: Beginnen Sie mit Unit-Tests, erweitern Sie schrittweise zu Integrations- und Systemtests; verwenden Sie CI, um Regressionen frühzeitig zu erkennen; kombinieren Sie manuelle Tests mit automatisierten, wo sinnvoll. - Qualitätsaspekte: Stabilität der Tests, Schnelligkeit, Wartbarkeit, klare Fehlermeldungen; nutzen Sie Mocks/Stubs dort sinnvoll, um externe Abhängigkeiten zu isolieren.