MC Aufgabenpool

Datenabstraktion

Welche der folgenden Aussagen gelten in Java für die unterschiedlichen Arten von Variablen und Parametern?

Richtig	Frage	Begründung
	Formale Parameter und lokale Variablen können gleich heißen.	Formale Parameter sind Parameter von Methoden und lokale Variablen würden in der Methode dann die Parameter überschreiben
	Lokale Referenzvariablen werden automatisch vorinitialisiert.	Variablen innerhalb eines bestimmten Blocks (bspw Methode), werden nicht automatisch vorinitialisiert.
	Lokale Variablen werden mit private deklariert.	Nein die sind automatisch nur innerhalb des Blocks zugreifbar
	Klassenvariablen werden bei der Objekterzeugung angelegt.	Nein die existieren nur ein mal pro Klasse unabhängig vom Objekt
x	Klassenvariablen werden mit static deklariert.	
x	Objektvariablen und lokale Variablen können gleich heißen.	
x	Objektvariablen werden automatisch vorinitialisiert.	
x	Objektvariablen werden bei der Objekterzeugung angelegt.	

Unterscheidung zwischen Klassenvariablen und Objektvariablen:

Klassenvariablen:

- Werden mit dem Schlüsselwort `static` in der Klassendefinition deklariert.
- Gehören der Klasse als Ganzes und nicht einzelnen Objekten.
- Werden einmalig initialisiert, wenn die Klasse geladen wird.
- Zugriff erfolgt über den Klassennamen oder über ein Objekt (wenn `public`).

Objektvariablen (Instanzvariablen):

- Werden ohne das Schlüsselwort `static` deklariert.
- Gehören zu einzelnen Objekten (Instanzen) der Klasse.
- Werden bei der Objekterzeugung (mit `new`) angelegt und beim Löschen des Objekts wieder freigegeben.
- Zugriff erfolgt über ein Objekt.

Beispiel (Java):

```java
class MyClass {
    static int classVariable = 10; // Klassenvariable
    int instanceVariable; // Objektvariable

    public MyClass(int value) {
        this.instanceVariable = value;
    }
}

public class Main {
    public static void main(String[] args) {
        MyClass obj1 = new MyClass(20);
        MyClass obj2 = new MyClass(30);

        System.out.println(MyClass.classVariable); // Zugriff auf Klassenvariable über Klassennamen
        System.out.println(obj1.classVariable);    // Zugriff auch über Objekt (wenn public)
        System.out.println(obj1.instanceVariable);
        System.out.println(obj2.instanceVariable);

        MyClass.classVariable = 15; // Änderung der Klassenvariable
        System.out.println(obj2.classVariable);    // Änderung ist für alle Objekte sichtbar
    }
}
```

Welche der folgenden Aussagen treffen auf Objektmethoden bzw. Klassenmethoden zu?

Richtig	Frage	Begründung
	Aufrufe von this(...) sind nur in Objektmethoden erlaubt.	Nein damit werden auch neue Objekte erstellt. Mit `this(...)` ist in dem Fall der Konstruktor gemeint.
	In Klassenmethoden bezeichnet this die aktuelle Klasse.	Nein das bezieht sich auf das Objekt, nicht auf die Klasse
x	In Klassenmethoden darf this nicht vorkommen.	(Klassenmethode ist static Methode)
	Jede Methode f() aus einer Klasse C ist durch C.f() aufrufbar.	Nein nur die, die `public` sind.
x	Jede nicht als static deklarierte Methode ist eine Objektmethode.	
x	Klassenmethoden haben keinen Zugriff auf Objektvariablen.	
	Objektmethoden haben keinen Zugriff auf Klassenvariablen.	Doch haben sie
x	Objektmethoden haben Zugriff auf Objekt- und Klassenvariablen.	

Welche der folgenden Aussagen stimmen in Bezug auf Datenabstraktion?

Richtig	Frage	Begründung
x	Das Resultat ist ein abstrakter Datentyp.	
	Data-Hiding behindert die Datenabstraktion.	Nein man braucht Data-Hiding für die Datenabstraktion
	Datenabstraktion verhindert Änderungen von Objektzuständen.	Nein, mit `getter` und `setter` Methoden noch möglich.
x	Datenkapselung fügt Variablen und Methoden zu einer Einheit zusammen.	
	Datenkapselung ist ein anderer Begriff für Data-Hiding.	Nein ist was anderes
x	Datenkapselung und Data-Hiding sind für Datenabstraktion nötig.	
x	Klassen implementieren abstrakte Datentypen.	
	Kommentare sind in abstrakten Datentypen bedeutungslos.	Nein sie helfen der Orientierung im Programmcode

Welche der folgenden Aussagen müssen für jede Verwendung von this(...) bzw. this in einem Konstruktor zutreffen?

Richtig	Frage	Begründung
x	this(...); darf nur als erste Anweisung vorkommen.	Ja, das ist der Konstruktor
	this(...); darf nur als letzte Anweisung vorkommen.	
	this darf in Konstruktoren nicht verwendet werden.	Doch für Variablenzuweisungen
	this ist nur in static Konstruktoren verwendbar.	
	this = null; darf nur als erste Anweisung vorkommen.	
x	this referenziert das Objekt, das gerade initialisiert wird.	
x	Wird this(...); aufgerufen, gibt es keinen Default-Konstruktor.	
	Zu Beginn gilt: this == null.	Nein, sobald this existiert ist es $\neq$ `null`
	'this == null' kann 'true' zurückgeben.	Kommt nie vor, dann hätte man auch keine Referenz mehr auf das Objekt und könnte diese Abfrage nicht durchführen
	'this' referenziert den Untertyp der aktuellen Klasse.	Nein die Objektinstanz
	Der Wert von 'this' ist nur schreibbar, nicht lesbar.	Andersherum

Welche der folgenden Aussagen stimmen in Bezug auf die Innen- und Außensicht eines abstrakten Datentyps?

Richtig	Frage	Begründung
x	Außen- und Innensicht betreffen einzelne Objekte, nicht ganze Klassen.	
x	Public Methoden betreffen die Außen- und Innensicht.	
	Zur besseren Wartbarkeit sollen Methoden public sein.	Methoden sollen nur dann public sein, wenn man von außen drauf zugreifen können muss, für bessere Wartbarkeit gibt es keinen Grund die Methoden public zu machen.
	Änderungen der Innensicht wirken sich stets auf die Außensicht aus.	Nein muss nicht sein
	Änderungen der Außensicht lassen die Innensicht unberührt.	Nein muss nicht sein

Datenstrukturen

Welche der folgenden Aussagen stimmen in Bezug auf die unterschiedlichen Arten linearer und assoziativer Datenstrukturen?

Richtig	Frage	Begründung
x	Assoziative Datenstrukturen erlauben wahlfreie Zugriffe.	Wahlfreie Zugriffe, man kann überall auf jeden möglichen Index zugreifen
	Assoziative Datenstrukturen haben wie Arrays eine fixe Größe.	Gegenbeispiel: Hashmaps, Treemaps
x	Assoziative Datenstrukturen verwenden Schlüssel zur Adressierung.	
	Einträge in Queues sind nach Schlüsseln sortiert.	Sind in der Reihenfolge wie man sie hinzugefügt hat
	put(k,v) gibt null zurück wenn der Schlüssel k schon existiert.	Gibt `false` zurück
x	Die Methodennamen push und pop weisen auf LIFO-Verhalten hin.	Last in first out (Beispiel Stack)
x	Double-Ended-Queues können auch wie Stacks verwendet werden.	Indem man immer auf einer Seite hinzufügt und von der Seite auch wieder raus-`pop`t
	Assoziative Datenstrukturen haben LIFO- oder FIFO-Verhalten.	Nein bei denen wird mit Schlüsseln gearbeitet.

Welche der folgenden Aussagen stimmen in Bezug auf rekursive Datenstrukturen?

Richtig	Frage	Begründung
x	Doppelt verkettete Listen sind in beide Richtungen traversierbar.	
x	Fortschritt erfolgt durch induktiven Aufbau und Dereferenzierung.	
x	Auch zyklische Datenstrukturen müssen fundiert sein.	Fundiert = Endpunkt haben
x	Zur Fundierung können spezielle Knoten verwendet werden.	Beispielsweise `NIL`
x	Zur Fundierung wird meist null verwendet.	
	Jeder Knoten, der mehrere Knoten referenziert, ist Teil eines Baums.	Muss kein Baum sein
	Schleifen erlauben kein vollständiges Traversieren.	Doch wenn man als `while` Abbruchbedingung den Fundierten Knoten meidet.

Welche der folgenden Aussagen stimmen in Bezug auf die Unterscheidung zwischen Datenstrukturen und abstrakten Datentypen?

Richtig	Frage	Begründung
x	Abstrakte Datentypen beschreiben vorwiegend Schnittstellen.	definiert das *Was* (die Funktionalität und das Verhalten) einer Datenstruktur, aber nicht das *Wie* (die Implementierung)
	Abstrakte Datentypen implementieren Algorithmen.	ADTs beschreiben nur die Schnittstelle
x	Abstrakte Datentypen lassen verwendete Algorithmen meist offen.	
	Abstrakte Datentypen müssen bestimmte Datenstrukturen festlegen.	Im Gegenteil, ADTs sind von der zugrundeliegenden Datenstruktur **unabhängig**. Ein ADT kann durch verschiedene Datenstrukturen implementiert werden (z.B. eine Liste als Array oder verkettete Liste).
x	Abstrakte Datentypen spezifizieren Typen von Methoden-Parametern.	Teil der Schnittstellenbeschreibung eines ADT ist es, die Datentypen der Ein- und Ausgabeparameter (Signaturen) der Methoden festzulegen, damit Benutzer wissen, wie sie mit dem ADT interagieren können
x	Datenstrukturen beschreiben, wie Operationen auf Daten zugreifen.	
	Datenstrukturen lassen offen, wie Daten zusammenhängen.	Nein das wird ganz genau geregelt
	Datenstrukturen legen die Typen ihrer Einträge fest.	Nein in eine Liste kann man ja auch sich als Nutzer aussuchen ob man `Strings` oder `Integer` Werte speichern will.
x	Datenstrukturen sind unabhängig von bestimmten Programmiersprachen.	
	Jede Datenstruktur hat eine festgelegte Maximalgröße.	Gegenbeispiel `Queue`
x	Abstrakte Datentypen müssen keine Datenstrukturen beschreiben.	
x	Datenstrukturen stehen in engem Zusammenhang mit Algorithmen.	
	Datenstrukturen implementieren abstrakte Datentypen.	

a sei eine Variable mit einer leeren assoziativen Datenstruktur, wobei Schlüssel und Werte vom Typ String sind (und null sein können). X und Y seien zwei voneinander verschiedene String-Konstanten (static final). Nach welchen der folgenden Aufruf-Sequenzen liefert a.get(X) den String in Y als Ergebnis?

Richtig	Frage

	a.put(a.get(X), a.get(Y));
	a.put(X, X);
	a.put(Y, Y);

	a.put(X, X);
	a.put(a.get(X), a.get(Y));
	a.put(Y, Y);

x	a.put(X, X);
	a.put(Y, Y);
	a.put(a.get(X), a.get(Y));

	a.put(X, X);
	a.put(Y, Y);
	a.put(a.get(Y), a.get(X));

x	a.put(X, Y);
	a.put(a.get(Y), a.get(X));
	a.put(Y, X);

	a.put(X, Y);
	a.put(X, X);
	a.put(a.get(X), a.get(Y));

x	a.put(X, Y);
	a.put(Y, X);
	a.put(a.get(X), a.get(Y));

x	a.put(Y, X);
	a.put(a.get(Y), a.get(X));
	a.put(X, Y);

x	a.put(Y, X);
	a.put(X, Y);
	a.put(a.get(Y), a.get(X));

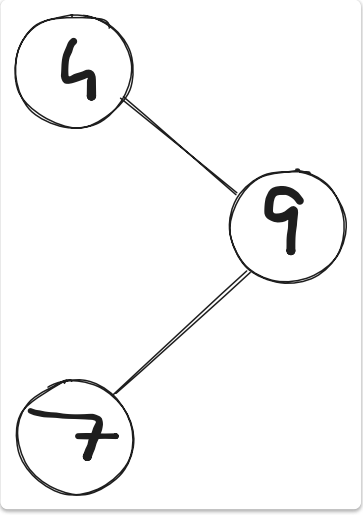
x, y und z seien Objektreferenzen ungleich null. Welche der folgenden Bedingungen müssen für jede Implementierung der Methoden boolean equals(Object obj) und int hashCode() in Java gelten?

Richtig	Frage	Begründung
x	Aus x.equals(y) folgt x.hashCode() == y.hashCode().	
	Aus !x.equals(y) folgt x.hashCode() != y.hashCode().	Hashcode kann trz gleich sein
x	Aus x.equals(y) folgt y.equals(x).	
x	Aus !x.equals(y) folgt !y.equals(x).	
	Aus x.equals(y) folgt !y.equals(x).	Wiederspruch
	null.equals(null) gibt true zurück.	Nein gibt eine `NullPointerException`
x	x.equals(null) gibt false zurück.	
	x.hashCode() >= 0 gibt true zurück.	
	Aus x.hashCode() == y.hashCode() folgt x.equals(y).	Muss nicht sein
	null.equals(x) gibt false zurück.	Nein gibt eine `NullPointerException`

Richtig	Frage	Begründung
	x.hashCode() >= 0 gibt true zurück.	
x	Aus x.equals(y) und y.equals(z) folgt x.equals(z).	Transitivität
x	Aus x.hashCode() != y.hashCOde() folgt !x.equals(y).	

t sei eine Variable mit einem einfachen (unbalancierten) binären Suchbaum ganzer Zahlen, der durch diese Anweisungen aufgebaut wurde:

```
STree t = new STree();
t.add(4);
t.add(9);
t.add(7);
```



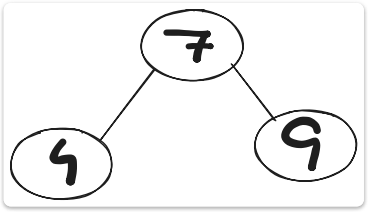
Welche der folgenden Aussagen treffen auf t zu?

Richtig	Frage
	Der Baum hat eine Tiefe von 2.
x	Der Baum hat eine Tiefe von 3.
	Der Knoten mit Wert 7 hat zumindest ein Kind.
	Der Knoten mit Wert 7 ist die Wurzel.
x	Der Knoten mit Wert 7 ist ein Blattknoten.
x	Der Knoten mit Wert 9 hat zumindest ein Kind.
	Der Knoten mit Wert 9 ist ein Blattknoten.
x	Der Knoten mit Wert 4 ist die Wurzel.

t sei eine Variable mit einem einfachen (unbalancierten) binären Suchbaum ganzer Zahlen, der durch diese Anweisungen aufgebaut wurde:

```
STree t = new STree();
t.add(7);
```

```
t.add(9);
t.add(4);
```



Welche der folgenden Aussagen treffen auf t zu?

Richtig	Frage
x	Der Baum hat eine Tiefe von 2.
	Der Baum hat eine Tiefe von 3.
x	Der Knoten mit Wert 7 ist die Wurzel.
	Der Knoten mit Wert 7 ist ein Blattknoten.
x	Der Knoten mit Wert 9 ist ein Blattknoten.

x sei eine Referenz auf einen Knoten (Typ Node) in einer einfach verketteten Liste mit mindestens einem existierenden Nachfolger (in der Objektvariablen next). Welche der folgenden Anweisungs-Sequenzen entfernen den direkten Nachfolger von x aus einer Liste, ändern sonst aber nichts?

Richtig	Frage
	x.next.next.next = x.next.next;
	Node n = x.next.next; n.next = x;
	Node d = x.next; d = d.next;
x	Node d = x.next; x.next = d.next;
x	Node n = x.next.next; x.next = n;
x	Node n = c; n.next = c.next.next;
x	x.next = x.next.next;

s sei eine Variable mit einem leeren Stack ganzer Zahlen. Nach welchen der folgenden Aufruf-Sequenzen liefert s.peek() die Zahl 1 als Ergebnis?

Richtig	Frage	Begründung
x	s.push(3); s.push(2); s.push(1);	
x	s.push(2); s.push(1); s.push(s.peek());	das was drunter liegt nochmal drauf
	s.push(1); s.push(2); s.push(s.pop());	
	s.push(3); s.push(1); s.push(2);	
x	s.push(2); s.push(1); s.push(s.pop());	

Richtig	Frage	Begründung
	s.push(1); s.push(s.peek()); s.push(2);	
x	s.push(2); s.push(s.peek()); s.push(1);	

q sei eine Variable, die eine leere Double-Ended-Queue ganzer Zahlen enthält. Nach welchen der folgenden Aufruf-Sequenzen liefert q.peekFirst() die Zahl 1 als Ergebnis?

Richtig	Frage	Begründung
	q.addLast(1); q.addLast(2); q.pollFirst();	
x	q.addFirst(1); q.addFirst(2); q.addFirst(q.peekLast());	wir geben das letzte an den Anfang
	q.addFirst(1); q.addFirst(2); q.peekFirst();	Peek entfernt nichts
x	q.addLast(1); q.addLast(2); q.pollLast();	Nur noch 1 in Queue
	q.addFirst(1); q.addFirst(2);	

Dynamische und statische Bindung

Welche der folgenden Aussagen stimmen in Bezug auf dynamisches und statisches Binden?

Richtig	Frage	Begründung
x	Bei statischem Binden kennt der Compiler die auszuführende Methode.	Der Compiler entscheidet zur Kompilierzeit, welche Methode aufgerufen wird, basierend auf dem statischen Typ der Variablen
x	Dynamisches Binden ist zusammen mit Untertypbeziehungen nötig.	Dynamisches Binden ist essenziell für Polymorphismus bei Untertypbeziehungen, da die Methode erst zur Laufzeit basierend auf dem tatsächlichen Objekttyp ausgewählt wird
x	Ein dynamischer Typ ist stets eine Klasse, kein Interface.	Ein Objekt wird immer von einer konkreten Klasse instanziiert; Interfaces können nicht instanziiert werden
	Ein statischer Typ ist stets ein Interface, keine Klasse.	
	Klassenmethoden werden immer dynamisch gebunden.	
	Objektmethoden werden immer dynamisch gebunden.	
x	Klassenmethoden werden immer statisch gebunden.	gehören zur Klasse, nicht zur Instanz, und können nicht überschrieben werden
	Objektmethoden werden in deklarierten Typen von Objekten ausgeführt.	
x	Private Methoden werden immer statisch gebunden.	Private Methoden sind nicht von außen zugänglich und können nicht überschrieben werden. Ihre Zuordnung ist daher zur Kompilierzeit eindeutig.
x	Objektmethoden werden in dynamischen Typen von Objekten ausgeführt.	Bei Instanzmethoden entscheidet die JVM zur Laufzeit, welche Methode aufgerufen wird, basierend auf dem tatsächlichen Objekttyp (Polymorphismus).
	Ein statischer Typ ist stets ein Interface, keine Klasse.	

R, S und T seien Referenztypen. Welche der folgenden Aussagen treffen zu?

Richtig	Frage	Begründung
x	Aus R Untertyp von S und S Untertyp von T folgt: R Untertyp von T.	Transitivität
	Aus 'R Untertyp von S' und 'S Untertyp von T' folgt: 'R.class==T.class'.	
	Aus S ist Klasse und T ist Interface folgt: S ist Untertyp von T.	
	Aus S Untertyp von T folgt: Kommentare in S und T sind gleich.	
x	Aus S Untertyp von T und T Untertyp von S folgt: S.class==T.class.	
x	T ist Untertyp von T.	

Richtig	Frage	Begründung
	Aus S untertyp von T folgt: jede Methode von S ist Methode von T.	Nein andersherum
	'null' ist ein Objekt von jedem Referenztyp T.	
x	Ist R kein Untertyp von java.lang.Object, dann ist R ein Interface.	

S und T seien Referenztypen, sodass der Compiler folgenden Programmtext fehlerfrei compiliert: T x = new S(); x.foo(); Welche der folgenden Aussagen treffen für alle passenden S, T, x und foo() zu?

Richtig	Frage	Begründung
	Die Methode foo() muss in S vorkommen, in T aber nicht.	Muss in beiden Vorkommen
x	Durch x.foo() wird die Methode in S ausgeführt.	
	Es gilt: x.getClass() == T.class	Diese Aussage ist falsch, weil `x.getClass()` den dynamischen Typ des Objekts zurückgibt, auf das `x` verweist. In der Zeile `T x = new S();` wird ein Objekt vom Typ S instanziiert. Daher wird `x.getClass()` zur Laufzeit `S.class` zurückgeben, nicht `T.class`. Die Bedingung `x.getClass() == T.class` wäre nur wahr, wenn `S` und `T` identisch wären (also `S` selbst `T` ist) oder wenn `S` der anonyme dynamische Typ wäre, aber nicht, wenn `S` ein echter Untertyp von `T` ist.
x	Kommentare zu foo() in T müssen auch auf foo() in S zutreffen.	
x	S ist Untertyp von T.	
x	Die Methode foo() muss in S und T vorkommen.	
x	S muss eine Klasse sein.	
	x kann verwendet werden, wo ein Objekt von S erwartet wird.	Variable `x` ist vom statischen Typ T. Das bedeutet, `x` kann überall dort verwendet werden, wo ein Objekt vom Typ `T` oder einem seiner Obertypen erwartet wird

T sei ein Referenztyp (Klasse oder Interface), und x sei eine durch R x = new S(); deklarierte Variable, wobei der Compiler keinen Fehler meldet. Welche der folgenden Aussagen treffen für alle passenden R, S, T und x zu?

Richtig	Frage	Begründung
	Mit S ist Untertyp von T gilt: ((T)x).getClass() == R.class	
x	Mit S ist Untertyp von T gilt: ((T)x).getClass() == S.class	
	Mit S ist Untertyp von T gilt: ((T)x).getClass() == T.class	
x	(T)null liefert zur Laufzeit keinen Fehler.	
x	(T)x ändert den deklarierten Typ von x auf T.	
x	(T)x liefert keinen Laufzeitfehler wenn R Untertyp von T ist.	
	(T)x liefert Laufzeitfehler wenn T nicht Untertyp von R ist.	
x	(T)x liefert Laufzeitfehler wenn S nicht Untertyp von T ist.	
	(T)x ändert den dynamischen Typ von x auf T.	nein nur den deklarierten

T sei ein Referenztyp (Klasse oder Interface), und x sei eine Variable eines Referenztyps mit x != null. Welche der folgenden Aussagen treffen für alle T und x zu?

Richtig	Frage	Begründung
	Aus x instanceof T folgt x.getClass() == T.class.	Nein weil `.getClass()` liefert den exakten dynamischen Typ und `instanceOf` prüft nur ob das Objekt auf das `x` verweist vom Typ `T` ist
x	Aus x.getClass() == T.class folgt x instanceof T.	
x	Gilt x.getClass() == T.class, dann ist T eine Klasse.	
	Gilt x instanceof T, dann ist T der deklarierte Typ von x.	`instanceof` prüft die Typ-Kompatibilität zur Laufzeit, während der deklarierte Typ von `x` der Typ ist, der bei der Deklaration der Variablen `x` angegeben wurde. Diese beiden müssen nicht gleich sein.
	Gilt x instanceof T, dann ist T der dynamische Typ von x.	`instanceof` prüft, ob das Objekt, auf das `x` verweist, vom Typ `T` ist oder von einem Untertyp von T. Der dynamische Typ von `x` ist jedoch der exakte, tatsächliche Typ des Objekts zur Laufzeit.
	Gilt x instanceof T, dann ist T eine Klasse.	Kann auch ein Interface sein
	x.getClass() liefert (interne Repr. vom) deklarierten Typ von x.	liefert das `Class`-Objekt, das den tatsächlichen, dynamischen (Laufzeit-)Typ des Objekts repräsentiert, auf das `x` verweist
x	x.getClass() liefert (interne Repr. vom) dynamischen Typ von x.	