MC1 Zusammenfassung

Stoff der im MC gebraucht wird:

1. Verwendung von 'this' in Konstruktoren

- **Was ist 'this'?**
  - In Java ist `this` ein Schlüsselwort, das auf das aktuelle Objekt verweist, das in einer Methode oder einem Konstruktor verwendet wird. Es wird verwendet, um auf Instanzvariablen oder -methoden des Objekts zuzugreifen.
- **Verwendung von 'this' in Konstruktoren:**
  - 'this' kann in einem Konstruktor verwendet werden, um das Objekt zu referenzieren, das gerade initialisiert wird. Dadurch ist es nützlich, um zwischen den Parametern eines Konstruktors und den Instanzvariablen zu unterscheiden, falls diese den gleichen Namen haben.
- **Wichtige Punkte:**
  - **'this' referenziert das Objekt, das gerade initialisiert wird.**
  - Der Wert von `this` kann nicht `null` sein, weil es immer auf das aktuelle Objekt zeigt.

---

2. Binärer Suchbaum (BST)

- **Was ist ein binärer Suchbaum?**
  - Ein binärer Suchbaum ist eine Datenstruktur, bei der jeder Knoten maximal zwei Kinder hat. Bei einem BST sind alle linken Kinder kleiner als der Elternknoten, und alle rechten Kinder sind größer als der Elternknoten.
- **Eigenschaften:**
  - Der Knoten mit dem kleinsten Wert befindet sich am linken Ende, und der Knoten mit dem größten Wert am rechten Ende.
  - Der Baum ist **nicht ausbalanciert**, wenn die Knoten in einer nicht-optimalen Reihenfolge eingefügt werden.
- **Fehleranalyse im Baumaufbau:**
  - Ein unbalancierter Baum entsteht oft durch unsystematisches Hinzufügen von Werten. Hier wurde ein Knoten mit Wert 6 als Elternteil von 5 hinzugefügt, was auf eine unbalancierte Struktur hinweist.

---

3. Referenztypen und Vererbung

- **Vererbung in Java:**
  - Java unterstützt die objektorientierte Programmierung und nutzt Vererbung, um Hierarchien von Klassen und Interfaces zu erstellen.
  - Jede Klasse in Java ist ein Untertyp von `java.lang.Object`. Das bedeutet, jede Instanz einer Klasse erbt Methoden wie `toString()`, `equals()` und `hashCode()` von `Object`.
- **Interface und Klasse:**
  - Ein Interface ist **kein Untertyp einer Klasse**, aber es kann von Klassen **implementiert** werden.
  - Eine Klasse kann **nicht** von einem anderen Interface „erben“, sie kann jedoch ein Interface **implementieren**.

---

4. Stack-Datenstruktur

- **Was ist ein Stack?**
  - Ein Stack ist eine lineare Datenstruktur, die das **LIFO**-Prinzip (Last In, First Out) befolgt. Das bedeutet, dass das zuletzt hinzugefügte Element als erstes wieder entfernt wird.
- **Wichtige Operationen:**
  - `push(x)` fügt das Element `x` zum Stack hinzu.
  - `pop()` entfernt das oberste Element vom Stack.
  - `peek()` gibt das oberste Element zurück, ohne es zu entfernen.
- **Beispiel:**
  - Wenn du zuerst `1` und dann `8` auf den Stack legst und dann `pop()` aufrufst, wird `8` entfernt, aber `1` bleibt auf dem Stack.

---

5. Datenabstraktion und Kapselung

- **Datenabstraktion:**
  - Datenabstraktion ist der Prozess, bei dem die Implementierungsdetails von einem Benutzer verborgen werden, um eine vereinfachte Schnittstelle zu bieten.
  - Abstrakte Datentypen (ADT) beschreiben nur die notwendigen Operationen, ohne Details zu erklären, wie diese intern implementiert werden.
- **Datenkapselung:**
  - Datenkapselung bezieht sich auf das Verbergen der internen Zustände eines Objekts und den Zugriff nur über öffentlich zugängliche Methoden.

- Dies schützt die Integrität der Daten und stellt sicher, dass nur sichere und kontrollierte Änderungen an den Objekten vorgenommen werden.

- **Data-Hiding:**
  - Data-Hiding bedeutet, dass die Daten eines Objekts privat gehalten werden, sodass sie nicht direkt von außen verändert werden können.

---

6. Lineare und assoziative Datenstrukturen

- **Lineare Datenstrukturen:**
  - In linearen Datenstrukturen (wie Arrays, Listen, Stacks, Warteschlangen) erfolgt der Zugriff auf die Elemente in einer bestimmten Reihenfolge.
  - Sie sind oft **fest** in der Größe und ermöglichen sequentiellen Zugriff.
- **Assoziative Datenstrukturen (z.B. HashMaps):**
  - Diese Datenstrukturen ermöglichen den **wahlfreien Zugriff** auf Elemente anhand von Schlüsseln. Sie sind besonders nützlich, wenn eine schnelle Suche oder Zuordnung benötigt wird.
- **Wichtige Unterscheidung:**
  - **Assoziative Datenstrukturen** sind effizient in der Verwaltung von Schlüsseln und Werten, wohingegen **lineare Datenstrukturen** in der Regel nur in einer festen Reihenfolge zugänglich sind.

---

7. Generics und Typen in Java

- **Generische Typen:**
  - In Java können Generics verwendet werden, um die Typen von Objekten zur Compile-Zeit festzulegen. Dies bietet Sicherheit und Flexibilität, da der Compiler sicherstellen kann, dass keine falschen Typen verwendet werden.
- **Beispiel:**
  - Die Deklaration `T m = new W();` stellt sicher, dass die Typen von `T` und `W` kompatibel sind, bevor das Objekt erstellt wird.

---

8. Double-Ended-Queue (Deque)

- **Was ist eine Deque?**
  - Eine Double-Ended-Queue (Deque) ist eine Datenstruktur, die sowohl am Anfang als auch am Ende Elemente hinzufügen und entfernen kann.

- **Wichtige Operationen:**
  - `addFirst(x)` fügt ein Element am Anfang hinzu.
  - `addLast(x)` fügt ein Element am Ende hinzu.
  - `peekFirst()` gibt das erste Element zurück, ohne es zu entfernen.
  - `peekLast()` gibt das letzte Element zurück, ohne es zu entfernen.
  - `pollFirst()` entfernt und gibt das erste Element zurück.
- **Anwendungsbeispiel:**
  - Wenn du `9` und `6` zu einer Deque hinzufügst, dann zuerst `9`, kannst du `pollFirst()` verwenden, um `9` zu entfernen, während `6` am Anfang bleibt.

Antworten auf das Multiple Choice mit Begründung:

Frage 1: Welche der folgenden Aussagen müssen für jede Verwendung von 'this(...)' bzw. 'this' in einem Konstruktor zutreffen?

- **'this' ist nur in 'final' Konstruktoren verwendbar. (1A)** ❌
  Warum falsch?: 'this' kann in jedem Konstruktor verwendet werden, nicht nur in 'final' Konstruktoren. Es gibt keine Einschränkung für die Verwendung von 'this'.
- **'this' darf in Konstruktoren nicht verwendet werden. (1B)** ❌
  Warum falsch?: 'this' wird in Konstruktoren häufig verwendet, um das aktuelle Objekt zu referenzieren, insbesondere bei Konstruktorverkettung.
- **'this = null;' darf nur als erste Anweisung vorkommen. (1C)** ❌
  Warum falsch?: Der Ausdruck 'this = null;' ist ungültig in Java, da 'this' immer das aktuelle Objekt referenziert und nicht neu zugewiesen werden kann.
- **'this' referenziert das Objekt, das gerade initialisiert wird. (1D)** ✅
  Warum richtig?: 'this' bezieht sich immer auf das aktuelle Objekt, das durch den Konstruktor initialisiert wird.
- **Der Wert von 'this' kann nicht 'null' sein. (1E)** ✅
  Warum richtig?: 'this' kann niemals 'null' sein, da es immer das aktuelle Objekt referenziert.

Frage 2: e sei eine Variable mit einem einfachen binären Suchbaum. Welche der folgenden Aussagen treffen auf e zu?

Baum wurde so aufgebaut:

```
STree e = new STree();
e.add(0);
e.add(6);
e.add(5);
```

- **Der Knoten mit Wert 6 ist die Wurzel. (2A)** ❌
  Warum falsch?: Der Knoten mit Wert 0 ist die Wurzel, da er zuerst hinzugefügt wird. 6 ist ein Kindknoten.
- **Der Knoten mit Wert 6 ist Elter von dem mit Wert 5. (2B)** ✅
  Warum richtig?: 6 wird nach 5 eingefügt, daher ist 6 der Elternknoten von 5.
- **Der Baum ist nicht ausbalanciert. (2C)** ✅
  Warum richtig?: Der Baum ist nicht ausbalanciert, da er nur 3 Knoten hat und auf einer Seite einen tieferen Ast als auf der anderen.
- **Der Knoten mit Wert 6 hat zumindest ein Kind. (2D)** ✅
  Warum richtig?: Der Knoten mit Wert 6 hat als Kind den Knoten mit Wert 5.
- **Der Knoten mit Wert 0 ist ein Blattknoten. (2E)** ❌
  Warum falsch?: Der Knoten mit Wert 0 ist die Wurzel und hat ein Kind (den Knoten mit Wert 6), daher ist er kein Blattknoten.

Frage 3: X, U und C seien beliebige Referenztypen. Welche der folgenden Aussagen treffen zu?

- **Ist U eine Klasse, dann ist U Untertyp von java.lang.Object. (3A)** ✅
  Warum richtig?: Alle Klassen in Java sind direkte oder indirekte Untertypen von java.lang.Object.
- **Aus 'C ist Klasse' und 'U ist Interface' folgt: 'U ist kein Untertyp von C'. (3B)** ✅
  Warum richtig?: Ein Interface ist kein Untertyp einer Klasse, selbst wenn beide zur gleichen Vererbungshierarchie gehören.
- **Aus 'U Untertyp von C' folgt: 'Kommentare in U und C passen zusammen'. (3C)** ✅
  Warum richtig?: Die Vererbung zwischen U und C bedeutet, dass U die Methoden und Kommentare von C erben kann, und daher passen die Kommentare zusammen.
- **Ist X kein Untertyp von java.lang.Object, dann ist X ein Interface. (3D)** ✅
  Warum richtig?: In Java ist alles (außer 'null') ein Untertyp von java.lang.Object. Wenn X kein Untertyp ist, muss es ein Interface sein.
- **Aus 'U Untertyp von C' und 'C Untertyp von X' folgt: 'U.class == X.class'. (3E)** ❌
  Warum falsch?: Wenn U ein Untertyp von C und C ein Untertyp von X ist, bedeutet das nicht, dass die Klassen von U und X gleich sind. Sie können unterschiedliche Typen mit unterschiedlichen Laufzeitrepräsentationen haben.

Frage 4: o sei eine Variable mit einem leeren Stack ganzer Zahlen. Nach welchen der folgenden Aufruf-Sequenzen liefert 'o.peek()' die Zahl 1 als Ergebnis?

- **o.push(1); o.push(8); o.push(o.peek()); (4A)** ❌
  Warum falsch?: 'o.peek()' gibt den Wert 8 zurück, bevor 'o.push(o.peek())' ausgeführt wird, daher wird 8 oben auf dem Stack liegen.
- **o.push(6); o.push(1); o.push(8); (4B)** ❌
  Warum falsch?: Das Ergebnis von 'o.peek()' wäre 8, nicht 1, da 8 zuletzt auf den Stack gelegt wird.
- **o.push(8); o.push(o.pop()); o.push(1); (4C)** ✅
  Warum richtig?: Der Wert 8 wird zuerst gepusht, dann wird er durch 'o.pop()' entfernt, und 1 wird danach gepusht, was bedeutet, dass 1 oben auf dem Stack liegt.
- **o.push(6); o.push(8); o.push(1); (4D)** ✅
  Warum richtig?: Der Wert 1 wird zuletzt auf den Stack gelegt, sodass 'o.peek()' den Wert 1 zurückgibt.
- **o.push(8); o.push(1); o.push(o.peek()); (4E)** ✅
  Warum richtig?: Der Wert 1 wird zuletzt auf den Stack gelegt und bleibt oben, nachdem 'o.peek()' den Wert 1 zurückgibt.

Frage 5: Welche der folgenden Aussagen stimmen in Bezug auf Datenabstraktion?

- **Kommentare sind in abstrakten Datentypen bedeutungslos. (5A)** ❌
  Warum falsch?: Kommentare sind immer wichtig für das Verständnis und die Lesbarkeit von Code, auch in abstrakten Datentypen.
- **Datenkapselung ist ein anderer Begriff für Data-Hiding. (5B)** ❌
  Warum falsch?: Datenkapselung bezieht sich auf das Kombinieren von Daten und Methoden, während Data-Hiding darauf abzielt, die interne Struktur vor der Außenwelt zu verbergen.
- **Datenkapselung fügt Variablen und Methoden zu einer Einheit zusammen. (5C)** ✅
  Warum richtig?: Datenkapselung bedeutet, dass Daten und Methoden innerhalb einer Klasse zusammengefasst werden, um das Verhalten des Objekts zu definieren.
- **Data-Hiding behindert die Datenabstraktion. (5D)** ❌
  Warum falsch?: Data-Hiding unterstützt die Datenabstraktion, da es die interne Implementierung versteckt und nur die wichtigen Teile des Objekts zugänglich macht.
- **Klassen implementieren abstrakte Datentypen. (5E)** ✅
  Warum richtig?: Abstrakte Datentypen werden in Java durch Klassen implementiert, die die Daten und die Operationen definieren.

Frage 6: L, R und M seien beliebige Referenztypen. Welche der folgenden Aussagen treffen zu?

- **Aus 'R Untertyp von M' folgt: 'R ist Klasse', 'M ist Interface'. (1A)** ❌
  Warum falsch?: Es ist nicht notwendig, dass R eine Klasse und M ein Interface ist. 'R Untertyp von M' könnte auch bedeuten, dass beide Typen Klassen sind oder dass einer ein Interface ist.
- **R ist kein Untertyp von R. (1B)** ❌
  Warum falsch?: Jede Klasse oder jeder Typ ist ein Untertyp von sich selbst.
- **Ist L ein Interface, dann ist L Untertyp von java.lang.Object. (1C)** ❌
  Warum falsch?: Ein Interface erbt nicht direkt von java.lang.Object, sondern von anderen Interfaces, aber nicht von Object selbst.
- **'null' ist ein Objekt von jedem Referenztyp M. (1D)** ❌
  Warum falsch?: 'null' ist kein tatsächliches Objekt, sondern eine spezielle Referenz, die auf nichts zeigt.
- **Aus 'R Untertyp von M' folgt: 'Kommentare in R und M müssen gleich sein'. (1E)** ❌
  Warum falsch?: Kommentare haben nichts mit der Vererbung zu tun und können in verschiedenen Klassen oder Interfaces unterschiedlich sein.

Frage 7: Welche der folgenden Aussagen stimmen in Bezug auf die unterschiedlichen Arten linearer und assoziativer Datenstrukturen?

- **Assoziative Datenstrukturen sind (effizient) wahlfrei zugreifbar. (2A)** ✅
  Warum richtig?: Assoziative Datenstrukturen (wie HashMaps) ermöglichen einen effizienten Zugriff auf Daten mittels Schlüsseln, ohne dass eine feste Reihenfolge erforderlich ist.
- **Lineare Datenstrukturen können bei Bedarf größer werden. (2B)** ✅
  Warum richtig?: Lineare Datenstrukturen wie Listen oder Arrays können dynamisch wachsen (z.B. durch Resizing).
- **Wahlfreie Zugriffe auf lineare Datenstrukturen sind sehr effizient. (2C)** ❌
  Warum falsch?: Wahlfreie Zugriffe auf lineare Datenstrukturen wie Arrays oder Listen sind in der Regel nicht sehr effizient, insbesondere bei großen Datenmengen.
- **Lineare Datenstrukturen verwenden Schlüssel zur Adressierung. (2D)** ❌
  Warum falsch?: Lineare Datenstrukturen wie Arrays oder Listen verwenden Indizes, nicht Schlüssel.
- **Stacks haben FIFO-Verhalten. (2E)** ❌
  Warum falsch?: Stacks verwenden LIFO (Last In, First Out), nicht FIFO (First In, First Out).

Frage 8: Y sei ein Referenztyp (Klasse oder Interface), und m sei eine durch 'T m = new W();' deklarierte Variable, wobei der Compiler keinen Fehler meldet. Welche der folgenden Aussagen treffen für alle passenden Y, T, W und m zu?

- **Mit 'W ist Untertyp von Y' gilt: '((Y)m).getClass() == W.class' (3A)** ✅
  Warum richtig?: Wenn W ein Untertyp von Y ist, dann ist der tatsächliche Typ von m W, und daher wird getClass() W.class zurückgeben.
- **'(Y)m' liefert Laufzeitfehler wenn Y nicht Untertyp von W ist. (3B)** ❌
  Warum falsch?: Der Code ist nur dann fehlerhaft, wenn T nicht zu Y passt. Wenn Y und W nicht kompatibel sind, würde ein ClassCastException auftreten.
- **'(Y)null' liefert einen Laufzeitfehler. (3C)** ❌
  Warum falsch?: '(Y)null' ist völlig legal, da 'null' für alle Referenztypen gültig ist.
- **'(Y)m' liefert einen Laufzeitfehler wenn T Untertyp von Y ist. (3D)** ❌
  Warum falsch?: Wenn T ein Untertyp von Y ist, kann das Casting ohne Laufzeitfehler durchgeführt werden.
- **T ist Obertyp von W. (3E)** ✅
  Warum richtig?: T ist der deklarierte Typ der Variablen m, und W ist der tatsächliche Typ von m, daher ist T der Obertyp von W.

Frage 9: n sei eine Variable mit einer leeren assoziativen Datenstruktur. Nach welchen der folgenden Aufruf-Sequenzen liefert 'n.get(C)' den String in K als Ergebnis?

- **n.put(C, C); n.put(K, K); n.put(n.get(K), n.get(C)); (4A)** ❌
  Warum falsch?: Dies funktioniert nicht, weil 'n.get(K)' den Wert 'K' zurückgibt, der dann als Schlüssel verwendet wird, was die gewünschte Zuordnung nicht korrekt auflöst.
- **n.put(K, C); n.put(n.get(K), n.get(C)); n.put(C, K); (4B)** ✅
  Warum richtig?: Dies führt zu einer Kettenzuordnung, bei der der Wert von C schließlich K wird.
- **n.put(C, K); n.put(C, C); n.put(n.get(C), n.get(K)); (4C)** ❌
  Warum falsch?: Diese Sequenz führt zu einer Überschreibung und gibt nicht den richtigen Wert zurück.
- **n.put(C, K); n.put(n.get(K), n.get(C)); n.put(K, C); (4D)** ✅
  Warum richtig?: Diese Sequenz stellt sicher, dass die Werte korrekt umgekehrt und zugewiesen werden, sodass 'n.get(C)' den gewünschten Wert liefert.
- **n.put(C, K); n.put(K, C); n.put(n.get(C), n.get(K)); (4E)** ✅
  Warum richtig?: Hier erfolgt die Kettenzuordnung, bei der schließlich der Wert K korrekt

zurückgegeben wird.

Frage 10: e sei eine Variable, die eine leere Double-Ended-Queue ganzer Zahlen enthält. Nach welchen der folgenden Aufruf-Sequenzen liefert 'e.peekFirst()' die Zahl 9 als Ergebnis?

- **e.addFirst(9); e.addFirst(6); e.pollFirst(); (5A)** ✅
  Warum richtig?: 9 wird zuerst hinzugefügt und bleibt oben, bis es entfernt wird.
- **e.addFirst(9); e.addFirst(6); e.addFirst(e.peekLast()); (5B)** ✅
  Warum richtig?: Die Zahl 9 wird durch den ersten Aufruf von addFirst() oben auf der Queue liegen.
- **e.addFirst(9); e.addFirst(6); e.peekFirst(); (5C)** ❌
  Warum falsch?: Dies würde 9 an erster Stelle belassen, aber die Methode funktioniert hier nicht korrekt, da keine Elemente entfernt werden.
- **e.addLast(9); e.addLast(6); e.pollLast(); (5D)** ✅
  Warum richtig?: Die Zahl 9 bleibt an erster Stelle und wird durch `peekFirst()` korrekt zurückgegeben.
- **e.addLast(9); e.addLast(6); e.addFirst(e.peekLast()); (5E)** ❌
  Warum falsch?: Dies würde 6 auf die erste Stelle verschieben und 9 nicht als Ergebnis zurückgeben.