Nachname	Vorname	Matrikelnummer	Studienkennzahl

Schriftliche Prüfung

LV-Titel: Grundzüge der Programmierung

LV-Nr.: 80

LV-Typ(en): LVP

LV-Leiter: Holmes, Ta'id

Institut: Department für Informationsverarbeitung und Prozessmanagement,
Institut für Informationswirtschaft,
Wirtschaftsuniversität Wien
Augasse 2-6, A-1090 Wien

Datum: 30. Juni 2007

Zeit: 09:30

Dauer: 60 min

Inhaltsverzeichnis

Software-Entwicklung (10)

1. Nennen Sie Phasen eines Softwareprojekts. **×10**

- Analyse
- Design
- Implementierung
- Test

Programmiersprachen (10)

2. Nennen Sie verschiedene Programmelemente und geben Sie Beispiele in Java Syntax. **×10**

Anweisungen

```
System.out.println("eine Anweisung");
```

Blöcke

```
if (true) { System.out.println("Eine Anweisung in einem Block."); }
```

Klassendefinitionen

```
public class Klassendefinition {...}
```

Methoden

```
private void methode(String parameter) {...}
```

Kommentare

```
// Das ist ein Kommentar.
```

Variablen und Datentypen (6)

3. Deklarieren Sie einen Array, in dem Sie ganze Zahlen speichern können. Speichern Sie den Wert 5 in der Variablen an dritter Position im Array. **×5**

```
int[] ganzeZahlen = new ganzeZahlen[2];
ganzeZahlen[2] = 5;
```

Operationen (12)

4. Beschreiben Sie folgende gültige Java Anweisung, indem Sie detailliert auf alle Symbole, sowie Syntax und Semantik eingehen. Wie erfolgt die Ausführung? ×20

```
flase = fasel == fasle == false;
```

=: Zuweisungsoperator, **==**: Gleichheitsoperator, **;**: Abschluss der Anweisung
flase: Eine bereits definierte Variable vom Typ boolean.
fasel bzw. **fasle**: Eine bereits definierte Variable, deren Typ kompatibel ist, um mit dem Variablentyp von fasle bzw. fasel auf Gleichheit geprüft werden zu können.
false: Ein reserviertes Wort in Java mit dem Wert logisch null vom Datentyp boolean.
Syntax ist korrekt: flase wird das Ergebnisses zweier Vergleichsoperationen zugewiesen.
Semantik ist ebenfalls korrekt, da es sich laut Angabe um eine gültige Java Anweisung handelt.
Ausführung zwischen Operationen mit Operatoren gleicher Priorität erfolgt von links nach rechts. Es wird also zuerst fasle mit fasel und das Ergebnis dann mit false auf Gleichheit geprüft.

Kontrollstrukturen (7)

5. Was lautet die Ausgabe des folgenden Programms? ×5

```
public class Main {

  public static void main(String[] args) {
    int i = 1;
    while (i <= 8) {
      if (i >= 8) {
        System.out.println();
        break;
      } else {
        System.out.print(i + " ");
      }
      i = i * 2;
    }
    System.out.println("Ergebnis = " + i);
  }

}
```

```
1 2 4
Ergebnis = 8
```

Methoden (15)

6. Beim Aufruf der Methode `kaufe` der Java Klasse `at.ac.wuwien.ai.holmes.gzp.automat.Automat` kann es zu einer `at.ac.wuwien.ai.holmes.gzp.automat.KeinArtikelException` bzw. einer `at.ac.wuwien.ai.holmes.gzp.automat.NichtGenugGeldException` kommen. Welche Möglichkeiten des Exception Handlings haben Sie bei Aufruf von Methoden? ×15

- Weitererreichen: die Methode von der aufgerufen wird, wirft ihrerseits: Änderung in der Signatur mit `throws KeinArtikelException, NichtGenugGeldException.`
- Lokale Fehlerbehandlung mit `try` & `catch`: im einfachsten Fall

```
try {
  automat.kaufe();
} catch (Exception ex) {
  System.out.println(ex);
}
```

- Mapping: die Methode, von der aufgerufen wird, wirft ihrerseits eine Exception unterschiedlichen Typs: Abfangen und Werfen einer neuen Exception.
- Mischformen aus beiden letzten Formen.

Klassen (20)

7. Übersetzen Sie `Knuth` aus folgendem UML Klassendiagramm in eine abstrakte Javaklasse unter Berücksichtigung des Information Hiding Prinzips. ×10

Knuth
-gewicht: int -groesse: int
+kauen(): void *+knuddeln(): void*

```
abstract class Knuth {
  private int gewicht, groesse;
  abstract void knuddeln();
  abstract void kauen();
}
```

8. Was wird bei Initialisierung einer neuen Instanz folgender Klasse ausgegeben? ×10

```
public class Servus {

  private String x = "servus";

  public Servus() {
    print(x);
    init("hallo");
    for (int x = 2; x < 4; x++) {
      print(x);
    }
    print(x);
  }

  public static void print(String x) {
    System.out.print(x + " ");
  }

  public void print(int x) {
    System.out.print(this.x + "! ");
  }

  private void init(String x) {
    print(x);
    x = x;
    print(x);
  }
}
```

servus hallo hallo servus! servus! servus

Collections (20)

In *Peter und der Wolf* von Sergej Prokofjew ist jeder Figur ein bestimmtes Instrument sowie ein musikalisches Thema zugeordnet. Anbei ein entsprechendes Java Interface für `Figur`**.**

```
interface Figur {
  String getInstrument();
  void spieleThema();
}
```

9. Speichern Sie in einem `java.util.Vector` die mit einem Standardkonstruktor bereits als Klassen implementierten `Figur`en aus *Peter und der Wolf*. Bis auf `Grossvater`, `Jaeger` und `Peter` sind die restlichen Figuren (`Vogel`, `Ente`, `Wolf`, und `Katze`) von einer auf das Interface `Figur` typisierten `java.util.Collection tiere` zu übernehmen. Geben Sie die Anzahl der Figuren aus. **×20**

```
java.util.Collection<Figur> figuren =
     new java.util.Vector<Figur>();
figuren.addAll(tiere);
figuren.add(new Grossvater());
figuren.add(new Jaeger());
figuren.add(new Peter());
System.out.println(figuren.size());
```

10. Es kommt zu einem tragischen Vorfall: Der Wolf frisst die Ente. Entfernen Sie das Enten-Objekt, das Sie über den Schlüssel "`Ente`" aus der `java.util.Hashtable name2figur` erhalten, aus Ihrer Liste der Figuren und geben Sie die Anzahl der Figuren aus. **×10**

```
figuren.remove(name2figur.get("Ente"));
System.out.println(figuren.size());
```

11. Verwenden Sie `java.util.Iterator` und spielen Sie für die polymorphen Figuren Ihrer Liste durch Methodenaufruf das Thema ab. Brechen Sie die Iteration ab, nachdem das Thema für eine Instanz von `Peter` gespielt wurde. Leeren Sie abschließend die Liste. **×20**

```
java.util.Iterator<Figur> it = figuren.iterator();
while(it.hasNext()) {
  Figur figur = it.next();
  figur.spieleThema();
  if (figur instanceof Peter) {
    break;
  }
}
figuren.clear();
```

java.util
Interface Collection<E>

Method Summary	
boolean	**add**(E e) Ensures that this collection contains the specified element (optional operation).
boolean	**addAll**(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	**clear**() Removes all of the elements from this collection (optional operation).
boolean	**contains**(Object o) Returns `true` if this collection contains the specified element.
boolean	**containsAll**(Collection<?> c) Returns `true` if this collection contains all of the elements in the specified collection.
boolean	**equals**(Object o) Compares the specified object with this collection for equality.
int	**hashCode**() Returns the hash code value for this collection.
boolean	**isEmpty**() Returns `true` if this collection contains no elements.
Iterator<E>	**iterator**() Returns an iterator over the elements in this collection.
boolean	**remove**(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	**removeAll**(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	**retainAll**(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	**size**() Returns the number of elements in this collection.
Object[]	**toArray**() Returns an array containing all of the elements in this collection.
<T> T[]	**toArray**(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.

java.util
Interface Iterator<E>

Method Summary	
boolean	**hasNext**() Returns `true` if the iteration has more elements.
E	**next**() Returns the next element in the iteration.
void	**remove**() Removes from the underlying collection the last element returned by the iterator (optional operation).

java.util

Class Hashtable<K,V>

Method Summary	
void	**clear**() Clears this hashtable so that it contains no keys.
Object	**clone**() Creates a shallow copy of this hashtable.
boolean	**contains**(Object value) Tests if some key maps into the specified value in this hashtable.
boolean	**containsKey**(Object key) Tests if the specified object is a key in this hashtable.
boolean	**containsValue**(Object value) Returns true if this hashtable maps one or more keys to this value.
Enumeration<V>	**elements**() Returns an enumeration of the values in this hashtable.
Set<Map.Entry<K,V>>	**entrySet**() Returns a Set view of the mappings contained in this map.
boolean	**equals**(Object o) Compares the specified Object with this Map for equality, as per the definition in the Map interface.
V	**get**(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	**hashCode**() Returns the hash code value for this Map as per the definition in the Map interface.
boolean	**isEmpty**() Tests if this hashtable maps no keys to values.
Enumeration<K>	**keys**() Returns an enumeration of the keys in this hashtable.
Set<K>	**keySet**() Returns a Set view of the keys contained in this map.
V	**put**(K key, V value) Maps the specified key to the specified value in this hashtable.
void	**putAll**(Map<? extends K,? extends V> t) Copies all of the mappings from the specified map to this hashtable.
protected void	**rehash**() Increases the capacity of and internally reorganizes this hashtable, in order to accommodate and access its entries more efficiently.
V	**remove**(Object key) Removes the key (and its corresponding value) from this hashtable.
int	**size**() Returns the number of keys in this hashtable.
String	**toString**() Returns a string representation of this Hashtable object in the form of a set of entries, enclosed in braces and separated by the ASCII characters ", " (comma and space).
Collection<V>	**values**() Returns a Collection view of the values contained in this map.