Kleiner Guide zum Fallenfinden

Nicht vergessen:
Die Ersteller der Prüfungsfragen sind fies, aber auch schreibfaul.
Wenn sie Wissen zu einem Thema abfragen wollen, müssen sie Befehle schreiben, die sich anhand der Syntax verraten!

(01) Warnsignal Packagedeklarationen und Sichtbarkeitsmodifier: Attributsichtbarkeit, nötige Imports

Zwei Klassen im Beispiel, in verschiedenen packages.

```
package parentpackage;
public class Parent {

   private int priv;

   int packagepriv;

   protected int prot;

   public int pub;
}
```

```
package childpackage;

import parentpackage.*;

class Child extends Parent{

}
```

Child erbt von Parent!
→ Augenmerk auf:
  → Attribute, die vererbt werden
  → Imports

→ Sind die Imports korrekt?
  → Child muss die Klasse Parent sehen, um von ihr erben zu können.
  → Import nötig

→ Hat Parent Attribute, die NICHT public sind?
  → Direktes Augenmerk auf die Vererbung:
    → private kennt nur Parent!
    → Kein modifier: Kennt nur das eigene Package!
    → protected: Sichtbar für das eigene package UND erbende Klassen (this.prot)!
    → public: Überall sichtbar!

→ Beispiel:
  → Auf priv und packagepriv kann Child nicht zugreifen
  → Child kann auf sein EIGENES prot zugreifen (this.prot)!
  → Child und alle anderen Klassen können auf pub zugreifen!

(02) Warnsignal Attribute:
Static (Klassenvariablen) vs non-static (Instanzvariablen)

Eine Klasse im Beispiel.

```
public class Test {

  private int priv;
  private static int miau;

  Test(){
    priv=1;
    miau=2; //soll verwirren!
  }

  public static void main(String[] args){
    priv=1;  //Falle!
    miau=2;
  }
}
```

→ Augenmerk auf:
→ Instanzvariable und static variable
→ static methode vorhanden

→ Wo wird auf die Variablen zugegriffen?
→ Suchen nach illegalen, direkten Zugriffen auf die Instanzvariablen in der static Methode
→ Ausnahme natürlich: Es wurde ein Objekt vom Typ der Klasse erstellt und über objekt.variable zugegriffen

→ Wie wirkt sich das auf den Code zur Laufzeit aus?
→ Alle Zugriffe auf die static variable ins Auge fassen. Änderungen an der Variable bleiben bestehen und können überschrieben werden!

→ Wie sieht es mit mehreren Variablen mit gleichem Namen aus?
→ Es gibt im Prinzip 3 Stellen, die sich nicht überschreiben würden:

-Static, Zugriff: Klassenname.variable oder objekt.variable, aber NUR, wenn das Objekt keine Instanzvariable dieses Namens hat! In der praxis den Zugriff über objekt.staticvariable unbedingt vermeiden!

-Instanzvariable, Zugriff: objekt.variable. Ausnahme natürlich private variablen, wenn man sie aus einer anderen Klasse heraus ansprechen will.

-Lokale Variable/Argument
→ Es gelten immer die „nächsten“ Variablen bei gleichem Namen.
→ Methode z.b.:
1. lokal,
2. instanzvariable (nur für NICHT-Static methoden!)
3. static variable

(3) Warnsignal: Imports und Packagedeklarationen in Kombination mit Imports

→ Immer genau zu untersuchen!
- → Knackpunkt Reihenfolge:
  1. packagedeklarationen,
  2. imports
  3. Klassendeklaration
- → Kommentare sind das Einzige, das man buchstäblich hinklatschen kann, wo man möchte
- → Knackpunkt Wildcard bzw *
  - → package.* referenziert alle KLASSEN in dem Paket.
  - → .* referenziert KEINE UNTERPAKETE!

  Beispiel animals.hund und animals.wuff.wolf: animals.* importiert hund, aber NICHT wolf. Dafür bräuchte es animals.wuff.* oder natürlich animals.wuff.wolf

- → Knackpunkt static import
  - → Wichtiger Unterschied zum klassischen import:
    1. Referenziert „Member“ einer Klasse. Das heißt, alles, was die Klasse nach außen anbietet, kann damit importiert werden. Beispiel: animals.hund.* würde alle Member von hund importieren.
    2. Referenziert niemals KLASSEN, immer nur die Member!
    3. Reihenfolge zwingend: import static!

(04) Warnsignal: Konstruktoren mit Argumenten und überladene Konstruktoren

→ Geschenkte Punkte, Fehler sind sehr leicht zu sehen

→ Knackpunkt Konstruktor mit Argumenten
→ Konstruktor ohne Argumente nicht mehr automatisch generiert!
→ Vererbung: Hat die Superklasse keinen Konstruktor ohne Argumente, MUSS die Childklasse einen eigenen Konstruktor bekommen, der den richtigen Konstruktor der Superklasse aufruft! Im Beispiel fehlt dieser!

```
public class Test {

  private int priv;

  static int miau;

  Test(int i){

  }

  Test (int i, int moep){
    this (i);
    super(moep);
  }

  public static void main(String[] args){
    new Test(1);
    new Test(2,3);
  }
}

class TestChild extends Test{

}
```

→ Knackpunkt Sichtbarkeit
→ Man kann Konstruktoren auch mit Modifiern versehen. Dann gelten dieselben Regeln wie für Methoden.
→ Spezialfall: Superklassenkonstruktor ist für Childklasse nicht sichtbar. Muss dann behandelt werden, als gäbe es ihn nicht!

→ Knackpunkt Syntax:
→ Klassenname(argumente){
  //Body
}
→ Erlaubt modifier, also z.b. public Test(). KEIN static!
→ void Test(){} ist KEIN Konstruktor!

→ Knackpunkt zusätzliche Konstruktorcalls
→ Aufruf anderer Konstruktoren zulässig,
Für eigene Konstruktoren: this() oder this(argumente)
Für die Superklasse: super(argumente) oder super()
→ Aufrufe anderer Konstruktoren oder die
der Superklassen unterliegen natürlich auch den Sichtbarkeitsmodifiern.
→ Erlaubt IMMER NUR EINEN weiteren Aufruf, der auch die ALLERERSTE Anweisung sein muss!
Im Beispiel: this(i); wäre zulässig, danach super(moep); wäre direkt illegal!
Es gibt hier keine Spezialfälle! Zwei Konstruktoraufrufe untereinander sind immer illegal!

(05) Collections und Arrays

Wichtige Unterscheidung! Arrays sind ihr eigener Fall!

→ Knackpunkt Größe
→ Was eine Sammlung ist, bietet die .size() Methode.
→ Arrays bieten stattdessen .length. Ohne Klammern, es ist KEINE Methode!
→ Knackpunkt Veränderbarkeit:
→ Arrays werden einmal erstellt und sind dann fest. Man kann den Inhalt ändern, aber NICHT das Objekt selbst (z.B. die Menge der Felder)
→ Collections sind in der Regel in der Lage, problemlos neue Werte hinzuzufügen Es gibt Sonderfälle, aber die werden selten genutzt und kommen in der Prüfung nicht dran.

(06) Deklaration von mehreren Variablen gleichzeitig, vor allem Arrays und Initialisierung

→ Syntax für nicht-Arrays
datentyp variablenname1, variablenname2, variablenname3,...;
→ Fehler/Knackpunkt zusätzliche Datentypen:
int i, double b, c;
int i, c, int f;
→ Knackpunkt Initialisierung:
→ zu nutzender Wert muss an die Variable geschrieben werden und gilt NICHT für andere Variablen in der Liste:
Beispiel int i,u=2,t=3;
i ist nicht initialisiert, u ist 2 und t ist 3!;
→ Syntax für Arrays:
→ zusätzlich zu dem Initialisieren können auch die Dimensionen verschieden angegeben werden!
→ Beispiele:
Int [][] a, b=new int [1][2], c[];
a und c sind nicht initialisiert
a und b sind [][], also 2-dimensional. C ist dreidimensional durch die extra []!

(7) Arrays im Speziellen

→ Deklaration mit Datentyp[]
→ Sind Objekte! Das array.toString() liefer also nur das Object.toString(), ergo Objektname und wilden Hexcode
→ Es gibt in der Klasse „Arrays“ einige Arrayspezifische Extrafunktionen wie z.b. Arrays.sort(array), Arrays.asList(array) u.ä.
→ Können NICHT verändert werden. Nur die Werte in den bestehenden Feldern können geändert werden!
→ Starten bei 0.
→ array.length für die Länge.
→ Letzter index ist array[array.length -1]. Verletzen der Indexgrenzen wirft eine IndexOutOfBoundsException!

Varargs vs Arrays in den Argumenten

→ Syntax modifier rückgabetyp methodenname(Datentyp...)
→ Müssen IMMER das letzte Argument in der Liste sein!
→ Akzeptieren Arrays, aber auch einzelne Werte, die dann zu Arrays umgeformt werden
→ Ein array in den Argumenten erzwingt, dass man ein array zur Übergabe erzeugen muss.
→ arrays in den Argumenten erlauben KEINE einzelnen Teile einer Liste
→ Varargs sind KEIN datentyp! String... s= new String...() o.ä. Sind NICHT erlaubt!

→ Im Beispiel sind die grünen Anweisungen erlaubt. Der rote Text zeigt einen illegalen Gebrauch

```
public static void miau(int a, int b, double d, String... args){

}

public static void wuff(String ... args, int i){

}

public static void main(String[] args){
  miau(0,1,2.0,"blubb","miau");
  miau(0,1,2.0," ichBinNurEinArgument");
  miau (0,1,2.5, new String[8]);
}
```