Lösungshinweise

Test 1 - Klassenbasierte Programmierung

Modul prog, WS23/24
Trier University of Applied Sciences
Informatik Fernstudium (M.C.Sc.)

31.10.2023

Thorsten Suckow-Homberg, `https://thorsten.suckow-homberg.de`

Das Abrufdatum aller in diesem Dokument aufgeführten Webseiten war der 29.10.2023.

Inhaltsverzeichnis

1

Bezeichner

Lösung

- anzahlFlaschen
- year365

Anmerkungen und Ergänzungen

Die im Kurs verwendeten Namenskonventionen werden im Skript auf Seite 35 vereinbart. Diese orientieren sich u.a. an den in den Java Tutorials empfohlenen Namenskonventionen für Variablen[1].

Die bevorzugte Schreibweise bei Variablen ist die *camelCase (sic!)*[2]-Schreibweise: Der erste Buchstabe des Bezeichners ist immer klein. Sollte der Bezeichner aus mehreren Wörtern zusammengesetzt sein, werden die Anfangsbuchstaben der Wörter groß geschrieben[3].
`_` und `$` sind grundsätzlich bei der Verwendung von Bezeichnern erlaubt, entsprechen aber nicht den vereinbarten Namenskonventionen.

Bei numerischen Literalen darf `_` seit Java 7 zur Verbesserung der Lesbarkeit als Separator verwendet werden[4]:

```
1  int eineMillionen = 1_000_000;
```

Seit Java 9 ist `_` ein Schlüsselwort und nicht mehr als Bezeichner erlaubt[5]:

```
1  int _ = 42;
```

[1] The Java™ Tutorials - Variables: `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/variables.html`; siehe ausserdem den archivierten Artikel "Naming Conventions"`https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html`

[2] "Camel case" (Eintrag bei Wikipedia): `https://en.wikipedia.org/wiki/Camel_case`

[3] bei Klassen und Schnittstellen wird i.A. *PascalCase* verwendet: Die Schreibweise entspricht der durch camelCase festgelegten Regeln, allerdings wird hier auch der erste Buchstabe groß geschrieben.

[4] Underscores in Numeric Literals: `https://docs.oracle.com/javase/7/docs/technotes/guides/language/underscores-literals.html`

[5] Java Platform, Standard Edition What's New in Oracle JDK 9:`https://docs.oracle.com/javase/9/whatsnew/toc.htm`

produziert folgenden Compiler-Fehler:

```
1  error: as of release 9, '_' is a keyword, and may not be used as an identifier
2      int _ = 42;
3          ^
```

`class` und `goto` sind reservierte Schlüsselwörter, die als Bezeichner nicht verwendet werden dürfen. Schlüsselwörter werden im Skript auf Seite 37 behandelt. Eine Auflistung findet sich ausserdem in den Java Tutorials[6].

Namenskonventionen machen auch bei der Erstellung von Schnittstellen Sinn. So vereinbart bspw. die *Date-Time API*[7] Präfixe, die Hinweise auf die Intention der Methoden liefern[8]. Domain-Driven Design[Eva04] versteht **Intention-Revealing Interfaces** als essentielles Stilmittel bei der Modellierung von Verhalten und Funktion:

> Name classes and operations to describe their effect and purpose, without reference to the means by which they do what they promise. ([Eva04, 247])

Robert C. Martin weist in diesem Zusammenhang darauf hin, dass man bei der Vergabe von *descriptive names* auch Mut zur Länge haben soll:

> Don't be afraid to make a name long. A long descriptive name is better than a short enigmatic name. ([Mar08, 39])

[6] The Java™ Tutorials - Java Language Keywords: `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html`

[7] The Java™ Tutorials - The Date-Time Packages: `https://docs.oracle.com/javase/tutorial/datetime/overview/packages.html`

[8] The Java™ Tutorials - Method Naming Conventions: `https://docs.oracle.com/javase/tutorial/datetime/overview/naming.html`

2

Variablen-Ausgabe

Lösung

1. a=2
2. b=27
3. c=5
4. m(a)=25
5. a=5
6. b=4
7. c=6

Anmerkungen und Ergänzungen

Klassen-und Objektattribute (sowie -Methoden) werden in dem Skript ab Seite 178 behandelt.

Besonderes Augenmerk ist bei der Analyse dem Modifizierer `static`[1] zu widmen, der in der Aufgabe für die Klassenattribute `a`, `b`, `c` und die Klassenmethode `m` verwendet wird.

In diesem Zusammenhang muss beachtet werden, dass der Parameter `c` der Methode `m` das Klassenattribut `c` überdeckt[2]. Um von `m` aus auf das Klassenattribut `c` zuzugreifen, wäre bei dem vorliegenden Code ein expliziter Zugriff über den Klassennamen notwendig.

Folgendes Beispiel illustriert das Überdecken eines statischen Klassenattributs durch einen Methodenparameter:

```
1  class Foo {
2
3      static int c = 42;
4
5      static void m(int c) {
6          System.out.println("Argument c: " + c);
7          System.out.println("Klassenattribut c: " + Foo.c);
```

[1] The Java™ Tutorials - Understanding Class Members: `https://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html`

[2] siehe hierzu Seite 213 im Skript

```
8
9       }
10
11      public static void main(String[] args) {
12          Foo.m(11);
13      }
14
15  }
```

Die Ausgabe lautet hier

```
1  Argument c: 11
2  Klassenattribut c: 42
```

Die Verwendung von `this.c` würde hingegen in der methode `m` zu einem Compiler-Fehler führen: `this` is ein Objektattribut und funktioniert in einem statischen Kontext nicht:

```
1  non-static variable this cannot be referenced from a static context
```

Siehe hierzu auch Seite 181 im Skript.

3

Überladen

Lösung

- Die Namen der Methoden sind gleich
- Der Rückgabe-Typ kann verschieden sein

Anmerkungen und Ergänzungen

Im Skript wird das Überladen von Methoden ab Seite 166 behandelt.

Bei der Methodenüberladung wird die Parameterliste der Signatur einer Methode geändert (und wahlweise auch der Rückgabetyp).

In Java gehört zu der Methodensignatur der Methodenname sowie die formale Parameterliste[1].

Um eine Methode zu überladen, muss eine Methode erstellt werden, die den gleichen Namen wie die zu überladende Methode besitzt. Die Parameterliste muss abgewandelt werden[2].

In dem folgenden Beispiel sieht man leicht, dass bei der Methodenüberladung auch eine Änderung des Rückgabetyps Sinn machen kann:

```
1
2  class Foo {
3
4      public int sum(int x, int y) {
5          return x + y;
6      }
7
8      public double sum(double x, double y) {
9          return x + y;
10     }
11 }
```

Dass der Compiler trotz unterschiedlicher Parameterliste Schwierigkeiten haben kann, den "richtigen" Methodenaufruf zu finden, behandelt das Skript auf Seite 167

[1] Java Language Specification - 8.4.2. Method Signature: https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.2

[2] The Java™ Tutorials - Defining Methods https://docs.oracle.com/javase/tutorial/java/java00/methods.html

(unten).

Das folgende Programm demonstriert dies:

```
1  class Foo {
2
3      public double sum(double x, int y) {
4          return x + y;
5      }
6
7      public double sum(int x, double y) {
8          return x + y;
9      }
10
11     public static void main(String args[]) {
12
13         Foo f = new Foo();
14
15         System.out.println(f.sum(0, 0));
16
17     }
18 }
```

Es wird der folgende Compiler-Fehler produziert:

```
1  error: reference to sum is ambiguous
2          System.out.println(f.sum(0, 0));
3                               ^
4    both method sum(double,int) in Foo and method sum(int,double) in Foo match
```

Diesbzgl. könnte man dazu neigen, die Antwort “Beim Aufruf der Methoden kann es zu Verwechslungen kommen.“ mit in die Lösung einzubeziehen. Allerdings muss zuerst der Begriff “Aufruf“ im richtigen Kontext verstanden werden. Der Compiler stellt fest, ob es zu Verwechslungen kommen kann: Ist das der Fall, unterbricht der Kompiliervorgang mit einer Fehlermeldung. Dadurch wird ausgeschlossen, dass es während der Laufzeit - also beim “Aufruf“ - zu Verwechslungen kommen kann.

Das Schlüsselwort `this` hat mit Methodenüberladung nichts zu tun.

4

Durchschnitt

Lösung

- Der Compiler meldet in Zeile 13 einen Fehler.
- Das Programm wird vom Compiler zurückgewiesen; daher gibt es keine Ausgabe.
- Die Methode berechne in Zeile 3 wird in Zeile 8 überladen.

Anmerkungen und Ergänzungen

Die Frage lässt sich mit dem Wissen beantworten, das man bereits bei der Lösung von **Überladen** (Frage 3) angewendet hat.

Der Compiler findet in Zeile 13 die Methode `berechne(int, int)`; die gleiche Signatur wird aber bereits in Zeile 3 definiert, was durch den Compiler bemängelt wird. Aus diesem Grund gibt es keine Programm-Ausgabe, wohl aber eine Fehler-Ausgabe durch den Compiler, der das Programm zurückweist.

5

Vergleichsoperator

Lösung

- a == null liefert true, wenn a eine Referenzvariable ist, der ein Wert zugewiesen wurde, die aber auf kein Objekt verweist.
- Mit dem Operator == wird getestet, ob zwei Referenzvariablen auf dasselbe Objekt verweisen.
- Bei primitiven Datentypen liefert a == b den Wert true, wenn beide Variablen denselben Wert haben.

Anmerkungen und Ergänzungen

Vergleichsoperationen werden im Skript ab Seite 77 behandelt.

Für die Beantwortung der Frage ist das Wissen um die verschiedenen Typen in Java wichtig, die sich unterteilen in **primitive Typen** und **Referenztypen**[1]: zu den primitiven Typen[2] gehören bspw. `boolean` und `double`.

Zu den Referenztypen gehören **Klassen**, **Interfaces**, **Typvariablen**[3] und **Arrays**[4].

Die *Java Language Specification* legt für `null` fest[5]:

> The null reference can always be assigned or cast to any reference type.

Daraus lässt sich schliessen, dass, wenn einer Referenzvariable ein Wert zugewiesen wurde, der auf kein Objekt verweist, dieser Wert nur `null` sein kann (im

[1] Java Language Specification - Chapter 4. Types, Values, and Variables: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html`

[2] Java Language Specification - 4.2. Primitive Types and Values: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html#jls-4.2`

[3] Java Language Specification - 4.4. Type Variables: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html#jls-4.4`

[4] Java Language Specification - 10.1. Array Types `https://docs.oracle.com/javase/specs/jls/se21/html/jls-10.html#jls-10.1`

[5] Java Language Specification - 4.1. The Kinds of Types and Values`https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html`

anderen Fall hätte der Kompiliervorgang bereits einen Fehler produziert), weshalb `a == null` trivialerweise zu `true` evaluiert.

Als Beispiel sei folgender Code gegeben:

```
1  class Foo {
2
3      public static void main(String[] args) {
4          Foo f = 42;
5      }
6  }
```

Beim Kompilieren wird folgende Fehler-Ausgabe erzeugt:

```
1  incompatible types: int cannot be converted to Foo
2  Foo f = 3;
3  ^
4  1 error
```

Als Notiz ist hinzuzufügen, dass sich das Verhalten je nach verwendeter Programmiersprache unterscheidet: So kennen bspw. Python[6] als auch JavaScript[7] `null` als Objekt.

Javas **Autoboxing** hat keine Auswirkungen auf die Beantwortung der Frage[8]. Zur Erinnerung: In Java ist es durchaus möglich, einem Objekt einen primitiven Datentypen zuzuweisen. Allerdings sorgt der Compiler dafür, dass der primitive Datentyp in seine Objektrepräsentation "verpackt" wird. Damit steht im Quelltext augenscheinlich ein primitiver Datentyp, der aber beim Kompiliervorgang angepasst wird. Somit verweist die Variable schließlich auf ein Objekt.
Der folgende Code illustriert das Verhalten:

```
1  class Foo {
2
3      public static void main(String[] args) {
4          Object x = 42;
5          Boolean y = true;
6          Character z = 'a';
7
8          System.out.println(x.getClass()); // class java.lang.Integer
9          System.out.println(y.getClass()); // class java.lang.Boolean
10         System.out.println(z.getClass()); // class java.lang.Character
11     }
12
13 }
```

Die Umkehrung dieses Vorganges wird **unboxing** genannt. Weitere Informationen, auch zur Anwendung von *boxing* und *unboxing*, finden sich in [Ull12, 732 ff].

Ob Objekte den gleichen Inhalt haben, wird nicht mit dem Vergleichsoperator `==` überprüft: Zwar lässt sich bei Referenztypen so feststellen, ob zwei Variablen

[6] The Python Standard Library - Built-in Types: `https://docs.python.org/3/library/stdtypes.html?highlight=null#the-null-object`
[7] JavaScript Reference - Expressions and operators (null): `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/null`
[8] im Skript auf Seite 237; ausserdem unter The Java™ Tutorials - Autoboxing and Unboxing: `https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html`

das gleiche Objekt referenzieren. Um aber auf den gleichen Inhalt zu überprüfen, kann bspw. die Methode `equals(Object)`[9] aus `java.lang.Object` überladen (oder überschrieben) werden.

Im folgenden Beispiel überlädt und überschreibt die Klasse `Foo` die Methode `equals`:

```
1  class Foo {
2
3      int x = 0;
4
5      public int getX() {
6          return x;
7      }
8
9      public void setX(int x) {
10         this.x = x;
11     }
12
13     // equals(Object) ueberschreiben
14     public boolean equals(Object f2) {
15         // durch die folgende    berprfung    wird sichergestellt, dass
16         // das explizite casten keine Exception wirft
17         if (!(f2 instanceof Foo)) {
18             return false;
19         }
20         return this.equals((Foo)f2);
21     }
22
23     // equals(Object) mit equals(Foo) ueberladen
24     public boolean equals(Foo f2) {
25         return this == f2 || this.x == f2.x;
26     }
27 }
```

Die Antwort "a == null liefert false, wenn a eine Referenzvariable ist und a noch nicht initialisiert wurde." ist falsch: Eine Referenzvariable kann als Objekt- bzw. Klassenattribute existieren[10]. Diese werden ohne explizite Wertzuweisung implizit mit dem Wert `null` initialisiert[11] (hier würde `a == null` also zu `true` evaluieren).

Eine uninitialisierte Referenzvariable kann aber als lokale Variable in einer Methode existieren: Hier muss vor Verwendung der Variable eine Wertzuweisung stattfinden, ansonsten liefert der Compiler einen Fehler. Aus diesem Grund kann auch keine Überprüfung auf `null` stattfinden, wie in der Antwort behauptet.

[9] Java API Docs - Class Object `https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)`

[10] Java Language Specification - 4.12.3. Kinds of Variables: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-4.html#jls-4.12.3`

[11] Im Skript auf Seite 139 vermerkt - hier wäre allerdings eine Unterscheidung zwischen lokalen Variablen und Objekt-/Klassenattributen förderlich für das Verständnis, da das dort beschriebene Verhalten nicht für lokale Variablen gilt (siehe hierzu auch Frage 6). Weiter Informationen finden sich in The Java™ Tutorials - Primitive Data Types (Default Values): `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`

6

InitTest

Lösung

- t.ok hat den Wert false.
- Bei Entfernen der Kommentarsymbole // meldet der Java-Compiler einen Fehler.
- t.s hat den Wert null.

Anmerkungen und Ergänzungen

Die Antworten lassen sich leicht durch Anwendung des Wissens finden, das bereits auf die vorhergehenden Fragen angewendet wurde:

`s` ist kein Klassenattribut, sondern ein Objektattribut (im Skript erklärt ab Seite 178).

`t.ok` hat den Wert false, weil Klassen- und Objektattribute vom Typ `boolean` automatisch mit dem Wert `false` initialisiert werden. Das gilt allerdings nicht für **lokale Variablen** - diese werden nicht automatisch initialisiert; folglich meldet der Compiler nach Entfernen der Kommentarsymbole auch einen Fehler[1]:

> Local variables are slightly different; the compiler never assigns a default value to an uninitialized local variable. If you cannot initialize your local variable where it is declared, make sure to assign it a value before you attempt to use it. Accessing an uninitialized local variable will result in a compile-time error. ([2])

[1] Default-Werte für primitive Datentypen sowie Hinweis auf Initialisierung im Skript auf Seite 41.
[2] The Java™ Tutorials - Primitive Data Types (Default Values): `https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html`

7

MeinPunkt

Lösung

1. 15
2. 8
3. true
4. false
5. 15
6. 7

Anmerkungen und Ergänzungen

Für das Nachvollziehen des Codes sollte das Prinzip von *Referenzen* verstanden sein - hierfür finden sich Erklärungen im Skript ab Seite 154[1].

[1] ergänzend hierzu The Java™ Tutorials - Creating Objects: `https://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html`

8

Konstruktoren II

Lösung

- Als Standard-Konstruktor wird jeder Konstruktor ohne Argumente bezeichnet.
- Ein Konstruktor-Aufruf wird durch den new-Operator ausgelöst.

Anmerkungen und Ergänzungen

Konstruktoren werden im Skript ab Seite 163 behandelt.

Zunächst müssen wir verstehen, was der Kurs mit **Standardkonstruktor** meint: Dieser wird hier gleichgesetzt mit einem parameterlosen Konstruktor und stimmt von der Methodensignatur her überein mit dem **default constructor**, den der Java Compiler zur Verfügung stellt, wenn kein expliziter Konstruktor implementiert ist.

> *Bloch* schreibt zu dem **default constructor** [1]:
>
> In the absence of explicit constructors, however, the compiler provides a public, parameterless default constructor [...] A default constructor is generated only if a class contains no explicit constructors ([Blo17, 19 f.])
>
> Bei *Ullenboom* findet sich:
>
> Wenn wir in unserer Klasse überhaupt keinen Konstruktor angeben, legt der Compiler automatisch einen an. Diesen Konstruktor nennt die Java Sprachdefinition (JLS) default constructor, was wir als vorgegebener Konstruktor (selten auch Vorgabekonstruktor) eindeutschen wollen." ([Ull12, 515])
>
> Der Autor merkt im folgenden bzgl. der Begrifflichkeit an:
>
> In der Java Language Specification gibt es bei den Konstruktoren nur die Trennungen in no-arg-constructor (parameterloser Konstruktor) und default constructor (vorgegebener Konstruktor), aber den Begriff »standard constructor« gibt es nicht. ([Ull12, 516])

[1] Java Language Specification - 8.8.9. Default Constructor: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.8.9`

sowie weiter:

> Einige Autoren nennen nur den vom Entwickler explizit geschriebenen parameterlosen Konstruktor »Standard-Konstruktor« und trennen dies sprachlich von dem Konstruktor, den der Compiler generiert hat, den sie weiterhin »Default-Konstruktor« nennen. ([Ull12, 517])

Anmerkungen zur Verwendung der Begrifflichkeiten im Kurs finden sich auch in der Aufzeichnung des ersten Online-Tutoriums vom 07.10.2023, ab 03:39 - 03:41 (hh:mm).

Wir dürfen also davon ausgehen, dass der Standardkontruktor im Kurs gleichbedeutend zu einem parameterlosen Konstruktor ist.

Zunächst ist festzuhalten, dass Konstruktoren in Java formal nicht mit Objekt-/Klassenmethoden gleichzusetzen sind und nicht nicht zu den vererbbaren Eigenschaften einer Klasse zählen, wie bspw. (je nach Sichtbarkeit) Methoden oder Attribute[2] (ein Zugriff über `super` auf den Konstruktor einer Elternklasse ist trotzdem möglich).

Wenn für eine Klasse kein Konstruktor definiert wurde, stellt der Compiler einen **default constructor** zur Verfügung[3]. Die Parameterliste des default constructors ist leer. Sie entspricht damit der Signatur des Standardkonstruktors.

```
1  class A {
2
3      // implizit deklarierter default constructor
4      // public A() {
5      // }
6
7      public static void main(String[] args) {
8          A a = new A(); // ruft den implizit deklarierten default constructor
9                         // auf
10     }
11
12 }
```

Wenn ein explizit deklarierter Konstruktor vorhanden ist, stellt der Compiler keinen default constructor zur Verfügung.

Im folgenden Beispiel führt `new A()` zu einem Compiler-Fehler, da kein parameterloser Konstruktor explizit oder implizit für `A` deklariert ist.

```
1  class A {
2
3      int a;
4
5      // explizit deklarierter Konstruktor
6      // unterbindet den impliziert deklarierten default constructor
7      public A(int value) {
```

[2] The Java Tutorials - Inheritance: `https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html`

[3] The Java Language Specification - 8.8.9. Default Constructor: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.8.9`

```
8           a = value;
9       }
10
11      public static void main(String[] args) {
12          A a1 = new A(i); // ruft den explizit deklarierten Konstruktor auf
13          A a2 = new A();  // fuehrt zu einem Compiler-Fehler
14      }
15
16  }
```

Wenn ein Konstruktor nicht explizit einen Konstruktor seiner Elternklasse aufruft, ruft er implizit `super()` auf.

Wenn die Elternklasse in solchen Fällen keinen parameterlosen Konstruktor deklariert hat, wird ein Compiler-Fehler erzeugt.

Im folgenden Beispiel erweitert `B` die Klasse `A`. `B` hat einen Konstruktor deklariert, dessen formale Parameterliste `int` ist. Dieser Konstruktor ruft implizit `super()` auf:

```
1   class A {
2
3   }
4
5   class B {
6
7       int val;
8
9       public B (int a) {
10          // implizit aufgerufen:
11          // super();
12          val = a;
13      }
14
15      public static void main(String[] args) {
16          B b = new B(1);
17      }
18
19  }
```

Ergänzend sei daran erinnert, dass jede Klasse in Java direkt oder indirekt von `java.lang.Object` erbt[4]: Der Konstruktor der Klasse `java.lang.Object` ist also in einer Konstruktor-Aufrufkette enthalten.

Im nächsten Beispiel erbt `C` von `B`. Ein default constructor wird implizit für `C` deklariert. Allerdings führt der Versuch, ein Objekt vom Typ `C` zu erzeugen, zu einem Compiler-Fehler, da `B` keinen parameterlosen Konstruktor - den Standardkonstruktor - deklariert:

```
1   class C extends B {
2
3
4       public static void main(String[] args) {
5           C c = new C(); // Compiler-Fehler
6       }
7
8   }
```

[4] The Java Tutorials - Object as a Superclass: `https://docs.oracle.com/javase/tutorial/java/IandI/objectclass.html`

Damit ein Objekt vom Typ `C` erzeugt werden kann, benötigt `C` einen Standardkonstruktor mit einem expliziten Aufruf des Konstruktors von `B` - da `B` selber keinen Standardkonstruktor deklariert hat, muss der Konstruktor mit der Signature `B(int)` aufgerufen werden:

```
1  class C extends B {
2
3      public C() {
4         super(2); // ruft den Konstruktor von B mit dem Argument 2 auf
5      }
6
7      public static void main(String[] args) {
8          C c = new C();
9      }
10
11 }
```

Wir sehen, dass in Java bei einem explizit deklarierten Konstruktor als erstes der Konstruktor der Elternklasse aufgerufen wird (implizit durch den Compiler) bzw. aufgerufen werden muss (falls explizit implementiert)[5].

Es folgen ergänzende Beispiele. Details sind den Quelltextkommentaren zu entnehmen.

```
1  class A {
2      // impliziter default constructor: vorhanden
3
4      // impliziter default constructor: ruft Konstruktor von java.lang.Object
5      // auf
6  }
7
8
9  class B extends A {
10
11     // impliziter default constructor: vorhanden
12
13     // impliziter default constructor: ruft Konstruktor von A auf
14 }
15
16
17 class C {
18
19     // impliziter default constructor: nicht vorhanden
20
21     public C() {
22         // impliziter Aufruf von java.lang.Object's Konstruktor
23         System.out.println("C created.");
24     }
25 }
26
27
28 class D extends C {
29
30     // impliziter default constructor: vorhanden
31
32     // impliziter default constructor: ruft Konstruktor von C auf
33
34 }
35
36
```

[5] The Java Language Specification - 8.8.7. Constructor Body: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.8.7`

```
37  class E extends D {
38
39      // impliziter default constructor: nicht vorhanden
40
41      public E() {
42          // impliziter Aufruf von D's Konstruktor
43          System.out.println("E created.");
44      }
45  }
46
47
48  class F extends E {
49
50      // impliziter default constructor: nicht vorhanden
51
52      public F(int x) {
53          // impliziter Aufruf von E's Standardkonstruktor
54          System.out.println("F created");
55      }
56  }
57
58
59  class G extends F {
60
61      // impliziter default constructor: nicht vorhanden
62
63      public G(int x) {
64          super(x); // expliziter Aufruf von F's Konstruktor
65
66          // Das Auskommentieren der o.a. Anweisung fuehrt zu einem impliziten
67          // Aufruf von 'super()', was einen Compiler-Fehler produziert:
68          // Da kein Standardkonstruktor in F deklariert ist, muessen wir dem
69          // Konstruktor explizit mitteilen, welcher Konstruktor der
70          // Elternklasse aufgerufen werden soll.
71          System.out.println("G created");
72      }
73  }
```

9

Standardkonstruktor

Lösung

- Klasse B
- Klasse C

Anmerkungen und Ergänzungen

Die Frage lässt sich mit dem Wissen beantworten, das man bereits bei der Lösung von **Konstruktoren II** (Frage 8) angewendet hat.

Da wir den Standardkonstruktor als **parameterlosen Konstruktor** verstehen, fällt Klasse `A` als Antwortmöglichkeit weg: Diese Klasse hat einen expliziten Konstruktor mit der Parameterliste `int` deklariert, somit wird kein impliziter **default constructor** zur Verfügung gestellt.

Dass in Klasse `C` ein parameterloser Konstruktor auch Implementierung im Konstruktor-Rumpf vorhält, ändert nichts daran, dass auch dieser Konstruktor als Standardkonstruktor zu verstehen ist.

10

Add

Lösung

1. 8
2. 21
3. 8
4. 30

Anmerkungen und Ergänzungen

Für die Beantwortung der Frage hilft das Wissen, das man sich zum Thema **Überladen** (Frage 3) angeeignet hat.

Des Weiteren helfen hier Kenntnisse über das Thema **Typanpassungen**[1]. Im Skript wird das Thema beginnend ab Seite 81 erklärt.

Weiterführende Beispiele und Erklärungen finden sich ausserdem in *2.4.10 Die Typanpassung (das Casting)* bei [Ull12, 170 ff.].

[1] Java Language Specification - Chapter 5. Conversions and Contexts: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-5.html`

Literaturverzeichnis

[Blo17] Joshua Bloch. *Effective Java, 3rd Edition*. Addison-Wesley Professional, 2017. ISBN: 978-0134686097.

[Eva04] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. ISBN: 978-0321125217.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series. Upper Saddle River, NJ: Prentice Hall, 2008. ISBN: 978-0-13235-088-4.

[Ull12] Christian Ullenboom. *Java ist auch eine Insel: Das umfassende Handbuch, 10. Auflage*. Galileo Computing, 2012. ISBN: 978-3-8362-1802-3.