Lösungshinweise

Test 3 - Objektorientierung

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

05.12.2023

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

Das Abrufdatum aller in diesem Dokument aufgeführten Webseiten war der 03.12.10.2023.

Die Lösungshinweise beinhalten nicht die Testaufgaben, bei denen eine bestimmte Programmausgabe nachvollzogen werden musste.

Um Konzepte des Lehrmaterials deutlicher von weiterführenden, für den Kurs weniger relevante Themen abzugrenzen, sind diese nun unter **Exkurs** in den jeweiligen Abschnitten zusammengefasst.

Inhaltsverzeichnis

1

Überschreiben

Lösung

- void f(int i)

Anmerkungen und Ergänzungen

Überschreiben und **Überladen** sind zwei unterschiedliche Konzepte[1]:

- Eine Methode wird in einer Unterklasse *überschrieben*, wenn Name, Signatur und Rückgabetyp der Methode der Unterklasse mit der Methode der Oberklasse übereinstimmen.
- Eine Methode wird *überladen*, wenn eine Methode mit gleichem Namen bereitgestellt wird, aber eine unterschiedliche Signatur aufweist. Der Rückgabetyp darf abweichen:

  > There is no required relationship between the return types or between the throws clauses of two methods with the same name, unless their signatures are override-equivalent. (Java Language Specification - 8.4.9. Overloading[2])

Exkurs - Kovarianz

Sobald es sich bei dem Rückgabetyp einer Methode um einen *Referenztypen* handelt, muß der Rückgabetyp der Methode in der Unterklasse folgende Bedingung erfüllen[3]:

- Der Rückgabetyp der Methode in der Unterklasse muss **kovariant** zu dem Rückgabetyp der überschriebenen Methode sein[4]

[1] Seite 210 im Skript
[2] `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.9`
[3] vgl. "Java Language Specification - 8.4.8.3. Requirements in Overriding and Hiding": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.8.3`
[4] vgl. "Java Language Specification - 8.4.5. Method Result": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.5`

Kovariant bedeutet, dass der Rückgabetyp in der Methode der Unterklasse nicht allgemeiner sein darf als der Rückgabetyp der Methode der Oberklasse: Sowohl Richtung der Vererbungshierarchie der als auch die Richtung der Typhierarchie stimmen überein:

- Unterklasse ← Oberklasse[5]
- Rückgabetyp Methode Unterklasse ← Rückgabetyp Methode Oberklasse[6]

Zur Verdeutlichung können die in dem Kurs bereits verwendeten geometrischen Figuren betrachtet werden:

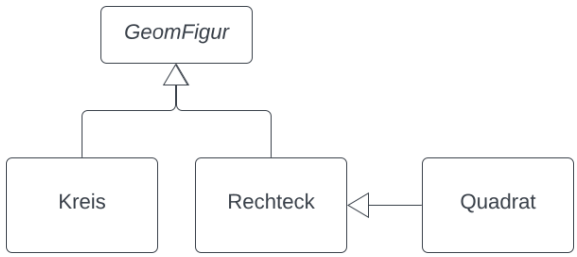

Abb. 1.1. Vererbungshierarchie der Klasse *GeomFigur* und ihrer Subklassen.

Eine Klasse `FigurenFabrik` sei folgendermaßen gegeben:

```
1
2  class FigurenFabrik {
3
4      /**
5       * Erzeugt ein neues Quadrat mit der spezifizierten Seitenl nge.
6       *
7       * @param seitenLaenge die spezifizierte Seitenl nge.
8       */
9      public Quadrat erzeugeQuadrat(double seitenLaenge) {
10         return new Quadrat(seitenLaenge);
11     }
12 }
```

Eine Klasse, die von `FigurenFarik` erbt und `erzeugeQuadrat` überschreiben möchte, muss sicherstellen, dass das von der Methode zurückgelieferte Objekt auch tatsächlich ein `Quadrat` ist.

Hierzu darf der Rückgabetyp nicht allgemeiner sein. ‘Allgemeiner‘ bedeutet in diesem Fall: Typ `Rechteck` oder `Figur`.

Ansonsten würden Zugriffe auf (vererbbare) spezifische Eigenschaften der Klasse `Quadrat` mit den zurückgelieferten Objekten zu Fehlern führen. Der Compiler verhindert dies bereits:

```
1  class KaputteFigurenFabrik extends FigurenFabrik {
2
3      public Rechteck erzeugeQuadrat(double seitenLaenge) {
4          return new Rechteck(seitenLaenge, seitenLaenge*2);
5      }
```

[5] “←“: Richtung d. Vererbungshierarchie
[6] “←“: Richtung d. Typhierarchie

```
6  }
7
8  public class FigurenMacher {
9
10     public static void main(String[] args) {
11         FigurenFabrik f = new FigurenFabrik();
12
13         // bevor es zum Programmabsturz waehrend der Laufzeit
14         // aufgrund eines Aufrufs einer nur in Quadrat bekannten
15         // Methode kommt, bricht der Compiler den Uebersetzungsvorgang
16         // bereits mit einem Fehler ab
17         f.erzeugeQuadrat().spezifischeMethodeAusQuadrat();
18     }
19
20 }
```

Der vom Compiler ausgegebene Fehler weist auf die Inkompatibilität der Rückgabetypen hin:

```
1  return type Rechteck is not compatible with Quadrat.
```

Für **Kovarianz** (und in diesem Zusammenhang **Kontravarianz**) und formalere Herleitungen sei auf *Das Liskovsche Substitutionsprinzip*[7] ([Lis87]) sowie den ausführlichen Beitrag im englischsprachigen Wikipedia [8] verwiesen.

[7] "Wikipedia - Liskov substitution principle": `https://en.wikipedia.org/wiki/Liskov_substitution_principle`

[8] "Wikipedia - Covariance and Contravariance": `https://en.wikipedia.org/wiki/Covariance_and_contravariance_(computer_science)`

2

Vererbung

Lösung

- Eine Variable vom Typ B kann eine Instanz von A referenzieren.
- Die Klasse A erbt alle nicht private deklarierten Objektattribute von B.
- Ein Exemplar von A kann auf alle nicht private deklarierten Objektattribute von B zugreifen.
- Geerbte Klassenattribute sind in Klasse A und B identisch.

Anmerkungen und Ergänzungen

Die Lösungen zu den Fragen lassen sich aus dem Skriptinhalt leicht herleiten:

- "Eine Variable vom Typ B kann eine Instanz von A referenzieren".

Der Sachverhalt wird im Skript zu Begin des Abschnitts "8.4 Zuweisungskompatibilität" erklärt.

- "Die Klasse A erbt alle nicht private deklarierten Objektattribute von B".

Siehe hierzu das Beispiel in der Sprachspezifikation[1], im offiziellen Java Tutorial[2] und im Skript auf Seite 224 "Der Modifizierer private".

- "Ein Exemplar von A kann auf alle nicht private deklarierten Objektattribute von B zugreifen."

Ein "Exemplar" von `A` ist ein Objekt vom Typ `A`. Das Exemplar kann auf geerbte Objektattribute zugreifen:

```
1  package programm;
2
3  class B {
4      int foo = 2;
5  }
6
```

[1] "Java Language Specification - Example 8.2-4. Inheritance of private Class Members": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#d5e13702`

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

```
7  class A extends B {
8
9      public int getFoo() {
10         return foo;
11     }
12 }
```

Ist das Objektattribut `protected` oder `package private`, kann auch über das Exemplar[3] auf das Objektattribut innerhalb desselben Package zugegriffen werden:

```
1  package programm;
2
3  public class Beispiel {
4
5      public static void main(String[] args) {
6          A a = new A();
7          System.out.println(a.foo);
8      }
9
10 }
```

Ist das Objektattribut `public`, kann von überall her auf das geerbte Attribut zugegriffen werden.

- Geerbte Klassenattribute sind in Klasse A und B identisch.

Ein Klassenattribut, das für Klasse `B` definiert wurde, wird (sofern durch den Zugriffsmodifizierer ermöglicht) auch von Klasse `A` geerbt. Wird der Wert des Klassenattributs geändert, ist diese Änderung in allen Unterklassen sichtbar - egal, in welchem Codeteil die Änderung implementiert ist.

```
1  class B {
2      public static int foo = 2;
3  }
4
5  class A extends B {
6
7      public void changeFoo() {
8          foo = 4;
9      }
10
11 }
12
13
14 public class KlassenAttribut {
15
16     public static void main(String[] args) {
17         System.out.println(A.foo); // 2
18         B.foo = 3;
19         System.out.println(A.foo); // 3
20         A a = new A();
21         a.changeFoo();
22         System.out.println(B.foo); // 4
23         System.out.println(A.foo); // 4
24
25
26     }
27
28 }
```

[3] hier greift Code, der nicht zu der Klasse *A* gehört, auf das Objektattribut zu.

Wird ein Klassenattribut in `A` angelegt, welches ein Klassenattribut in `B` verdeckt[4], steuert der Aufrufkontext Zuweisung und Rückgabe des Klassenattributs:

```
1  class B {
2      public static int foo = 2;
3  }
4
5  class A extends B {
6  public static int foo = 2;
7
8      public void printFoo() {
9          System.out.println(foo);
10     }
11
12     public void printParentFoo() {
13         System.out.println(super.foo);
14         System.out.println(B.foo);
15     }
16
17 }
18
19 public class KlassenAttribut {
20
21     public static void main(String[] args) {
22         System.out.println(A.foo); // 2
23         B.foo = 3;
24         System.out.println(A.foo); // 2
25         System.out.println(B.foo); // 3
26         A.foo = 1;
27         System.out.println(A.foo); // 1
28         A a = new A();
29         a.printFoo();              // 1
30         a.printParentFoo();        // 3\n3
31     }
32
33 }
```

[4] im Skript auf Seite 212 ff. Im Englischen ist der Begriff hierzu “hiding“. Ein ausführliches Beispiel findet sich in der Sprachreferenz unter “Java Language Specification - Example 8.4.9-2. Overloading, Overriding, and Hiding“: `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#d5e15760`, weitere Hinweise in “The Java™ Tutorials - Hiding Fields“`https://docs.oracle.com/javase/tutorial/java/IandI/hidevariables.html` sowie bei Ullenboom in [Ull12, 483 ff.].

3

Aufruf von Konstruktoren

Lösung

- Der Konstruktor der Unterklasse ruft automatisch den Standard-Konstruktor der Oberklasse auf, falls kein expliziter super(...)-Aufruf vorhanden ist.
- Gibt es in der Oberklasse keinen Standard-Konstruktor, dann muss in der Unterklasse ein parametrisierter Konstruktor der Oberklasse mit super(arg1,...) aufgerufen werden.

Anmerkungen und Ergänzungen

- Der Konstruktor der Unterklasse ruft automatisch den Standard-Konstruktor der Oberklasse auf, falls kein expliziter super(...)-Aufruf vorhanden ist. werden.

  Im Skript findet sich hierzu im Abschnitt "8.3 Das Schlüsselwort super":

  > Der Aufruf von super() darf nur in Konstruktoren verwendet werden und muss dann die erste Anweisung im Konstruktorrumpf sein.

  Ist der erste Aufruf in einem Konstruktor nicht `this` und nicht `super(...)`, wird der Aufruf des Elternklassen-Konstruktors durch den Compiler implizit durchgeführt (mittels `super()`. Findet sich in der Elternklasse kein parameterloser Konstruktor (der *Standardkonstruktor*), kommt es zu einem Compilerfehler[1].

- Konstruktoren der Oberklassen werden automatisch von der Unterklasse geerbt.

  Die Java Tutorials wiesen diesbzgl. darauf hin, dass:[2]:

  > Constructors are not members, so they are not inherited by subclasses, but the constructor of the superclass can be invoked from the subclass.

- Ist in der Oberklasse ein Konstruktor mit Parametern definiert, wird dieser an Stelle des Standard-Konstruktors in der Unterklasse automatisch aufgerufen.

[1] Im Skript auf Seite 214, Abschnitt "Konstruktoren in Subklassen", außerdem bei "The Java™ Tutorials - Using the Keyword super"': `https://docs.oracle.com/javase/tutorial/java/IandI/super.html`

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

Bis hierhin sollte klar geworden sein, dass das nicht korrekt ist:

> If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass.

[3]

Für weitere ausführliche Anmerkungen und Ergänzungen bzgl. Konstruktoren in Java sei auf den Abschnitt "Konstruktoren II" in den Lösungshinweisen von Test 1 verwiesen.

[3] "The Java™ Tutorials - Using the Keyword super"': `https://docs.oracle.com/javase/tutorial/java/IandI/super.html`

4

Modifikatoren

Lösung

- Eine Methode kann nicht private abstract erklärt werden.
- Von Klassen, die als abstract gekennzeichnet sind, können keine Exemplare erzeugt werden. werden.

Anmerkungen und Ergänzungen

Im Folgenden werden die Antworten anhand der Sprachspezifikationen hergeleitet bzw. widerlegt.

- Eine Methode kann nicht private abstract erklärt werden.

Eine Implementierung einer als `private` markierten abstrakten Methode in einer Kindklasse wäre u.a. aus folgenden Gründen nicht möglich.

Die Sprachspezifikationen legen fest:

> It is a compile-time error to declare an abstract class type such that it is not possible to create a subclass that implements all of its abstract methods.[1]

Eine abstrakte Klasse stellt eine unvollständige Klasse dar[2]. Unterklassen müssen entweder selber als `abstract` gekennzeichnet werden oder die abstrakten Methoden implementieren.

Folglich führt eine als `private` markierte abstrakte Methode zu dem Compilerfehler[3]

```
1 illegal combination of modifiers: abstract and private
```

- Von Klassen, die als abstract gekennzeichnet sind, können keine Exemplare erzeugt werden.

[1] "Java Language Specification - 8.1.1.1. abstract Classes":`https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.1.1.1`

[2] "An abstract class is a class that is incomplete, or to be considered incomplete." in "Java Language Specification - 8.1.1.1. abstract Classes": `8.1.1.1.abstractClasseshttps://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.1.1.1`

[3] Fehlermeldung generiert durch JDK21.0.0

Die Sprachspezifikation legt fest:

> It is a compile-time error if an attempt is made to create an instance of an abstract class using a class instance creation expression.[4]

- Von Klassen, die als abstract gekennzeichnet sind, können keine Unterklassen gebildet werden.

Abstrakte Klassen sind unvollständige Klassen. Es muss deshalb möglich sein, Unterklassen von ihnen zu erzeugen. Die Sprachspezifikationen legen hierzu fest:

> The declaration of an abstract method m must appear directly within an abstract class (call it A).[5]

und weiter

> Every subclass of A that is not abstract[...] must provide an implementation for m, or a compile-time error occurs.[6]

- Eine final erklärte Klasse darf keine abstract erklärten Methoden haben.

Von als `final` deklarierte Klassen können keine Unterklassen gebildet werden[7].
Wenn die Klasse abstrakte Methoden beinhaltet, muss die Klasse als `abstract` implementiert werden. Dies schließt die gleichzeitige Verwendung von `final` aus:

> It is a compile-time error if a class is declared both final and abstract, because the implementation of such a class could never be completed (§8.1.1.1).[8]

[4] "Java Language Specification - 8.1.1.1. abstract Classes": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.1.1.1`
[5] "Java Language Specification - 8.4.3.1. abstract Methods": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.4.3.1`
[6] ebenda
[7] siehe hierzu Skript Seite 38, Absatz "final".
[8] "Java Language Specification - 8.1.1.2. sealed, non-sealed, and final Classes": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-8.html#jls-8.1.1.2`

5

Pakete

Lösung

- Auf Elemente, die in einer Klasse als public deklariert sind, kann aus allen Klassen aller Pakete zugegriffen werden.
- Auf Elemente, für die kein Modifizierer angegeben ist, kann aus allen Klassen desselben Pakets zugegriffen werden.

Anmerkungen und Ergänzungen

Zugriffsmodifizierer werden ausführlich im Skript auf Seite 39 behandelt.

Es lohnt, sich die Zugriffstabelle[1] einzuprägen:

Modifizierer	Klasse	Package	Unterklasse	Welt
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	–
kein Modifizierer	✓	✓	–	–
`private`	✓	–	–	–

Tabelle 5.1. Zugriffsmodifizierer und ihre Sichtbarkeiten

Wenn kein expliziter Zugriffsmodifizierer angegeben wurde, ist die Sichtbarkeit implizit *package-private* (Zeile 3 in Tabelle 5.1).

Erwähnenswert ist ausserdem, dass Java so etwas wie "innere Pakete" nicht kennt. Die Java Tutorials erklären dazu, dass das Package-Konzept kein hierarchisches Modell repräsentiert:

[1] im Skript auf Seite 225, ausserdem bei "The Java™ Tutorials - Controlling Access to Members of a Class": `https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html`

At first, packages appear to be hierarchical, but they are not.[2]

Sie verdeutlichen dies Anhand des Beispiels von `import`-Deklarationen, die Wildcards ("*") in Verbindung mit einem qualifizierten Namen nutzen[3]

So können alle öffentlichen Klassen aus dem Paket `java.awt` in Typdeklarationen einer Übersetzungseinheit[4] über ihren Namen referenziert werden, wenn wie folgt importiert wird:

```
1 import java.awt.*;
```

Dies schließt allerdings nicht die Klassen ein, die in Paketen liegen, die ihrem qualifizierten Namen nach scheinbar "in" `java.awt` liegen, wie bspw. `java.awt.color`. Diese müssen separat importiert werden:

```
1 import java.awt.*;
2 import java.awt.color.*;
```

[2] "The Java™ Tutorials - Apparent Hierarchies of Packages": `https://docs.oracle.com/javase/tutorial/java/package/usepkgs.html`

[3] Eine sog. "Type-Import-on-Demand Declaration". Siehe "Java Language Specification - 7.5.2. Type-Import-on-Demand Declarations": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-7.html#jls-7.5.2`

[4] "Compilation Unit". Siehe hierzu "Java Language Specification - 7.3. Compilation Units": `https://docs.oracle.com/javase/specs/jls/se8/html/jls-7.html#jls-7.3`

6

Schnittstellen

Lösung

- Jede nicht abstrakte Klasse, die eine Schnittstelle implementiert, muss alle Methoden dieser Schnittstelle bereitstellen.
- Alle Methoden einer Schnittstelle sind öffentlich.

  (für beides siehe auch den Exkurs unter "Anmerkungen und Ergänzungen")

Anmerkungen und Ergänzungen

Das Konzept von Schnittstellen mit Anwendungsbeispielen findet sich im Skript auf Seite 230 ff.

Die in der Aufgabe getroffenen Aussagen lassen sich darüber hinaus über die Sprachspezifikationen herleiten bzw. widerlegen.

- Jede Klasse kann nur eine Schnittstelle implementieren.

Widerlegt durch das Skript auf Seite 232, Absatz 3, außerdem wird in den Java Tutorials darauf hingewiesen, dass

> Your class can implement more than one interface, so the implements keyword is followed by a comma-separated list of the interfaces implemented by the class.[1]

- Schnittstellen müssen als abstract gekennzeichnet werden.

Ein `interface` ist in Java implizit `abstract`[2]. Der Modifizierer darf zwar angegeben werden, aber da er obsolet ist, sollte er nicht verwendet werden[3]

- Jede nicht abstrakte Klasse, die eine Schnittstelle implementiert, muss alle Methoden dieser Schnittstelle bereitstellen.

[1] "The Java™ Tutorials - Implementing an Interface": `https://docs.oracle.com/javase/tutorial/java/IandI/usinginterface.html`

[2] im Skript erklärt auf Seite 231, Absatz 2

[3] "Java Language Specification - 9.1.1. Interface Modifiers": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-9.html#jls-9.1.1`

Ist die Methode einer Schnittstelle `abstract`[4], gilt hier das gleiche Prinzip wie bei den abstrakten Klassen: Ist eine implementierende Klasse selber nicht abstrakt, muss die implementierende Klasse die Schnittstellenfunktionen zur Verfügung stellen[5].

Exkurs - Alle Methoden einer Schnittstelle sind öffentlich

Im Kontext des im Skript enthaltenen Lehrmaterials wird diese Antwort als richtige Antwort gezählt. Denn ein Interface beschreibt eine Schnittstelle, deren öffentliche Methoden in implementierenden Klassen zur Verfügung gestellt werden müssen.

Allerdings bietet Java seit Version 8 bereits die Möglichkeit, Methoden in Schnittstellen auszuimplementieren, sofern diese mit dem Modfizierer `default` deklariert werden[6].

Seit Java 9 is es außerdem erlaubt, als `private` deklarierte Methoden in einem `interface` zu implementieren[7]. Als `private` deklarierte Schnittstellenmethoden erlauben eine weitere Kapselung von Funktionalität, die von den *default*-Methoden benötigt wird: Werden mehrere default-Methoden implementiert, die Funktionalität wiederverwenden, kann diese Funktionalität in private Methoden ausgelagert werden.

Als einfaches Beispiel sei folgender Code gegeben, der von JDK21.0.0.0 problemlos übersetzt wird:

```
1
2  interface Fooable {
3      default String getFoo() {
4          return foo();
5      }
6
7      private String foo() {
8          return "foo";
9      }
10
11 }
12
13 public class B implements Fooable {
14     public static void main(String[] args) {
15         B b = new B();
16         System.out.println(b.getFoo());
17     }
18 }
```

In diesem Zusammenhang ist die Aussage wie folgt zu verstehen:

"Alle **abstrakten** Methoden einer Schnittstelle sind öffentlich".

---

[4] siehe nachfolgender Exkurs

[5] siehe hierzu auch "Java Language Specification - 9.4.1. Inheritance and Overriding": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-9.html#jls-9.4.1`

[6] "The Java™ Tutorials - Default Methods": `https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html`

[7] siehe hierzu "JEP 213: Milling Project Coin" unter `https://bugs.openjdk.org/browse/JDK-8042880` sowie "Java Language Specification - 9.4. Method Declarations": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-9.html#jls-9.4`

Die Sprachspezifikationen halten hierzu fest:

> If no access modifier is given, the method is implicitly public.

sowie

> An interface method lacking a private, default, or static modifier is implicitly abstract.[8]

[8] beides zu finden in "Java Language Specification - 9.4. Method Declarations": `https://docs.oracle.com/javase/specs/jls/se21/html/jls-9.html#jls-9.4`

Literaturverzeichnis

[Lis87] Barbara Liskov. „Keynote address - data abstraction and hierarchy“. In: *Addendum to the proceedings on Object-oriented programming systems, languages and applications (Addendum) - OOPSLA '87* (1987). DOI: 10.1145/62138.62141.

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