Inhalt der Vorlesung

1. Klassen und Objekte
2. Vererbung
3. Enums, Wrapper und Autoboxing
4. Interfaces
5. Generics
6. Exceptions
7. Polymorphismus
8. Grafische Oberflächen
9. Streams und Lambda Expressions
10. Leichtgewichtige Prozesse – Threads

Kapitel 4: Interfaces

Inhalt

Kapitel 4: Interfaces

Lernziele

[LZ 4.1] Erklären können, was ein Java Interface ist und was es enthalten kann
[LZ 4.2] Die Interface-Syntax anwenden können für Definition und Nutzung von Interfaces
[LZ 4.3] Das Comparable-Interface kennen und anwenden können
[LZ 4.4] Das Comparator-Interface kennen und anwenden können
[LZ 4.5] Die UML-Darstellung von Interfaces in Klassenmodellen kennen und anwenden

4. Interfaces

4.1 Einführung

Im Rahmen der Entwicklung größerer Programme stellen sich Fragen wie etwa

- wie können mehrere Entwicklerteams parallel entwickeln?
- wie können Frameworks (z.B. JavaFX, s. Kapitel 9) elegant in die eigenen Klassen eingebunden werden?
- wie kann Mehrfachvererbung umgesetzt werden? Java bietet nur Einfachvererbung!
  Beispiele „Ein Amphibienfahrzeug ist ein Landfahrzeug und ein Wasserfahrzeug“
  „Ein AllesFresser ist ein Fleichfresser und ein Pflanzenfresser“
- wie kann Software leicht änderbar, also flexibel bei Änderungen gestaltet werden?

Java bietet als Antwort das Interface-Konzept an, welches eine strikte Trennung von Schnittstelle und Implementierung erlaubt:

> Ein Interface definiert eine Aufruf-Schnittstelle für Objekte. Wenn man von einem Objekt weiß, dass es eine gewisse Schnittstelle bietet, können andere Objekte diese nutzen, d.h. deren Methoden aufrufen.

Objekte können damit andere Objekte aufrufen, von denen sie nur das Interface (also die Methoden) kennen, nicht jedoch die Klasse des aufgerufenen Objekts. Dies eröffnet große Freiheiten bei der Implementierung von Klassen, da deren Implementierungsdetails (Attribute bzw. weitere Methoden außerhalb der Schnittstelle), die beim Aufrufer nicht bekannt sein müssen.

4. Interfaces

4.1 Einführung

Ein Interface entspricht im Prinzip einer vollständig abstrakten Klasse, es enthält i.W.

- Prototypen ohne Implementierung für öffentliche Methoden (nicht-static, keine Konstruktoren!) sowie
- konstante Attribute (entspricht static public final-Deklaration in einer Klasse). Ab Java 8 können Interfaces auch Default-Implementierungen für Methoden bieten, so dass diese dann nicht notwendigerweise implementiert werden müssen.

Einfaches Beispiel: Figur-Schnittstelle → jede Figur kann ihre Fläche berechnen

```
interface Figur {
    double flaeche();
}
```

Anwendung des Interface-Konzepts am Beispiel

1. Man definiert ein Interface, z.B. Figur:

```
interface Figur {
    double flaeche();
}
```

2. Die Klasse Kreis behauptet (*keyword implements*), das Interface Figur zu implementieren. Der Compiler prüft, ob alle Methoden des Interface tatsächlich implementiert werden!

```
class Kreis implements Figur {
    …
    public double flaeche() {
        return Math.PI * radius * radius;
    }
    public double addiereFlaeche(Figur f) {
        return flaeche() + f.flaeche();
    }
}
```

Jedes Kreisobjekt bietet die flaeche-Methode

Implementierung wird von Compiler gefordert!

liefert die Summe der Kreisfläche und der Fläche des Parameter-Objekts

4. Interfaces

4.1 Einführung

Die Methode `addiereFlaeche(Figur f)` gibt die Summe der Flächen des Kreises und eines beliebigen Objekts, welches ebenfalls eine Fläche hat, zurück. Durch die Verwendung des Figur-Interface als Parameter wird die Methode flexibel einsetzbar!

Analog implementiere die Klasse Rechteck das Figur-Interface: `class Rechteck implements Figur { … }`

3. Nutzungs-Beispiele:

```
Kreis kreis1 = new Kreis(2.9)
Kreis kreis2 = new Kreis(2.3);
Rechteck rEck = new Rechteck(2.9, 3.0);
System.out.println("Kreis1 + Rechteck: " + kreis1.addiereFlaeche(rEck));
System.out.println("Kreis1 + Kreis2: " + kreis1.addiereFlaeche(kreis2));
```

Das Besondere an obigem Beispiel ist die rot markierte Parametrisierung der addiereFlaeche-Methode. Diese kann mit jedem Objekt parametrisiert werden, welches das Figur-Interface implementiert. Es wird also keine bestimmte Klasse für den Parameter verlangt, woraus sich die Flexibilität ergibt.

Das Beispiel zeigt, dass Interfaces analog zu Klassen als Datentyp verwendet werden. Die Deklarationen

```
Figur f1 = new Kreis(2.0)
Figur f2 = new Rechteck(1.0, 1.5);
```

sind korrekt, wenn Kreis und Rechteck beide das Figur-Interface implementieren. f1 ist also eine Referenz auf ein Objekt, das das Figur-Interface implementiert.

Beachte: Mit obigen Deklarationen ist der Aufruf `f1.flaeche()` korrekt, der Aufruf, `f1.addiereFlaeche(f2)` jedoch nicht, weil f1 keine Kreisreferenz ist!

4. Interfaces

4.2 Das Interface Comparable

Häufig müssen Objekte miteinander verglichen werden (engl. *to compare*) , etwa, um eine Liste sortiert darzustellen. Die Java-Standard-Bibliothek bietet hierzu das Interface Comparable an, welches eine Methode für den Objektvergleich vorgibt:

```
interface Comparable {
     public int compareTo(Object o);
}
```

Das Interface wird folgendermaßen genutzt, hier gezeigt am Beispiel String, welche das Interface Comparable implementiert:

```
String s1 = "Hallo", s2 = "Hugo";
int vergleich = s1.compareTo(s2);
```

Bezüglich des Ergebnisses von s1.compareTo(s2) ist folgendes gefordert:

< 0 ➔ s1 ist kleiner als s2;
= 0 ➔ s1 ist gleich s2;
> 0 ➔ s1 ist größer als s2

Basierend auf obiger Festlegung können Objekte verglichen und bspw. auch sortiert werden, da Sortierverfahren auf dem paarweisen Vergleich zweier Objekte beruhen. In der Klasse Arrays gibt es einige sehr effiziente Sortiermethoden, die z.B. so genutzt werden können:

```
int[] liste = { 3, 7, 9 };
java.util.Arrays.sort(liste);
for (int wert : liste)
     System.out.println(wert);
```

Beispiel: Vergleich von Quadraten anhand ihrer Fläche

Die Quadrat-Klasse implementiert `Comparable`, damit sind Quadrate mit Hilfe der `compareTo-Methode` vergleichbar.

```
public class Quadrat implements Comparable, Figur {
    public double flaeche() {
        return seitenLaenge * seitenLaenge;
    }
    public int compareTo(Object o) {
        Quadrat q = (Quadrat) o; // nur Quadrate werden verglichen!
        double f1 = flaeche(), f2 = q.flaeche();
        if (f1 < f2)
            return -1; // aufgerufenes Quadrat < übergebenes Q.
        else if (f1 > f2)
            return 1; // aufgerufenes Quadrat > übergebenes Q.
        else
            return 0; // aufgerufenes Quadrat gleich übergebenes Q.
    }
    public static void main(String args[]) {
        Quadrat q1 = new Quadrat(2.01), q2 = new Quadrat(2.0);
        int vergleich = q1.compareTo(q2);
        if (vergleich < 0)
            System.out.println("q1 < q2!");
        else if (vergleich > 0)
            System.out.println("q1 > q2!");
        else
            System.out.println("q1 gleich q2!");
    }}
```

compareTo implementiert den Vergleich zweier Quadrate anhand der Quadrat-Fläche. Alternativ hätte man auch einfach nur die Seitenlängen vergleichen können. Wie verglichen wird, hängt von der jeweiligen Anwendung ab!

Anmerkungen:

- werden float- bzw. double-Werte verglichen, so sollte der Vergleich („==“) vermieden werden, da hier Genauigkeitsprobleme auftreten können. Im Beispiel auf der letzten Seite werden daher zuerst kleiner- und größer-Bedingungen geprüft. Optimal wäre ein Vergleich mit einer definierten Grenze THRESHOLD:

```
if (Math.abs(float1 - float2) < THRESHOLD) // z.B. final double THRESHOLD = .0001;
    System.out.println("f1 and f2 are equal using threshold\n");
else
    System.out.println("f1 and f2 are not equal using threshold\n");
```

- vergleich man zwei ganze Zahlen (Typen „int“, „long“, „short“, „char“), so genügt es, die beiden Werte voneinander abzuziehen, um ein korrektes compareTo-Ergebnis zu erzeugen.
- Das Beispiel auf der letzten Seite erzeugt eine Typecast-Exception, falls compareTo nicht mit einem Quadrat-Parameter aufgerufen wird (was anhand des Parametertyps zulässig ist!). Im Kapitel 5 „Generics“ besprechen wir Möglichkeiten, Interfaces mit Datentypen (=Klassen bzw. Interfaces) zu parametrisieren. Entsprechend würde man dann fordern, dass anstelle eines Object-Parameters ein Quadrat-Parameter übergeben werden muss. Der Compiler könnte dann bereits prüfen, ob compareTo korrekt genutzt wird.

Ein Quadrat-Array basierend auf der Quadrat-Klasse auf der vorherigen Seite kann aufsteigend nach dem Flächeninhalt sortiert werden:

```
Quadrat[] liste = {new Quadrat(1.), new Quadrat(0.1), new Quadrat(3.)};
Arrays.sort(liste);
for (Quadrat q : liste)
     System.out.println("seitenLaenge: " + q.seitenLaenge);
```

In vielen Situationen (z.B. Dateiexplorer: Sortierung durch Klick auf Spaltenkopf) benötigt man jedoch verschiedene Sortierkriterien, d.h. der durch `compareTo` vorgegebene Standard-Vergleich ist zu starr und unflexibel. Der Lösungsansatz besteht darin, für jeden Anwendungsfall ein eigenes Vergleicher-Objekt mit einer Methode zum Vergleich zweier Objekte zu nutzen. Die Vergleichsmethode wird in dem Interface `Comparator` vorgegeben:

```
interface Comparator {
     int compareTo(Object o1, Object o2);
}
```

Sollen nun beispielsweise Quadrate nach ihrem Umfang und nach ihrer Seitenlänge sortierbar sein, so könnte man zwei Comparator-Klassen definieren, die dann einer speziellen Sortiermethode der Klasse Arrays übergeben werden, um die jeweils gewünschte Sortierung zu erreichen. Eine Beispiel-Implementierung findet sich auf der Folgeseite.

4. Interfaces

4.3 Das Interface Comparator

Die sort-Methode sortiert das Quadrate-Array zunächst nach Seitenlänge, dann nach dem Umfang. Dies wird durch Übergabe eines entsprechenden Comparator-Objekts erreicht. Im Zuge der Sortierung wird die compare-Methode des jeweiligen Comparator-Objekts genutzt.

```
class UmfangVergleicher implements Comparator {
       public int compare(Object o1, Object o2) {
              double umfang1 = ((Quadrat) o1).umfang();
              double umfang2 = ((Quadrat) o2).umfang();
              if (umfang1 < umfang2)       return -1;
              else if (umfang1 > umfang2)  return 1;
              else                         return 0;
}}
class SeitenlaengenVergleicher implements Comparator {
       public int compare(Object o1, Object o2) {
              double seite1 = ((Quadrat) o1).getSeitenLaenge();
              double seite2 = ((Quadrat) o2).getSeitenLaenge();
              if (seite1 < seite2)         return -1;
              else if (seite1 > seite2)    return 1;
              else                         return 0;
}}

// sortiere die Quadrateliste zuerst nach Seitenlänge, dann nach Umfang
Quadrat[] quadrate = { new Quadrat(1.0), new Quadrat(0.1), new Quadrat(3.0) };
Arrays.sort(quadrate, new SeitenlaengenVergleicher());
Arrays.sort(quadrate, new UmfangVergleicher());
```

Vergleich nach Umfang

Vergleich nach Seitenlänge

Implementierung mehrerer Interfaces
Eine Klasse kann beliebig viele Interfaces implementieren, jedoch nur von einer Klasse erben:

```
class BuntesQuadrat extends Quadrat implements Figur, Comparable {
    ...
}
```

Die Klasse BuntesQuadrat muss jede Methode der beiden Interfaces implementieren. Tut sie das nicht, muss sie als abstrakte Klasse definiert werden, wodurch die konkreten Unterklassen gezwungen werden, die fehlenden Implementierungen zu definieren.

Ableitung von Interfaces
Über das von der Vererbung bekannte Schlüsselwort „extends“ kann ein Interface beliebig viele Interfaces „beerben“. Dies bedeutet, dass die Schnittstelle der Basis-Interfaces übernommen wird und um zusätzliche Methoden ergänzt werden kann. Sind in den Basis-Interfaces gleiche Methoden-Prototypen definiert, stellt das kein Problem dar. Die Implementierung der geforderten Methode erfüllt gleichzeitig die Anforderungen sämtlicher Interfaces.

UML-Darstellung
In UML wird eine implements-Beziehung mit einem Vererbungspfeil dargestellt. Führt der Pfeil von einer Klasse zu einem Interface, so bedeutet der Pfeil „implements“. Führt er von einem Interface zu einem anderen Interface, so bedeutet der Pfeil „extends“.