Gottlieb-Daimler-Schule 2
Technisches Schulzentrum Sindelfingen
mit Abteilung Akademie für Datenverarbeitung

1 Interfaces vs. abstrakte Klassen

Die zentrale Aufgabe von abstrakten Klassen besteht im logischen Verbinden verwandter Klassen und dem Vererben von Gemeinsamkeiten (Prinzip der Generalisierung/Spezialisierung). Eine Klasse wird abstrakt, wenn sie durch das Schlüsselwort `abstract` dazu erklärt wird und insbesondere, wenn sie abstrakte Methoden enthält.

Beim Sprachkonstrukt „Interface“ geht es zwar auch um abstrakte Methoden, allerdings definiert ein Interface nicht eine Generalisierung von verwandten Unterklassen, sondern lediglich eine gemeinsame Methoden-Schnittstelle zwischen ansonsten logisch unabhängigen Klassen.

2 Eigenschaften von Interfaces

Interfaces dürfen im Gegensatz zu abstrakten Klassen ausschließlich abstrakte Objektmethoden und symbolische Konstanten enthalten[1]. Ein Interface legt somit fest, welche Methoden existieren müssen, aber nicht, wie diese konkret ausprogrammiert sein müssen.

Beispiel:

Datei: ZuVerkaufen.java:

```
public interface ZuVerkaufen
{
   // Die Schlüsselworte public und abstract sind hier implizit
   // gegeben und können weggelassen werden

   double berechnePreis();  // Abstrakte Methode
}
```

Datei Artikel.java:

```
// Die Klasse Artikel verpflichtet sich, alle Methoden aus dem
// Interface ZuVerkaufen zu implementieren.

public class Artikel implements ZuVerkaufen
{
  // Anfang Attribute
  private int iArtikelNr;
  private String sBezeichnung;
  private double dVerkaufsPreis;


...  // weitere Attribute und Methoden

     // Implementierung der Methode aus dem Interface
  @Override
  public double berechnePreis()
  {
    return this.dVerkaufsPreis;
  }
}
```

[1] Daneben können sie seit Java 8 auch statische Methoden (Interfacemethoden) und sogenannte Default-Methoden besitzen.

Datei: Geraet.java:

```
// Die Klasse Geraet verpflichtet sich, alle Methoden aus dem
// Interface ZuVerkaufen zu implementieren.

public class Geraet implements ZuVerkaufen
{
    private String raumID;
    private double bKosten;
    private LocalDate bDatum;

...  // weitere Attribute und Methoden

    public double aktuellerGeraeteWert()
    {
        ... // Bestimmung des Alters Datum

        if (iAlterJ>=4)
        {
            wert = bKosten * faktor[4];
        }
        else
        {
            wert = bKosten * faktor[iAlterJ];
        }

        return wert;
    }

     // Implementierung der Methode aus dem Interface
  @Override
  public double berechnePreis()
  {
    return aktuellerGeraeteWert();
  }
}
```

- Durch das Schlüsselwort `implements` hinter dem Klassennamen wird deklariert, dass die Klasse die abstrakten Methoden des genannten Interfaces konkret implementiert. Tut die Klasse das nicht, so wird sie zu einer abstrakten Klasse, die ihrerseits durch weitere Kindklassen vervollständigt werden muss, um konkret zu werden.
- Eine Klasse kann nur von einer Klasse erben, aber gleichzeitig mehrere Interfaces implementieren.
- Ein Interface wird in der Regel von mehreren Klassen implementiert.
- *Interfaces* werden genutzt um Schnittstellen (Gemeinsamkeiten) *logisch unabhängiger Klassen* zu beschreiben ohne dabei Vererbungsketten zu beeinflussen.
  Im Gegensatz dazu wird *Vererbung* genutzt um „ist-ein-Beziehungen“ zu modellieren, d.h. um Gemeinsamkeiten *logisch abhängiger Klassen* zu beschreiben.

Anwendung des Interface (Beispiel):

Datei ZuVerkaufenStartUI.java:

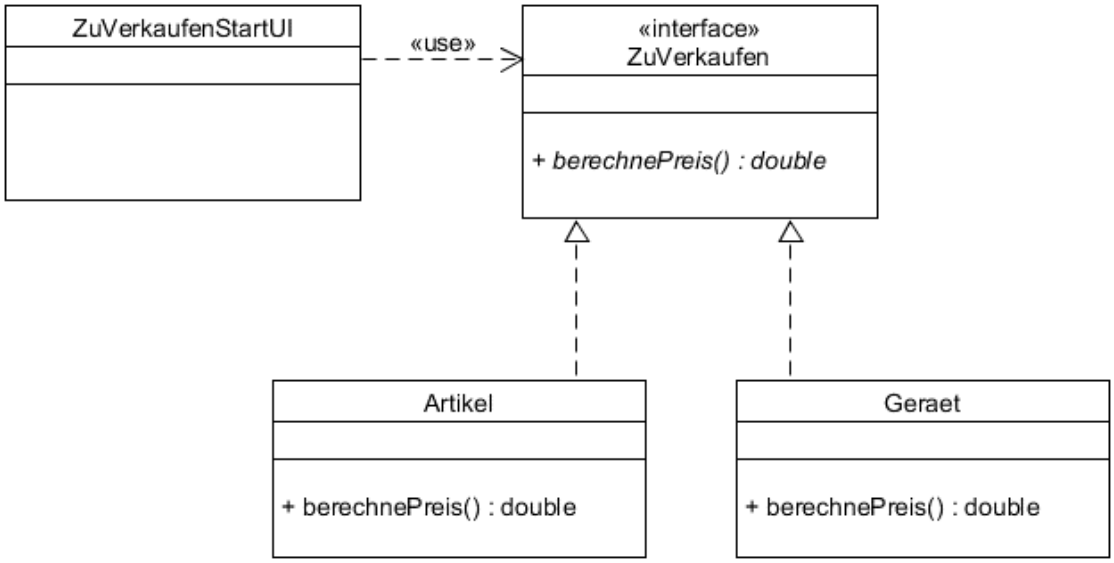
```
public class ZuVerkaufenStartUI
{
  public static void main(String[] args)
  {
    double dGesamtPreis = 0;

    ZuVerkaufen[] aVerschiedenes = new ZuVerkaufen[4];
    // Das Array kann auf beliebige Objekte verweisen, deren Klasse
    // das Interface ZuVerkaufen implementiert.

    aVerschiedenes[0] = new Artikel(4711, "Ladenhüter", 25.60);
    aVerschiedenes[1]= new Artikel(4712, "Bestseller", 15.40);
    aVerschiedenes[2] = new Geraet("B243", 500,
                                   LocalDate.of(2015, 1, 18));
    aVerschiedenes[3] = new Geraet("D114", 450,
                                   LocalDate.of (2017, 2, 14));

    for (ZuVerkaufen einTeil : aVerschiedenes)
    {
      dGesamtPreis = dGesamtPreis + einTeil.berechnePreis();
    }

    System.out.printf("Gesamtpreis = %6.2f €\n", dGesamtPreis);
  }
}
```

Vergleich abstrakter Klassen mit Interfaces

	Abstrakte Klassen		Interfaces
	Objektvariablen und Objektmethoden können vererbt werden. Enthalten abstrakte Methoden.		Enthalten symbolische Konstanten, Klassen- und abstrakte Methoden.
-	Keine Mehrfachvererbung. Eine Kind-klasse kann neben einer abstrakten Vaterklasse keine weiteren Vater-klassen besitzen.	+	Eine Klasse kann beliebig viele Interfaces implementieren und zusätzlich eine Vaterklassen besitzen.
	werden abgeleitet (**extends**)		werden implementiert (**implements**)
	Schlüsselwörter müssen angegeben werden (public, abstract, final,…)		Schlüsselwörter können weggelassen werden, da sie implizit gegeben sind (public, abstract, final,…)

3 Darstellung im UML-Klassendiagramm

Die Implementierung eines Interface durch eine Klasse wird durch einen gestrichelten Vererbungspfeil dargestellt. Die Verwendung eines Interface wird durch einen gestrichelten Pfeil mit einer offenen Spitze und dem Stereotyp «use» abgebildet.



4 Das Interface `Comparable`

4.1 Das Interface `Comparable` ohne Typangabe (Typ-Parameter)

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

Das `Interface Comparable` ist ein Java-System-Interface (aus dem Paket `java.lang`), das lediglich eine abstrakte Methode enthält.

Das Interface kann von beliebigen Klassen implementiert werden, deren Objekte paarweise miteinander vergleichbar sind. Da das Interface von unterschiedlichsten Klassen implementiert werden soll, muss die Methode `compareTo()` mit einem Parameter vom Typ `Object` arbeiten, da sonst die Methode nicht universell einsetzbar ist.

4.1.1 Anwendung des Interface `Comparable`

In der Klasse Lottostatistk (siehe LSG_05_JavaArrays A2.6) soll ein Array sortiert werden, dessen Elemente Objekte der Klasse `MerkmalWertPaar` sind.

Die Klassenmethode `public static void sort(Object[] a)` der Standardklasse `Arrays` (aus dem Paket `java.util`) kann beliebige Arrays sortieren, sofern die entsprechende Klasse das Interface `Comparable` implementiert. In dem Beispiel Lottostatistik muss die Klasse `MerkmalWertPaar` das Interface implementieren, da Objekte dieser Klasse sortiert werden sollen.

Beispiel:

```
public class MerkmalWertPaar implements Comparable
{
    private int iMerkmal;
    private int iWert;
...
    public int compareTo(Object anderesPaar)
    { // Das erste Minuszeichen sorgt für absteigende Sortierung
      return (-(this.iWert - ((MerkmalWertPaar) anderesPaar).iWert));
    }
}
```

Die Methode `compareTo()` muss in der Klasse `MerkmalWertPaar` genau die gleiche Signatur haben wie im Interface `Comparable`. Da der Parameter `anderesPaar` mit Typ `Object` deklariert ist, muss er in der Methode mit einem Typecast versehen werden, damit ein Zugriff auf das Attribut `iWert` möglich ist.

Ein Array der Klasse `MerkmalWertPaar` kann jetzt mit Hilfe der Klassenmethode `Arrays.sort()` sortiert werden. Dies ist möglich, da eine Klasse, die das Interface `Comparable` implementiert, sicherstellt, dass es die `compareTo()` gibt, und diese die geforderte Signatur besitzt.

```
public class LottoStatistik
{
    MerkmalWertPaar[] aVerteilung;
...
    public MerkmalWertPaar[] getNachHäufigkeit()
    {
        MerkmalWertPaar[] aSortiert;

        aSortiert = Arrays.copyOf(aVerteilung, aVerteilung.length);
        Arrays.sort(aSortiert);
        return aSortiert;
    }
}
```

4.2 Das Interface `Comparable<T>` mit Typangabe (Typ-Parameter)

```
public interface Comparable<T>
{
   int compareTo(T o);
}
```

Die Verwendung des Parameter-Typs `Object` hat das Interface `Comparable` (s. 4.1) universell einsetzbar gemacht. Allerdings kann der Compiler dann im konkreten Anwendungsfall (s. 4.1.1) keine Typüberprüfungen vornehmen. Mit Java 2 wurde deshalb eine Spracherweiterung, die sogenannten generischen Typen (Generics) eingeführt. Im Falle von Interfaces bedeutet dies, dass bei der Interface Deklaration ein Typ-Parameter <T> (als Platzhalter) statt

der Klasse `Object` verwendet wird. Dieser Typ-Parameter wird dann im Anwendungsfall durch einen konkreten Typ (eine Klasse) ersetzt.

Beispiel:

```
public class MerkmalWertPaar implements Comparable<MerkmalWertPaar>
{
    private int iMerkmal;
    private int iWert;
...
    public int compareTo(MerkmalWertPaar anderesPaar)
    { // Das erste Minuszeichen sorgt für absteigende Sortierung
      return (-(this.iWert - anderesPaar.iWert));
    }
}
```

Durch die Angabe eines konkreten Typs anstelle des Typ-Parameters wird das Interface auf einen bestimmten Typ festgelegt. In der Methode `compareTo()` der Klasse `MerkmalWertPaar` muss der Parameter `anderesPaar` jetzt mit dem Typ `MerkmalWertPaar` deklariert werden. Dadurch muss in der Methode kein Typecast verwendet werden, da der Compiler den genauen Typ kennt.

Vorteile der Typangabe

1. **Typsicherheit**
   Datentypkonflikte können bereits vom Compiler erkannt werden und führen nicht erst während der Programmlaufzeit zum Programmabsturz.

2. **Kein Typecast notwendig**
   Da die verwendeten Datentypen bekannt sind, ist ein aufwendiges und unübersichtliches Typ-Casting nicht mehr nötig.