Arbeitsblatt Java Interfaces

1 Einleitendes Beispiel

Eine Applikation kann mit verschiedenen geometrischen Objekten umgehen und dabei deren Fläche berechnen. Sie haben somit die gemeinsame Methode:

- berechneFlaeche(): berechnet die Fläche des Objekts

Der Applikation können beliebig viele solche Objekte hinzugefügt werden, wobei dann für jedes die Fläche berechnet und auf die Konsole ausgegeben werden kann. Somit muss für alle Objekte obige Methode aufgerufen werden.

Wie kann dies nun realisiert werden? (Nachfolgender Code liegt ebenfalls auf BSCW bereit, so können Sie die Erklärungen direkt in Eclipse nachvollziehen)

1.1 Variante 1 – gemeinsame Oberklasse

Man könnte versuchen mit einer gemeinsamen Oberklasse zu arbeiten.

```
public class GeoObjekt {

    public GeoObjekt(){}
    protected double berechneFlaeche(){return 0;}
}
```

Eine Unterklasse davon würde dann z.B. so aussehen:

```
public class Dreieck extends GeoObjekt {

    private int grundflaeche;
    private int hoehe;

    public Dreieck(int grundflaeche, int hoehe) {
        super();
        this.grundflaeche = grundflaeche;
        this.hoehe = hoehe;
    }

    public double berechneFlaeche() {
        return (this.grundflaeche * this.hoehe) / 2;
    }
}
```

Eine Verwaltungsklasse hält nun eine Liste von den Objekten und zwar vom Typ der Oberklasse:

```
public class GeoManager {


    private List<GeoObjekt> geoObjects = new ArrayList<GeoObjekt>();

    public void addObj(GeoObjekt obj) {
        this.geoObjects.add(obj);
    }

    public void print() {
        for(GeoObjekt g : this.geoObjects) {
            System.out.println("Fläche: " + g.berechneFlaeche());
        }

    }
}
```

Über die Methode addObj(…) können Objekte hinzugefügt werden, print() berechnet die Fläche für jedes Objekt in der Liste und gibt dies auf die Konsole aus.

Betrachten Sie die Methode print() etwas genauer. Es wird mittels foreach die Liste durchlaufen und auf dem jeweiligen Objekt g kann dann die Methode berechneFlaeche() aufgerufen werden. Diese Variante ist so grundsätzlich schön, sie hat aber einen Haken.

Die Klasse GeoObjekt beinhaltet keine Implementierung für die Berechnung der Fläche, sondern gibt lediglich 0 zurück. Das ist auch logisch, denn die Fläche eines Kreises berechnet sich anders als diejenige eines Dreiecks.

In der Klasse Dreieck wird dann zwar die Methode überschrieben und ermöglicht so eine korrekte Berechnung, **dies ist aber nicht zwingend!** Wir könnten in der Klasse Dreieck die Methode berechneFlaeche() weglassen und würden keinen Compilefehler bekommen. Das Resultat wäre, dass wenn wir print() aufrufen, die Fläche bei diesen Objekten 0 ist, da in der Klasse GeoObjekt 0 zurückgegeben wird.

Und diese Tatsache ist unbefriedigend, da wir ein gewisses Verhalten aller GeoObjekte vorschreiben wollen. Daher verfolgen wir im nächsten Abschnitt einen anderen Ansatz.

1.2 Variante 2 – Abstrakte Oberklasse

Das Problem von Variante 1 kann grundsätzlich durch das Einführen einer abstrakten Oberklasse eliminiert werden:

```
public abstract class GeoObjekt {

    public GeoObjekt(){}
    public abstract double berechneFlaeche();
}
```

Dies ist die einzige Anpassung die vorgenommen werden muss, nun sind alle Unterklassen gezwungen, berechneFlaeche() zu implementieren.

Beobachtungen:

- Die Klasse GeoObjekt beinhaltet keine Instanzvariablen
- Sie hat ebenfalls einen leeren Standardkonstruktor
- Sie beinhaltet lediglich eine abstrakte Methode
- Sie ist daher so abstrakt wie möglich
- Definiert, **WAS** abgeleitete Klassen leisten müssen

Mit diesen Erkenntnissen kommen wir zur Variante 3.

1.3 Variante 3 - Interface

Soll nur deklariert werden, WAS eine Klasse können muss, kann dies über ein Java Interface geschehen:

```
public interface GeoObjekt {

    double berechneFlaeche();
}
```

Die Klasse Dreieck muss dann wie folgt angepasst werden:

```
public class Dreieck implements GeoObjekt {

    private int grundflaeche;
```

2 Interface Definition

2.1 Grundlagen

- Java Interfaces legen Verhalten ohne Implementation fest (WAS)
- Schlüsselwort **interface**
- Interfaces haben keinen Konstruktor
- Alle Methoden eines Interfaces sind implizit **public abstract** (dies muss also nicht geschrieben werden)
- Ein Interface wird auch als Schnittstelle bezeichnet

Beispiel:

```
public interface GeoObjekt {

    double berechneFlaeche();
}
```

2.2 Mehrfachvererbung

Java erlaubt nur die Vererbung von **einer** Oberklasse. Eine Klasse darf jedoch beliebig viele Interfaces implementieren. Dies sogar in Kombination mit der Vererbung einer anderen Klasse.

Folgende Kombinationen sind somit denkbar:

```
public class MyClass implements MyInterface {…}


public class MyClass2 extends BaseClass implements MyInterface
{…}
public class MyClass3 implements MyInterface, MyInterface2 {…}


public class MyClass4 extends BaseClass implements
MyInterface, MyInterface2 {…}
```

2.3 Abstrakte Klasse oder Interface

Wie Sie bereits erkannt haben, ist ein Interface die abstrakteste Form einer abstrakten Klasse. Wann verwenden Sie nun eine abstrakte Klasse und wann ein Interface. Grundsätzlich gilt: Wenn Sie auf der höheren Ebene Implementierungen, welche für alle Unterklassen gilt, machen können, verwenden Sie eine abstrakte Klasse. In den anderen fällen können Sie beides wählen, das Interface ist aber vorzuziehen, so verbauen Sie sich nicht die Möglichkeit von einer anderen Klasse zu erben. Denn Interfaces können Sie so viele implementieren wie Sie wollen, von einer anderen Klasse erben können Sie nur einmal.

Hinweis: Seit Java 8 gibt es auch bei Interfaces die Möglichkeit, sogenannte „Default-Methoden“ zu implementieren. Dies widerspricht etwas dem ursprünglichen Gedanken, die Spezifikation von der Implementierung zu trennen. Trotzdem kann es Situationen geben, wobei dies sinnvoll sein kann. Die Entscheidung des Einsatzes ist aber im speziellen Falle zu analysieren und stellt eine eigene Designentscheidung dar.

2.4 Interface als Typ

Ein Interface definiert lediglich die Methoden, welche eine Klasse implementieren muss. Was bringt es denn nun, ein Interface zu erstellen und in anderen Klassen zu implementieren?

Sie kennen zwei Vorteile der Vererbung:

- Die Unterklasse erbt alle Eigenschaften und Methoden der Oberklasse, dies ermöglicht die Wiederverwendung von Code, Codeduplikation wird verhindert
- Die Unterklasse stellt ein Untertyp der Oberklasse dar. Sie können also eine Variable vom Obertyp definieren, worin ein Objekt des Untertyps gespeichert werden kann und so auf einer allgemeineren Ebene damit arbeiten

Der erste Punkt fällt bei Interfaces weg (ausgenommen Default-Methoden seit Java 8), der zweite wird aber erfüllt. Es ist also möglich, in eine Variable vom Interfacetyp ein Objekt einer Klasse, welches das Interface implementiert zu speichern.

Um auf das einleitende Beispiel zurück zu kommen. GeoObjekt wird in Variante 3 als Interface geführt und die Klasse Dreieck implementiert es. Ein Dreieck-Objekt könnte also nun in eine Variable vom Typ GeoObjekt gespeichert werden. Dies geschieht im Beispiel auch, wobei die Methode print() eine Liste vom Interfacetyp GeoObjekt abarbeitetet und die berechneFlaeche()-Methode darauf aufruft. So können alle Klassen, welche das Interface implementieren, gleich behandelt werden, es ist kein zusätzlicher Code notwendig.

2.5 Interface als Spezifikation

Sie kennen bis jetzt verschiedene Gründe, Interfaces einzusetzen. Die wichtigste Eigenschaft eines Interfaces ist es jedoch, die Definition der Funktionalität (WAS) komplett von der Implementierung zu trennen (WIE).

Ein konkretes Beispiel finden Sie bei den Java-Collections. Für alle Listen existiert das Interface List.

Das Interface definiert die komplette Funktionalität (WAS). Die Klassen, welche das Interface implementieren beinhalten die Implementation (WIE). Die beiden Implementierungen ArrayList und LinkedList unterscheiden sich jedoch erheblich bei der Effizienz von gewissen Funktionen. Bei der ArrayList sind z.B. wahlfreie Zugriffe auf Elemente in der Mitte der Liste viel schneller. Einfügen und Entfernen von Elementen kann wiederum bei einer LinkedList viel schneller sein.

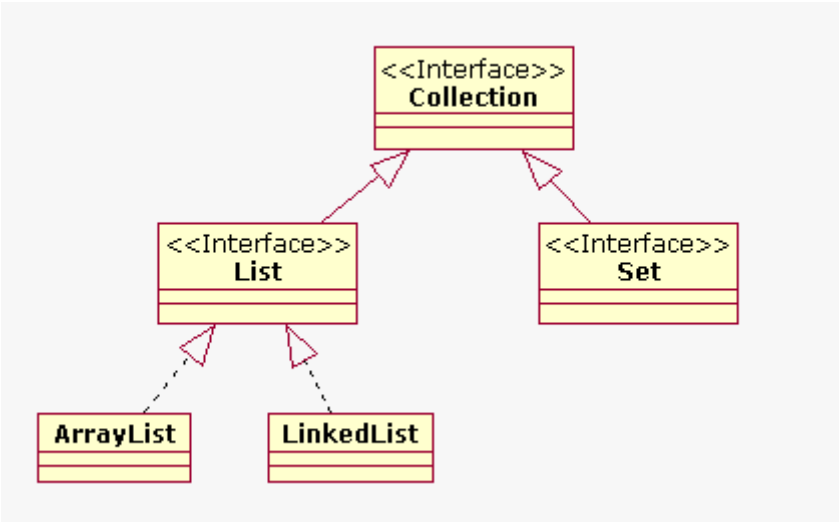

Abbildung 1 – UML Java Collections

Welche von beiden Implementierungen Sie nun wählen bleibt ihre eigene Entscheidung. Ein Wechsel von der Einen zur Anderen ist aber sehr einfach zu realisieren, indem Sie sämtliche Variablen immer vom Typ List<> definieren. Lediglich an einem Ort müssen Sie den konkreten Typ angeben und zwar bei der Erstellung des Objekts:

```
private List<GeoObjekt> geoObjects = new ArrayList<GeoObjekt>();
```

Eine entsprechende Änderung zu LinkedList müsste also nur hier erfolgen.