Inhaltsverzeichnis

1. Implementierung mit Interfaces

[Inhalt | Demo | Übungen]

In Java, ein **Interface** ist ein abstrakter Datentyp, der einer Sammlung von Methoden und/oder Konstanten enthält. Dies ist ein wichtiges **Kernkonzept** in Java und wird vor allem eingesetzt, um Abstraktion, Polymorphism und Mehrfach-Vererbung umzusetzen.

Die Signatur des Interfaces erfordert das Schlüsselwort

```
interface
```

Eine Beispiel-Signatur

```
public interface MyInterface {
    // Inhalt des Interfaces
}
```

Die Implementierung eines Interfaces erfolgt mit dem Schlüsselwort

```
implements
```

Eine Beispiel-Realisierung

```
public class Car implements Vehicle {
    // Implementierung des Interfaces
}
```

1.1. Erzeugungsregeln

In einem **Interface** ist gestattet:

- konstante Variablen
- abstrakte Methoden

- statische Methoden
- `default` Methoden

Darüber hinaus ist **wichtig**, dass ...

- Interfaces nicht direkt instanziiert werden können,
- ein Interface *"leer"* sein kann, also ohne Konstanten oder Methoden,
- das Schlüsselwort `final` nicht genutzt werden kann, da sonst ein Compiler Error entsteht,
- alle Interface Deklarationen `public` oder `default` access haben müssen; der `abstract` Modifizierer wird vom Compiler automatisch hinzugefügt,
- Interface Methoden nicht `protected` oder `final` sein können,
- Seit Java 9 erlaubt ein Interface die Möglichkeit, private Methoden in Interfaces zu definieren,
- Interface Variablen sind `public`, `static`, und `final` per Definition.

Die *grafische* Darstellung der Beziehung zwischen Interfaces und deren Implementierung sieht folgendermaßen aus:

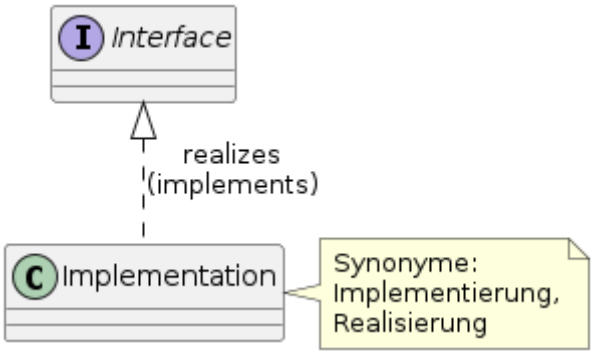

Figure 1. Interface & Realisierung

1.2. Was kann mit Interfaces eigentlich erreicht werden?

Verhaltensvorschrift

Schnittstellen werden verwendet, um bestimmte Verhaltensfunktionen zu definieren, die von "beliebigen" Klassen umgesetzt werden können. Beispiele für Java-Schnittstellen sind `Comparable`, `Comparator` und `Cloneable`. Sie können durch konkrete Klassen implementiert werden.

Für mehr Information dazu siehe → Comparable.html

Mehrfach-Vererbung

Java-Klassen unterstützen nur die einfache (singuläre) Vererbung. Durch die Verwendung von Schnittstellen sind wir jedoch auch in der Lage, Mehrfachvererbungen zu implementieren.

Polymorphismus

Bei Polymorphismus handelt es sich um die Fähigkeit eines Objekts, während der Laufzeit unterschiedliche Formen anzunehmen. Genauer gesagt handelt es sich um die Ausführung der `Override`-Methode, die sich zur Laufzeit auf einen bestimmten Objekttyp bezieht.

In Java kann Polymorphismus mithilfe von Schnittstellen erreicht werden. Beispielsweise kann die Shape-Schnittstelle verschiedene Formen annehmen – es kann ein Kreis oder ein Quadrat sein.

Am **Beispiel** der Klasse `Shape`:

```
public interface Shape {
  String name();
}
```

Die Klasse `Kreis`:

```
public class Circle implements Shape {

    @Override
    public String name() {
        return "Circle";
    }
}
```

Und noch die `Quadrat` Klasse:

```
public class Square implements Shape {

    @Override
    public String name() {
        return "Square";
    }
}
```

```
@Test
@DisplayName("Demo 2: Polymorphismus durch Interfaces")
public void canRealizePolymorphism() {
    // given
    List<Shape> shapes = new ArrayList<>();

    Shape circle = new Circle();
    Shape square = new Square();
    Triangle triangle = new Triangle();

    shapes.add(circle);
    shapes.add(square);
    shapes.add(triangle);
```

```
    // when
    for (Shape shape : shapes) {
        System.out.println(shape.name());
    }

    // then
    assertEquals(3, shapes.size());
}
```

Demo:

Die Nutzung des Interfaces im Zusammenhang mit Polymorphismus anhand einer Demo

```
→ src/test/java/de/dhbw/demo/InterfaceDemoTest.java
```

2. Abstrakte Klassen

Man könnte behaupten, eine abstrakte Klasse sei eine Art **Hybrid** zwischen

1. einer **konkreten** *Klasse* auf der einen Seite und
2. einem **abstrakten** *Interface* auf der anderen.

Mit einer abstrakten Klasse können die Merkmale beider Welten (konkret & abstrakt) umgesetzt bzw. die Vorteile beider Konzepte in *einem* bereitgestellt werden.

Schlüsselkonzepte

Das **Schlüsselwort** ("abstrakter Modifikator") zur Umsetzung von Abstraktionen in Java ist

```
abstract
```

Die wichtigsten **Merkmale** dieser Abstraktionen sind:

- Eine abstrakte Klasse enthält den abstrakten **Modifikator** `abstract` in der Klassensignatur
- Eine abstrakte Klasse kann in Unterklassen unterteilt, aber **nicht instanziiert** werden
- Wenn eine Klasse eine oder mehrere **abstrakte Methoden** definiert, muss die Klasse selbst als abstrakt deklariert werden
- Eine abstrakte Klasse kann sowohl abstrakte als auch konkrete Methoden deklarieren
- Eine von einer abstrakten Klasse abgeleitete Unterklasse muss entweder alle abstrakten Methoden der Basisklasse implementieren oder selbst abstrakt sein

Die zugehörige **grafische Darstellung** dieser Beziehung mit Beispiel:

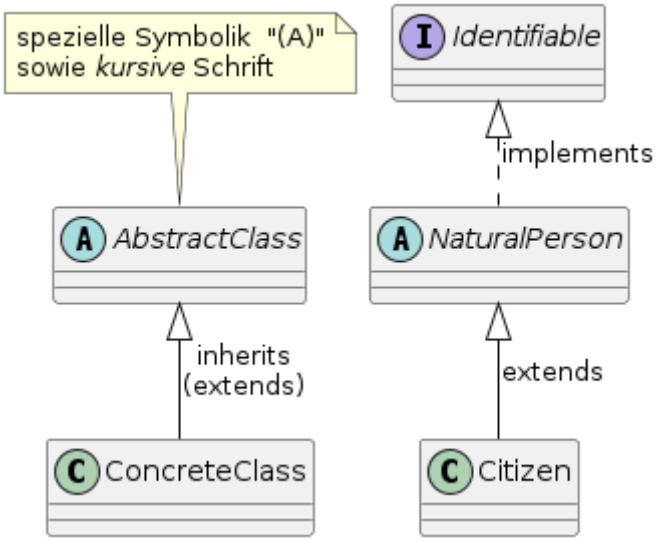

Figure 2. Abstrakte Klassen & Vererbung

Beispiel abstrakte Klasse & Methode:

```
public abstract class MyClass {

    public abstract void myMethod();

}
```

Demo:

```
→ src/test/java/de/dhbw/demo/AbstractsDemoTest.java
```

Nutzung abstrakter Klassen

Java `interface` und `abstract class` sind beides Abstraktionen. Abstrakte Klassen werden häufig bei den folgenden Szenarien eingesetzt:

- **Kapselung** allgemeiner Funktionen in einer Klasse (**Wiederverwendung** von Code). Diese sollen von mehreren verwandten Unterklassen gemeinsam genutzt werden
- Es muss nur teilweise eine API definiert werden, die von Unterklassen leicht **erweitert** und verfeinert werden können
- Die Unterklassen müssen eine oder mehrere gemeinsame Methoden oder Felder mit **geschützten** Zugriffsmodifikatoren erben
- Da sich die Verwendung abstrakter Klassen außerdem implizit mit Basistypen und Untertypen befasst, werden außerdem die Vorteile des **Polymorphismus** genutzt

Zu beachten ist auch, dass die Wiederverwendung von Code oft ein "zwingender" Grund für die Verwendung abstrakter Klassen ist. Dazu in anderen Modulen mehr (→ Beziehungsarten zwischen Klassen)

Übungen:

Übung 1 - Interface

Erzeuge folgende **Klassen**:

1. Ein **Interface** `Zug` mit den Schnittstellenmethoden
   a. `getNumber` (soll den Wert des Feldes `number` vom Typ `String` zurückgeben) sowie
   b. `setNumber` (soll den Wert des Feldes `number` setzen)
2. Eine konkrete Klasse `Regionalzug`, die die Schnittstelle `Zug` realisiert
3. Schreibe einen Test, setze eine Zugnummer, hole diese wieder und teste den korrekten Wert

Umsetzung in

```
→ src/test/java/de/dhbw/exercise/InterfaceExerciseTest.java
```

Übung 2 - Mini-Modell mit abstrakter klassen und Interface

Ein Anwendungsfall (*echte Aussage aus einem Kunden-Interview)*:

> *Es gibt zwei unterschiedliche Typen von (Zug-) Waggons. Es gibt Waggons für "Fahrgäste" und für "Frachtgüter". Als Teil eines Zuges sind die Waggons immer geordnet, sie bilden die sog. "Wagen-Reihung".*

Erzeuge ein kleines, aber "vollständiges" **Klassenmodell** daraus:

1. Eine Schnittstelle `Wagon` mit Methoden
   a. zum *Holen* und *Setzen* der Wagon-Reihenfolge (engl. `order` (Datentyp `int`))
2. Eine abstrakte Klasse `DefaultWagon`, die die Schnittstelle und die dortigen Methoden realisiert
   a. (*Optional*) Das Feld `order` soll dabei möglichst stark geschützt sein
3. Zwei konkrete Klassen, die beide von der abstrakten Klasse *erben*, nämlich
   a. `PassengerWagon` und
   b. `FreightWagon`

```
→ src/test/java/de/dhbw/exercise/ModelExerciseTest.java
```

▼ *Aufklappen mit grafischer Darstellung als kleine Hilfe...*

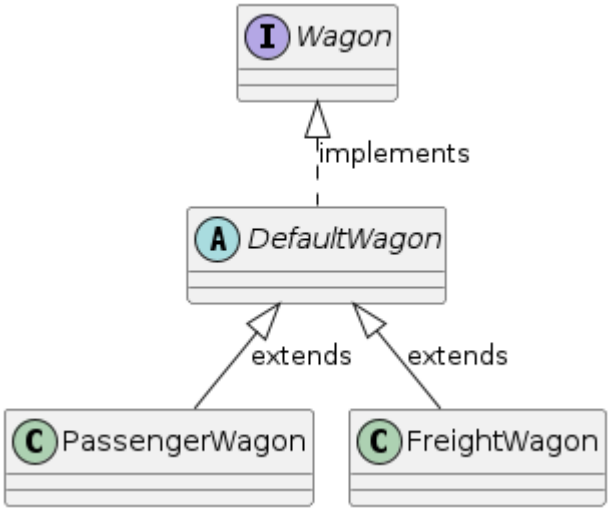

Figure 3. Ein kleines Klassenmodell

3. Referenzen