Überblick

Schnittstellen

- Die Objekt-Methoden einer Klasse definieren die Verhaltensweisen von Objekten dieser Klasse
- Aus Sicht eines Anwenders der Klasse können diese Verhaltensweisen auch als *Funktionalitäten* oder *Dienstleistungen* bezeichnet werden
- Dabei interessiert sich der Anwender (z.B. eine andere Klasse) wirklich nicht für die Implementierungsdetails, er muss nur die Signatur einer Methode kennen, um sie verwenden zu können
- Andererseits hängen die Implementierungsdetails nicht von der konkreten Verwendung durch einen Anwender ab

Schnittstellen

- Der Implementierer möchte also nur wissen, welche Funktionalitäten bereitgestellt werden müssen
- Der Anwender hingegen möchte nur wissen, welche Funktionalitäten bereitgestellt werden
- Beide richten sich nach einer gemeinsamen “Schablone” (*Schnittstelle*, *Interface*), der Implementierer “von innen”, der Anwender “von außen”.

Schnittstellen

- Eine Schnittstelle definiert also Dienstleistungen, die für Anwender (z.B. aufrufende Klassen) zur Verfügung stehen müssen und die der Implementierer bereitstellen muss
- Dabei werden in der Schnittstelle die Implementierungsdetails der Dienstleistungen (im Ggs. zu Klassen) *nicht* festgelegt.
- Es werden *funktionale Abstraktionen* in Form der Methodensignaturen bereitgestellt, die das WAS, aber nicht das WIE festlegen
- Interfaces bestehen i.d.R. nur aus Methodensignaturen, d.h. sie besitzen insbesondere keine Methodenrümpfe und keine Attribute (in Java dürfen sie zusätzlich statische Konstanten spezifizieren)
- Interfaces sind daher ähnlich zu abstrakten Klassen, die ausschließlich abstrakte Methoden besitzen
- Interfaces sind gültige Objekttypen für Variablen, es gibt aber keine Objekte (Instanzen) dieses Typs

Realisierungsbeziehung

- Ein Interface spezifiziert also eine gewisse Funktionalität, ist selbst aber nicht instanziierbar
- Implementiert eine (nicht-abstrakte) Klasse K ein Interface I, müssen alle Methoden des Interfaces in der Klasse implementiert werden
- Zwischen K und I besteht dann eine Realisierungsbeziehung

Schnittstellen/Interfaces in Java

- In Java werden Schnittstellen mit dem Schlüsselwort **`interface`** anstelle von **`class`** vereinbart
- Alle Eigenschaften für den Namensraum von Klassen bzgl. Packages gelten analog für Interfaces
- Alle Methoden eines Interfaces sind grundsätzlich **`public`**, daher ist keine Sichtbarkeitsspezifikation nötig (**`public`** darf aber explizit geschrieben werden)

Beiepiel: Interfaces in Java

- Als Beispiel definieren wir ein Interface `RaeumlichesObjekt`, das den Zugriff auf die Ausdehnung eines (3-dimensionalen) räumlichen Objekts festlegt

```
public interface RaeumlichesObjekt {
    /** Die L&auml;nge des Objekts in mm. */
    int laenge();

    /** Die H&ouml;he des Objekts in mm. */
    int hoehe();

    /** Die Breite des Objekts in mm. */
    int breite();
}
```

Interfaces in Java

- In einem zweiten Beispiel definieren wir ein Interface `Farbig`, das den Zugriff auf die Farbe eines Objekts festlegt und einige wichtige Farben als Konstanten definiert
- Alle *Attribute* eines Interfaces sind grundsätzlich globale, statische Konstanten, daher werden die Zusätze **`public static final`** nicht benötigt (dürfen aber natürlich wieder explizit geschrieben werden)

Interfaces in Java

- Die Definition des Interface `Farbig`

```
import java.awt.*; // fuer die Klasse Color

public interface Farbig {
    /** Die Farbe (als RGB-Zahl) des Objekts. */
    int farbe();

    /** Die Farbe "schwarz". */
    int SCHWARZ = Color.BLACK.getRGB();
    /** Die Farbe "weiss". */
    int WEISS = Color.WHITE.getRGB();
    /** Die Farbe "rot". */
    int ROT = Color.RED.getRGB();
    /** Die Farbe "gr&uuml;n". */
    int GRUEN = Color.GREEN.getRGB();
    /** Die Farbe "blau". */
    int BLAU = Color.BLUE.getRGB();
}
```

Implementierung von Interfaces in Java

- Das Interface `RaeumlichesObjekt` beschreibt zunächst nur die gewünschte Funktionalität, stellt diese aber noch nicht zur Verfügung
- Die Funktionalität kann nur von einer konkreten Klasse zur Verfügung gestellt werden
- Dazu muss die entsprechende Klasse das Interface *implementieren* – dies wird mit dem Schlüsselwort **`implements`** `<InterfaceName>` nach dem Klassennamen angezeigt
- Die Klasse muss dann Methodenrümpfe für alle Methoden des Interfaces definieren
- Eine Klasse kann auch mehrere Interfaces implementieren (dabei muss sie dann *alle* Methoden *aller* Interfaces implementieren)
- Im folgenden sehen wir drei Beispielklassen, die alle das Interface `RaeumlichesObjekt` implementieren
- Zwei Klassen implementieren zudem das Interface `Farbig`

Implementierung von Interfaces

```
public class Auto implements RaeumlichesObjekt, Farbig {
    private int laenge;
    private int hoehe;
    private int breite;
    private int farbe = WEISS;
    // weitere Attribute ...

    public int laenge() {
        return this.laenge;
    }

    public int hoehe() {
        return this.hoehe;
    }

    public int breite() {
        return this.breite;
    }

    public int farbe() {
        return this.farbe;
    }
}
```

Implementierung von Interfaces

```
public class FussballPlatz implements RaeumlichesObjekt, Farbig {
    public int laenge() {
        return 105000;
    }

    public int hoehe() {
        return 0;
    }

    public int breite() {
        return 65000;
    }

    public int farbe() {
        return GRUEN;
    }
}
```

Implementierung von Interfaces

```
public class PapierBlatt implements RaeumlichesObjekt {
    private final int FORMAT;

    public int laenge() {
        int erg = 0;
        switch(FORMAT)
        {
            case 0 : erg = 1189; break;
            case 1 : erg = 841; break;
            case 2 : erg = 594; break;
            case 3 : erg = 420; break;
            case 4 : erg = 297; break;
            // usw. ...
        }
        return erg;
    }

    ... // Fortsetzung naechste Seite
```

Implementierung von Interfaces

```
    ...

    public int hoehe()
    {
        return 0;
    }

    public int breite()
    {
        int erg = 0;
        switch(FORMAT)
        {
            case 0 : erg = 841; break;
            case 1 : erg = 594; break;
            case 2 : erg = 420; break;
            case 3 : erg = 297; break;
            case 4 : erg = 210; break;
            // usw. ...
        }
        return erg;
    }
}
```

Verwendung von Interfaces

- Nützlich sind Interfaces u.a. dann, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in ihrer “normalen” Vererbungshierarchie abgebildet werden können
- Hätten wir `RaeumlichesObjekt` als abstrakte Vaterklasse definiert, und `Auto`, `FussballPlatz` und `PapierBlatt` daraus abgeleitet, ergäbe das eine etwas unnatürliche Vererbungshierarchie
- Durch die Implementierung des Interfaces `RaeumlichesObjekt` können die drei Klassen die Methoden `laenge`, `hoehe` und `breite` dagegen *unabhängig* von ihrer Vererbungslinie garantieren
  (Dies wird später auch bei der Umsetzung einer Mehrfachverebung genutzt)

Verwendung von Interfaces

- Definierte Interfaces können ähnlich wie abstrakte Klassen verwendet werden
- Das folgende einfache Beispiel illustriert die Anwendung des Interfaces `RaeumlichesObjekt`:

  Eine Methode `volumen` (die z.B. eine Hilfsmethode einer `main`-Methode sein kann), die das Volumen von räumlichen Objekten berechnet, kann nun wie folgt definiert sein

```
public static int volumen(RaeumlichesObjekt obj)
{
    return obj.laenge() * obj.hoehe * obj.breite();
}
```

Verwendung von Interfaces

- Die Methode `volumen` akzeptiert als Eingabe nun nur Objekte von Klassen, die das Interface `RaeumlichesObjekt` implementieren, also in unserem Beispiel nur Objekte der Klassen `Auto`, `FussballPlatz` und `PapierBlatt`
- So ist sichergestellt, dass die Methoden `laenge`, `hoehe` und `breite` tatsächlich zur Verfügung stehen
- Wie bereits erwähnt, kann man für Interfaces keine Objekte instanziieren, d.h. die Methode `volumen` kann nicht für ein Objekt vom Typ `RaeumlichesObjekt` aufgerufen werden, sondern nur für Objekte vom Typ einer Klasse, die `RaeumlichesObjekt` implementiert
- Dennoch ist ein Interface ein gültiger Objekttyp, d.h. man kann Variablen vom Typ eines Interfaces vereinbaren (wie z.B. die Parameter-Variable `obj` im obigen Beispiel).

Interfaces und Vererbung

- Eine Klasse, die ein Interface implementiert, kann auch Vaterklasse für ein oder mehrere abgeleitete Klassen sein
- Dann erben alle abgeleiteten Klassen natürlich auch alle Methoden des Interfaces (die ja in der Vaterklasse implementiert wurden und ggf. nochmals überschrieben werden können)
- Dadurch “implementieren” auch alle abgeleiteten Klassen die Interfaces, die von der Vaterklasse implementiert werden
- Auch Interfaces selbst können abgeleitet werden
- Das abgeleitete Interface erbt alle Methoden des Vater-Interface
- Eine implementierende Klasse muss damit auch alle Methoden aller Vater-Interfaces implementieren

Beispiel: Vererbung von Interfaces

```
public interface EinDimensional {
  int laenge();
}

public interface ZweiDimensional extends EinDimensional {
  int breite();
}

public interface DreiDimensional extends ZweiDimensional {
  int hoehe();
}
```

Beispiel: Vererbung von Interfaces

```
public class Auto2 implements DreiDimensional {
  private int laenge;
  private int hoehe;
  private int breite;
  // weitere Attribute ...

  public int laenge()
  {
    return this.laenge;
  }

  public int hoehe()
  {
    return this.hoehe;
  }

  public int breite()
  {
    return this.breite;
  }
}
```

Interfaces und abstrakte Klassen: Unterschiede

- Offensichtlich haben Interfaces und abstrakte Klassen ähnliche Eigenschaften, z.B. können keine Objekte von Interfaces und abstrakten Klassen instanziiert werden
- Im Unterschied zu Interfaces können abstrakte Klassen aber auch konkrete Methoden enthalten, d.h. Methoden mit Rumpf
- Es ist sogar möglich, dass abstrakte Klassen nur konkrete Methoden spezifizieren; mit dem Schlüsselwort `abstract` in der Klassendeklaration ist die Klasse dennoch abstrakt (und daher nicht instanziierbar).
- Alle Methoden eines Interfaces sind dagegen immer abstrakt
- Abstrakte Klassen dürfen im Gegensatz zu Interfaces Attribute spezifizieren

Interfaces und abstrakte Klassen: Verwendung

- Welches Konzept sollte man nun verwenden?
- Interfaces sind flexibler und können in unterschiedlichen Klassenhierarchien verwendet werden, da sie keinerlei Möglichkeiten bereitstellen, Implementierungsdetails festzulegen, sondern lediglich abstrakte Funktionalitäten
- Implementierungsdetails können allerdings in der Dokumentation vorgeschrieben werden – ob diese Vorschrift wirklich eingehalten wird, kann aber der Compiler nicht überprüfen
- Abstrakte Klassen bieten darüberhinaus die Möglichkeit, einen Teil der Implementierungsdetails bereits festzulegen und damit die Wiederverwendbarkeit von Code-Teilen zu unterstützen, wo dies möglich/sinnvoll ist

Interfaces und Mehrfachvererbung

- Wie bereits angedeutet, kann man mit Hilfe von Interfaces auch Mehrfachvererbung in Java modellieren
- Beispiel: `AmphibienFahrzeug` wird von `WasserFahrzeug` und `Landfahrzeug` abgeleitet
- Problem war, dass beide Vaterklassen eine Methode `getPS` implementieren, die nicht überschrieben wird
- Falls die Methode `getPS` für ein Objekt der Klasse `AmphibienFahrzeug` aufgerufen wird, kann nicht entschieden werden, welche ererbte Version ausgeführt werden soll

Interfaces und Mehrfachvererbung

- Lösung: Nur eine der Vaterklassen wird als Klasse realisiert, alle anderen (in unserem Fall nur eine) werden als Interfaces angegeben
- Die (von der übriggebliebenen Vater-Klasse) abgeleitete Klasse muss nun zusätzlich zur Vererbung alle Interfaces implementieren
- Beispiel: `WasserFahrzeug` wird als Interface spezifiziert, wohingegen `Landfahrzeug` eine Klasse ist
- Die Klasse `AmphibienFahrzeug` ist von `Landfahrzeug` abgeleitet und implementiert `WasserFahrzeug`
- Die Methode `getPS` muss (falls sie im Interface `WasserFahrzeug` verlangt wird) in der abgeleiteten Klasse implementiert werden
- Wird die Methode allerdings bereits von der Vaterklasse ererbt, muss `getPS` in `AmphibienFahrzeug` *nicht* implementiert werden (kann aber selbstverständlich überschrieben werden)
- In beiden Fällen ist die Methode `getPS` für Objekte der Klasse `AmphibienFahrzeug` eindeutig bestimmt.

Interfaces und Mehrfachvererbung

- Achtung: Die Realisierung von Mehrfachvererbung in Java mittels Interfaces schränkt das eigentliche Konzept der Mehrfachvererbung ein
- Offensichtlich ist es z.B. nicht möglich, Objekte aller Vaterklassen zu erzeugen
- In unserem Beispiel ist es nicht möglich, Objekte vom Typ `WasserFahrzeug` zu instanziieren
- Ebenfalls eingeschränkt ist die Möglichkeit, Funktionalitäten in Vaterklassen zu implementieren und dann zu vererben (Wiederverwendung)
- Diese Einschränkungen sind allerdings nötig, um die oben angesprochenen Probleme der Mehrfachvererbung zu lösen.