LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Skript zur Vorlesung:

Einführung in die Programmierung

WiSe 2009 / 2010

Prof. Dr. Christian Böhm
Annahita Oswald
Bianca Wackersreuther

Ludwig-Maximilians-Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Gliederung

Schnittstellen

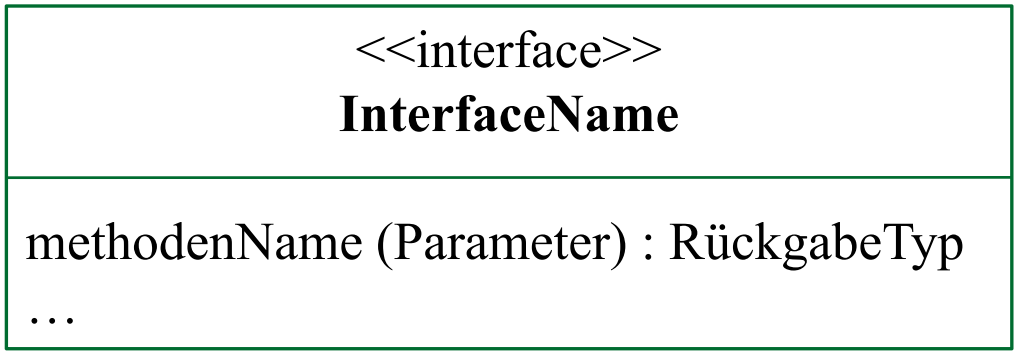
- Die Objekt-Methoden einer Klasse definieren die Verhaltensweisen von Objekten dieser Klasse.
- Aus Sicht eines Anwenders können diese Verhaltensweisen auch als *Funktionalitäten* oder *Dienstleistungen* bezeichnet werden.
- Der Anwender interessiert sich nicht für die Implementierungsdetails, er muss nur die Signatur einer Methode kennen.
- Andererseits hängen die Implementierungsdetails nicht von der konkreten Verwendung durch einen Anwender ab.
- 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* als 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.

Interfaces in UML

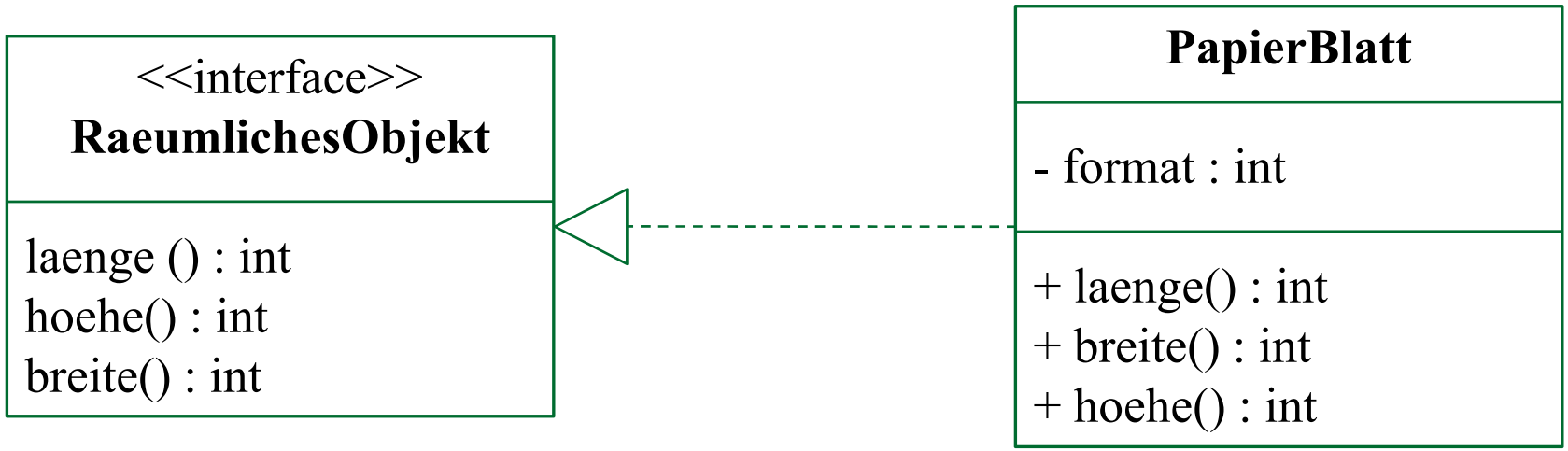
- Ein Interface spezifiziert also eine gewisse Funktionalität, ist selbst aber nicht instanziierbar.
- In UML werden Interfaces wie Klassen dargestellt, allerdings wird vor dem Interfacenamen der Stereotyp "`«interface»`" notiert:



- **Welche OO Modellierungsideen werden vom Konzept der Interfaces realisiert?**

Realisierungsbeziehung

- Implementiert eine (nicht-abstrakte) Klasse ein Interface, müssen alle Methoden des Interfaces in der Klasse implementiert werden.
- Eine Realisierungsbeziehung zwischen einem Interface und einer (implementierenden) Klasse wird in UML folgendermaßen umgesetzt:



Definition von Interfaces

- In Java werden Schnittstellen mit dem Schlüsselwort **`interface`** anstelle von **`class`** vereinbart.
- Alle Methoden eines Interfaces sind grundsätzlich **`public`**.
- 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();
}
```

Definition von Interfaces

- 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.

```
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

- Das Interface `RaeumlichesObjekt` *beschreibt* zunächst nur die gewünschte Funktionalität.
- Die Funktionalität kann nur von einer konkreten Klasse zur Verfügung gestellt werden.
- Dazu muss diese Klasse das Interface *implementieren*. Dies wird mit dem Schlüsselwort **`implements`** `<InterfaceName>` 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 Auto2 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;
    }

    ...
```

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 `Auto2`, `FussballPlatz` und `PapierBlatt` daraus abgeleitet, ergäbe das eine etwas unnatürliche Vererbungs-hierarchie.
- Durch die Implementierung des Interfaces `RaeumlichesObjekt` können die drei Klassen die Methoden `laenge`, `hoehe` und `breite` dagegen *unabhängig* von ihrer Vererbungslinie garantieren.

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 nur Objekte der Klassen `Auto2`, `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. Auf der anderen Seite 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 Auto3 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 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.

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 (hier: nur eine) werden als Interfaces angegeben.
- Die abgeleitete Klasse muss nun 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. Falls sie bereits von der Vaterklasse ererbt wurde, muss sie 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.
- Diese Einschränkung ist allerdings nötig, um die oben angesprochenen Probleme der Mehrfachvererbung zu lösen.

Was sind Marker-Interfaces?

- Interfaces, die weder Methoden noch Konstanten definieren, also einen leeren Rumpf haben, werden *Marker-Interfaces* (auch: *Flag-Interfaces*, *Tagging-Interfaces*) genannt.
- Marker-Interfaces sind dazu gedacht, gewisse (teilweise abstrakte) Eigenschaften von Objekten sicher zu stellen, die typischerweise im Kommentar des Interfaces spezifiziert sind.
- Implementiert eine Klasse ein Marker-Interface, sollte sich der Implementierer an diese Spezifikationen halten.
- Der Programmierer signalisiert durch `implements <MarkerInterface>`, dass seine Klasse die Eigenschaften hat, die im angegebenen Marker-Interface spezifiziert sind.
- Vorsicht bei geerbten Marker-Interfaces!

Beispiel: Das Interface `Cloneable`

- Zur Erinnerung: `clone` erzeugt eine Kopie des aktuellen Objekts.
- Die ursprüngliche Fassung von `clone` erzeugt dabei eine sog. *flache* Kopie (*shallow copy*):
  - Es werden lediglich die Verweise auf die entsprechenden Attribute (d.h. die "Zettel" auf dem Keller) kopiert, nicht aber die dahinterstehenden Objekte.
  - Bei Attributen mit primitiven Typen macht das keinen Unterschied. Der Wert des neuen "Zettels" kann nicht von anderen Benutzern verändert werden, denn nur das neue Objekt hat Zugriff auf diesen "Zettel".
  - Bei Attributen mit Objekttypen wird allerdings nur ein neuer "Zettel" angelegt, der auf das selbe Objekt auf der Halde verweist. Dieses Objekt wird auch von anderen "Zetteln" referenziert, kann also von anderen Benutzern verändert werden.
  - Damit ist nicht sichergestellt, dass die Kopie unabhängig vom ursprünglichen Objekt ist!

Beispiel: Das Interface `Cloneable`

- Die Java-API stellt das Marker-Interface `Cloneable` zur Verfügung, das für die Methode `clone` (die jede Klasse von der impliziten Vaterklasse `Object` erbt) eine spezielle Eigenschaft spezifiziert.
- Implementiert eine Klasse das Interface `Cloneable`, so garantiert der Implementierer, dass die Methode `clone` eine sog. *tiefe* Kopie (*deep copy*) erzeugt: Für alle Attribute mit Objekttypen müssen Kopien der entsprechenden Objekte angelegt werden.
- Die Methode `clone` muss dazu entsprechend überschrieben werden.
- Achtung: Beim Erstellen einer tiefen Kopie muss man darauf achten, dass die Objekte eines Attributs mit Objekttyp selbst wieder Attribute mit Objekttypen haben können.

Beispiel: Das Interface `Cloneable`

Overview Package **Class** Use Tree Deprecated Index Help

Java™ Platform Standard Ed. 6

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

java.lang

Interface Cloneable

All Known Subinterfaces:

AclEntry, Attribute, AttributedCharacterIterator, Attributes, CertPathBuilderResult, CertPathParameters, CertPathValidatorResult, CertSelector, CertStoreParameters, CharacterIterator, CRLSelector, Descriptor, GSSCredential, Name

```
public interface Cloneable
```

A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class.

Invoking Object's clone method on an instance that does not implement the `Cloneable` interface results in the exception `CloneNotSupportedException` being thrown.

By convention, classes that implement this interface should override `Object.clone` (which is protected) with a public method. See `Object.clone()` for details on overriding this method.

Note that this interface does *not* contain the `clone` method. Therefore, it is not possible to clone an object merely by virtue of the fact that it implements this interface. Even if the clone method is invoked reflectively, there is no guarantee that it will succeed.

Since:

JDK1.0

See Also:

`CloneNotSupportedException`, `Object.clone()`

Beispiel: Das Interface `Cloneable`

```
public class DeepCopy implements Cloneable
{
    private int zahl;
    private int[] zahlen;

    public DeepCopy(int zahl, int[] zahlen)
    {
        this.zahl = zahl;
        this.zahlen = zahlen;
    }

    public Object clone()
    {
        int neueZahl = this.zahl;
        int[] neueZahlen = new int[this.zahlen.length];
        for(int i=0; i<this.zahlen.length; i++)
        {
            neueZahlen[i] = this.zahlen[i];
        }
        DeepCopy kopie = new DeepCopy(neueZahl, neueZahlen);
        return kopie;
    }
}
```

Default-Implementierungen von Interfaces

- Wir hatten bisher Interfaces und abstrakte Klassen als ähnliche, aber doch unterschiedlich verwendbare Konzepte kennengelernt.
- Beide Konzepte kann man aber auch sinnvoll vereinen.
- Wird ein Interface `I` voraussichtlich sehr häufig implementiert werden und spezifiziert `I` sehr viele Methoden, dann führt das offensichtlich zu sehr hohem Aufwand.
- In diesem Fall kann es sinnvoll sein, eine Basisklasse zur Verfügung zu stellen, die das Interface `I` und alle sinnvoll realisierbaren Methoden implementiert. Diese Basisklasse wird *Default-Implementierung* von `I` genannt.
- Die Default-Implementierung muss nicht unbedingt *alle* Methoden des Interfaces implementieren. In diesem Fall ist diese Klasse abstrakt.

Default-Implementierungen von Interfaces

- Besitzt eine Klasse, die das Interface `I` implementieren muss, keine andere Vaterklasse, so kann sie von der Default-Implementierung abgeleitet werden.
- Damit erbt die Klasse bereits einen Teil der Methoden aus `I`, die implementiert werden müssen.
- Um dies zu illustrieren, ist auf den folgenden Folien ein beispielhaftes Interface `A`, eine Klasse `DefaultA` als Default-Implementierung von `A` und eine Klasse `B`, die `A` implementiert und von `DefaultA` abgeleitet ist, gegeben.

Default-Implementierungen von Interfaces

```
public interface A
{
    int m1();

    int m2();

    void m3(double d);

    double m4(int i);
}
```

Default-Implementierungen von Interfaces

```
public class DefaultA implements A
{

    private int a1;
    private int a2;

    public int m1()
    {
        return a1 + a2;
    }

    public int m2()
    {
        return a2 - a1;
    }

    public void m3(double d)
    {

    }
    public double m4(int i)
    {
        return 2.0 * i;
    }
}
```

Default-Implementierungen von Interfaces

```
public class B extends DefaultA
{

    private int a3;
    private double a4;

    public void m3(double d)
    {
        a4 = d * d;
        a3 = super.m1() + super.m2();
    }

    public double m4(int i)
    {
        return a4 * (a3 + i);
    }
}
```

Delegation an die Default-Implementierung

- Wenn sich eine Klasse `C`, die `A` implementieren soll, nun nicht von `DefaultA` ableiten lässt (da `C` bereits eine andere explizite Vaterklasse besitzt), müsste sie alle Methoden aus `A` selbst implementieren.
- Da die Default-Implementierung `DefaultA` allerdings bereits nennenswerte Funktionalitäten implementiert, wäre es sehr fehlerträchtig (und auch schlechter Stil), diese ein zweites Mal in der Klasse `C` zu implementieren.
- Stattdessen ist es möglich, die Implementierung an die bereits vorhandene Klasse `DefaultA` zu *delegieren*.

Delegation an die Default-Implementierung

- Dazu muss die Klasse `C` ein Attribut vom Typ `DefaultA` anlegen, und alle Aufrufe der Interface-Methoden an dieses Objekt weiterleiten.
- Achtung: Die Delegation funktioniert nur, wenn die Default-Implementierung *keine* abstrakte Klasse ist!
- Die Klasse `C` ist auf der folgenden Folie beispielhaft dargestellt.

Delegation an die Default-Implementierung

```
public class C extends D implements A
{
    private DefaultA defaultA = new DefaultA();

    public int m1()
    {
        return this.defaultA.m1();
    }

    public int m2()
    {
        return this.defaultA.m2();
    }

    public void m3(double d)
    {
        this.defaultA.m3(d);
    }

    public double m4(int i)
    {
        return this.defaultA.m4(i);
    }
}
```

Die leere Implementierung

- Manchmal wird ein Interface entworfen, bei dem nicht immer alle definierten Methoden benötigt werden.
- Da aber eine implementierende Klasse (wenn sie nicht abstrakt sein soll) immer alle Methoden implementieren muss, kann es sinnvoll sein, eine leere Default-Implementierung zur Verfügung zu stellen.
- Implementierende Klassen können dann ggf. von dieser abgeleitet werden und müssen dann nur noch die Methoden überschreiben, die tatsächlich benötigt werden.