

# Objektorientierte Programmierung

---

## Kapitel 13: Interfaces

Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Wintersemester 2014/15

<http://www.informatik.uni-halle.de/~brass/oop14/>

# Inhalt

## 1 Einführung, Motivation

Vergleich mit abstrakten Klassen, Mehrfache Vererbung

## 2 Syntax, Beispiel

Motivation, Beispiel

Interface-Definition

Definition von Klassen, die Interfaces implementieren

Benutzung von Interface-Typen

## 3 Sub-Interfaces

Erweiterung von Interfaces (Subtyp-Hierarchie)

# Einführung, Motivation (1)

- Ein Interface ist im Prinzip eine abstrakte Klasse mit nur abstrakten Methoden (also auch keine Attribute).

Formal ist ein Interface keine Klasse. Die Referenz-Typen in Java sind Klassen, Interfaces, und Arrays.

- D.h. es ist angegeben, welche Methoden mit welchen Argument- und Resultat-Typen es geben muss, aber es ist keine Implementierung für diese Methoden angegeben, d.h. kein Methoden-Rumpf.

Mit einem Interface trennt man klar Schnittstelle und Implementierung.

- Eine Klasse kann nur eine Oberklasse haben, aber beliebig viele Interfaces implementieren.

D.h. sie muss die im Interface angegebenen Methoden zur Verfügung stellen (so als wenn das Interface eine abstrakte Oberklasse wäre).

## Einführung, Motivation (2)

- Interfaces sind Typen. Man kann also z.B. eine Variable von einem Interface-Typ deklarieren.
- Man kann keine direkten Instanzen von Interfaces anlegen.  
Sie sind ja abstrakt, d.h. den Methoden würde die Implementierung fehlen.
- Eine Klasse **C**, die ein Interface **I** implementiert, ist ein Untertyp von **I**.  
Da es sich nicht um Klassen handelt, spricht man von “Untertyp” und “Obertyp”, aber diese verhalten sich z.B. bei Zuweisungen wie “Unterklasse” und “Oberklasse”.
- Man kann also ein Objekt der Klasse **C** einer Variablen vom Typ **I** zuweisen (wenn **C** das Interface **I** implementiert).

## Einführung, Motivation (3)

### Beispiel:

- Ein achsenparalleles Rechteck ist ein spezielles Viereck.
- Daher sollte die Subtyp-Beziehung gelten: Überall, wo man ein allgemeines Viereck übergeben kann, sollte man natürlich auch so ein Rechteck übergeben können.

Das Rechteck ist ja ein Viereck.

- Während ein allgemeines Viereck die  $(x,y)$ -Koordinaten aller vier Punkte speichern muss, reichen für ein achsenparalleles Rechteck ein Punkt sowie Breite und Höhe.
- Wäre `Rechteck` aber eine Subklasse von `Viereck`, würde sie die Attribute für die vier Punkte erben.

Subklassen sind nicht nur Subtypen, sondern erben auch Implementierung.

## Einführung, Motivation (4)

### Beispiel, Forts.:

- Daher sollte `Viereck` ein Interface sein, das von der Klasse `Rechteck` implementiert wird.

Außerdem: Klasse `AllgemeinesViereck` (implementiert auch das Interface).

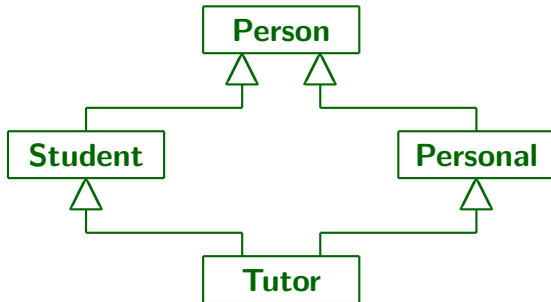
- Man könnte `Viereck` auch als abstrakte Klasse deklarieren.

Das wäre richtig, wenn einige Methoden doch schon in `Viereck` implementiert werden können (die Attribute für die vier Punkte fehlen aber, s.o.).

- Vorteil der Lösung mit Interface:  
Die Subklassen-Hierarchie wird nicht eingeschränkt.
- Z.B. gibt es schon die Klasse `Rectangle` im Paket `java.awt`.  
Die Klasse `Rechteck` könnte `Rectangle` als Oberklasse haben, und gleichzeitig `Viereck` implementieren.

# Mehrfache Vererbung (1)

- Mehrfache Vererbung (“Multiple Inheritance”) bedeutet, dass eine Klasse mehrere Oberklassen haben kann.
- Z.B. ist ein Tutor (wissenschaftliche Hilfskraft)
  - ein Student (mit Matrikelnummer, Studiengang, etc.) und
  - ein Angestellter der Universität (mit Bankverbindung, etc.).



## Mehrfache Vererbung (2)

- Java erlaubt nur einfache Vererbung (“Single Inheritance”), d.h. jede Klasse kann maximal eine Oberklasse haben.

Genauer hat jede Klasse außer `Object` genau eine Oberklasse.

- Manche anderen Sprachen, wie z.B. C++, lassen mehrfache Vererbung (“Multiple Inheritance”) zu.
- Das kann zu unübersichtlichen Situationen führen, wenn z.B. ein Attribut auf verschiedenen Wegen an eine Klasse vererbt wird.
- Mit Interfaces bekommt Java eine eingeschränkte Form von “mehrfacher Vererbung”, die einfacher zu überblicken ist.

In der Typ-Hierarchie kann eine Klasse mehrere Obertypen haben (eine Klasse und beliebig viele Interfaces). Attribute und Methoden-Implementierungen werden aber nur von der einen Klasse geerbt.



# Trennung von Funktionen einer Klasse

- Ein Interface kann genau die Methoden enthalten, die man für eine gegebene Aufgabe braucht.  
Es ist nicht selten, dass ein Interface nur eine einzige Methode enthält.
- Dann kann man später leichter eine andere Klasse verwenden.  
Beispiel:

- Ein großer Teil eines Programms braucht von der GUI-Klasse “`TextArea`” nur die Methode “`append`”.
- Dann führt man ein Interface ein, das ausschliesslich diese Methode enthält, und eine Subklasse von “`TextArea`”, die dieses Interface implementiert.

Die Methode braucht man dabei nicht neu zu schreiben.

- Man verwendet dann das Interface (und nicht die Klasse) an allen Stellen, wo man nur diese eine Methode braucht.

# Inhalt

## 1 Einführung, Motivation

Vergleich mit abstrakten Klassen, Mehrfache Vererbung

## 2 Syntax, Beispiel

Motivation, Beispiel

Interface-Definition

Definition von Klassen, die Interfaces implementieren

Benutzung von Interface-Typen

## 3 Sub-Interfaces

Erweiterung von Interfaces (Subtyp-Hierarchie)

## Beispiel: Motivation (1)

- Es soll eine Software zur Verwaltung von Abbrennplänen für Feuerwerke erstellt werden.
- Ein Abbrennplan legt fest, welcher Feuerwerksartikel zu welcher Zeit gezündet werden soll.

Zur Vereinfachung nehmen wir an, dass die Artikel immer nur einzeln gezündet werden. Das ist nicht besonders realistisch, oft werden mehrere Artikel gleichzeitig gezündet (zumindest mehrere Stück eines Artikels). Die Information über die gleichzeitige Zündung ist in der Praxis wichtig, weil sie dann einem Zünd-Stromkreis zugeordnet werden können (einem "Kanal").

- Für eine einfache Abbrennplan-Verwaltung werden nur zwei Eigenschaften der Artikel benötigt:
  - Name des Artikels (ein `String`)
  - Brenndauer des Artikels (in Sekunden, ein `int`).

## Beispiel: Motivation (2)

- Die Klasse für die Abbrennpläne soll möglichst wenig Vorgaben über die Klasse(n) für die Feuerwerksartikel machen.
- Eventuell hat jemand schon existierende Klassen für die Verwaltung von Feuerwerksartikeln und möchte unsere Software für die Verwaltung von Abbrennplänen nutzen.
- Da die existierenden Klassen selbst in eine Klassenhierarchie eingebunden sein können (d.h. schon Oberklassen haben), wäre die Anpassung schwierig, wenn die Feuerwerksartikel eine von uns definierte Oberklasse haben müssen.
- Daher beschreiben wir die Anforderungen in einem Interface.
  - Die existierenden Klassen für Feuerwerksartikel können meist relativ einfach so erweitert werden, dass sie auch unser Interface implementieren.

## Beispiel: Motivation (3)

- Hinzu kommt, dass bei dieser Anwendung vermutlich nicht viel an Implementierung von einer Oberklasse “Feuerwerksartikel” übernommen werden könnte.
- Schon der Name eines Feuerwerksartikels ist nicht unbedingt immer einfach ein Attribut:
  - Bei Feuerwerksbomben sollte der Name das Kaliber (Durchmesser) enthalten, z.B. “Trauerweide gold (100mm)”.
  - Weil das Kaliber z.B. auch zur Berechnung des Sicherheitsabstands benötigt wird, wird man es in einem eigenen Attribut (als `int`) speichern.
  - Der Name wird dann bei Bedarf berechnet aus der Effektbezeichnung und dem Kaliber.

# Interface-Definition (1)

- Ein Interface kann Folgendes enthalten:

- Abstrakte Methoden
- Konstanten
- Geschachtelte Klassen und Interfaces

Geschachtelte Klassen und Interfaces können in dieser Vorlesung leider nicht mehr behandelt werden.

- Ein Interface kann nicht enthalten:

- Statische Methoden (die können nicht `abstract` sein).
- Attribute (das wäre schon Implementierung).
- Konstruktoren (Objekte gibt es nur von Klassen).

Eine Klasse kann dann das Interface implementieren, aber man kann nicht direkt Interface-Objekte erzeugen.

## Interface-Definition (2)

```
(1) interface Feuerwerksartikel {  
(2)  
(3)     // Name des Artikels:  
(4)     String name();  
(5)  
(6)     // Brenndauer in Sekunden:  
(7)     int dauer();  
(8)  
(9)     // Konstante fuer Brenndauer,  
(10)    // falls unbekannt:  
(11)    int DAUER_UNBEKANNT = -1;  
(12)  
(13) }
```

## Interface-Definition (3)

- Weil es um die Schnittstelle geht, sind alle Bestandteile eines Interfaces implizit `public`.

Man könnte den Modifier `public` schreiben, es ist aber üblich, dies nicht zu tun.

- Das Interface selbst braucht nicht `public` zu sein, so kann man doch eine Einschränkung auf das Paket bekommen.
- Man läßt auch den Modifier "`abstract`" bei Methoden weg.

Man dürfte ihn hinschreiben, aber er versteht sich bei einem Interface von selbst.

- Und entsprechend "`static final`" bei Konstanten.

Sie sehen dann wie initialisierte Attribute aus. Ein Interface kann aber keine Attribute haben, nur Konstanten.



# Implementierung von Interfaces (1)

- Wenn man eine Klasse `C` mit Oberklasse `O` definieren will, die die Interfaces `I1`, `I2`, `I3` implementiert, schreibt man:

```
class C extends O implements I1, I2, I3 { ... }
```

- Die Teile für Oberklasse und implementierte Interfaces kann man einzeln weglassen, z.B.

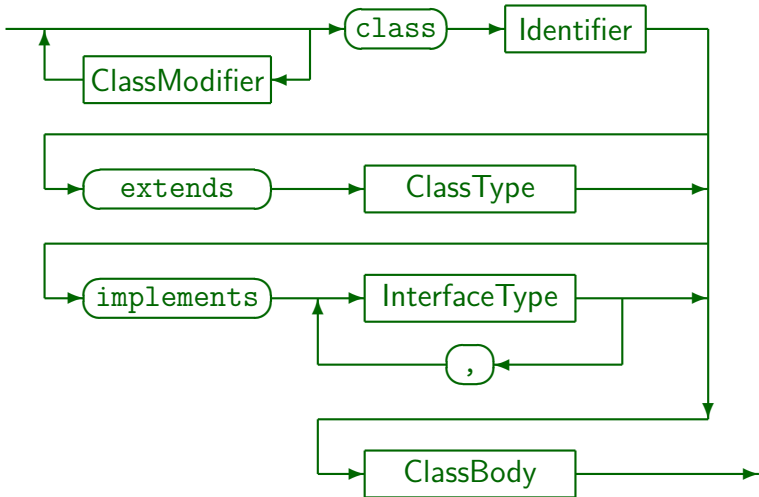
```
class C implements I { ... }
```

- Falls die Klasse nicht `abstract` ist, muss sie alle Methoden aus allen implementierten Interfaces überschreiben und damit eine Implementierung (Methoden-Rumpf) angeben.

Ist die Klasse mit dem Schlüsselwort `abstract` gekennzeichnet, erbt sie die Methoden aus dem Interface auch, braucht sie aber nicht selbst zu überschreiben. Dies muss dann aber in einer Subklasse geschehen, und nur von dieser können Objekte angelegt werden. Auch von einer Oberklasse geerbte Methoden können vom Interface geforderte Methoden implementieren.

## Implementierung von Interfaces (2)

- NormalClassDeclaration (ohne Typ-Parameter):



## Implementierung von Interfaces (3)

- Die implementierten Methoden müssen `public` sein.  
Weil sie im Interface implizit `public` sind, und die Zugriffsrechte in einer Subklasse nicht abgeschwächt werden dürfen (sonst wäre das Substitutionsprinzip verletzt, bzw. der Compiler könnte die Zugriffsrechte nicht überwachen).
- “`@Override`” verwendet man eher nicht.  
In Version 5 war es noch verboten, in Version 7 erlaubt. (Version 6 ?)
- Eine schwierige Situation entsteht, wenn eine Klasse zwei Interfaces implementieren soll, die beide eine Methode mit gleichem Namen fordern.  
Es ist grundsätzlich möglich, dass eine Methode der Klassen gleichzeitig die Implementierungs-Verpflichtungen mehrerer Interfaces erfüllt. Wenn die beiden Methoden unterschiedliche Argument-Typen haben, kann der Compiler sie auch auseinander halten (s. Kap. 14). Bei gleichen Argumenttypen und unterschiedlichem Ergebnistyp oder Zweck gibt es dagegen keine Lösung.

## Implementierung von Interfaces (4)

```
(1) class Fontaene implements Feuerwerksartikel {
(2)     private String name;
(3)     private int hoehe;
(4)     private int dauer;
(5)
(6)     public Fontaene(String n, int h, int d) {
(7)         this.name = n;
(8)         this.hoehe = h;
(9)         this.dauer = d;
(10)    }
(11)
(12)    public String name() { return name; }
(13)
(14)    public int hoehe() { return hoehe; }
(15)
(16)    public int dauer() { return dauer; }
(17) }
```

# Benutzung von Interfaces (1)

- Man kann Variablen (inkl. Parameter, Attribute) und Arrays vom Interface-Typ deklarieren.

Und auch Methoden, die den Interface-Typ zurückgeben. Es ist eben ein Typ.

- Für diese Variablen kann man dann die im Interface deklarierten Methoden aufrufen.
- Im Beispiel werden die Daten des Abbrennplans in zwei parallelen Arrays gehalten:

- `zuendung[i]`: Zeitpunkt der Zündung des  $i$ -ten Artikels.

In Sekunden vom Start des Feuerwerks an gerechnet. In der Realität muss es mindestens bei Musikfeuerwerken etwas genauer sein, z.B. in Zehntel-Sekunden.

- `artikel[i]`: zugehöriger Artikel.

## Benutzung von Interfaces (2)

```
(1) class Abbrennplan {
(2)     // Konstante (Beschraenkung Arraygroesse):
(3)     private final int MAX_KANAELE = 70;
(4)
(5)     // Attribute:
(6)     private int numKanaele;
(7)     private int[] zuendung;
(8)     private Feuerwerksartikel[] artikel;
(9)
(10)    // Konstruktor:
(11)    public Abbrennplan()
(12)        this.numKanaele = 0;
(13)        this.zuendung = new int[MAX_KANAELE];
(14)        this.artikel =
(15)            new Feuerwerksartikel[MAX_KANAELE];
(16)    }
(17)
```

## Benutzung von Interfaces (3)

```
(18) // Neuen Punkt an Abbrennplan anfüegen:  
(19) void punkt(int zeit, Feuerwerksartikel a) {  
(20)     if(numKanaele == MAX_KANAELE) {  
(21)         System.err.println("Plan voll!");  
(22)         return;  
(23)     }  
(24)     zuendung[numKanaele] = zeit;  
(25)     artikel[numKanaele] = a;  
(26)     numKanaele++;  
(27) }  
(28)  
(29) void print() {  
(30)     // Sonderfall "Leere Liste" behandeln:  
(31)     if(numKanaele == 0) {  
(32)         System.out.println("Plan leer.");  
(33)         return;  
(34)     }
```

## Benutzung von Interfaces (4)

```
(35) // Abbrennplan ausdrucken, Forts.:
(36) for(int i = 0; i < numKanaele; i++) {
(37)     int min = zuendung[i] / 60;
(38)     int sec = zuendung[i] % 60;
(39)     String art = artikel[i].name();
(40)     if(min < 10)
(41)         System.out.println(' ');
(42)     System.out.println(min);
(43)     System.out.println(':');
(44)     if(sec < 10)
(45)         System.out.println('0');
(46)     System.out.println(sec);
(47)     System.out.println(' ');
(48)     System.out.println(art);
(49) }
(50) }
(51) }
```



## Benutzung von Interfaces (5)

- Objekte von Klassen, die das Interface implementieren, kann man als Argumente vom Interface-Typ verwenden.

Und entsprechend in Variablen vom Interface-Typ speichern. Genauso, wie man Objekte einer Unterklasse Variablen einer Oberklasse zuweisen kann.

```
(1) public static void main(String[] args) {
(2)     Abbrennplan a = new Abbrennplan();
(3)     Fontaene f1 = new Fontaene(
(4)         "Schweizer Supervulkan II",
(5)         6, 60);
(6)     Feuerwerksartikel f2 = new Fontaene(
(7)         "Barockfontaene", 4, 30);
(8)     a.punkt( 0, f1);
(9)     a.punkt(58, f2);
(10)    a.print();
(11) }
```

# Inhalt

## 1 Einführung, Motivation

Vergleich mit abstrakten Klassen, Mehrfache Vererbung

## 2 Syntax, Beispiel

Motivation, Beispiel

Interface-Definition

Definition von Klassen, die Interfaces implementieren

Benutzung von Interface-Typen

## 3 Sub-Interfaces

Erweiterung von Interfaces (Subtyp-Hierarchie)

# Sub-Interfaces

- So wie Klassen Ober-Klassen haben können, können Interfaces Ober-Interfaces haben.

Allerdings kann ein Interface auch mehrere Ober-Interfaces haben, während eine Klasse nur eine Ober-Klasse haben kann.

- Die Syntax ist

```
interface I extends I1, I2, I3 { ... }
```

- Wenn eine Klasse nun das Interface I implementieren will, muss sie alle Methoden aus I1, I2, I3 implementieren, und natürlich die explizit in { ... } gelisteten.

I ist ein Untertyp von I1 u.s.w., daher ist z.B. die Zuweisung eines I-Objektes (Objekt einer Klasse, die I implementiert) an eine Variable vom Typ I1 möglich.

- Dabei hat sie Zugriff auf alle in diesen Interfaces deklarierten Konstanten.