# Das
# Interface-Konzept

## am Beispiel
## der Sprache Java

*Klaus Kusche, November 2013*

# Inhalt

- Motivation:
  Wozu braucht man Interfaces?
- Interfaces in Java
- Was spricht gegen
  die “große” Lösung?

# Voraussetzungen

Kenntnisse der *Objektorientierung*:

- *Klassen*

  … und deren praktische Anwendung
  zur Strukturierung von Code!

- *Vererbung*
- *Abstrakte* Klassen

# Praktische Erkenntnis

- Klassen beschreiben die

*Funktionalität von Objekten*

- Oft haben Objekte

*ganz verschiedener Klassen*

*idente (Teil-) Funktionalitäten*

# Beispiele (1)

- Funktionalität "**vergleichbar**":

  Die Objekte der Klasse sind

  *größenmäßig vergleichbar*

  (d.h. es gibt so etwas wie ein "**<**")

  Gilt für Zahlen, Strings, Datum & Zeit,
  selbstdefinierte Objekte, ...

  Als "**`Comparable`**" vordefiniert in Java, Bedeutung:
  Es gibt eine *Methode* "**`compareTo(...)`**" ==> -1, 0, 1

# Beispiele (2)

- Funktionalität "**sortierbar**":

  Die Objekte der Klasse enthalten mehrere Werte und bieten eine *Methode* "**`sort()`**" zum Sortieren

- Funktionalität "**durchlaufbar**":

  Die Objekte der Klasse enthalten mehrere Werte und bieten z.B. *Methoden* "**`getFirst()`**" und "**`getNext(…)`**"

  (ähnlich in Java: "**`Enumeration`**" bzw. "**`Iterable`**")

  *Beides* (meist, nicht immer!) gilt für

  *zahlreiche strukturell verschiedene Datenstrukturen*
  (Array, Liste, Baum, …)

# Beispiele (3)

- Funktionalität "**`Runnable`**" (Java):

  Die Objekte der Klasse bieten eine *Methode* "**`run()`**", mit der ein *paralleler Thread* gestartet werden kann.

- Funktionalität "**`Serializable`**" (Java):

  Die Objekte der Klasse sind
  *in einen Byte-Strom verwandelbar*
  (und damit in einer Datei speicherbar,
  via Netz versendbar, ...)

# Gemeinsame Eigenschaften

Derartige Funktionalitäten sind …

- … nach außen durch eine *Schnittstelle* charakterisiert

  = *Vorhandensein bestimmter* ***Methoden***

- … *nicht* an eine bestimmte *Datendarstellung* gebunden
- … *unabhängig* von *anderen Methoden* der Klasse (und deren Implementierung)
- … nur ein *kleiner Teil* der Gesamt-Funktionalität einer Klasse (meist 1 bis 5 Methoden, fast nie mehr!)
- … *erblich* (gelten auch für abgeleitete Klassen)

# Erster Implementierungs-Versuch

Für jede solche Funktionalität:

- Definiere eine

  *gemeinsame, abstrakte Basisklasse*

- Deklariere *alle Klassen* mit dieser Funktionalität als

  *davon abgeleitet*

# Was ist dafür nötig?

- Viele Klassen vereinigen
  *mehrere / viele* solcher Funktionalitäten!

  (z.B. “String”: vergleichbar, durchlaufbar, serialisierbar, ...)

- Ziel von O-O ist u.a.: Doppelten Code vermeiden!

  ==> Die “*Hauptvererbungslinie*” sollte
  *interne Datendarstellung & gemeinsamen Code*
  weitergeben, nicht nur Schnittstellen

  (z.B. “String” ist Erweiterung von “Sequence” oder “Vector”, ...)

==> *Eine Klasse müsste*

***von mehreren Klassen gleichzeitig abgeleitet sein!***

# Allgemeine Mehrfach-Vererbung ...

- ... wird in einigen Programmiersprachen unterstützt
  (vor allem in **C++**: *Keine Interfaces, nur Mehrfach-Vererbung!*)
- ... kann das Problem tatsächlich lösen
- ... wird zur Implementierung solcher Fälle verwendet
  (der *Großteil* aller Mehrfach-Vererbungen
  wird *nur* für solche "Funktionalitäten" verwendet)
- ... kann *viel mehr* als dafür notwendig
- ... ist *extrem kompliziert*
  (konzeptionell und implementierungstechnisch)

  ==> *siehe letztes Kapitel!*

# Java (und C#) will einfach sein!

==> Mehrfach-Vererbung so einschränken, dass sie...

- ... *verständlich* und nachvollziehbar ist
- ... leicht *implementierbar* bleibt
- ... diese Fälle immer noch *komfortabel löst*
  (aber viele ausgefallenere Anforderungen nicht mehr)

==> ***Interfaces*** (deutsch: "*Schnittstellen*") sind

*eine eingeschränkte Form*
*(aber die praktisch wichtigste/häufigste Form!)*
*der Mehrfach-Vererbung*
*nur für gemeinsame Schnittstellen!*

# Interfaces in Java

- Idee und Konzept
- Deklaration
- Verwendung in Klassen
- Verwendung in Deklarationen
- Prüfung zur Laufzeit
- “Leere” Interfaces
- Interface-Hierarchien
- Sonderfälle

# Idee

*Trenne* zwischen

- reiner "**Typ-Vererbung**"

  = ***Interface***
  (nur *Methoden-Deklarationen*)

  ==> *Mehrfach-Vererbung erlaubt!*

- kompletter "**Implementierungs-Vererbung**"

  = ***normale Klasse***
  (mit *Code & Daten*)

  ==> *Nur einfache Vererbung erlaubt!*

# Konzept: Ein Interface...

... ist eine *eingeschränkte* Form einer

***abstrakten*** *Klasse*

==> *Objekte* eines Interfaces sind *nicht direkt erzeugbar*,
nur Objekte davon abgeleiteter Klassen!

==> "Erbt" eine Klasse ein Interface,
so bleibt sie ebenfalls so lange abstrakt, bis sie

für alle Methoden des Interfaces
*Code defininiert* hat!

(d.h. alle Methoden eines Interfaces
müssen *überschrieben* werden!)

# Die Deklaration eines Interfaces...

- ... beginnt mit dem Schlüsselwort

  "**`interface`**" *statt* "**`class`**"

- ... darf *nur* enthalten: Die *Deklarationen*

  - von **`public`**-*Methoden*

  - und von *Konstanten*

- ... darf *keinen Code* enthalten
- ... darf *keine Member-Variablen* enthalten
- ... darf *keinen Konstruktor* / Destruktor enthalten
- ... darf *keine statischen Methoden* enthalten

# Hinweise zur Deklaration

- "**`abstract`**" (beim Interface),

  "**`public`**" (bei allen Deklarationen)

  und "**`static final`**" (bei Konstanten)

  sind *implizit* (weil ohnehin die einzig erlaubte Möglichkeit)

  ==> *Nicht hinschreiben!*

- **Generics** (Typ-Parameter)
  sind *auch bei Interfaces* möglich.

# Deklaration, Beispiel

```
interface Sortable
{
  // sortiere das eigene Objekt
  void sort();
}
```

# Verwendung in Klassen

- **Terminologie:**

  Bei Interfaces heißt es "*implementieren*",
  nicht "*erben*"!

- Daher:

  **`class`** *myClass* **`extends`** *myBaseClass*
  **`implements`** *myInterf1*`,` *myInterf2*`,` … `{` … `}`

- In Java: *Eine* Vaterklasse,
  aber *beliebig viele* Interfaces!

- Implementierte Interfaces
  werden *automatisch weitervererbt*!

# Verwendung in Deklarationen

Interface-Namen sind (wie Klassennamen)

***Typnamen***

==> Können zur ***Deklaration von Objekt-Variablen***
verwendet werden
(im Besonderen auch von Parametern)

==> Die Variable kann dann eine Referenz
auf ein *Objekt einer beliebigen Klasse* enthalten,
die *dieses Interface implementiert*

*Beispiel:*

```
void StartAndJoin(Runnable r) { …
```

# Prüfung zur Laufzeit

Analog zur Prüfung auf Klassen-Zugehörigkeit:

Prüfung, ob ein Objekt
ein bestimmtes *Interface implementiert*:

*obj* **`instanceof`** *interf*

liefert **`true`** / **`false`**

# “Leere” Interfaces

Java erlaubt “*leere*” Interfaces (*ohne* Methoden)
(inzwischen teilweise ersetzt durch Annotationen)

**Sinn:** Indikator für das

*Vorhandensein bestimmter Eigenschaften*

==> Primär zum Test mit “**`instanceof`**”

**Bezeichnung:**
“*Marker Interfaces*” (“*Markierungsschnittstellen*”)

**Beispiele:**
**`Cloneable`** (“*deep Copy*” mittels **`clone`** möglich)
**`Serializable`** (in Byte-Strom verwandelbar)

# Interface-Hierarchien

- Kann man Interfaces auch *voneinander **ableiten***?
- Kann man *bestehende Interfaces*
  um neue Methoden ***erweitern***?

***Ja!***

Aber diesmal mit "**`extends`**", nicht mit "**`implements`**":

**`interface`** *extInterf* **`extends`** *baseInterf* **`{ … }`**

Hier ist sogar *Mehrfach-Vererbung* erlaubt:

… **`extends`** *baseInterf1*`,` *baseInterf2*`,` …

*Beispiel:*

"*Bidirektional durchlaufbar*" = "*durchlaufbar*" + **`getPrev`**

# Sonderfälle (1): Was passiert bei ...

- ... ***selber Methode***
  in *mehreren* implementierten Interfaces?

  ==› *Erlaubt*, bezeichnet *ein und dieselbe Methode*!

- ... ***selber Konstante***
  in mehreren implementierten Interfaces?

  ==› *Fehlermeldung* bei direkter Verwendung!
  (auch bei gleichem Wert!)

  ==› *Erlaubt* mit expliziter Angabe des Interfaces:

  *interf*.*const*

  ==› Verwirrt, am besten *vermeiden*!

# Sonderfälle (2): Was passiert bei ...

- ... ***“Komplexen” Konstanten***
  (Initialwert zur Laufzeit *berechnet*)?

  ==> *Möglich*, aber kompliziert und verwirrend

  ==> Auf *“echte” Konstanten* beschränken!

  ==> *Dynamisch erzeugte Objekte meiden*!

# Probleme Mehrfach-Vererbung (1)

C++: Werden *zwei Methoden* mit *identem Aufruf*

von ***zwei Vaterklassen*** geerbt,

so sind es ***zwei verschiedene Methoden***!
(können auch *verschiedenen Code* haben!)

==> *Welche* wird aufgerufen???

==> Praktische Faustregel: *Muss überschrieben werden!*

Wenn nicht überschrieben: Aufruf von außen *liefert Fehler!*
(d.h. die Klasse *verletzt ihre Schnittstelle*!)

==> Es gibt *kein* "**super**" in C++!

Bei *jedem* Vaterklassen-Methoden-Aufruf muss
die gewünschte *Vaterklasse explizit angegeben* werden!

# Probleme Mehrfach-Vererbung (2)

Analog für

von ***zwei Vaterklassen*** geerbte
***Member gleichen Namens***:

Sind ***zwei verschiedene Member***!

==> “Normaler” Zugriff
- … ist nicht eindeutig
- … liefert daher einen Fehler!

==> Bei jedem un-eindeutigen Member-Zugriff muss die *Vaterklasse explizit dazugeschrieben* werden!

# Probleme Mehrfach-Vererbung (3)

Die "***diamantene Vererbung***":

Dieselbe "*Großvater-Klasse*"
wird *über zwei verschiedene Vaterklassen* geerbt.

==> Sind ihre Member jetzt ***einfach oder doppelt***
in jedem abgeleiteten Objekt gespeichert???
(in C++: *Per default doppelt*, mit **`virtual`** einfach)

==> Wenn *doppelt*:
- Bei Zugriff auf *geerbte Member*
- Bei *Typumwandlung* auf den Großvater-Typ

***Welches der beiden?***
*(explizite Angabe nötig!)*

# Probleme Mehrfach-Vererbung (4)

Implementierungstechnische Probleme:

*Member mehrerer Vaterklassen müssen*
*in einem abgeleiteten Objekt Speicherplatz bekommen*

==> Geerbte Member müssen
*frisch angeordnet* werden

==> Dasselbe Member steht innerhalb des Objektes
in verschiedenen Klassen
*an verschiedenen Offsets*!

==> Der Code zum Zugriff wird viel komplizierter:
Er muss zuerst die *Objektklasse auswerten*!

# “The end”

*Fragen?*