DHBW
Duale Hochschule
Baden-Württemberg

AIRBUS
DEFENCE & SPACE

Ergänzungen zu Datentypen&Wildcards

Lukas Abelt
lukas.abelt@airbus.com

DHBW Ravensburg
Wirtschaftsinformatik

Ravensburg
3. April 2019

Outline

Inhalt

Mehrfachvererbung oder Interfaces?

Was ist besser?

Was kann Mehrfachvererbung, das Interfaces nicht können?

- Kurzgesagt: Funktional lässt sich beides äquivalent nutzen

Mehrfachvererbung oder Interfaces?

Was ist besser?

Was kann Mehrfachvererbung, das Interfaces nicht können?

- Kurzgesagt: Funktional lässt sich beides äquivalent nutzen
- Der Unterschied liegt vielmehr in formaler Betrachtung der OOP

Mehrfachvererbung oder Interfaces?

Was ist besser?

Was kann Mehrfachvererbung, das Interfaces nicht können?

- Kurzgesagt: Funktional lässt sich beides äquivalent nutzen
- Der Unterschied liegt vielmehr in formaler Betrachtung der OOP
- Mehrfachvererbung birgt aber viele Risiken

Mehrfachvererbung oder Interfaces?

Was ist besser?

Was kann Mehrfachvererbung, das Interfaces nicht können?

- Kurzgesagt: Funktional lässt sich beides äquivalent nutzen
- Der Unterschied liegt vielmehr in formaler Betrachtung der OOP
- Mehrfachvererbung birgt aber viele Risiken
  - Diamond-Problem

Mehrfachvererbung oder Interfaces?

Was ist besser?

Was kann Mehrfachvererbung, das Interfaces nicht können?

- Kurzgesagt: Funktional lässt sich beides äquivalent nutzen
- Der Unterschied liegt vielmehr in formaler Betrachtung der OOP
- Mehrfachvererbung birgt aber viele Risiken
  - Diamond-Problem
  - Komplexe (=undurchsichtige) Vererbungshierarchien

Mehrfachvererbung oder Interfaces?

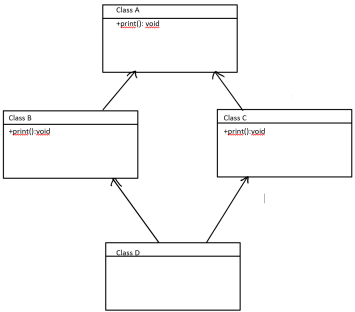
Was ist besser?

Was kann Mehrfachvererbung, das Interfaces nicht können?

- Kurzgesagt: Funktional lässt sich beides äquivalent nutzen
- Der Unterschied liegt vielmehr in formaler Betrachtung der OOP
- Mehrfachvererbung birgt aber viele Risiken
  - Diamond-Problem
  - Komplexe (=undurchsichtige) Vererbungshierarchien
  - Falsche Verwendung

Diamond-Problem

Visuelle Darstellung



Diamond-Problem

Kurze Erklärung

- Tritt auf, wenn die „Großvaterklasse“ von zwei Basisklassen gleich ist
- Führt zu einer Mehrdeutigkeit in Methoden und Member Variablen
- Diese wird jedoch in den meisten Sprachen durch den Compiler abgefangen
- Diamond Problem *kann* auch in Java auftreten
  - Weil Default-Implementierungen von Interfaces erlaubt sind
  - Werden auch durch Compiler erkannt
  - Entwickler **muss** Klassenspezifische Implementierung erstellen

Diamond-Problem

Am Java Beispiel

```
1  public interface A{
2      public void say(){
3          print("I am A!");
4      }
5  }
6
7  public interface B{
8      public void say(){
9          print("I am B");
10     }
11 }
12
13 public class C implements A,B{
14     //Compilerfehler, weil say() mehrdeutig ist!
15 }
```

Das Problem der Mehrfachvererbung

- Oft falsch verwendet
- Besonders bei Anfängern
- Wo keine Mehrfachvererbung ist, kann sie nicht falsch verwendet werden
  $\Rightarrow$ Der Java Ansatz
- Statt Mehrfachvererbung hilft oft:
  - Assoziation
  - Aggregation
  - Komposition
  - Delegation

Assoziation

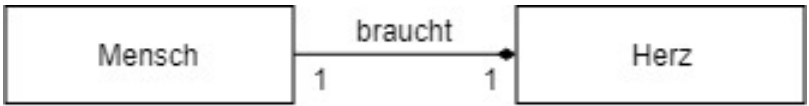
- Beziehung zwischen zwei Objekten
- Es besteht jedoch keine Abhängigkeit
- Beide Objekte können unabhängig voneinander existieren
- Beispiel: Relation zwischen Sprecher und Zuhörer

Aggregation

- Sonderfall der Assoziation
- Hier eine unidirektionale Beziehung („**benötigt ein**“ Beziehung)
- Kindobjekt kann unabhängig von Elternobjekt existieren
- Aber Elternobjekt nicht ohne Kind
- Beispiel: Auto und Räder
  - Räder können auch ohne Auto sinnvoll sein
  - Autos ohne Räder eher weniger...

Komposition

- Sonderfall der Aggregation
- Strenge Abhängigkeit zwischen kind- und Elternobjekt
- Beide können nicht unabhängig voneinander existieren
- Beispiel: Mensch und Herz



Delegation

- Ist ein alternativer Ansatz zur Vererbungshierarchien
- Hierbei wird ein Objekt als Instanzvariable genutzt
- Funktionsaufrufe werden „durchgereicht“ an dieses
- Vorteil gegenüber Vererbung/Interfaces:
  - Nicht alle Methoden müssen nach außen sichtbar gemacht werden
  - Für Delegate kann spezielle Unterklasse genutzt werden

Delegation

Codebeispiel

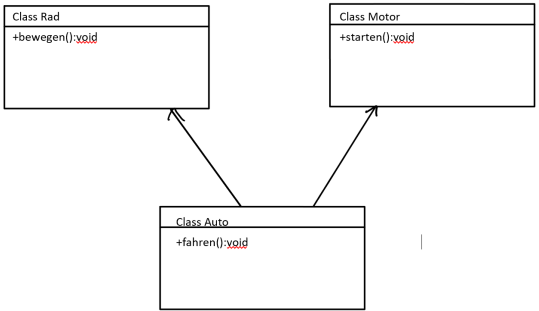
```
1  public class Delegate(
2      public void print(){
3          System.out.println("I am a Delegate!");
4      }
5      public void sayHello(){
6          System.out.println("Hello!");
7      }
8  )
9
10 public Class Example{
11     private Delegate del = new Delegate();
12
13     public void print(){
14         del.print();
15     }
16 }
```

Schlechte Mehrfachvererbung

- Wir haben eine Klasse `Rad` und `Motor`
- Man möchte nun eine `Auto` Klasse implementieren
- Ein unerfahrener Nutzer denkt:
  - Ein Auto braucht Fähigkeiten vom `Rad`
  - ...und vom `Motor`
  - und entwirft folgende Klasse:

Schlechte Mehrfachvererbung

Am Beispiel



Gute Verwendung von Mehrfachvererbung

- Sind relativ selten
- Wenige reale Anwendungsfälle
- Ein Beispiel: „Mix-in Klassen“
  - Nutzt im Grunde Mehrfache Verkettung von Template Klassen (Generics)
  - Link für mehr Info(C++): Mix-in (Siehe [1])

Der Zusammenhang zu Interfaces

- Laut formalen OOP:

Der Zusammenhang zu Interfaces

- Laut formalen OOP:
  - Definiert *Vererbung* eine **„ist ein“** Beziehung

Der Zusammenhang zu Interfaces

- Laut formalen OOP:
  - Definiert *Vererbung* eine **„ist ein“** Beziehung
  - Und ein *Interface* eine **„Hat die Fähigkeit“** Beziehung

Der Zusammenhang zu Interfaces

- Laut formalen OOP:
  - Definiert *Vererbung* eine **„ist ein“** Beziehung
  - Und ein *Interface* eine **„Hat die Fähigkeit“** Beziehung
- Heißt, formal „darf“ man ein Interface nicht als Mehrfachvererbung betrachtet werden

Der Zusammenhang zu Interfaces

- Laut formalen OOP:
  - Definiert *Vererbung* eine **„ist ein“** Beziehung
  - Und ein *Interface* eine **„Hat die Fähigkeit“** Beziehung
- Heißt, formal „darf“ man ein Interface nicht als Mehrfachvererbung betrachtet werden
- ...ist aber irgendwie nicht weit davon weg

Der Zusammenhang zu Interfaces

- Laut formalen OOP:
  - Definiert *Vererbung* eine **„ist ein“** Beziehung
  - Und ein *Interface* eine **„Hat die Fähigkeit“** Beziehung
- Heißt, formal „darf“ man ein Interface nicht als Mehrfachvererbung betrachtet werden
- ...ist aber irgendwie nicht weit davon weg
- Verstößt Java damit gegen die Prinzipien der OOP?

Der Zusammenhang zu Interfaces

- Laut formalen OOP:
  - Definiert *Vererbung* eine **„ist ein“** Beziehung
  - Und ein *Interface* eine **„Hat die Fähigkeit“** Beziehung
- Heißt, formal „darf“ man ein Interface nicht als Mehrfachvererbung betrachtet werden
- ...ist aber irgendwie nicht weit davon weg
- Verstößt Java damit gegen die Prinzipien der OOP?
  - Ganz klares: „Jain“

Der Zusammenhang zu Interfaces

- Laut formalen OOP:
  - Definiert *Vererbung* eine **„ist ein“** Beziehung
  - Und ein *Interface* eine **„Hat die Fähigkeit“** Beziehung
- Heißt, formal „darf“ man ein Interface nicht als Mehrfachvererbung betrachtet werden
- ...ist aber irgendwie nicht weit davon weg
- Verstößt Java damit gegen die Prinzipien der OOP?
  - Ganz klares: „Jain“
  - Kaum eine Sprache erfüllt alle Anforderungen an OOP

„Richtige“ Eigenschaften von Interfaces

- Rein abstrakte Definition von Schnittstellen
- Keine Attribute
- Keine Assoziationen
- Definiert eine Menge von Operationen $\Rightarrow$ Eigentlich ohne Implementierung

Inhalt

Generics

...und die Vererbung nochmal

- Hauptgrund für Wildcards: Die „eigenartige“ Vererbung bei generischen Klassen:
  - Wenn `Integer` von `Number` erbt...
  - Warum dann nicht auch `List<Integer>` von `List<Number>`?
- Würde dazu führen, dass man inkompatible Typen zuweisen kann
  - Zum Beispiel einen Double in eine `List<Integer>`

Generic-Vererbung

Ein Gegenbeispiel

```
1 List<Number> ln = new List<Number>();
2 List<Integer> li = {7,12,42,46};
3 ln = li;     //li wird als Referenz übergeben!
4 //Hier würde man li einen Double hinzufügen:
5 ln.add(new Double(2.7182818284590));
```

Wildcards

Und deshalb braucht man sie

- Durch „ungewöhnliche“ Vererbungshierarchie von Generics benötigt
- Definieren einen unbekannten Typ
  - Der jedoch eingeschränkt werden kann
  - ...Wie bereits letztes mal besprochen
- Verwendung als Argument von Funktionen
- Oder auch als Rückgabewert (Eher vermeiden)

Wildcards

Verwendung

```
1 void someFunc(List<?> in){ /* ... */ }    //OK
2 List<?> func(){ /* ... */ }               //OK
3
4 List<?> l1;    //OK
5 List<?> l2 = new ArrayList<>();    //OK
6 List<?> l3 = new ArrayList<?>();   //Fehler
```

Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben

Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben
- Dadurch gemeinsame Funktionalitäten, egal was übergeben wird

Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben
- Dadurch gemeinsame Funktionalitäten, egal was übergeben wird
- <?> ist formal gesehen ein `<? extends Object>`

Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben
- Dadurch gemeinsame Funktionalitäten, egal was übergeben wird
- <?> ist formal gesehen ein `<? extends Object>`
- Können jedoch nur lesend verwendet werden

Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben
- Dadurch gemeinsame Funktionalitäten, egal was übergeben wird
- <?> ist formal gesehen ein `<? extends Object>`
- Können jedoch nur lesend verwendet werden
- Warum?

Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben
- Dadurch gemeinsame Funktionalitäten, egal was übergeben wird
- <?> ist formal gesehen ein `<? extends Object>`
- Können jedoch nur lesend verwendet werden
- Warum?
  - Beispiel `add(T)` auf eine `List<? extends Number>`

Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben
- Dadurch gemeinsame Funktionalitäten, egal was übergeben wird
- <?> ist formal gesehen ein `<? extends Object>`
- Können jedoch nur lesend verwendet werden
- Warum?
  - Beispiel `add(T)` auf eine `List<? extends Number>`
  - Es gibt keinen Typ `T`, der auf alle Varianten von `<? extends Number>` passt

Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben
- Dadurch gemeinsame Funktionalitäten, egal was übergeben wird
- <?> ist formal gesehen ein `<? extends Object>`
- Können jedoch nur lesend verwendet werden
- Warum?
  - Beispiel `add(T)` auf eine `List<? extends Number>`
  - Es gibt keinen Typ `T`, der auf alle Varianten von `<? extends Number>` passt
  - Kurz: Wir können nicht sicherstellen, ob der Typ den wir schreiben überhaupt mit der speziellen Instanz kompatibel ist!

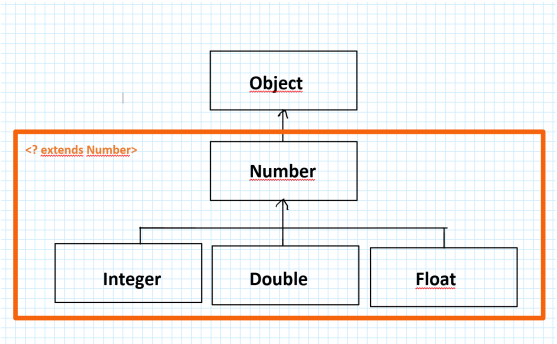
Upper-Bounded Wildcards

Hoffentlich etwas klarer

- Einschränkung des Generics nach oben
- Dadurch gemeinsame Funktionalitäten, egal was übergeben wird
- <?> ist formal gesehen ein `<? extends Object>`
- Können jedoch nur lesend verwendet werden
- Warum?
  - Beispiel `add(T)` auf eine `List<? extends Number>`
  - Es gibt keinen Typ `T`, der auf alle Varianten von `<? extends Number>` passt
  - Kurz: Wir können nicht sicherstellen, ob der Typ den wir schreiben überhaupt mit der speziellen Instanz kompatibel ist!
  - Einzige Ausnahme: `null`

Upper-Bounded Wildcards

Visualisiert



Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten

Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten
- Umgekehrter Effekt zu Upper-Bound Wildcards:

Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten
- Umgekehrter Effekt zu Upper-Bound Wildcards:
  - Wir keinen lesenden Zugriff (Außer über Methoden von `Object`)

Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten
- Umgekehrter Effekt zu Upper-Bound Wildcards:
  - Wir keinen lesenden Zugriff (Außer über Methoden von `Object`)
  - Dafür aber schreibeneden

Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten
- Umgekehrter Effekt zu Upper-Bound Wildcards:
  - Wir keinen lesenden Zugriff (Außer über Methoden von `Object`)
  - Dafür aber schreibeneden
  - Dadurch keine gemeinsamen Schnittstellen

Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten
- Umgekehrter Effekt zu Upper-Bound Wildcards:
  - Wir keinen lesenden Zugriff (Außer über Methoden von `Object`)
  - Dafür aber schreibeneden
  - Dadurch keine gemeinsamen Schnittstellen
  - Aber gemeinsamer Typ, der auf alle möglichen übergebenen Klassen passt

Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten
- Umgekehrter Effekt zu Upper-Bound Wildcards:
  - Wir keinen lesenden Zugriff (Außer über Methoden von `Object`)
  - Dafür aber schreibeneden
  - Dadurch keine gemeinsamen Schnittstellen
  - Aber gemeinsamer Typ, der auf alle möglichen übergebenen Klassen passt
- Frage: Welches ist dieser gemeinsame Typ? (Zum Beispiel für `List<? super Number>`)

Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten
- Umgekehrter Effekt zu Upper-Bound Wildcards:
  - Wir keinen lesenden Zugriff (Außer über Methoden von `Object`)
  - Dafür aber schreibeneden
  - Dadurch keine gemeinsamen Schnittstellen
  - Aber gemeinsamer Typ, der auf alle möglichen übergebenen Klassen passt
- Frage: Welches ist dieser gemeinsame Typ? (Zum Beispiel für `List<? super Number>`)
  - Immer „unterste“ Typ (Im Sinne der Vererbung)

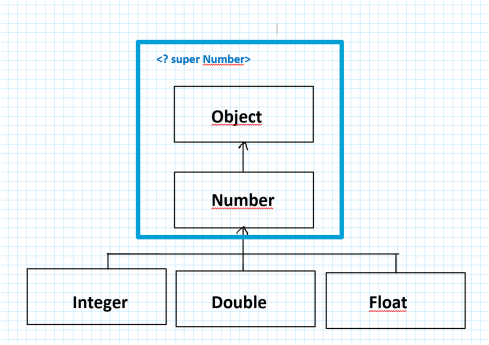
Lower-Bounded Wildcards

Nochmal erklärt

- Beschränken die Klasse nach unten
- Umgekehrter Effekt zu Upper-Bound Wildcards:
  - Wir keinen lesenden Zugriff (Außer über Methoden von `Object`)
  - Dafür aber schreibeneden
  - Dadurch keine gemeinsamen Schnittstellen
  - Aber gemeinsamer Typ, der auf alle möglichen übergebenen Klassen passt
- Frage: Welches ist dieser gemeinsame Typ? (Zum Beispiel für `List<? super Number>`)
  - Immer „unterste“ Typ (Im Sinne der Vererbung)
  - Durch die **„ist ein“** Beziehung passt dieser auf alle anderen Klassen

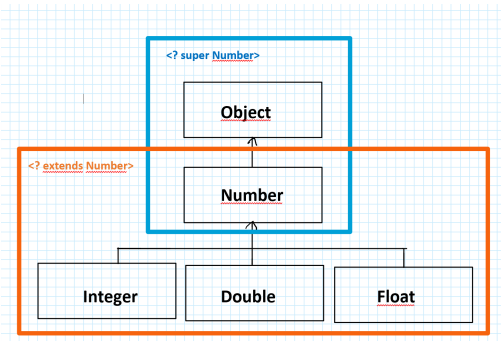
Lower-Bounded Wildcards

Visualisiert



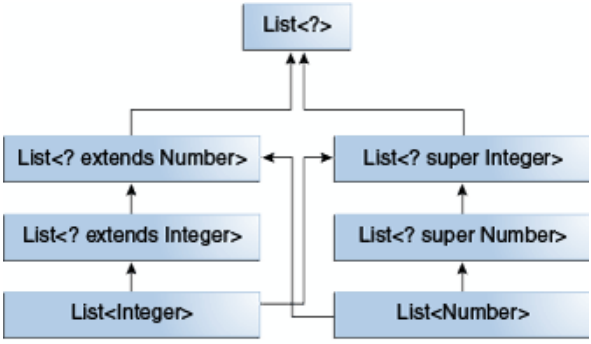
Bounded Wildcards

Nochmal in der Übersicht



Wildcards

Und deren Vererbung



Quellen I

[1] daniel.paull. *C++ Mixins*. 2011. URL: http://www.thinkbottomup.com.au/site/blog/C%20%20_Mixins_-_Reuse_through_inheritance_is_good (besucht am 31. 04. 2019).

Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt