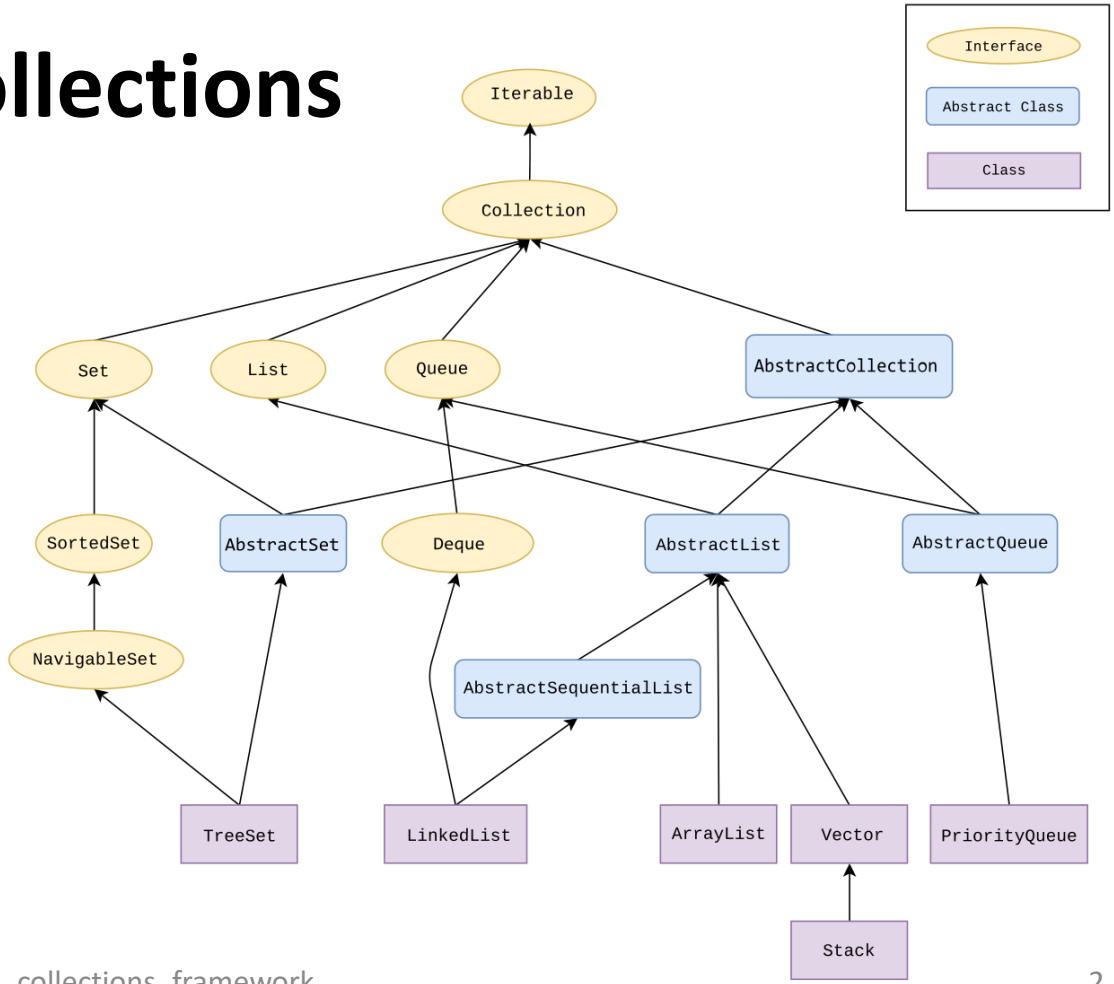
252-0027
Einführung in die Programmierung

13. Interfaces

Manuela Fischer, Malte Schwerhoff

Departement Informatik
ETH Zürich

Motivation: Collections



Grafik von https://en.wikipedia.org/wiki/Java_collections_framework

Vererbung vs abstrakte Klassen vs Interfaces

	Superklassen	Abstrakte Klassen	Interfaces
Implementation	vollständig vererbt	teilweise vererbt (Code mit Lücken)	Interfaces haben keine Implementation*
Attribute	vererbt	vererbt	Interfaces haben keine Attribute*
Instanzen	ja	nein	nein
Subtyping	ja, Subtyp ist ein Spezialfall	ja, Subtyp ist ein konkreter Fall	ja, Subtyp kann, was Interface verspricht

* Es gibt Ausnahmen, siehe spätere Folien.

Interface («Schnittstelle») intuitiv

Interface in Java: Vorschrift für gemeinsames Verhalten

- Menge von abstrakten Methoden (ohne Implementation)
  - Versprechen (wie ein Vertrag), dass Methoden existieren müssen; Implementation können sich aber komplett unterscheiden
  - Anders formuliert: Arbeit muss gemacht werden, egal wie…
    - `Savable`: muss Methode `save` haben (mit der man speichern kann)
    - `Comparable`: muss `compareTo`-Methode haben (mit der man vergleichen kann)
- Keine Attribute
  - Interne Repräsentation/Zustand spielt keine Rolle

Interface deklarieren

Interface-Deklaration («**interface declaration**») enthält abstrakte Methoden

- Keine* Implementation, nur Signatur + Rückgabetyp
- Methoden immer `public` – kann, aber muss nicht, explizit angegeben werden
- Attribute müssen `final` und `public` sein (→ i.d.R. Konstanten)

```
public interface InterfaceName {
  public type methodName1(type name, ..., type name);
  public type methodName2(type name, ..., type name);
  ...
}
```

Beliebige Sichtbarkeit des Interface möglich, i.d.R. aber `public`

Nur Deklaration, keine Implementation, d.h. abstrakte Methoden

* Seit Java 8 sind auch statische Methoden mit Code erlaubt. Nicht relevant für uns.

Interface verwenden (=implementieren)

- Klasse kann **Interface implementieren** («**implement an interface**»)
  - Deklaration mit `implements`
  - Muss Implementation für alle Methoden des Interface angeben
    (sonst Compilerfehler – oder die Klasse ist `abstract`)
  - Alle Klassen, die ein Interface implementieren, haben diese Methoden gemeinsam

```
class ClassName implements InterfaceName {
  public type methodName1(type name, ..., type name) { ... }
  public type methodName2(type name, ..., type name) { ... }
  ...
}
```

Konkrete Implementationen der Interface-Methoden

Referenzvariablen auf Interface

```
class ClassName implements InterfaceName {
  public type methodName1(type name, ..., type name) { ... }
  public type methodName2(type name, ..., type name) { ... }
  ...
}
```

- Interface `InterfaceName` definiert einen **neuen Typ**
  - Interface ist abstrakt, es kann also keine Instanzen von diesem Typ geben
  - Referenzvariablen können auf Instanzen von nicht-abstrakten Klassen zeigen, die das Interface implementieren
- **Subtyping:** `ClassName` ist ein `InterfaceName` (ist-ein-Beziehung!)
  - Wo ein `InterfaceName` gefragt ist, kann ein `ClassName`-Objekt verwendet werden
  - Verhalten dann wie aus Vererbung bekannt (Dynamic Binding)

Beispiel: Liste

```
interface List {
  ...
  public int get(int i)
  ...
}
```

```
class LinkedList implements List {
  ...
  public int get(int i) { ... }
  ...
}
```

```
class ArrayList implements List {
  ...
  public int get(int i) { ... }
  ...
}
```

- Jedes Interface (hier `List`) definiert einen **neuen Typ**
  - Interface ist abstrakt, es kann also keine Instanzen von diesem Typ geben
- Jede Interface-implementierende Klasse (hier z.B. `ArrayList`) ist ein **Subtyp** des Interfaces (hier `List`)
- Verhalten dann wie aus Vererbung bekannt (Dynamic Binding)
  - Wo ein Interface-Typ gefragt ist, kann ein Subtyp-Objekt verwendet werden

```
void fill(List data) { ... }
```

```
fill(new ArrayList()) { ... }
```

```
fill(new LinkedList()) { ... }
```

Intuition: Interface als Garantie

- Interfaces sind eine Art Versprechen/Garantie:
  Auf Interface-Variable können alle Interface-Methoden aufgerufen werden!
  - Variable mit statischem Typ des Interface kann nur auf Instanzen von Klassen zeigen, die das Interface implementieren.
  - Jede Klasse, die das Interface implementiert, muss die Methoden des Interface implementieren.

```
interface I {
  public void f();
  public void g();
}
```

```
class A implements I {
  public void f() { ... }
  public void g() { ... }
}
```

```
class B implements I {
  public void f() { ... }
  public void g() { ... }
}
```

```
I i = ...;
i.f();
i.g();
```

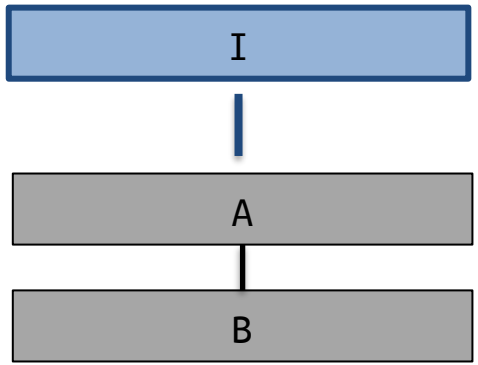
Wir dürfen davon ausgehen, dass Interface-Methoden `f()` und `g()` für Variable `i` existieren, unabhängig vom dynamischen Typ von `i`!

Interfaces und Vererbung

```
public interface I {
  ...
}
```

```
class A implements I {
  ...
}
```

```
class B extends A {
  // implicit declaration
}
```




```
class B extends A implements I {
  // explicit declaration
}
```

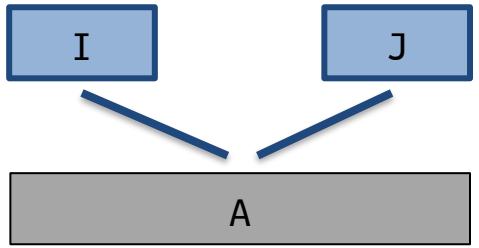
- Subklassen von Klassen, die ein Interface implementieren, implementieren dieses Interface auch (kann deklariert werden, muss aber nicht).
- Auch Interface-Methoden können in Subklasse überschrieben werden.

Mehrere Interfaces implementieren

```
public interface I {
  public void f();
  public void g();
}
```

```
public interface J {
  public void f();
  public void h();
}
```

```
class A implements I, J {
  public void f() { ... }
  public void g() { ... }
  public void h() { ... }
}
```



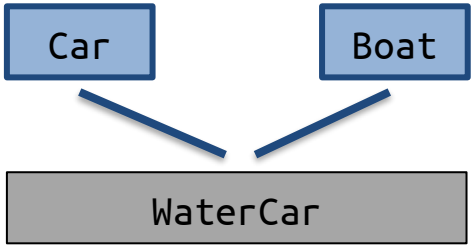
- Klasse kann beliebig viele Interfaces implementieren
- Muss Vereinigung aller Interface-Methoden implementieren
- Falls Methode mehrmals vorkommt, braucht es nur eine Implementation

Beispiel

```
public interface Car {
  public void start();
  public void stop();
  public void cruise();
}
```

```
public interface Boat {
  public void start();
  public void stop();
  public void swim();
}
```

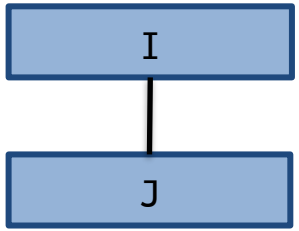
```
class WaterCar implements Car, Boat {
  public void start() { ... }
  public void stop() { ... }
  public void swim() { ... }
  public void cruise() { ... }
}
```



Erweiterung von Interfaces

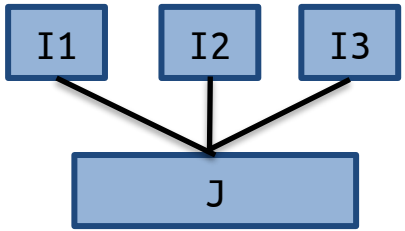
```
public interface I {
  public void f();
  public void g();
}
```

```
public interface J extends I {
  public void h();
  public void i();
}
```



- Interface kann ein Interface erweitern: Keyword `extends`
- Erweiterndes Interface (hier `J`) ist Subtyp vom erweiterten Interface (hier `I`)
- Erweitern des Interfaces durch Hinzufügen zusätzlicher (abstrakter) Methoden
- Klasse, die Subinterface (hier `J`) erweitert, muss Methoden aus beiden Interfaces (hier `I` und `J`) implementieren

- Interface kann auch mehrere Interfaces erweitern mit `public interface J extends I1, I2, I3`



Beispiel