Java: Interfaces (Schnittstellen)

Ziele

- Grundidee und Deklaration eines Interfaces verstehen können.
- Implementierung und Anwendung eines Interfaces verstehen und beherrschen können.
- Lokale und anonyme Klassen effizient einsetzen können.
- Die Konzepte Default-Implementierung und Delegierung an die Default-Implementierung verstehen und anwenden können.

Deklaration

```
public interface Groesse
{
  long getLaenge();
  long getBreite();
  long getHoehe();
}
```

Enthält ausschließlich *abstrakte* Methoden und Konstanten.

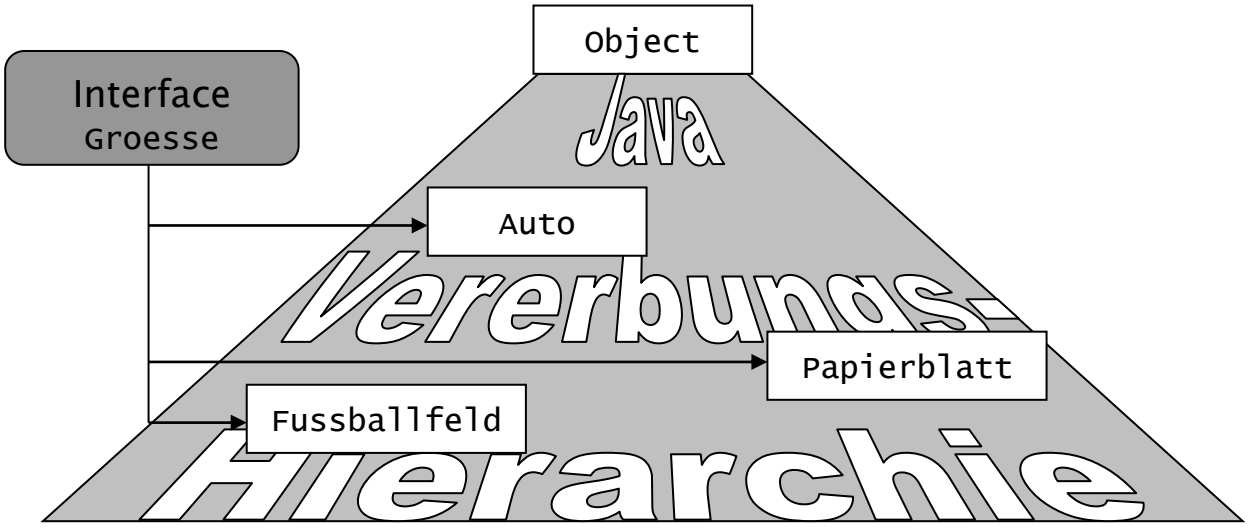
Konstanten werden in Klasse als Klassenkonstanten eingefügt.

Klasse muss als abstract deklariert werden, wenn sie nicht alle Methoden des Interfaces implementiert.

Keine statischen Methoden erlaubt.

Implementierung

```
public class Auto extends Object
  implements Groesse
{
  …
  public long getLaenge() {
    return …
  }
  public long getHoehe() {
    return this.hoehe;
  }
  public long getBreite() {
    return this.breite;
  }
  public int getPs() { … }
}
```



Autos, Fußballfelder und Papierblätter haben nun eine Länge, Breite und Höhe, die nicht direkt in der Vererbungshierarchie abgebildet sind.

Interfaces werden verwendet, um Eigenschaften auszudrücken, die auf Klassen aus unterschiedlichen Klassenhierarchien zutreffen.

Verwendung

```
Groesse g = new Auto();
public long grundflaeche(Groesse g)
System.out.println(g.getPS());
```

- Interface kann zur Deklaration von lokalen Variablen, Membervariablen und Methodenparametern verwendet werden.
- Interface-Variable ist kompatibel zu allen Objekten deren Klassen dieses Interface implementieren.
- Über eine Interface-Variable können nicht Methoden aufgerufen werden, die nur zur *Klasse* gehören.

```
public class Test1
{
  public static long grundflaeche(Groesse g) {
    return (g.getLaenge() * g.getBreite());
  }

  public static void main(String[] args) {
    Auto auto = new Auto();
    System.out.println(auto.getLaenge());
    Fussballfeld fussballfeld = new Fussballfeld();
    System.out.println(fussballfeld.getLaenge());
    Papierblatt papierblatt = new Papierblatt();
    System.out.println(papierblatt.getLaenge());

    Groesse g = new Auto();
    System.out.println(g.getLaenge());

    System.out.println(grundflaeche(auto));
    System.out.println(grundflaeche(fussballfeld));
    System.out.println(grundflaeche(papierblatt));
  }
}
```

oder allgemeiner

```
  public static long grundflaeche(Object o) {
    long ret = 0;
    if (o instanceof Groesse) {
      Groesse g = (Groesse)o;
      ret = g.getLaenge() * g.getBreite();
    }
    return ret;
  }
```

- Klassen können mehrere Schnittstellen gleichzeitig implementieren, dadurch wird *„Mehrfachvererbung“* realisiert.
- Eine Unterklasse kann auch die Implementierung der Schnittstelle Ihrer Oberklasse überschreiben.
- Eine Schnittstelle kann eine oder mehrere Oberschnittstellen besitzen.

```
public interface A
{
  void operation1();
  void operation2();
{
public interface B extends A
{
  void operation3();
}
```

```
public class Klasse
  extends Object
  implements B, Groesse
{
  public void operation1() { … }
  public void operation2() { … }
  public void operation3() { … }
  public int getLaenge() { … }
  public int getHoehe() { … }
  public int getBreite(){ … }
}

public class UnterKlasse
  extends Klasse
{
  public int getHoehe() { … }
}
```

`UnterKlasse` kompatibel zu *Interface* `A`, `B` und `Groesse` und
zu *Klasse* `Unterklasse`, `Klasse` und `Object`

`B b = new UnterKlasse();` oder `Klasse k = new UnterKlasse();`

Anwendung von Interfaces am Beispiel von Collections

Collections sind Datenstrukturen, welche eine Menge von Daten aufnehmen und verarbeiten können.

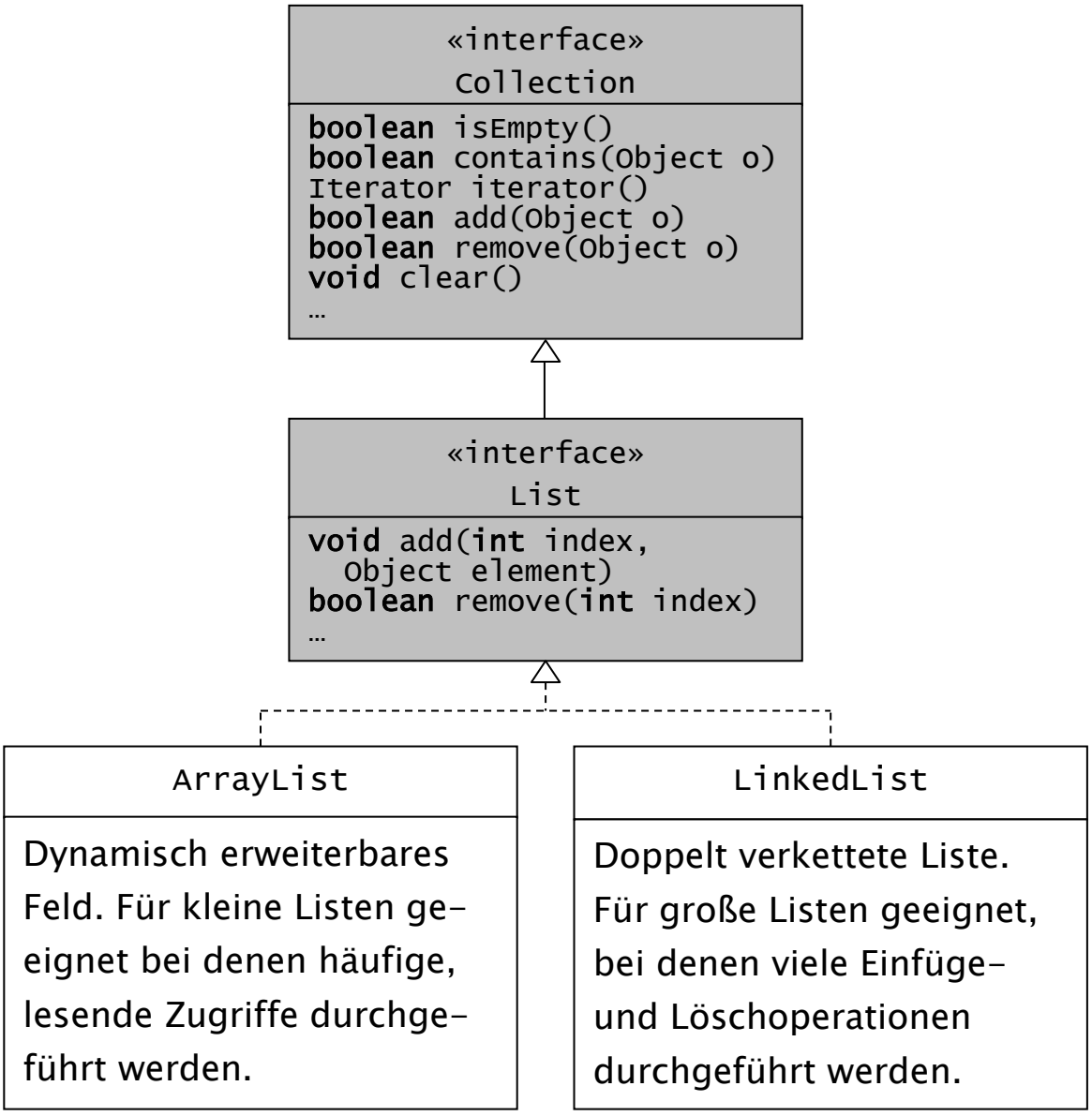
Collections kapseln die Daten und erlauben den Zugriff darauf nur über vorgegebene Methoden.

Beispielklassen in JDK 1.1 (in `java.util`):

`Vector`, `Stack`, `Hashtable`, `BitSet`

Interfaces in JDK 1.2 (in `java.util`):

`List`, `Set`, `Map` (speichert Objektpaare)



Collections selbstgemacht am Beispiel von `MeineListe`

```
public interface MeineListe
{
  // Fügt am Beginn der Liste ein neues Objekt ein
  boolean einfuegenErstesElement(Object element);

  // Löscht das erste Element der Liste
  boolean loeschenErstesElement();

  // Kontrolliert ob Liste leer ist
  boolean istLeer();

  // Löscht die gesamte Liste
  void leeren();

  // Liefert einen Iterator zurück, welcher es erlaubt die Elemente
  // der Liste nacheinander zu bearbeiten
  MeinIterator elemente();
}
```

```
public interface MeinIterator
{
  // Kontrolliert ob ein nächstes Element in der Liste vorhanden ist
  boolean hatNaechstesElement();

  // Liefert das nächste Element in der Liste zurück
  Object naechstesElement();

  // Fügt nach dem Element auf dem Iterator zeigt das übergebene
  // Element ein
  boolean einfuegenElement(Object element);
}
```

So werden Iteratoren angewendet:

```
for (MeinIterator i = l.elemente(); i.hatNaechstesElement(); )
  System.out.println(i.naechstesElement().toString());
```

Die Default-Implementierung eines Interfaces

Diese implementiert alle in einem Interface deklarierten Methoden und stellt eine voll funktionstüchtige Klasse bereit.

Vorteile

- Wird eine Klasse von der Default-Implementierung abgeleitet, so erbt sie bereits einen Teil der sonst manuell zu implementierenden Funktionalität.
- Nützt zur Dokumentation und stellt Referenzimplementierung dar.

```
public class MeineDefaultListe implements MeineListe
{
  private ListenElement erstesElem = null;

  class ListenElement extends Object
  {
    private Object element = null;
    private ListenElement naechstesElem = null;

    private ListenElement(
      Object element, ListenElement naechstesElem) {
      this.element = element;
      this.naechstesElem = naechstesElem;
    }
  }
```

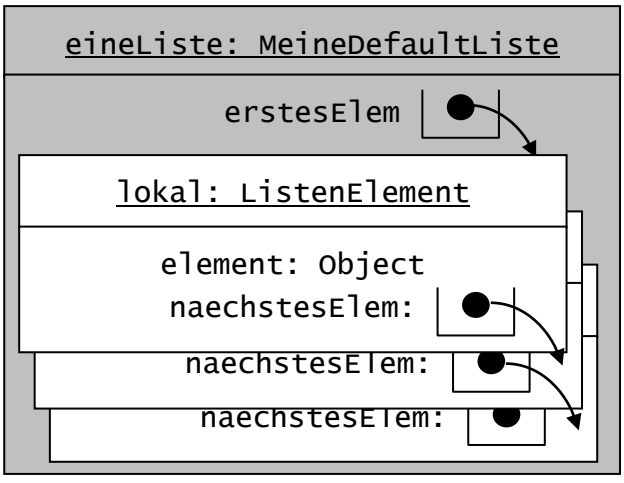
Lokale Klasse

```
  public boolean einfuegenErstesElement(Object element) {
    boolean ret = false;
    if (element != null) {
      ret = true;
      this.erstesElem = new ListenElement(element, erstesElem);
    }
    return ret;
  }

  public boolean loeschenErstesElement() {
    boolean ret = false;
    if (erstesElem != null) {
      ret = true;
      this.erstesElem = this.erstesElem.naechstesElem;
    }
    return ret;
  }

  public boolean istLeer() {
    return this.erstesElem == null;
  }

  public void leeren() {
    this.erstesElem = null;
  }
  // Fortsetzung folgt …
```



Lokale Klassen

Diese werden innerhalb einer beliebigen Klasse definiert.

Die Instanzierung kann nur innerhalb der äußeren Klasse erfolgen.

Innere Klasse kann auf die Membervariablen der äußeren Klasse zugreifen und umgekehrt (siehe dazu anonyme Klassen).

```
  // Fortsetzung Klassendeklaration
  public MeinIterator elemente() {
    return new MeinIterator() {
      private ListenElement aktuellesElem = null;

      public boolean hatNaechstesElement() {
        boolean ret = false;
        if (this.aktuellesElem == null)
          ret = MeineDefaultListe.this.erstesElem != null;
        else
          ret = this.aktuellesElem.naechstesElem != null;
        return ret;
      }

      public Object naechstesElement() {
        Object ret = null;
        if (this.aktuellesElem == null) {
          if (MeineDefaultListe.this.erstesElem != null) {
            this.aktuellesElem = MeineDefaultListe.this.erstesElem;
            ret = this.aktuellesElem.element;
          }
        } else
          if (this.aktuellesElem.naechstesElem == null)
            this.aktuellesElem = null;
          else {
            this.aktuellesElem = this.aktuellesElem.naechstesElem;
            ret = this.aktuellesElem.element;
          }
        return ret;
      }

      public boolean einfuegenElement(Object element) {
        boolean ret = false;
        if (element != null) {
          ret = true;
          if (this.aktuellesElem == null) {
            MeineDefaultListe.this.erstesElem =
              new ListenElement(element,
              MeineDefaultListe.this.erstesElem);
            this.aktuellesElem = MeineDefaultListe.this.erstesElem;
          } else {
            this.aktuellesElem.naechstesElem =
              new ListenElement
              (element,this.aktuellesElem.naechstesElem);
            this.aktuellesElem = this.aktuellesElem.naechstesElem;
          }
        }
        return ret;
      }
    };
  }
}
```

Anonyme Klasse

Anonyme Klassen

- Haben keinen Namen, sondern Definition und Instanzierung erfolgt in einer kombinierten Anweisung.
- `new MeinIterator() { … }`
  gibt die Basisklasse oder Basisinterface der Klasse an.
- Zugriff auf die Membervariablen der äußeren Klasse durch
  `MeineDefaultListe.this.erstesElem = …;`

Delegierung an die Default-Implementierung

Lässt sich eine potentielle `MeineListe`-Klasse nicht von `MeineDefaultListe` ableiten, muss sie das Interface selbst implementieren. Besitzt die Default-Implementierung bereits nennenswerte Funktionalitäten, wäre es schlechter Stil (und auch sehr fehlerträchtig), diese ein zweites Mal zu implementieren. Stattdessen ist es möglich, die Implementierung an die bereits vorhandene `MeineDefaultListe` zu delegieren.

```
public class AutoBesitzer extends Person implements MeineListe
{
  private MeineDefaultListe autoListe = new MeineDefaultListe();

  public AutoBesitzer(String name) {
    this.name = name;
  }

  // Alle Aufrufe der Interface-Methoden werden an das Objekt
  // autoListe weitergereicht
  public boolean einfuegenErstesElement(Object element) {
    return autoListe.einfuegenErstesElement(element);
  }
  public boolean loeschenErstesElement() {
    return autoListe.loeschenErstesElement();
  }
  public boolean istLeer() {
    return autoListe.istLeer();
  }
  public void leeren() {
    autoListe.leeren();
  }
  public MeinIterator elemente() {
    return autoListe.elemente();
  }

  public int getAnzahlAutos() {
    int ret = 0;
    for (MeinIterator i = autoListe.elemente();
      i.hatNaechstesElement();) {
      i.naechstesElement(); ret = ret + 1;
    }
    return ret;
  }
}
```

Nachtrag zu anonymen und lokalen Klassen

Bei der Instanzierung einer anonymen Klasse kann auch ein parameterbehafteter Konstruktor der Basisklasse aufgerufen werden. Die anonyme Klasse darf keinen eigenen parameterbehafteten Konstruktor haben.

```
public class Main
{
  public static void main(String[] args) {
    NeueKlasse o = new NeueKlasse(3) {
      public void tuewas() {
        System.out.println("Anonyme Klasse.tuewas");
      }
      public NeueKlasse(int i) {
        System.out.println("Anonyme Klasse.Konstruktor");
      }
    };
    o.tuewas();
  }
}
```

```
public class NeueKlasse
{
  public NeueKlasse(int i) {
    System.out.println("NeueKlasse.Konstruktor");
  }
  public void tuewas() {
    System.out.println("NeueKlasse.tuewas");
  }
}
```

Eine Klassendeklaration ist auch *innerhalb einer Methode* möglich:

```
public void print() {
  final int value = 10;
  class Inner
  {
    public void print() {
      System.out.println(value);
    }
  }
  Inner i = new Inner();
  i.print();
}
```

Innere Klasse kann auf Variablen der Methode zugreifen.

Diese Variablen müssen aber als `final` deklariert werden.

Statische lokale Klassen

```
package net.gobbz.statischlokal;

public class Outer
{
  private Inner inner = null;

  public static class Inner
  {
    public void print() {
      System.out.println("Inner");
    }
  }

  public Outer() {
    this.inner = new Inner();
    this.inner.print();
  }
}
```

```
import net.gobbz.statischlokal.Outer;
oder
import net.gobbz.statischlokal.Outer.Inner;

public class StaticClassTest
{
  public static void main(String[] args) {
    Outer o = new Outer();
    Outer.Inner i = new Outer.Inner();
    oder
    Inner i = new Inner();
    i.print();
  }
}
```

- Compiler erzeugt gewöhnliche globale Klasse, die auch von außerhalb instanziiert werden kann.
- Instanz von `Inner` kann NICHT auf Membervariablen von `Outer` zugreifen.
- Name der Inneren Klasse enthält als Präfix den Namen der äußeren Klasse.

Beispiel: `android.widget.AdapterView.OnItemClickListener`