Java I – Vorlesung 5
Collections

24.5.2004

Abstrakte Klassen und Interfaces
Arrays
Java Collections Framework

Abstrakte Klassen: Motivation

- Häufig will man eine Klasse schreiben, die nur als Basisklasse für andere Klassen dienen soll.
- Manche Methoden haben dort nur Dummy-Implementierungen und sind nur dazu da, dass ihre Existenz garantiert ist.
- Objekte der Basisklasse sollen nicht erzeugt werden.
- Lösung: Abstrakte Methoden in abstrakten Klassen.

Abstrakte Klassen: Motivation

```
class IntList {
  public int first() {
    return -1; // Fehlerwert
  }

  public IntList rest() {
    return null; // Fehlerwert
  }

  public int size() {
    return -1; // Fehlerwert
  }
}
```


Abstrakte Klassen: Motivation

```
abstract class IntList {
  abstract public int first();
  abstract public IntList rest();
  abstract public int size();
}

class IntCons extends IntList {
  private int firstElement;

  public int first() {
    return firstElement;
  }

  public IntList rest() { ... }
  public int size() { ... }
}
```

Abstrakte Methoden

- Eine abstrakte Methode ist eine Methode, die keine Implementierung hat.
- Zweck:
  - Überschreiben in abgeleiteter Klasse
  - Compiler garantiert, dass die Methode existiert.
- Deklaration mit dem Schlüsselwort `abstract`.
- Abstrakte Methoden dürfen nicht `static` sein.

Abstrakte Klassen

- Eine abstrakte Klasse ist eine Klasse, die eine abstrakte Methode deklariert.
- Deklaration mit Schlüsselwort `abstract`.
- Von einer abstrakten Klasse A können keine Objekte erzeugt werden: Ausdruck `new A()` ist syntaktisch falsch.
- Es kann aber Variablen von Typ A geben.

Abstrakte Klassen und Vererbung

- Von A abgeleitete Klasse B erbt alle abstrakten Methoden und darf sie überschreiben.
- Wenn nicht alle Methoden durch nicht-abstrakte Methoden überschrieben sind, muss auch B als `abstract` deklariert werden.
- Wenn B alle abstrakten Methoden mit nicht-abstrakten Methoden überschreibt, darf B als nicht-abstrakt deklariert werden.
- Objekte von konkreter Klasse B dürfen Variablen von Typ A zugewiesen werden.

Abstrakte Klassen und Ableitung

```
abstract class IntList {
  abstract public int first();
  abstract public IntList rest();
  abstract public int size();
}

class IntCons extends IntList {
  public int first() { ... }
  public IntList rest() { ... }
  public int size() { ... }
}

...
IntList fehler = new IntList(); // Fehler!
IntList list = new IntCons(1, new IntNil());
```

Abstrakte Klassen und Ableitung

```
abstract class IntList {
  abstract public int first();
  abstract public IntList rest();
  abstract public int size();
}

class IntList2 extends IntList {
  public int size() { ... }
}
```


Abstrakte Klassen und Ableitung

```
abstract class IntList {
  abstract public int first();
  abstract public IntList rest();
  abstract public int size();
}

abstract class IntList2 extends IntList {
  public int size() { ... }
}
```

Zweck von abstrakten Klassen

- Abstrakte Basisklasse kann Implementierungsdetails offen lassen.
- Abstrakte Methoden können in anderen Methoden verwendet werden. In jedem Objekt ist die abstrakte Methode dann garantiert überschrieben ("Template-Pattern").

Abstrakte Klasse mit nicht-abstrakten Methoden

```
abstract class GeoFigur {
  abstract public void draw();
  abstract public int inhalt();

  public void test() {
    draw();
    System.out.println("Inhalt = " + inhalt());
  }
}
```

Interfaces

- Ein Interface spezifiziert Methoden, die von einer Klasse definiert werden müssen, kann aber gar keine Implementierung enthalten.
- Eine (abstrakte oder konkrete) Klasse kann ein Interface implementieren.

Interfaces als Datentypen

- Interfaces sind Referenz-Datentypen.
- Implementierende Klassen sind speziellere Typen als die Interfaces, d.h. Upcast ist möglich.
- Eine Klasse implementiert ein Interface nur dann, wenn sie ausdrücklich so deklariert ist -- selbst wenn sie alle Methoden implementiert.

IntList als Interface

```
interface IntList {
  int first();
  IntList rest();
  int size();
}

class IntCons implements IntList {
  public int first() { ... }
  public IntList rest() { ... }
  public int size() { ... }
}

...
IntList fehler = new IntList(); // Fehler!
IntList list = new IntCons(1, new IntNil());
```


IntList als Interface

```
interface IntList {
  int first();
  IntList rest();
  int size();
}

class IntCons {
  public int first() { ... }
  public IntList rest() { ... }
  public int size() { ... }
}

...
IntList list = new IntCons(1, new IntNil());
  // Fehler: IntCons nicht "implements IntList"!
```

Members von Interfaces

- Alle Methodendeklarationen in Interfaces sind implizit `public` und `abstract`.
- Alle Felddeklarationen in Interfaces sind implizit `public` und `static` (und `final`, d.h. ihr Wert darf nicht verändert werden).
- Wenn implementierende Klasse nicht alle Methoden definiert, muss sie als `abstract` deklariert sein.

Zweck von Interfaces

- Vollständiges Verbergen von Implementierungsdetails nach außen.
- Methoden können Interfaces als Parameter nehmen, wenn sie nur wissen müssen, dass bestimmte Methoden definiert werden (egal wie).

Interfaces: Mehrfachvererbung

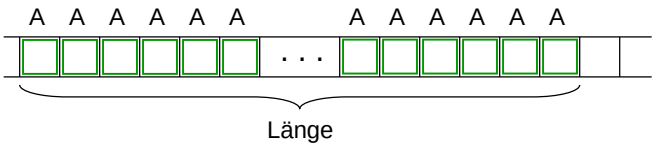
- Eine Klasse darf in Java nur von einer einzigen anderen Klasse abgeleitet sein.
- Eine Klasse darf aber beliebig viele Interfaces implementieren. Sie ist dann Untertyp jedes dieser Interfaces.
- Manchmal wird das ausgenutzt, um leere Interfaces als Markierung zu verwenden (z.B. `Serializable` in Java-API).

Interfaces vs. abstrakte Klassen

- Interfaces sind Extremfall auf Abstraktionsskala: In etwa eine Klasse ohne Felder und nur mit abstrakten Methoden.
- Wenn Klasse irgendwelche Implementierung (konkrete Methoden, Felder) enthalten soll, muss es eine abstrakte Klasse sein.
- Ansonsten ist oft Interface sinnvoll (Mehrfachvererbung!).
- Entscheidung zwischen Interface und abstrakter Klasse nicht immer einfach.

Arrays

- Ein Array ist eine Sammlung von Variablen des gleichen Datentyps, die zur Laufzeit erzeugt werden.
- Jedes Array hat einen eindeutigen Datentyp: `A[]` ist Typ "Array mit Einträgen von Typ A".
- Jedes Array hat eine eindeutige, unveränderliche Länge.

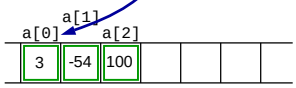


Arrays

- Zugriff auf Elemente mit eckigen Klammern: `expr[i]`.
- `expr` ist ein Ausdruck von Array-Typ; `i` ist ein Ausdruck von Typ `int`.
- Indices laufen ab 0.
- Zugriff auf zu große oder kleine Indices gibt einen Laufzeit-Fehler.

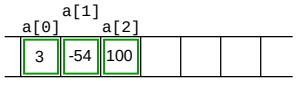
Arrays

```
class ArrayTest {
  public static void main(String[] args) {
    int[] array = new int[3];

    array[0] = 3;
    array[1] = -54;
    array[2] = 100;

    for( int i = 0; i < 3; i++ )
      System.out.println(array[i]);
  }
}
```



Arrays: Mögliche Fehler

```
class ArrayTest {
  public static void main(String[] args) {
    int[] array = new int[3];

    array[-1] = 0;      // Index < 0
    array[3] = 0;       // Index > Array-Länge
    array[1] = "hallo"; // Falscher Datentyp
  }
}
```



Arrays als Parameter von Methoden

```
class Test {
  public static void main(String[] args) {
    for( int i = 0; i < args.length; i++ )
      System.out.println(args[i]);
  }
}
```


Arrays deklarieren

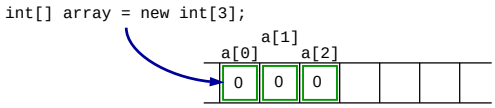
- Ein Array wird genauso deklariert wie jede andere Variable auch:
  ```
  A[] varname;
  ```
- Dabei wird aber nur das Array-Objekt angelegt und noch kein Speicher für die Elemente des Arrays.

Arrays erzeugen

- Die Elemente des Arrays werden mit besonderer Form von new erzeugt:

```
new A[expr];
```

- `expr` muss ein Ausdruck von Typ `int` sein, der Länge des Arrays angibt.



Arrays erzeugen: Initialisierer

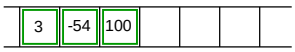
- Arrays können bequem mit Initialisierern erzeugt werden:

```
A[] varname = { x, y, z };
```

- x, y, z sind Ausdrücke von Typ A. Länge des neuen Arrays ist Länge der Liste in Klammern.
- Achtung: `{ ... }` ist kein eigener Ausdruck, sondern nur im Kontext der Variablen-Initialisierung zulässig!
- Kombination mit `new` ist erlaubt.

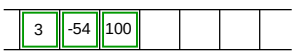
Initialisierer

```
class ArrayTest {
  public static void main(String[] args) {
    int[] array = {3, -54, 100};

    for( int i = 0; i < 3; i++ )
      System.out.println(array[i]);
  }
}
```



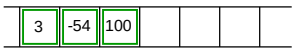
Initialisierer

```
class ArrayTest {
  public static void main(String[] args) {
    int[] array = new int[] {3, -54, 100};

    for( int i = 0; i < 3; i++ )
      System.out.println(array[i]);
  }
}
```



Initialisierer

```
class ArrayTest {
  public static void main(String[] args) {
    int[] array = new int[3] {3, -54, 100};

    for( int i = 0; i < 3; i++ )
      System.out.println(array[i]);
  }
}
```



Initialisierer

```
class ArrayTest {
  public static void main(String[] args) {
    int[] array;
    array = {3, -54, 100};  // FEHLER!

    for( int i = 0; i < 3; i++ )
      System.out.println(array[i]);
  }
}
```

Länge von Arrays

- Jedes Array hat eine feste Länge.
- Die Länge eines Arrays kann mit dem Feld `length` ermittelt werden:

  ```
  expr.length
  ```

- Das ist v.a. nützlich, wenn man Array als Parameter in einer Methode übergeben bekommt.

Länge von Arrays

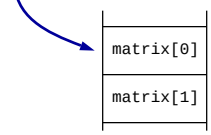
```
class Test {
  public static void main(String[] args) {
    for( int i = 0; i < args.length; i++ )
      System.out.println(args[i]);
  }
}
```

Mehrdimensionale Arrays

- Der Element-Datentyp eines Arrays darf seinerseits ein Array-Datentyp sein.
- Deren Element-Typen dürfen ihrerseits Array-Typen sein, und so weiter.
- Damit können mehrdimensionale Informationen gespeichert werden.
- Die verschiedenen Elemente des Arrays dürfen Arrays verschiedener Länge sein.

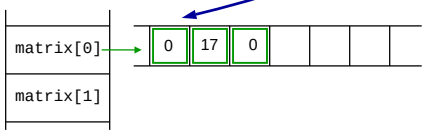
Matrizen

```
class Matrix {
  public static void main(String[] args) {
    int[][] matrix;

    matrix = new int[2][];
    for( int i = 0; i < 5; i++ )
      matrix[i] = new int[3];

    matrix[0][1] = 17;
  }
}
```



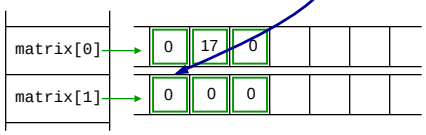
Matrizen

```
class Matrix {
  public static void main(String[] args) {
    int[][] matrix;

    matrix = new int[2][];
    for( int i = 0; i < 5; i++ )
      matrix[i] = new int[3];

    matrix[0][1] = 17;
  }
}
```



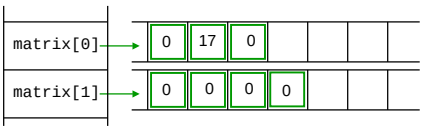
Matrizen

```
class Matrix {
  public static void main(String[] args) {
    int[][] matrix;

    matrix = new int[2][];
    for( int i = 0; i < 5; i++ )
      matrix[i] = new int[3];

    matrix[0][1] = 17;
  }
}
```



Länge muss nicht einheitlich sein

```
class Matrix2 {
  public static void main(String[] args) {
    int[][] matrix;

    matrix = new int[2][];
    matrix[0] = new int[3];
    matrix[1] = new int[4];

    matrix[0][1] = 17;
  }
}
```



Mehrdimensionale Arrays: Erzeugen

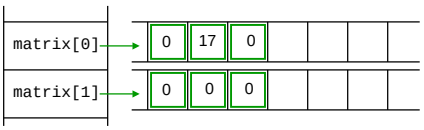
- Das erste `new A[expr][]` legt nur in der ersten Dimension Speicher an.
- Jedem einzelnen Eintrag neues Array zuweisen: Schleife.
- Wenn alle Einträge gleiche Länge haben, abkürzende Schreibweise:

  ```
  new A[expr1][expr2]
  ```

- Mehrdimensionale Arrays können mit geschachtelten Initialisierern angelegt werden.

Matrizen

```
class Matrix {
  public static void main(String[] args) {
    int[][] matrix;

    matrix = new int[2][3];

    matrix[0][1] = 17;
  }
}
```



Collections

- Manchmal weiß man zunächst noch nicht, wie viele Einträge ein Array haben soll.
- Das wird aber von Java nicht unterstützt: Länge des Arrays ist unveränderlich.
- Ausweg: Listen.
- Listen und andere Sammlungen (collections) von Objekten werden von Java Collections Framework bereitgestellt.

Java Collections Framework

- Definiert Interfaces für Datentypen, v.a.
  - `List<T>`: Angeordnete Liste von Einträgen
  - `Set<T>`: Ungeordnete Menge von Einträgen
  - `Map<S,T>`: Abbildungen
- Verschiedene Implementierungen für jedes Interface.
- Java 1.5: Kann Typ S, T der Einträge angeben.
- http://java.sun.com/j2se/1.5.0/docs/guide/collections/

Listen

- Eine Liste ist eine geordnete Abfolge von Werten. Gleicher Wert darf mehrfach vorkommen.
- Im JCF gibt es Interface `List<T>` für Listen mit Element-Typ T. T muss Referenz-Typ sein (nicht `int` o.ä.!).
- Wichtige Methoden:
  - `add`: Element einfügen
  - `size`: Länge der Liste
  - `get`: Element an Position auslesen
  - `remove`: Element entfernen

Listen

```
import java.util.*;

class Test {
  public static void main(String[] args) {
    List<Integer> liste =
       new LinkedList<Integer>();

    for( int i = 0; i < args.length; i++ )
      liste.add(new Integer(args[i]));

    System.out.println("Elemente: " +
                       liste.size());
    System.out.println("1. Element: " +
                       liste.get(0));
  }
}
```


Listen: Andere Implementierung

```
import java.util.*;

class Test {
  public static void main(String[] args) {
    List<Integer> liste =
       new ArrayList<Integer>();

    for( int i = 0; i < args.length; i++ )
      liste.add(new Integer(args[i]));

    System.out.println("Elemente: " +
                       liste.size());
    System.out.println("1. Element: " +
                       liste.get(0));
  }
}
```

Interfaces und Implementierungen

- Indem man im Programm nur die Interfaces verwendet, macht man sich von Implementierung unabhängig.
- Implementierungen von `List<T>` im JCF:
  - `ArrayList<T>`: Liste intern mit Arrays implementiert
  - `LinkedList<T>`: Als verkettete Liste implementiert
- Implementierungen haben Vor- und Nachteile.

Iteratoren

- Jedes JCF-Objekt kann einen Iterator zurückgeben, mit dem man mit for-Schleife durch die Elemente des Objekts iterieren kann.
- Aufruf `o.iterator()` gibt ein `Iterator<T>`-Objekt zurück.
- Iteratoren haben Methoden `hasNext()`, `next()`.
- In Java 1.5 gibt es verkürzte Schreibweise für for-Schleifen.

Iteratoren

```
import java.util.*;

class Test {
  public static void main(String[] args) {
    List<Integer> liste =
       new ArrayList<Integer>();

    // ...

    for( Iterator<Integer> it = liste.iterator();
         it.hasNext() ; ) {
      Integer x = it.next();
      System.out.println(x);
    }
  }
}
```


Iteratoren: Verkürzte Syntax

```
import java.util.*;

class Test {
  public static void main(String[] args) {
    List<Integer> liste =
       new ArrayList<Integer>();

    // ...

    for( Integer x : liste ) {
      System.out.println(x);
    }
  }
}
```

Mengen

- Eine Menge ist eine ungeordnete Sammlung von Werten. Jeder Wert kommt nur einmal vor.
- Interface `Set<T>` für Mengen mit Einträgen von Typ T.
- Wichtige Methoden:
  - `add`, `remove`, `size` wie bei Listen
  - `get` macht hier keinen Sinn!

Mengen

- Implementierungen:
  - `HashSet<T>`: Über Hashtabellen implementiert; effizient
  - `TreeSet<T>`: Menge nach Ordnung der Werte sortiert.
  - `LinkedHashSet<T>`: Menge nach Ordnung der Einfüge-Operationen sortiert.

Mengen: Ein Beispiel

```
import java.util.*;

class Test {
  public static void main(String[] args) {
    Set<Integer> s = new HashSet<Integer>();

    s.add(new Integer(1));
    s.add(new Integer(2));
    s.add(new Integer(1));

    System.out.println(s.size());  // -> 2
  }
}
```


Abbildungen (Wörterbücher)

- Eine Abbildung bildet Werte eindeutig auf andere Werte ab.
- Interface `Map<S,T>` für Abbildungen von Typ S nach Typ T.
- Wichtigste Operationen:
  - `put(key, val)`: Legt Wert val unter Schlüssel key ab.
  - `containsKey(key)`: Gibt es den Eintrag key?
  - `get(key)`: Liest Wert für Schlüssel key aus

Abbildungen (Wörterbücher)

- Implementierungen:
  - `HashMap<S,T>`: Über Hashtabellen implementiert, effizient
  - `TreeMap<T>`: Einträge nach Ordnung auf Schlüsseln sortiert.
  - `LinkedHashMap<T>`: Einträge nach Ordnung der Einfüge-Operationen sortiert.

Abbildungen

```
import java.util.*;

class Test {
  public static void main(String[] args) {
    Map<String,Integer> map =
       new HashMap<String,Integer>();

    map.put("abcd", new Integer(3));
    map.put("Hallo", new Integer(-2));

    for( String key : map.keySet() ) {
      System.out.println( key + " -> " +
                          map.get(key) );
    }
  }
}
```

Zusammenfassung

- Abstrakte Klassen und Interfaces erlauben die Definition von abstrakten Methoden ohne Implementierung.
- Arrays sind Sammlungen von gleichartigen Werten.
- Collections definieren Listen, Mengen und Abbildungen und nutzen konsequent Interfaces.