Programmierkurs Java

UE 15 - Zugriffsrechte

Dr.-Ing. Dietrich Boles

Gliederung

- Klassen
- Attribute und Methoden
- Beispiel
- Klassen als ADT
- ADT-Klasse DynamicArray
- Zusammenfassung

Klassen (1)

- In Java existieren zwei verschiedene Möglichkeiten, Zugriffsrechte auf Klassen zu definieren:
  - **`public`**
  - **`<ohne>`** (Zugriffsschlüsselwort fehlt!)

- **public**-Klassen sind von überall her zugreifbar/nutzbar (`import`)
- **<ohne>**-Klassen sind nur in dem Paket zugreifbar/nutzbar, in dem sie definiert werden

- In einer Datei können mehrere Klassen definiert werden, aber nur eine darf eine **public**-Klasse sein! Der Name der Datei muss in diesem Fall gleich dem Namen der **public**-Klasse sein.

- Richtlinien:
  - will man öffentlich zugängliche Klassen definieren, muss man sie als **public** deklarieren
  - Hilfsklassen sollten immer **<ohne>** deklariert werden

Klassen (2)

Beispiel (Datei **`DynamicArray.java`**):

```
package util;

class ListElem {   // Hilfsklasse
  int value;       // Speicher fuer den Wert
  ListElem next;   // Verweis auf naechstes Element
  ListElem(int v) { value = v; next = null; }
}

public class DynamicArray {
  ListElem first; // verkettete Liste

  public DynamicArray() { ... }
  public void add(int value) { ... }
  public void remove(int value) { ... }
  public boolean isElement(int value) { ... }
  ...
}
```

Attribute und Methoden (1)

- In Java existieren vier verschiedene Möglichkeiten, Zugriffsrechte auf Attribute und Methoden zu definieren (in absteigender Reihenfolge):
  - `public`
  - `protected`
  - `<ohne>` (Zugriffsschlüsselwort fehlt!)
  - `private`

- **public**-Attribute/Methoden sind von überall her zugreifbar
- **protected**-Attribute/Methoden sind nur zugreifbar in allen Klassen desselben Paketes sowie in abgeleiteten Klassen (auch anderer Pakete)
- **<ohne>**-Attribute/Methoden sind zugreifbar in allen Klassen desselben Paketes
- **private**-Attribute/Methoden sind nur zugreifbar in derselben Klasse

Attribute und Methoden (2)

Beispiel (Datei **`DynamicArray.java`**):

```
package util;

class ListElem {        // Hilfsklasse
  private int value;    // Speicher fuer den Wert
  ListElem next;        // Verweis auf naechstes Element
  ListElem(int v) { this.value = v; next = null; }
  int getValue() { return this.value; }
}

public class DynamicArray {
  protected ListElem first; // verkettete Liste

  public DynamicArray () { ... }
  public void add(int value) { ... }
  public void remove(int value) { ... }
  public boolean isElement(int value) { ... }
  ...
}
```

Attribute und Methoden (3)

- Richtlinien (Attribute):
  - Definieren Sie Attribute möglichst niemals als **public** (→ Datenkapselung, set/get-Methoden)
  - Definieren Sie Attribute, die lediglich innerhalb der Klassenimplementierung benötigt werden, als **private** oder **<ohne>**
  - Definieren Sie Attribute, auf die evtl. jemand zugreifen muss, wenn er eine Klasse von der Klasse ableitet, immer als **protected**
  - Konstanten werden oft als **public** definiert (kann man eh nicht manipulieren)

- Richtlinien / Anmerkungen (Methoden):
  - Definieren Sie nur die Methoden als **public**, die Sie anderen zur Verfügung stellen wollen
  - bei in abgeleiteten Klassen überschriebenen Methoden dürfen die Rechte **ausschließlich erweitert** werden

    (**`private` → `<ohne>` → `protected` → `public`**)

Beispiel (1)

Übung: wo liefert der Compiler Fehlermeldungen?

```
Verzeichnis: misc
Datei: X.java

public class X {
  private   int i1;
            int i2;
  protected int i3;
  public    int i4;

  void zugriffOK(X x2) {
    this.i1 = 3;
    this.i2 = 4;
    this.i3 = 5;
    this.i4 = x2.i1;
} }
```

```
Verzeichnis: misc
Datei: Y.java

public class Y {
  X x1;
  void zugriffOK() {
    x1 = new X();
    x1.i1 = 3;
    x1.i2 = 4;
    x1.i3 = 5;
    x1.i4 = 6;
    x1.zugriffOK(new X());
} }
```

Beispiel (2)

Übung: wo liefert der Compiler Fehlermeldungen?

```
Verzeichnis: misc
Datei: X.java

package misc;
public class X {
  private   int i1;
            int i2;
  protected int i3;
  public    int i4;

  void zugriffOK(X x2) {
    this.i1 = 3;
    this.i2 = 4;
    this.i3 = 5;
    this.i4 = x2.i1;
} }
```

```
Verzeichnis: util
Datei: Y.java

package util;
import misc.*;
public class Y {
  X x1;
  void zugriffOK() {
    x1 = new X();
    x1.i1 = 3;
    x1.i2 = 4;
    x1.i3 = 5;
    x1.i4 = 6;
    x1.zugriffOK(new X());
} }
```

Beispiel (3)

Übung: wo liefert der Compiler Fehlermeldungen?

```
Verzeichnis: misc
Datei: X.java

package misc;
public class X {
  private   int i1;
            int i2;
  protected int i3;
  public    int i4;

  void zugriffOK(X x2) {
    this.i1 = 3;
    this.i2 = 4;
    this.i3 = 5;
    this.i4 = x2.i1;
} }
```

```
Verzeichnis: util
Datei: Y.java

package util;
import misc.*;
public class Y extends X {

  void zugriffOK(X x2) {
    this.i1 = 3;
    this.i2 = 4;
    this.i3 = 5;
    this.i4 = x2.i3;
  }
}
```

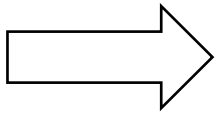
Zugriff nur auf eigene geerbte protected-Attribute erlaubt

Klassen als ADT

- Datentyp = Wertebereich + Funktionen/Operationen
- Abstrakter Datentyp =
  (gekapselte) Datenstruktur + (öffentliche) Funktionen
- Klasse = (syntaktisches) Konzept, Datenstruktur und Funktionen zu einer Einheit zusammenzufassen
- Zugriffsrechte: Steuerung von Datenkapselung und Öffentlichkeit
- Protokoll einer Klasse: Menge der nach außen sichtbaren Elemente

Imperative Programmierung:

```
class X {
  „Datenstruktur“
}

class XFunktionen {
  „Funktionen für X“
}
```



Objektorientierte Programmierung:

```
class X {
  „Datenstruktur“
       +
  „Funktionen“
}
```

Klassen als ADT / Motivationsbeispiel (1)

```
package math;

public class Bruch {
  // gekapselte Datenstruktur
  private int zaehler;
  private int nenner;

  // Öffentliche Funktionen/Methoden auf der Datenstruktur
  public Bruch(int z, int n) {  // Konstruktor
    this.zaehler = z;
    this.nenner = n;
    this.kuerzen();
  }

  public Bruch() {  // Default-Konstruktor
    this(0, 1);
  }
```

Klassen als ADT / Motivationsbeispiel (2)

```
public void multTo(Bruch b2) { // Multiplikation
  this.zaehler *= b2.zaehler;
  this.nenner *= b2.nenner;
  this.kuerzen();
}

public void addTo(Bruch b2) { // Addition
  this.zaehler = b2.nenner * this.zaehler +
                 this.nenner * b2.zaehler;
  this.nenner = this.nenner * b2.nenner;
  this.kuerzen();
}

public String toString() {
  return this.zaehler + "/" + this.nenner;
}
```

Klassen als ADT / Motivationsbeispiel (3)

```
  // interne Hilfs-Methoden
  private void kuerzen() {
    int ggt = Bruch.ggT(this.zaehler, this.nenner);
    this.zaehler /= ggt;
    this.nenner /= ggt;
  }

  private static int ggT(int z1, int z2) {
    return (z2 == 0) ? z1 : Bruch.ggT(z2, z1 % z2);
  }

  // Testprogramm
  public static void main(String[] args) {
    Bruch b1 = new Bruch(12, 9);
    Bruch b2 = new Bruch(3, 2);
    b1.multTo(b2); // b1 = b1 * b2;
    b1.addTo(b2);  // b1 = b1 + b2;
    IO.println("Bruch 1: " + b1.toString());
    IO.println("Bruch 2: " + b2.toString());
  }
}
```

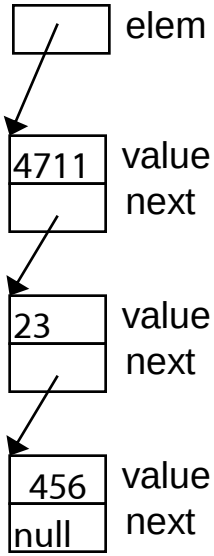
ADT-Klasse DynamicArray

- gesucht: Klasse **`DynamicArray`**, die ein Array beliebiger Größe realisiert
- wichtig ist das Protokoll, nicht die interne Datenstruktur

```
public class DynamicArray { // gewünschtes Protokoll
  public void add(int value)
  public boolean isElem(int value)
}

public static void main(String[] args) { // Test
    DynamicArray l = new DynamicArray();
    int input = IO.readInt("Number: "); // Fuellen
    while (input > 0) {
      l.add(input);
      input = IO.readInt("Number: ");
    }
    while (true) { // auf Liste arbeiten
      int check = IO.readInt("Check: ");
      if (l.isElem(check)) IO.println("Is Element");
      else IO.println("Is NOT Element");
}   }
```

ADT-Klasse DynamicArray / Lösung 1 (1)

```
class ListElem {   // Hilfsklasse (nicht public)
  int value;       // Speicher fuer den Wert
  ListElem next;   // Verweis auf naechstes Element

  ListElem(int v) { value = v; next = null; }
}


public class DynamicArray{

  // gekapselte Datenstruktur
  private ListElem elem; // verkettete Liste

  // Konstruktoren

  public DynamicArray() { // Default-Konstruktor
    this.elem = null;
  }
```

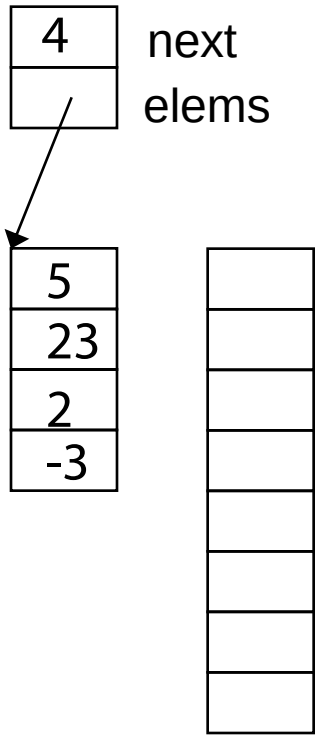


ADT-Klasse DynamicArray / Lösung 1 (2)

```
  // Methoden
  public void add(int v) {
    if (this.elem == null) {
      this.elem = new ListElem(v);
    } else {
      ListElem help = this.elem;
      while (help.next != null)
        help = help.next;
      help.next = new ListElem(v);
    }
  }

  public boolean isElem(int v) {
    ListElem help = this.elem;
    while (help != null) {
      if (help.value == v) return true;
      help = help.next;
    }
    return false;
} }
```

ADT-Klasse DynamicArray / Lösung 2

```
public class DynamicArray{
  private int next;
  private int[] elems;  // "austauschbares" Array

  public DynamicArray() {
    this.elems = new int[4];  this.next = 0;
  }

  public void add(int v) {
    if (this.next == this.elems.length) {
      int[] copy = new int[this.elems.length * 2];
      for (int i=0; i<this.elems.length; i++)
        copy[i] = this.elems[i];
      this.elems = copy; // Austausch des Arrays
    }
    this.elems[this.next++] = v;
  }

  public boolean isElem(int v) {
    for (int i=0; i<this.next; i++)
      if (this.elems[i] == v) return true;
    return false;
} }
```



Zusammenfassung

- Klassen als auch Elemente von Klassen (Attribute, Methoden) können mit Hilfe von Zugriffsrechten vor unberechtigtem Zugriff geschützt werden
- Zugriffsrechte in Java realisieren die Datenkapselung
- Zugriffsrechte unterstützen die Definition von Abstrakten Datentypen
- ADT: (gekapselte) Datenstruktur + (öffentliche) Funktionen auf der Datenstruktur
- Klasse: (syntaktisches) Konzept, Datenstruktur und Funktionen zu einer Einheit zusammenzufassen