Grundlagen der Programmierung

Dr. Christian Herzog
Technische Universität München

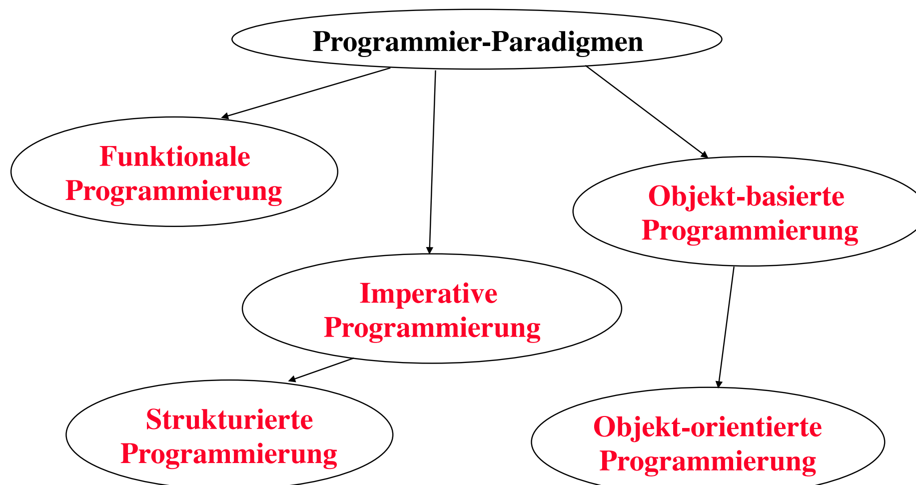
Wintersemester 2017/2018

Kapitel 8: Objektorientierter Programmierstil

Überblick: Wo stehen wir?

- Funktionale Programmierung: Ein Programm ist eine Menge von Funktionen. Die Ausführung des Programms besteht in der Berechnung eines Ausdrucks mit Hilfe der Funktionen und liefert Ergebniswerte
- Imperative Programmierung: Ein Programm besteht aus einer Menge von Daten und Operationen (Prozeduren, Funktionen). Die Ausführung des Programms verändert die Werte dieser Daten. Die Operationen sind nicht an die Daten gebunden.
  - Strukturierte Programmierung: Alle imperativen Programme können mit Zuweisungs-, **`if`**-, **`for`**-, **`while`**- und **`do-while`**-Anweisungen realisiert werden.
- Objekt-basierte Programmierung: Ein Programm ist eine Menge von kooperierenden Klassen, die Daten und Operationen (Methoden genannt) enthalten. *Methoden können nicht außerhalb von Klassen existieren*.
- ***Ein neuer Begriff:***
  Objekt-orientierte Programmierung: Ein Programm ist eine Menge von kooperierenden und *wieder verwendbaren* Klassen. Klassen können ihre Eigenschaften *vererben*.

Überblick über die Programmier-Paradigmen



Überblick über diesen Vorlesungsblock

- Konzepte der objekt-orientierten Programmierung
  - Vererbung
  - Abstrakte Klassen
  - Generische Klassen
  - Schnittstellen (*interfaces*)
- Ziele:
  - Sie verstehen die Gründe für objekt-orientierte Programmierung.
  - Sie können aktiv mit Vererbung umgehen, insbesondere mit Vererbung durch Spezifikation und Vererbung durch Implementierung.
  - Sie kennen den Unterschied in Java zwischen Klasse, abstrakter Klasse und Schnittstelle.
- Beispiel:
  - Hierarchie von Mengendarstellungen
  - Enumeratoren (Iteratoren)

Objekt-Orientierung

- **Objekt-orientierte Programmierung:** Hauptziel ist die Wiederverwendung (reuse) von Bausteinen (components). Drei Konzepte werden wir jetzt kennen lernen:
  1. **Implementierungsvererbung** (implementation inheritance), auch reale Vererbung genannt:

     Wiederverwendung von Implementierungen
  2. **Spezifikationsvererbung** (interface inheritance), auch virtuelle Vererbung genannt:

     Wiederverwendung von Schnittstellen
  3. **Generische Klassen**:

     Wiederverwendbarkeit von Datenstrukturen mit unterschiedlichen Basistypen.
- Im folgenden besprechen wir Javakonstrukte für diese Konzepte.

Copyright 2017 Bernd Brügge, Christian Herzog

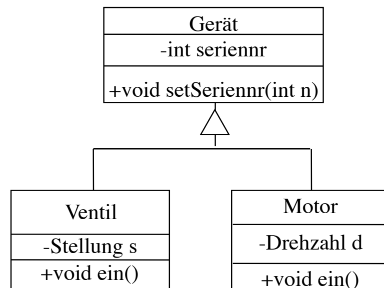
Die Vererbungsbeziehung (vgl. Kapitel 3, Folie 32)

- Zwei Klassen stehen in einer **Vererbungsbeziehung** (*inheritance relationship*) zueinander, falls die eine Klasse, auch **Unterklasse** (Subklasse) genannt, alle Merkmale der anderen Klasse, auch **Oberklasse** genannt, besitzt, und darüber hinaus noch zusätzliche Merkmale.
- Eine Unterklasse wird also durch Hinzufügen von Merkmalen **spezialisiert**.
- Umgekehrt verallgemeinert die Oberklasse die Unterklasse dadurch, dass sie spezialisierende Eigenschaften weglässt. Wir nennen das auch **Verallgemeinerungsbeziehung** (*generalization relationship*).
- **Alternativer Sprachgebrauch (Java):** Eine von einer **Superklasse `B`** abgeleitete Klasse **`A` erbt** die Attribute und Methoden, die **`B`** anbietet.

Copyright 2017 Bernd Brügge, Christian Herzog

Beispiel für Vererbung

- **Modell:**




```
// Irgendwo in main() oder in einer anderen Klasse:
….
Ventil v = new Ventil();
v.setSeriennr(1508);
Geraet g = new Ventil(); // ein Ventil ist ein Gerät
```

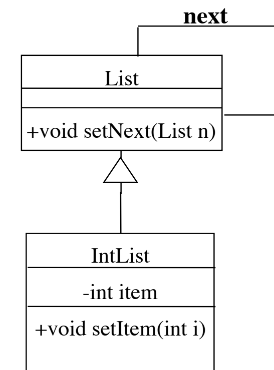
Java Code:

```
class Geraet {
   private int seriennr;
   public void setSeriennr(int n) {
      seriennr = n;
   }
}

class Ventil extends Geraet {
   private Stellung s;
   public void ein() {
     s.an = true;
   }
}
class Motor extends Geraet {
   private Drehzahl d;
   public void ein() {
     d.on = true;
   }
}
```

Copyright 2017 Bernd Brügge, Christian Herzog

Noch ein Beispiel

- **Modell:**



Java Code:

```
class List {
    private List  next;
    public void setNext(List n) {
       next = n;
    }
}
class IntList extends List {
   private int item;
   public void setItem(int i) {
      item = i;
   }
}
…..
List list = new  IntList();
list.setNext(null);
…..
```

- **`IntList`** erweitert **`List`** um ein neues Attribut **`item`** und eine neue Methoden **`setItem()`**.

Copyright 2017 Bernd Brügge, Christian Herzog

Überschreibbare Methoden

- Falls die Unterklasse tatsächlich die Implementierung einer Methode aus der Superklasse wiederverwendet, sprechen wir von **Implementierungs-Vererbung** (implementation inheritance).
- Oft ist Flexibilität gefordert: Die Implementierung der Methode aus der Superklasse muss an die Besonderheiten der Unterklasse angepasst werden, und zwar durch eine Reimplementierung.
- Methoden, die eine Reimplementierung zulassen, heißen **überschreibbare Methoden**.
  - In Java sind überschreibbare Methoden Standard, d.h. es gibt kein Schlüsselwort, um überschreibbare Methoden zu kennzeichnen.
  - Wenn eine Methode nicht überschrieben werden darf, muss sie mit dem Schlüsselwort **`final`** gekennzeichnet sein.

Beispiel für die Überschreibung einer Methode

Ursprünglicher Java-Code:

```
class Geraet {
    private int seriennr;
    public final void help() {….}
    public void setSeriennr(int n) {
      seriennr = n;
    }
}
class Ventil extends Geraet {
    private Stellung s;
    public void ein() {
      s.an = true;
    }
}
```

Neuer Java-Code :

```
class Geraet {
    protected int seriennr;
    public final void help() {….}
    public void setSeriennr(int n) {
      seriennr = n;
    }
}
class Ventil extends Geraet {
    private Stellung s;
    public void ein() {
      s.an = true;
    }
    public void setSeriennr(int n) {
      seriennr = n + 10000;
    }
} // class Ventil
```

nicht über-schreibbar

`protected` erlaubt Zugriff von Unterklassen aus

Überschreibt die Methode `setSeriennr()` aus der Klasse `Geraet`

Der Sichtbarkeits-Modifikator „protected“

- Zur Erinnerung
  - Die **Sichtbarkeit** eines Attributs oder einer Operation regelt, welche Objekte dieses Merkmal verwenden dürfen.
  - Die Sichtbarkeit ist zwischen Klassen definiert, d.h. alle Objekte einer Klasse K_1 haben auf ein Merkmals eines Objekts einer Klasse K_2 dieselben Zugriffsrechte.
- Nun kennen wir drei Sichtbarkeiten:
  - **public**: jedes Objekt jeder Klasse hat unbeschränkten Zugriff;
  - **private**: nur die Objekte derselben Klasse dürfen das Merkmal verwenden.
  - **protected**: auch Objekte von abgeleiteten Klassen (Unterklassen) haben Zugriff
- In UML werden diese Sichtbarkeiten durch ein vorangestelltes „+“, „-“ bzw. „#“gekennzeichnet.

Abstrakte Methoden und Abstrakte Klasse

- Wenn eine Methode noch keine Implementierung hat, nicht einmal einen leeren Methodenrumpf, dann sprechen wir von einer **abstrakten Methode** (abstract method)
  - Eine Klasse, die mindestens eine abstrakte Methode enthält, ist eine **abstrakte Klasse** (abstract class). Sie wird mit den Schlüsselworten **`abstract class`** gekennzeichnet.

Beispiel einer abstrakten Methode

Ursprünglicher Java Code:

```
class Geraet {
   private int seriennr;
   public void setSeriennr(int n) {
      seriennr = n;
   }
}

class Ventil extends Geraet {
   private Stellung s;
   public void ein() {
     s.an = true;
   }
}
```

Java Code mit abstrakter Methode:

```
abstract class Geraet {
   protected int seriennr;
   public abstract void setSeriennr(int n);
}

class Ventil extends Geraet {
   private Stellung s;
   public void ein() {
    s.an = true;
   }
   public void setSeriennr(int n) {
     seriennr = n;
   }
} // class Ventil
```

Abstrakte Methode: hat keinen Rumpf, nicht einmal Klammern {}

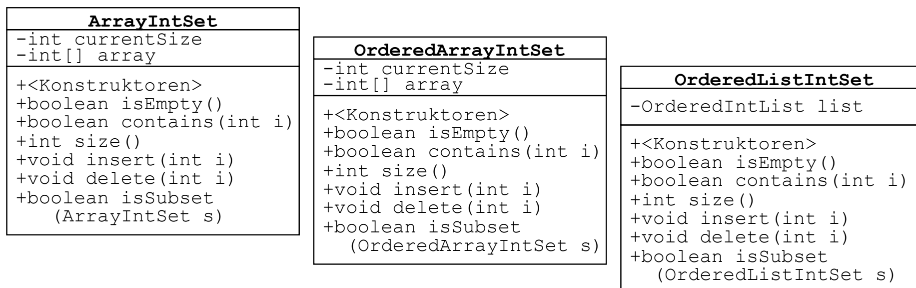
Implementierung der abstrakten Methode `setSeriennr()`

Eigenschaften von Klassen mit abstrakten Methoden

- Wenn wir eine Methode als abstrakt deklarieren, dann überlassen wir die Implementierung einer Unterklasse, die von der Superklasse erbt.
  - Die Funktionalität der Methode in der Unterklassenimplementierung muss mit der Funktionalität der abstrakten Methode identisch sein.
- Eine Klasse, die eine abstrakte Methode enthält, ist automatisch auch abstrakt, und muss deshalb als abstrakt deklariert werden.
- **Abstrakte Klassen können nicht instanziiert werden:**
  - Nur Unterklassen, in denen alle Methoden implementiert sind, können instanziiert werden.
- Wenn eine Unterklasse einer abstrakten Klasse nicht alle abstrakten Methoden implementiert, dann ist die Unterklasse selber auch abstrakt.

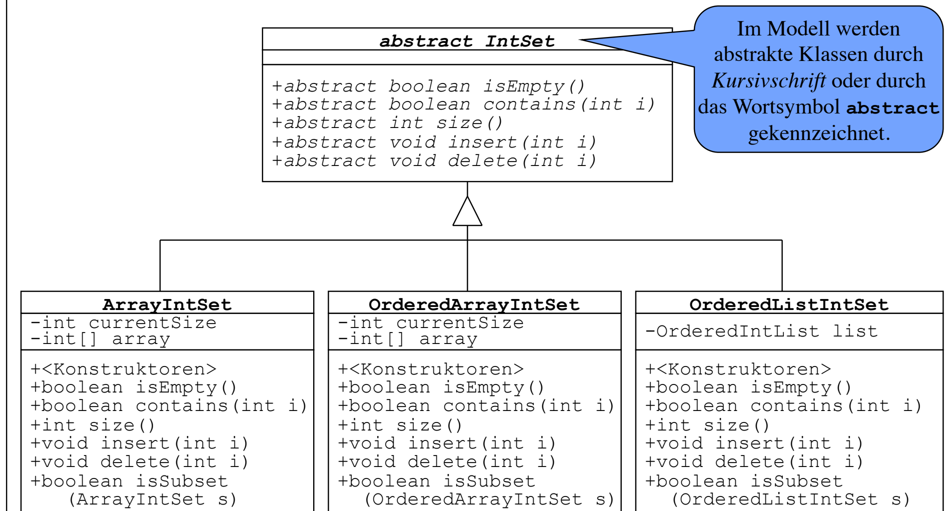
Zurück zur Darstellung von Integer-Mengen

- In Kapitel 7 hatten wir drei verschiedene Klassen zur Darstellung von Mengen über ganzen Zahlen entwickelt:



- Die Schnittstellen der Klassen unterscheiden sich nur in den Konstruktoren und in der Funktionalität der Methode **`isSubset()`**.
- Es bietet sich also die Generalisierung an, eine Oberklasse mit dem gemeinsamen Teil der Schnittstellen zu bilden.
- Da wir (zunächst) nur Schnittstellen vererben, wird die Klasse abstrakt.

Eine abstrakte Oberklasse für die Mengendarstellungen



Die abstrakte Klasse `IntSet` in Java

```
abstract class IntSet {
    public abstract boolean isEmpty();
    public abstract boolean contains(int i);
    public abstract int size();
    public abstract void insert(int i);
    public abstract void delete(int i);
}
```

Die anderen Klassen sind Ableitungen dieser Klasse, z.B.:

```
class ArrayIntSet extends IntSet{

    ... Klassendefinition wie bisher ...

}
```


Wiederverwendung von Code: die Methode `isEmpty()`

- In allen bisherigen Implementierungen von Mengendarstellungen haben wir **`isEmpty()`** auf **`size()`** abgestützt, die Implementierung war jeweils identisch.
  - **`isEmpty()`** selbst ist also unabhängig von der konkreten Datenstruktur und kann bereits auf abstrakter Ebene implementiert werden:

```
abstract class IntSet {
            // die datenstrukturunabhaengige Methode isEmpty
            // wird bereits hier implementiert:
    public boolean isEmpty() {
        return size() == 0;
    }
    public abstract boolean contains(int i);
    public abstract int size();
    public abstract void insert(int i);
    public abstract void delete(int i);
}
```

- In den abgeleiteten Klassen muss die Implementierung von **`isEmpty()`** nicht wiederholt werden; der Code wird wieder verwendet.

Variablen vom Typ `IntSet`

- Was haben wir durch die Einführung der abstrakten Klasse **`IntSet`** bisher erreicht?
  - Der Code der Methode **`isEmpty()`** wurde wieder verwendet.
  - Wir können mit Variablen von Typ **`IntSet`** arbeiten, wenn wir nur über die Schnittstelle von **`IntSet`** zugreifen müssen:

```
class SetTest {
    public static void besetze (IntSet set) {
        for (int i=101; i<=110; i++)
          set.insert(i);
        System.out.println(set);
    }
    public static void main (String[] args) {
        ArrayIntSet arrSet = new ArrayIntSet(10);
        OrderedArrayIntSet oArrSet = new OrderedArrayIntSet(10);
        OrderedListIntSet oListSet = new OrderedListIntSet();

        besetze(arrSet);
        besetze(oArrSet);
        besetze(oListSet);
    }
}
```

Dynamische Bindung: Die zum konkreten Objekt „passenden“ Versionen der Methoden **`insert()`** und **`toString()`** werden ermittelt und angewendet.

Instanziierung mittels **`new`** ist nur bei konkreten Klassen möglich!

Verallgemeinerung des Copy-Konstruktors

- „Natürlich“ sind wir mit dem Erreichten noch nicht zufrieden:
  - Als nächstes Ziel wünschen wir uns, dass jede der drei konkreten Klassen einen Copy-Konstruktor erhält, der es erlaubt, die Kopie einer Menge **`set`** in der gewünschten Darstellung zu erhalten, gleichgültig wie **`set`** selbst dargestellt ist.
  - Ein Beispiel für die Klasse **`OrderedListIntSet`**:

```
IntSet set;
... // set wird irgendwie besetzt

// nun wird eine Kopie in der Darstellung OrderedListIntSet erzeugt:
OrderedListIntSet oListSet = new OrderedListIntSet(set);
```

- Die Klasse **`OrderedListIntSet`** benötigt also einen allgemeinen Copy-Konstruktor mit folgender Schnittstelle:

```
public OrderedListIntSet(IntSet s) { ... }
```

Iteratoren (bzw. Enumeratoren)

- Für den verallgemeinerten Copy-Konstruktur müssen wir unsere Mengendarstellungen um die Möglichkeit erweitern, die dargestellten Integer-Werte der Reihe nach aufzuzählen (zu enumerieren)
  - bzw. durch die Menge zu „laufen“ (zu iterieren)
- Dafür verwendet man eine eigene Objektklasse, die **Iteratoren** oder **Enumeratoren:**
- Ein Iterator lässt sich gut mit einem **Lesezeichen** zu einem Buch vergleichen:
  - Das Lesezeichen gehört nicht selbst zum Buch, kennt sich aber mit der Struktur des Buches gut aus.
  - Das Lesezeichen kann folgende Fragen beantworten:
    - Kommt noch eine weitere Seite im Buch?
    - Wenn ja, welches ist die nächste Seite?
- Analog besteht die Schnittstelle eines Iterators aus den zwei Methoden `boolean hasMoreElements()` und `Data nextElement()` (dabei ist `Data` der Typ der Elemente).

Die abstrakte Klasse `IntEnumeration`

- Die Klasse `IntEnumeration` definiert die Schnittstelle für `int` - Iteratoren:
  - Sie ist abstrakte Oberklasse für Enumeratoren über Ansammlungen von Elementen vom Typ `int` (z.B. über unsere Integer-Mengen);
  - Sie ist unabhängig von konkreten Datenstrukturen und besitzt deshalb keine eigenen Attribute.

```
abstract class IntEnumeration {

                // Methode, die angibt, ob noch weitere Elemente
                // folgen:
    public abstract boolean hasMoreElements();

                // Methode, die nur aufgerufen werden darf, falls noch
                // weitere Elemente folgen;
                // in diesem Fall wird das nächste Element ausgeliefert
                // und gleichzeitig die Enumeration um ein Element
                // weiterschaltet:
    public abstract int nextElement();

}
```


Die Klasse `ArrayIntSetEnumeration`

- **`ArrayIntSetEnumeration`**
  - ist eine Ableitung von `IntEnumeration` ;
  - zählt die Elemente von Integer-Mengen auf, die mittels Reihungen dargestellt sind;
  - „kennt“ die Struktur der Darstellungen, d.h. sie bekommt über ihren Konstruktur direkten Zugriff auf die Attribute `currentSize` und `array`.

```
class ArrayIntSetEnumeration extends IntEnumeration {

//Attribute:

              // direkter Zugriff auf die Attribute der
              // zugehoerigen Menge:
    private int currentSize;
    private int[] array;

              // Index, der die Menge durchlaeuft
              // (mit erstem Element initialisiert):
    private int index = 0;

  ...

}
```


Konstruktor und Methoden der Klasse `ArrayIntSetEnumeration`

```
//Konstruktor:
                    // die Attribute der zugehoerigen Menge werden
                    // als Parameter uebergeben:
    public ArrayIntSetEnumeration(int currentSize, int[] array) {
       this.currentSize = currentSize;
       this.array = array;
    }

// Implementierung der Methoden:
    public boolean hasMoreElements() {
                    // genau dann, wenn index noch nicht currentSize
                    // erreicht hat:
       return index < currentSize;
    }
    public int nextElement() {
                    // vor dem Weiterschalten Inhalt merken:
       int element = array[index];
                    // Enumeration weiterschalten:
       index++;
                    // gemerkten Inhalt ausliefern:
       return element;
    }
```

Analog: die Klasse `OrderedListIntSetEnumeration`

```
class OrderedListIntSetEnumeration extends IntEnumeration {
// Attribute:
              // direkter Zugriff auf das Attribut list der
              // zugehoerigen Menge:
    private OrderedIntList list;
// Konstruktor:
              // das Attribut list der zugehoerigen Menge wird als
              // Parameter uebergeben:
    public OrderedListIntSetEnumeration(OrderedIntList list) {
       this.list = list;
    }
// Implementierung der Methoden:
    public boolean hasMoreElements() {
                     // genau dann, wenn Liste nicht leer ist:
       return list != null;
    }
    public int nextElement() {
                     // vor dem Weiterschalten Inhalt merken:
       int item = list.getItem();
                     // Enumeration weiterschalten:
       list = list.getNext();
                     // gemerkten Inhalt ausliefern:
       return item;
    }
}
```


Erweiterung der Mengendarstellungen um Iteratoren

- Die abstrakte Menge **`IntSet`** erhält nun eine zusätzliche abstrakte Methode:

```
abstract public IntEnumeration getEnumeration();
```

- Sie wird in **`ArrayIntSet`** und **`OrderedArrayIntSet`** gleichlautend folgendermaßen implementiert:

```
public IntEnumeration getEnumeration() {
    return new ArrayIntSetEnumeration(currentSize, array);
}
```

- Damit ist das Ziel erreicht, über eine allgemeine Schnittstelle (**`IntEnumeration`**) eine Möglichkeit zu erhalten, durch die konkrete Datenstruktur zu navigieren.
- Für die Klasse **`OrderedListIntSet`** wird die Methode folgendermaßen implementiert:

```
public IntEnumeration getEnumeration() {
    return new OrderedListIntSetEnumeration(list);
}
```


Der verallgemeinerte Copy-Konstruktur für `ArrayIntSet`

- Mit den zur Verfügung gestellten Iteratoren lassen sich nun tatsächlich die gewünschten verallgemeinerten Copy-Konstruktoren realisieren.
- Beginnen wir mit dem für die Klasse **`ArrayIntSet`**:

```
// Konstruktor, der die Kopie einer beliebigen Menge liefert:

             // die Reihungsgroesse wird wieder so gewaehlt, dass
             // zusaetzliche Elemente Platz finden
   public ArrayIntSet(IntSet s) {
      currentSize = s.size();

      if (currentSize < DEFAULT_CAPACITY)
          array = new int[DEFAULT_CAPACITY];
      else
          array = new int[currentSize + DEFAULT_CAPACITY_INCREMENT];

             // die Elemente aus s werden mittels
             // Iterator uebertragen:
      int index=0;
      IntEnumeration enu = s.getEnumeration();
      while (enu.hasMoreElements()) {
         array[index] = enu.nextElement();
         index++;
      }
   }
```

Der Iterator ist ein eigenständiges Objekt, der wie ein Lesezeichen in **s** „steckt“ und weitergeschaltet werden kann.

Der verallgemeinerte Copy-Konstruktur für `OrderedArrayIntSet`

- Bei allgemeinen Mengen kann (leider) nicht angenommen werden, dass die Elemente vom Iterator in aufsteigender Reihenfolge geliefert werden.
- Die Elemente müssen also einzeln in die richtige Ordnung gebracht (einsortiert) werden.
  - Dafür stützen wir uns ab auf die Methode **`insert()`** :

```
public OrderedArrayIntSet(IntSet s) {
   if (s.size() < DEFAULT_CAPACITY)
       array = new int[DEFAULT_CAPACITY];
   else
       array = new int[s.size() + DEFAULT_CAPACITY_INCREMENT];

   currentSize = 0;    // vorerst ist Menge leer
         // Uebertragung der Elemente mittels Iterator
         // und insert():
   IntEnumeration enu = s.getEnumeration();
   while (enu.hasMoreElements())
      insert(enu.nextElement());
}
```

Der verallgemeinerte Copy-Konstruktur für `OrderedListIntSet`

- Analog der verallgemeinerte Konstruktor für die Darstellung auf sortierten Listen in der Klasse **`OrderedListIntSet`**.
- Diesmal stützen wir uns auf die Methode **`insertElement()`** der Klasse **`OrderedIntList`** ab, um die Elemente einzusortieren:

```
public OrderedListIntSet(IntSet s) {
  list = null;
  IntEnumeration enu = s.getEnumeration();
  while (enu.hasMoreElements()) {
    list = OrderedIntList.insertElement(enu.nextElement(), list);
  }
}
```


Beispiel: Anwendung der neuen Copy-Konstruktoren

- Die verallgemeinerten Copy-Konstruktoren erlauben es uns nun, Mengen beliebiger Darstellungen ineinander überzuführen:

```
class SetTest {
    public static void besetze (IntSet set) {
        for (int i=101; i<=110; i++)
            set.insert(i);
        System.out.println(set);
    }

    public static void main (String[] args) {
        ArrayIntSet arrSet = new ArrayIntSet(10);
        besetze(arrSet);
        OrderedArrayIntSet oArrSet = new OrderedArrayIntSet(arrSet);
        System.out.println(oArrSet);
        OrderedListIntSet oListSet = new OrderedListIntSet(oArrSet);
        System.out.println(oListSet);
    }
}
```

Die **dynamische Bindung** sorgt dafür, dass der jeweils „passende“ Iterator die Elemente der Menge aufzählt.

Zusammenfassung Iteratoren bzw. Enumeratoren

- Unseren Mengendarstellungen aus Kapitel 7 hat die Möglichkeit gefehlt, alle Elemente der Menge der Reihe nach aufzuzählen.
- Mit den Iteratoren haben wir eine Lösung gefunden, die Elemente aufzuzählen ohne get-Methoden für die spezielle Datenstruktur in die Mengen-Schnittstelle aufnehmen zu müssen.
  - Wir haben z.B. vermieden, eine Methode **`getArray()`** in die Schnittstelle von **`IntArraySet`** aufnehmen zu müssen.
- Ein Iterator ist ein eigenständiges Objekt
  - das die Datenstruktur der zugehörigen Menge kennt,
  - aber mit **`hasMoreElements()`** und **`nextElement()`** eine datenstruktur-unabhängige Schnittstelle bietet.
- Die abstrakte Klasse **`IntEnumeration`** *generalisiert* die allen Integer-Iteratoren gemeinsame Schnittstelle und erlaubt es damit, von der Datenstruktur zu *abstrahieren*.
- Java bietet einen Typ **`Enumeration`** mit exakt unserer Schnittstelle an, allerdings
  - als Interface statt abstrakter Klasse (siehe später)
  - und in generischer Form (siehe später).

Die datenstrukturunabhängige Implementierung weiterer Methoden der Klasse `IntSet` mittels Iteratoren

- Die Iteratoren erlauben es nun, weitere Methoden bereits in der Klasse **`IntSet`** (auf abstraktem Niveau) zu implementieren.
- Bei Verwendung von Iteratoren müssen **`contains()`**, **`size()`**, **`isSubset()`** und **`toString()`** nicht auf die konkrete Datenstruktur zugreifen.
  - Sie können also bereits auf abstrakter Ebene implementiert werden.
  - Die Unterklassen können entscheiden, ob sie diese Implementierung der Methoden wieder verwenden (erben) oder die Methoden re-implementieren (überschreiben).
    - Reimplementierung ist dann zu empfehlen, wenn Eigenschaften der Datenstruktur ausgenutzt werden können, um die Methoden effizienter zu realisieren.
    - Beispiel: Ausnutzen von Sortiertheit

Implementierung der Methoden `contains()` und `size()` in der Klasse `IntSet`

```
         // Abfrage, ob Element enthalten ist:
public boolean contains(int i) {
   IntEnumeration enu = getEnumeration();
   while (enu.hasMoreElements()) {
      int item = enu.nextElement();
         // Falls gefunden:
      if (item == i)
         return true;
   }
         // i nicht gefunden:
   return false;
}
         // Abfrage nach Groesse der Menge:
public int size() {
   int result = 0;
   IntEnumeration enu = getEnumeration();
   while (enu.hasMoreElements()) {
      result++;
      int dummy = enu.nextElement();
   }
   return result;
}
```

Auf dieser Ebene kann eine eventuell vorhandene Sortiertheit der Elemente nicht ausgenutzt werden!

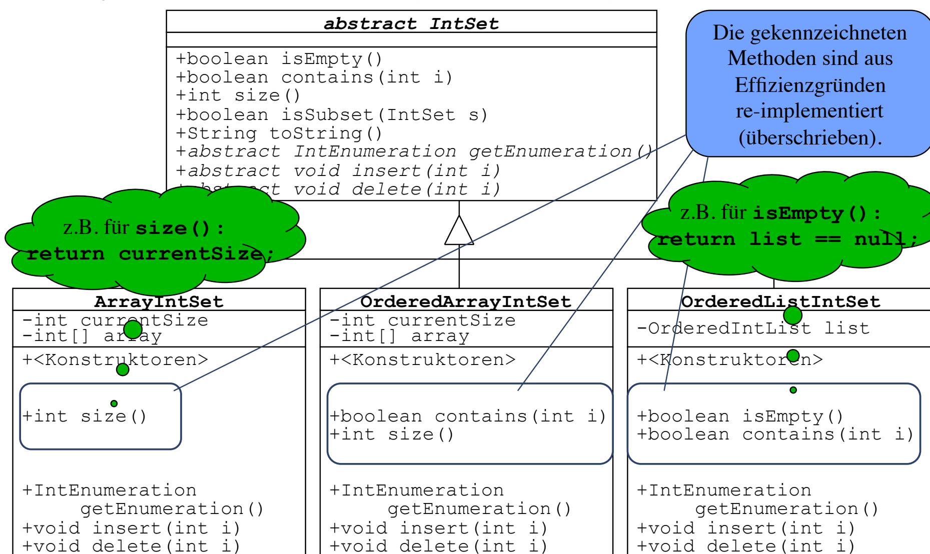
Das Funktionsergebnis wird hier nicht benötigt.

Implementierung der Methoden `isSubset()` und `toString()` in der Klasse `IntSet`

```
         // Abfrage nach Teilmengeneigenschaft:
public boolean isSubset(IntSet s) {
   IntEnumeration enu = getEnumeration();
   while (enu.hasMoreElements())
      if (! s.contains(enu.nextElement()))
         return false;
         // Teilmengeneigenschaft ist nie verletzt:
   return true;
}
         // Ausgabefunktion:
public String toString() {
   String result = "{";
   IntEnumeration enu = getEnumeration();
   while (enu.hasMoreElements()) {
      result += enu.nextElement();
      if (enu.hasMoreElements())
         result += ",";
   }
   return result + "}";
}
```

Auch hier kann eine eventuell vorhandene Sortiertheit der Elemente nicht ausgenutzt werden!

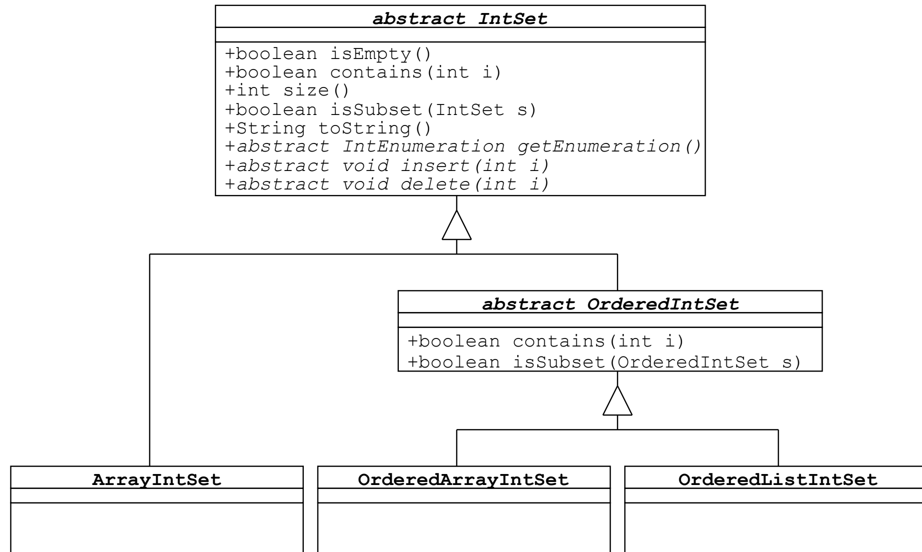
Der jetzige Stand der Mengendarstellungen im Modell



Die abstrakte Klasse `OrderedIntSet`

- Wir wollen noch eine abstrakte Menge **`OrderedIntSet`** als Oberklasse der sortierten Mengendarstellungen einführen.
  - **`OrderedIntSet`** generalisiert also die Eigenschaft, dass die Elemente der Menge sortiert sind.
  - Iteratoren über Unterklassen von **`OrderedIntSet`** liefern die Elemente in aufsteigender Reihenfolge.
  - Bereits in **`OrderedIntSet`** kann damit die effizientere Fassung der Methode **`contains()`** implementiert werden.
  - Auch in **`OrderedArrayIntSet`** kann dann beispielsweise eine effizientere Fassung des Copy-Konstruktors, speziell für Parameter vom Typ **`OrderedIntSet`**, realisiert werden.
  - In **`OrderedIntSet`** kann bereits die effizientere Fassung der Methode **`isSubset()`**, speziell für Parameter vom Typ **`OrderedIntSet`**, realisiert werden.
- **Definition Überladen**: Wenn es in einer Klasse mehrere Konstruktoren oder Methoden mit demselben Bezeichner aber unterschiedlicher Funktionalität gibt, so sprechen wir von **Überladen**.

Die Klasse `OrderedIntSet` *im Modell*



Reimplementierung von `contains()` *in* `OrderedIntSet`

```
abstract class OrderedIntSet extends IntSet {

// Methoden, die bei Sortiertheit effizienter implementierbar sind:

              // Abfrage, ob Element enthalten ist:
    public boolean contains(int i) {
       IntEnumeration enu = getEnumeration();
       while (enu.hasMoreElements()) {
          int item = enu.nextElement();
              // Falls gefunden:
          if (item == i)
             return true;
              // Ordnung wird ausgenutzt:
          if (item > i)
             return false;
       }
              // i nicht gefunden:
       return false;
    }
  ...
}
```

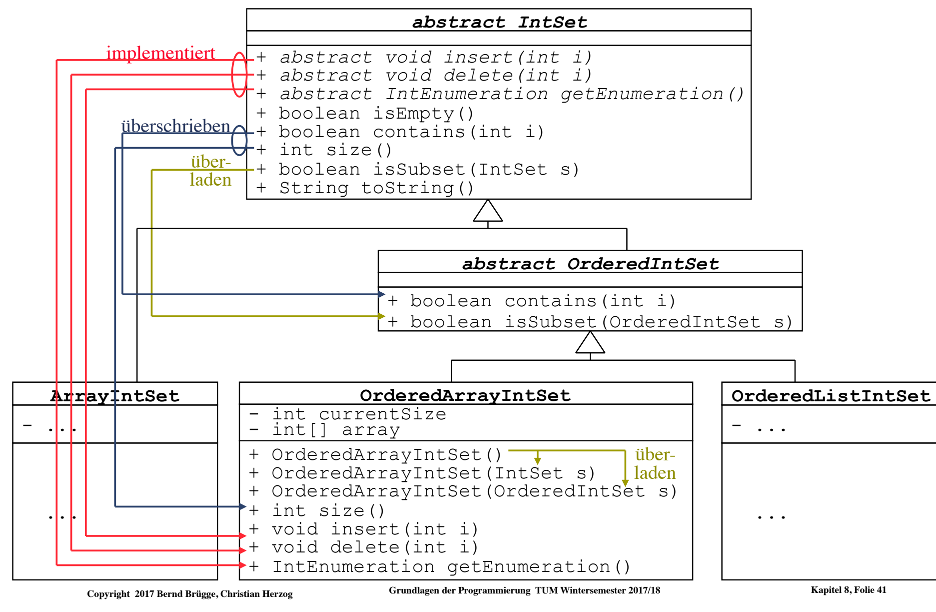

Überladen von `isSubset()` *in* `OrderedIntSet` *für sortierte Parametermengen*

```
    public boolean isSubset(OrderedIntSet s) {
              // Enumeration für die Menge selbst:
       IntEnumeration enu = getEnumeration();
              // Enumeration für andere Menge:
       IntEnumeration enuS = s.getEnumeration();

       while (enu.hasMoreElements()) {
           int item = enu.nextElement();

           if (!enuS.hasMoreElements())         // keine Teilmenge
              return false;

              // kleinere Elemente in s ueberspringen:
           int itemS;
           do {
              itemS = enuS.nextElement();
           } while (item > itemS && enuS.hasMoreElements());

           if (item != itemS)
              // Element der Menge kann nicht auch in s sein
              return false;
       }
              // Teilmengeneigenschaft ist nie verletzt:
       return true;
    }
```

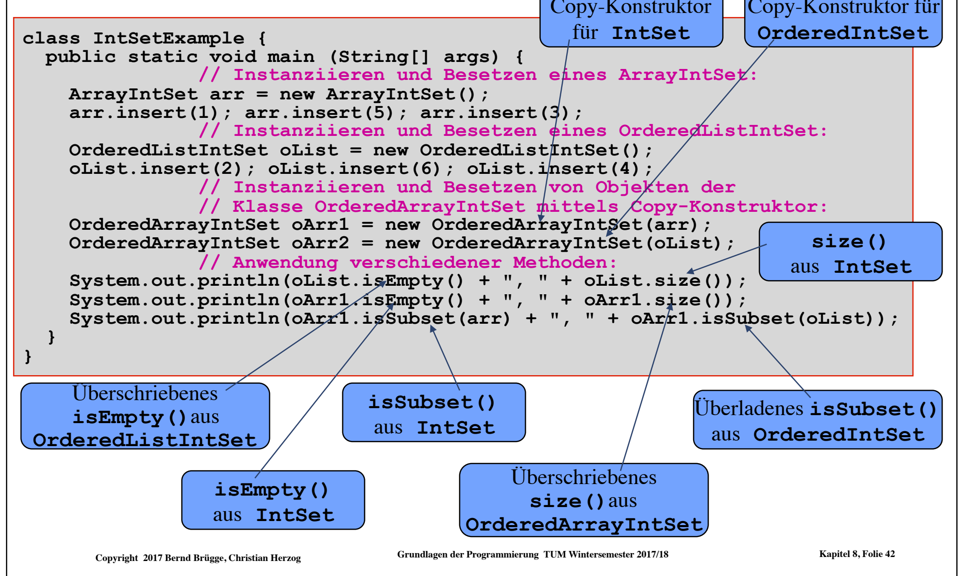

Zusammenfassung: Implementieren, Überschreiben, Überladen

- **Implementieren**:
  - Unterklassen **implementieren** abstrakte Methoden einer abstrakten Oberklasse. Die Funktionalität ist identisch.
  - Unterklassen, in denen nicht alle abstrakten Methoden der Oberklasse implementiert sind, sind selbst abstrakt.
- **Überschreiben**:
  - Unterklassen **überschreiben** (oder **reimplementieren**) bereits implementierte Methoden der Oberklasse, wenn in der Unterklasse eine speziellere Behandlung nötig oder eine effizientere Realisierung möglich ist. Die Funktionalität ist identisch.
  - Man bezeichnet dies auch als **Polymorphie** („Vielgestaltigkeit“ der Methode“).
- **Überladen**:
  - Methoden oder Konstruktoren werden **überladen**, wenn für eine **spezielle Funktionalität** eine speziellere Behandlung nötig oder eine effizientere Realisierung möglich ist.

Die Begriffe am Beispiel der Mengendarstellungen



Ein Beispiel für die Anwendung überschriebener bzw. überladener Methoden




```
class IntSetExample {
  public static void main (String[] args) {
                // Instanziieren und Besetzen eines ArrayIntSet:
    ArrayIntSet arr = new ArrayIntSet();
    arr.insert(1); arr.insert(5); arr.insert(3);
                // Instanziieren und Besetzen eines OrderedListIntSet:
    OrderedListIntSet oList = new OrderedListIntSet();
    oList.insert(2); oList.insert(6); oList.insert(4);
                // Instanziieren und Besetzen von Objekten der
                // Klasse OrderedArrayIntSet mittels Copy-Konstruktor:
    OrderedArrayIntSet oArr1 = new OrderedArrayIntSet(arr);
    OrderedArrayIntSet oArr2 = new OrderedArrayIntSet(oList);
                // Anwendung verschiedener Methoden:
    System.out.println(oList.isEmpty() + ", " + oList.size());
    System.out.println(oArr1.isEmpty() + ", " + oArr1.size());
    System.out.println(oArr1.isSubset(arr) + ", " + oArr1.isSubset(oList));
  }
}
```


Konzepte für Wiederverwendbarkeit in Java

- ✓ Vererbung
- ✓ Abstrakte Klassen
- ➔ Generische Klassen
- ❖ Schnittstellen (*interfaces*)

Generische Klassen

- Wir haben die Klassen zur Darstellung von Mengen und Listen in den letzten Vorlesungen eingeführt, um grundsätzliche Konzepte zu erklären.
  - Aus diesem Grund hatten wir uns auf Mengen und Listen beschränkt, deren Knoten nur applikationspezifische Daten vom Typ **`int`** speichern konnten.
  - Was uns jetzt interessiert, ist die Frage, ob wir diese Strukturen auch für andere Klassen aus der Applikationsdomäne (Personen, Autoteile, Flugzeugreservierungen, …) nehmen können.
- Wir wollen deshalb jetzt Mengen- und Listen-Klassen entwickeln, die eine generelle Knotenklasse benutzen, in der wir beliebige Daten speichern und verarbeiten können.
- Als Beispiel führen wir die Klasse **`ArrayIntSet`**, die nur Mengen von **`int`**-Elementen darstellen kann, in eine Klasse **`ArraySet`** über, die Mengen beliebiger Elemente darstellen kann.
  - Zur Vereinfachung gehen wir dabei von der „ursprünglichen“ Klasse **`ArrayIntSet`** aus Kapitel 7 aus, die noch nicht in eine Hierarchie eingebettet ist.

Von ArrayIntSet zu ArraySet (1. Version)

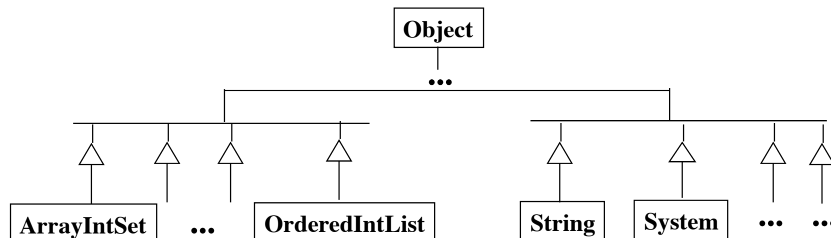
```
class ArraySet {
// Attribute:
    private static final int DEFAULT_CAPACITY = 5;
    private static final int DEFAULT_CAPACITY_INCREMENT = 5;
    private int currentSize;     // aktuelle Groesse der Menge
    private Object[] array;   // speichert die Elemente der Menge
// einer der Konstruktoren fuer die leere Menge:
    public ArraySet (int capacity) {
       array = new Object[capacity];
       currentSize = 0;
    }
...
// ausgewählte Methoden:
...
    public boolean contains(Object o) {
       for(int index=0; index<currentSize; index++) {
          if (array[index] == o)
             return true;
       }
       return false;
    }
...
```


Verallgemeinerung der Methode delete() (1. Version)

```
public void delete(Object o) {

   // Indexposition von o ermitteln:
   int index = 0;
   while (index < currentSize && array[index] != o)
      index++;

   if (index >= currentSize) {
      // in diesem Fall ist o nicht in Menge enthalten
      System.out.println("delete: " + o + " nicht enthalten!");
      return;
   }

   // Sonst steht o auf Position index; o wird geloescht, indem
   // das rechteste Element auf Position index umgespeichert wird
   array[index] = array[currentSize-1];

   // Konsistente Verminderung von currentSize:
   currentSize--;
}
```


Die Klassenhierarchie in Java

- In Java unterscheiden wir benutzerdefinierte Klassen und vordefinierte Klassen.
  - Benutzerdefinierte Klassen: **ArrayIntSet**, **OrderedIntList**, ...
  - Vordefinierte Klassen: **String**, **System** ,...
- Alle Klassen in Java bilden eine Klassenhierarchie.
  - Die Superklasse der Klassenhierarchie heißt **Object**
  - Jede Java-Klasse, bis auf **Object**, hat eine Superklasse.



Die Java-Klasse Object

```
public class Object {
  ...
   // ausgewaehlte Instanzmethoden:

   public String toString() {…}
      // Konvertiert die Werte der Attribute eines
      // Objekts in eine Zeichenkette
      // Wird in der Regel von Unterklassen geeignet
      // überschrieben
  ...
   public boolean equals(Object obj) {…}
      // true, wenn beide Objekte gleich sind.
      // Voreinstellung: Referenzgleichheit (wie bei == )
      // Wird in der Regel von Unterklassen geeignet
      // überschrieben
  ...
}
```

Vollständige Definition von Object => Java Referenz-Manual

Die Operationen == und != in den Methoden `contains()` und `delete()` der Klasse `ArraySet`

```
public boolean contains(Object o) {
     for(int index=0; index<currentSize; index++) {
        if (array[index] == o)
           return true;
     }
     return false;
}

 public void delete(Object o) {
    int index = 0;
    while (index < currentSize && array[index] != o)
       index++;
    if (index >= currentSize) {
        System.out.println("delete: " + o + " nicht enthalten!");
        return;
    }
    array[index] = array[currentSize-1];
    currentSize--;
 }
```

Auf diese Weise kann nur die Gleichheit bzw. Ungleichheit der **Referenzen** ermittelt werden.

Verbesserung: Verwendung der Methode **`equals()`** der Klasse **`Object`**

Verbesserte Version der Methoden `contains()` und `delete()` der Klasse `ArraySet`

```
public boolean contains(Object o) {
    for(int index=0; index<currentSize; index++) {
       if (array[index].equals(o))
          return true;
    }
    return false;
}

public void delete(Object o) {
   int index = 0;
   while (index < currentSize && ! array[index].equals(o))
      index++;
   if (index >= currentSize) {
       System.out.println("delete: " + o + " nicht enthalten!");
       return;
   }
   array[index] = array[currentSize-1];
   currentSize--;
}
```

`equals()` kann in den benutzerdefinierten Unterklassen von **`Object`** so überschrieben werden, dass es die jeweils gewünschte Gleichheit realisiert.

Das Problem mit den Grundtypen

- Java unterscheidet zwischen **Grundtypen** und **Objekttypen.**
  - Objekttypen sind z.B. **`String`** und alle benutzerdefinierten Klassen wir **`ArrayIntSet`**.
    - Objekttypen sind Unterklassen von **`Object`** (eventuell über mehrere Stufen).
  - Die 8 Grundtypen in Java sind **`char`**, **`boolean`** und die 6 Typen zur Darstellung von Zahlen:
    - **`byte`**, **`short`**, **`int`** und **`long`** stellen ganze Zahlen mit Vorzeichen mit 8, 16, 32 bzw. 64 Bit dar.
    - **`float`** und **`double`** sind Gleitkommazahlen mit 32 bzw. 64 Bit.
- **Problem**: Grundtypen stellen **keine Unterklassen** von **`Object`** dar.
  - Unsere generische Klasse **`ArraySet`** kann damit keine Elemente vom Typ **`int`** aufnehmen.
- **Lösung**: Grundtypen werden in Objekttypen (sog. **Hüll-Klassen**) eingebettet.

Eine Hüll-Klasse für `int`

- Um mit der generischen Klasse **`ArraySet`** auch Mengen ganzer Zahlen darstellen zu können, bilden wir um **`int`** herum eine Hüllklasse **`MyInteger`**:

```
class MyInteger {
// Attribute (Datenstruktur):
    private int value;
// Konstruktor huellt int ein:
    public MyInteger(int value) {
        this.value = value;
    }

// sonstige Methoden:
    // liefert den int-Wert aus:
    public int getValue() {
       return value;
    }

    // Ausgabefunktion;
    // ueberschreibt entsprechende
    // Methode von Object:
    public String toString() {
       return "" + value;
    }

    // vergleicht, ob zwei MyIntegers
    // denselben Wert haben;
    // ueberschreibt entsprechende
    // Methode aus Object:
    public boolean equals(Object o) {
      if (o == null)
        return false;
      if (getClass() != o.getClass())
        return false;

      // Typkonvertierung von Object auf
      // MyInteger, damit Zugriff auf
      // nicht vererbte Attribute
      // moeglich werden:
      MyInteger i = (MyInteger) o;
      return value == i.getValue();
    }
} // class MyInteger
```

`getClass()` ist eine in **`Object`** definierte Methode

Hüllklassen und Typkonvertierung

- Mit Hilfe der Hüllklasse `MyInteger` lassen sich jetzt ganze Zahlen in unsere Mengendarstellung eingeben:

```
ArraySet set = new ArraySet();
set.insert(new MyInteger(27));
```

- Wenn eine Variable vom Typ `Object` auf ein Objekt vom Typ `MyInteger` verweist, muss vor dem Zugriff auf nicht vererbte Merkmale der Klasse `MyInteger` eine Typkonvertierung (*type casting*) vorgenommen werden:

```
Object o = new MyInteger(27);
System.out.println(o.getValue());
```

`getValue()` für Variable vom Typ `Object` nicht definiert!

Statt dessen Typkonvertierung vor dem Zugriff:

```
MyInteger i = (MyInteger) o;
System.out.println(i.getValue());
```

Oder in einer Anweisung:

```
System.out.println( ((MyInteger) o).getValue() );
```

- In Java gibt es bereits vorgefertigte Hüllklassen für alle Grundtypen.
  - Z.B. die Klasse `Integer` für `int`.

`Integer` **im Referenz-Manual**

Typkonvertierung

- **Definition**: Eine **Typkonvertierung** konvertiert einen Typ in einen anderen.
  Die Konvertierung wird mittels eines Typ-Bezeichners gemacht, der geklammert vor dem zu konvertierenden Ausdruck steht.
- Beispiele:
  - `char c = 'a';` // Variable `c` vom Type `char`
  - `int k;` // Variable `k` vom Typ `int`
  - `k = (int) c;` // Die Variable k bekommt den `int`-Wert von `a` // (der `int`-Wert ist 97, der sogenannte ASCII- // Wert von `'a'`)
  - `Object o = new Integer(27);` // Die Java-Hüllklasse
  - `if ( ((Integer) o).intValue() > 0) ...`

Typkonvertierungen in Java

- Typkonvertierung in Java folgt vielen Regeln und Konventionen.
  In einigen Fällen kann der Programmierer sogar implizite Typkonvertierungen veranlassen. Beispiele:

char → int:

```
char ch;
int k;
k = ch; // konvertiert einen Buchstaben
        // in eine 32-Bit-Zahl
```

int → double:

```
int i;
double d;
d = i;  // konvertiert eine ganze
        // 32-Bit- Zahl in eine 64-Bit-
        // Gleitkommazahl
```

- In anderen Fällen muss der Programmierer explizite Typkonvertierungen machen. Beispiele

double → int:

```
int i;
double d;
i = d;  // geht nicht implizit!
    // Eine Zahl vom Typ double passt
    // nicht in eine Zahl vom Typ int.
    // Explizite Typkonvertierung:
```

- `i = (int) d;`
- ***int → String: mit valueOf()***
  - String s;
  - int i;
  - i = 45;
  - s = String.valueOf(i);

Zusammenfassung generische Klassen und Hinweis auf Neuerungen seit Java 5

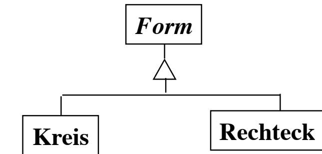
- Eine **generische Klasse** erlaubt eine allgemeine Formulierung von Datenstrukturen wie Listen und Mengen.
- Anstelle eines festen Basistyps wird die Klasse `Object` verwendet.
  - Die üblichen Operationen (contains, insert, delete) müssen auf Daten vom Typ `Object` definiert werden.
- In der generischen Klasse wird der Basistyp also nicht festgelegt, sondern erst in der Anwendung, die die generische Klasse für ihre applikationsspezifischen Daten (`Person`, `Addresse`, `Kunde`, `Bauteil`, ...) wieder verwenden will.
- Eine Menge vom Typ `ArraySet` kann aber gleichzeitig sowohl Elemente vom Typ `Integer` als auch beispielsweise vom Typ `String` enthalten.
  - Für die Konsistenz der Daten muss der Programmierer sorgen!
- Seit Java-Version 5 sind auch **parametrisierte Klassen** erlaubt, wie beispielsweise die die sog. *Templates* in der Sprache C++. Diese heißen dann wieder „generische Klassen“:
  - Beispiel: `public class ArraySet<T> extends Set<T> { ... }`
  - Verwendung: `ArraySet<Integer> set = new ArraySet<Integer>(20);`

Konzepte für Wiederverwendbarkeit in Java

- ✓ Vererbung
- ✓ Abstrakte Klassen
- ✓ Generische Klassen
- ➔ Schnittstellen (*interfaces*)

Copyright 2017 Bernd Brügge, Christian Herzog

Noch ein Beispiel für abstrakte Methoden



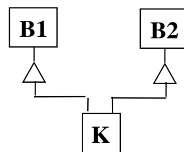

```
public abstract class Form {
    public abstract double flaeche();
    public abstract double umfang();
}
class Kreis extends Form{
    private double r;
    public Kreis() { this(1.0); }
    public Kreis(double radius) { r = radius; }
    public double flaeche() { return Math.PI * r * r; }
    public double umfang() { return 2 * Math.PI * r; }
    public double getRadius() { return r; }
}
```

```
class Rechteck extends Form {
    private double b, h;
    public Rechteck() { this(0.0, 0.0); }
    public Rechteck(double breite, double hoehe)
      {b = breite; h = hoehe; }
    public double flaeche() { return b * h; }
    public double umfang() { return 2 * (b + h); }
    public double getBreite() { return b; }
    public double getHoehe() { return h; }
}
```

Copyright 2017 Bernd Brügge, Christian Herzog

Mehrverfachvererbung

- **Definition Mehrfachvererbung:** Eine Klasse K erbt von mehreren Superklassen $B_1,\ldots, B_n$.

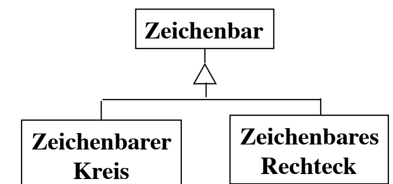


- Mehrfachvererbung kommt in der Modellierung ganz natürlich vor. Schauen wir uns noch einmal die Klasse **`Form`** an.
  - Wir wollen diese Klasse so erweitern, dass wir eine Anzahl von Formen auf dem Bildschirm malen können.

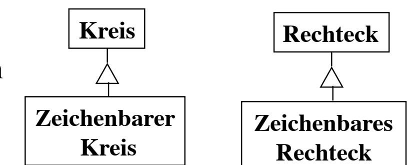
Copyright 2017 Bernd Brügge, Christian Herzog

Modellierung von zeichenbaren Formen

- Wir könnten eine abstrakte Klasse **`Zeichenbar`** definieren, und dann wieder verschiedene Unterklassen definieren, wie z.B. **`ZeichenbarerKreis`**, **`ZeichenbaresRechteck`**, usw.

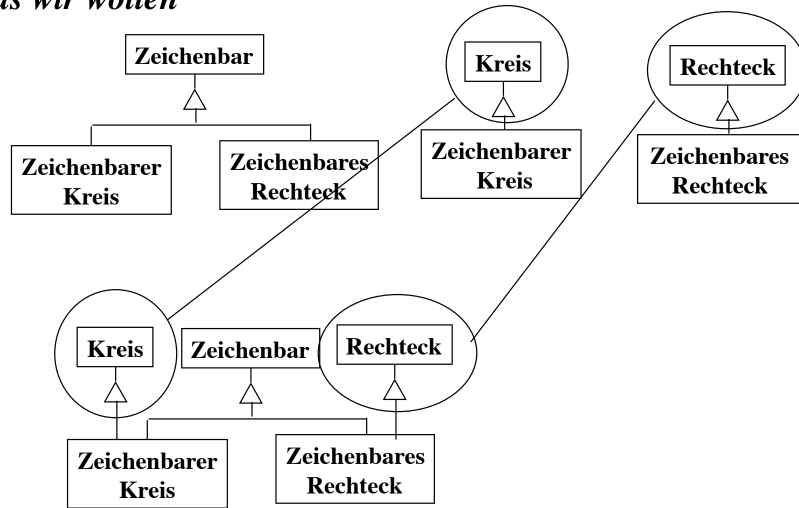


- Wir wollen aber auch, daß diese **`Zeichenbar`** Typen die Methoden **`flaeche`** und **`umfang`** anbieten.
  - Um diese Methoden zu reimplementieren, würden wir gern **`ZeichenbarerKreis`** zu einer Unterklasse von **`Kreis`** machen, und **`ZeichenbaresRechteck`** zu einer Unterklasse von **`Rechteck`**.



Copyright 2017 Bernd Brügge, Christian Herzog

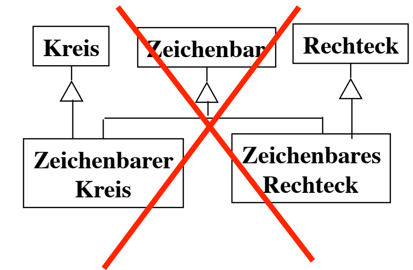
Was wir wollen



Copyright 2017 Bernd Brügge, Christian Herzog

Mehrfachvererbung ist in Java nicht erlaubt

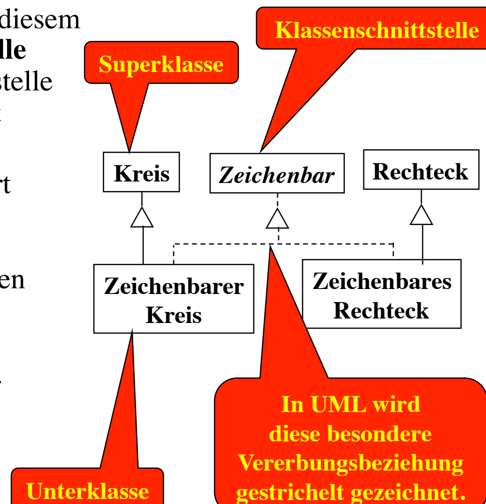
- Java verbietet Mehrfachvererbung.
  - In Java können Klassen nur eine Superklasse haben.
  - Wenn **`ZeichenbarerKreis`** Unterklasse von **`Kreis`** ist, dann kann sie nicht gleichzeitig Unterklasse von **`Zeichenbar`** sein.



Copyright 2017 Bernd Brügge, Christian Herzog

Java-Implementierung mit Klassenschnittstelle

- Java hat eine andere Lösung zu diesem Problem: die **Klassenschnittstelle** (interface). Eine Klassenschnittstelle ist wie eine abstrakte Klasse mit folgenden Unterschieden:
  - Sie benutzt das Schlüsselwort "**`interface`**" anstatt "**`abstract class`**".
  - Sie darf keine Instanzvariablen haben, nur Konstanten ("**`final static`**").
  - Alle Methoden sind abstrakt.



Copyright 2017 Bernd Brügge, Christian Herzog

Beispiel einer Klassenschnittstelle in Java

Schlüsselwort: interface

```
public interface Zeichenbar {
    public abstract void setFarbe(Farbe f);
    public abstract void setPosition(double x, double y);
    public abstract void zeichne();
}
```

Keine Instanzvariablen: nicht erlaubt!

Ein Interface enthält nur abstrakte Methoden: keinen Rumpf, keine Klammern {}

- Sprachgebrauch:
  - Eine Klasse erweitert **(extends)** ihre Superklasse
  - Eine Klasse implementiert **(implements)** eine Schnittstelle.
- Beispiel:

```
public class ZeichenbaresRechteck extends Rechteck
                                  implements Zeichenbar {... }
```

Copyright 2017 Bernd Brügge, Christian Herzog

Implementierung einer Java-Schnittstelle

```
interface Zeichenbar {
  public void setFarbe(Farbe f);
  public void setPosition(double x,
   double y);
  public void zeichne();
}
```

```
public class ZeichenbaresRechteck extends
   Rechteck implements Zeichenbar {
   private Farbe lf;
   private double lx, ly;
   private void zeichneRechteck () {
     // Zeichnet ein Rechteck in Farbe lf in
     // Position lx, ly. Breite und Höhe
     // bekommt man durch Aufruf von
     // getBreite() und getHoehe() (öffentliche
     // Methoden in Rechteck).
   };
   // Implementierungen von Zeichenbar:
   public void setFarbe(Farbe f)
     { lf = f; }
   public void setPosition(double x, double y)
     { lx = x; ly = y; }
   public void zeichne()
     { zeichneRechteck(); }
}
```


Wie benutzt man Klassenschnittstellen in Java?

- Im Gegensatz zur Vererbung von Klassen, bei denen in Java nur die Einfachvererbung zulässig ist, können auch mehrere (Klassen-)Schnittstellen mittels der **`implements`**-Klausel aufgeführt und implementiert werden.
- Eine Klasse, die eine Schnittstelle realisiert, muss alle in der Schnittstelle genannten Methoden implementieren.
- Eine Schnittstelle ist in Java ein Referenztyp (reference type), d.h. man kann Variablen oder Parameter mit einer Schnittstelle als Typ deklarieren.
  - Dies wird oft bei der Programmierung generischer Klassen gemacht.

Eine Basistyp für generische Mengen auf sortierten Reihungen

- Auf den Elementen einer sortierten Reihung muss eine Ordnung definiert sein.
  - Deshalb scheidet **`Object`** als Basistyp für eine generische Klasse aus.
  - Wir definieren deshalb einen abstrakten Basistyp **`Comparable`**, dessen Schnittstelle eine Vergleichsoperation enthält.
  - Um für abgeleitete Klassen zusätzliche Vererbungen zuzulassen, definieren wir **`Comparable`** als Schnittstelle:

```
interface Comparable {
       // die Vergleichsoperation compareTo liefert
       // -1, falls this kleiner als c ist
       // 0,  falls this gleich c ist
       // 1,  falls this groesser als c ist
   public abstract int compareTo(Comparable c);
}
```

- **`Comparable`** kann nun als Basistyp für die generische Mengendarstellung auf sortierten Reihungen verwendet werden:

Eine generische Klasse `OrderedArraySet` für sortierte Mengen

- Die Attribute der Klasse **`OrderedArraySet`**:

```
   private int currentSize;        // aktuelle Groesse der Menge
   private Comparable[] array;     // speichert die Elemente
```

- Die Methode **`contains()`** der Klasse **`OrderedArraySet`**:

```
public boolean contains (Comparable c) {
  for(int index=0; index<currentSize; index++) {
    if (array[index].compareTo(c) == 0)
       // Element gefunden
      return true;
    if (array[index].compareTo(c) == 1)
       // größeres Element erreicht
      return false;
  }
       // Ansonsten Element nicht enthalten
  return false;
}
```

Die Methode `delete()` der Klasse `OrderedArraySet`

```
public void delete(Comparable c) {

              // Indexposition von c ermitteln:
  int index = 0;
  while (index < currentSize && array[index].compareTo(c) == -1)
    index++;

  if (index >= currentSize || array[index].compareTo(c) == 1) {
              // in diesem Fall ist c nicht in Menge enthalten
    System.out.println("delete: " + c + " nicht enthalten!");
    return;
  }

              // Sonst steht c auf Position index; c wird geloescht,
              // indem die Elemente rechts von Position index nach
              // links umgespeichert werden
  for (int k=index+1; k<currentSize; k++)
    array[k-1] = array[k];

              // Konsistente Verminderung von currentSize:
  currentSize--;
}
```


Erweiterung von `MyInteger` um die Methode `compareTo()`

- Damit unsere Klasse **`MyInteger`** für Elemente einer generischen Liste vom Typ **`OrderedList`** verwendet werden kann, muss sie die Schnittstelle **`Comparable`** implementieren, d.h.
  - der Klassenkopf muss ergänzt werden um **`implements Comparable`**
  - Die Vergleichsoperation **`compareTo()`** muss implementiert werden:

```
class MyInteger implements Comparable {
  ...
      // Konstruktor und Methoden wie bisher
  ...
  public int compareTo(Comparable c) {
      // Zunaechst Typkonvertierung von c auf MyInteger:
    MyInteger i = (MyInteger) c;
    if (value < i.value)
        return -1;
    if (value == i.value)
        return 0;
    return 1;
  }
}
```


Zusammenfassung: Wiederverwendung durch Vererbung

- Klassenvererbung durch Implementierungs- und Schnittstellenvererbung.
- **Implementierungsvererbung**: Der Programmierer schreibt Java-Methoden und erlaubt, dass einige von ihnen überschrieben werden.
- **Schnittstellenvererbung**:
  - Ein Klassenschnittstelle (Java-Interface) ist wie eine abstrakte Klasse, kann aber keine Attribute (Instanzvariablen) haben.
  - In einer Schnittstellendefinition sind alle Methoden abstrakt, d.h. keine Methode hat einen Methodenrumpf.
  - Der Ersteller einer Schnittstelle spezifiziert, welche Methoden von den Klassen realisiert werden müssen, die diese Schnittstelle implementieren.
  - Die Methodenrümpfe werden in den Klassen definiert, die die Methoden bzw. die Schnittstelle implementieren.

Hinweis

- Auf der Homepage der Vorlesung steht unter.
  - **`HierachieGenerisch.zip`**

  die komplette Hierarchie von Mengendarstellungen als generische Klassen zur Verfügung.